

## Chapter 12

# ASPECT COMPOSITION USING COMPOSITION FILTERS

Lodewijk Bergmans, Mehmet Akşit and Bedir Tekinerdoğan

*TRESE group, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands. email: {bergmans, aksit, bedir}@cs.utwente.nl, www: <http://trese.cs.utwente.nl>*

**Key words:** composition, aspects, multiple views, view partitioning, view extension, view refinement, history sensitiveness, synchronization, composition filters

**Abstract:** This chapter first discusses a number of software reuse and extension problems in current object-oriented languages. For this purpose, a *change case* for a simplified mail system is presented. Each evolution step in the change case consists of the addition or refinement of certain aspects to existing classes. These examples illustrate that both inheritance and aggregation mechanisms cannot adequately express certain aspects of evolving software. This deficiency manifests itself in the number of superfluous (method) definitions that are required to realize the change case. As a solution to these problems, the composition filters model is introduced. We evaluate the effectiveness of various language mechanisms in coping with evolving software as in the presented change case.

## 1. INTRODUCTION

One of the most important principles in software engineering is the separation of concerns principle [7]. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendibility and reusability. Despite a common agreement on the necessity of the application of the separation of concerns principle, its application for large and complex

applications that involve multiple concerns may become problematic. The reason for this is that concerns that are separated at the design level may become scattered at the programming code level and be tangled with implementations of other concerns within methods [10].

Problems in composing various concerns have been described in different publications. For example, in [12, 13, 4] the conflicts between the implementation of synchronization concerns and inheritance in object-oriented concurrent programming languages are described. In [3] the problems in composing concerns for real-time specifications are discussed. In [1, 6] the so-called *multiple views* composition problems have been addressed. In all these cases, a *conceptually sound* composition cannot be adequately expressed in a given language. The term *inheritance anomaly* was coined in [12, 13] to denote a more specific case of *composition anomaly* where the embedding of synchronization code in method implementations causes unnecessary redefinitions if the synchronization code has to be reused and/or extended through inheritance. In those cases, it typically appears that the problems can be patched by overriding in a subclass substantial parts of the methods defined by its superclass. This conflicts, however, with the intended reuse and causes reduced maintainability.

In section 2 we will illustrate various composition anomalies by using an example change scenario of a simple mail system. Using this example, we show that the conventional object-oriented composition techniques cannot deal with certain concern compositions satisfactorily. In section 3 we will introduce the composition filters model, which is an extension to the object-oriented model. Composition filters offer a better support for reusing and extending software with certain concerns, for example that are presented in the mail system case. In section 4 we will provide the composition filters solution to the composition anomalies that have been addressed in section 2. Finally, we will provide our conclusions in section 5.

## 2. EXAMPLE: DESIGN OF A MAIL SYSTEM

Figure 1 shows the class diagram of a simple mail system, which consists of classes *Originator*, *Email*, *MailDelivery* and *Receiver*. Class *Email* represents the electronic messages sent in this system and provides methods for defining, delivering and reading mails. For example, methods to write and read the attributes *originator*, *receiver*, and *content* of a mail object. The methods *putRoute()*, *getRoute()*, *deliver()* and *isDelivered()* are used by class *MailDelivery* while routing and delivering the messages from originators to receivers. The method *reply()* is used by receiver objects to send a reply

message. In this text, *Email* will be used as the base class that can be specialized into various kinds of email objects.



Figure 1: The interface methods of class *Email*

To illustrate a number of composition anomalies, class *Email* will be extended to support additional concerns. The concerns that we will address are the following:

1. *adding multiple views*, whereby the access to the mail interface is distinguished for a user and system view.
2. *view partitioning*, whereby the existing views are partitioned into additional sub-views.
3. *view extension*, whereby the views are extended.
4. *history sensitive behavior*, whereby information on history is logged.
5. *synchronization to multiple classes*, whereby locking mechanisms are added to multiple classes.

The motivation for these change cases is to show that it is in many cases impossible to define such extensions in the object-oriented model without superfluous redefinitions, i.e. composition anomalies.

Object-orientation provides two different mechanisms for composing concerns; either through aggregation or through inheritance (see also chapter 2, section 2.2.5). For each change case, we will discuss the application of both mechanisms.

## 2.1 Adding Multiple Views

The current implementation of class *Email* allows any client object to access e.g. the contents of a mail. We specialize class *Email* into *USViewMail* (User/System-View) and restrict access to its methods based on the class of the client object (i.e. the object that was the sender of the invocation). If the client is of the *User* type (i.e. an *Originator* or a *Receiver*), it is allowed to execute the methods *putOriginator()*, *putReceiver()*, *putContents()*, *getContents()*, *send()* and *reply()*. The methods *approve()*, *putRoute()* and *deliver()* are used by the clients of the *System* type (i.e. an instance of class *MailDelivery*). No restrictions are required for the other methods.

We assume that the identity of the client object (the sender of the message) can be obtained<sup>1</sup>. The following two subsections discuss aggregation-based and inheritance-based approaches within the conventional object-oriented model as supported by programming languages such as Java, C++ and Smalltalk.

In the case of aggregation-based composition, the *USViewMail* object encapsulates an instance of class *Email* and implements the view checking operations *userView()* and *systemView()*<sup>2</sup>. For each method that requires a view constraint to be enforced, additional code must be inserted that implements this constraint. Because the methods have already been implemented in class *Email*, invoking the appropriate method in the encapsulated *Email* object reuses this implementation. For example method *putOriginator()*, which is subject to the ‘User’ view can be implemented as follows in pseudocode:

```
USViewMail::putOriginator(Object anOriginator)
  if self.userView() // returns true if view applies
  then return imp.putOriginator(anOriginator)
  else self.viewError();
```

The class diagram in Figure 2 shows aggregation-based composition of multiple views. Notice that in this implementation strategy, all the methods have to be declared and implemented by class *USViewMail*, even those methods that do not require any view enforcement. This is because, these methods must be accessible through the *USViewMail* object.

<sup>1</sup> Note that in most language implementations this is far from trivial, if not impossible. For example in Smalltalk and Java there are –computationally expensive– ways to access the identity of the client through the calling stack.

<sup>2</sup> We implement view checking in separate methods for the purpose of reuse.

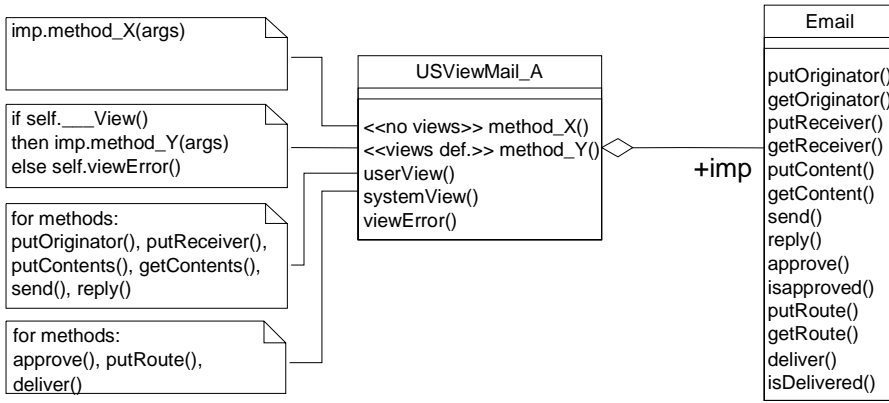


Figure 2: Aggregation-based composition of multiple views

Figure 3 provides the class diagram for the inheritance-based composition. View checking is again implemented at the start of each view-constrained method, reuse is now realized through *super* calls. Only the methods that require views have to be redefined; other methods can be inherited from the superclasses.

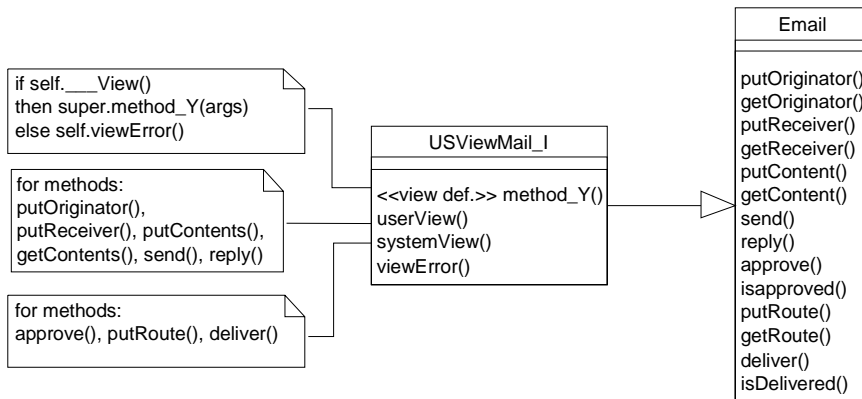


Figure 3: Inheritance-based composition of multiple views

The method *putOriginator()* is implemented as follows:

```

USViewMail::putOriginator(Object anOriginator)
    if self.userView()
    then return super.putOriginator(anOriginator)
    else self.viewError();
    
```

Adopting aggregation-based composition, *USViewMail* implements 16 methods. Among these, 9 methods implement view checking and forwarding, 5 methods are used for forwarding only, and 2 methods implement the views (excluding the *viewError()* method). The inheritance-based implementation requires 11 methods. Here, 9 methods implement view checking and super class calls and 2 methods implement the views. Ideally, we should only implement the two view implementation methods and a mapping between these methods and the methods to which they apply (i.e. can be prefixed). The following table summarizes these numbers:

Table 1: Evaluation of composition anomalies in *USViewMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2+ 1 view mapping
Aggregation	16
Inheritance	11

## 2.2 View Partitioning

Assume that class *ORViewMail* partitions the *User* view into *Originator* and *Receiver* views. Only *originator* clients are allowed to invoke the methods *putOriginator()*, *putReceiver()*, *putContent()* and *send()*. *Receiver* clients are only allowed to invoke the method *reply()*. For other methods, the restrictions (if any) defined by *USViewMail* apply.

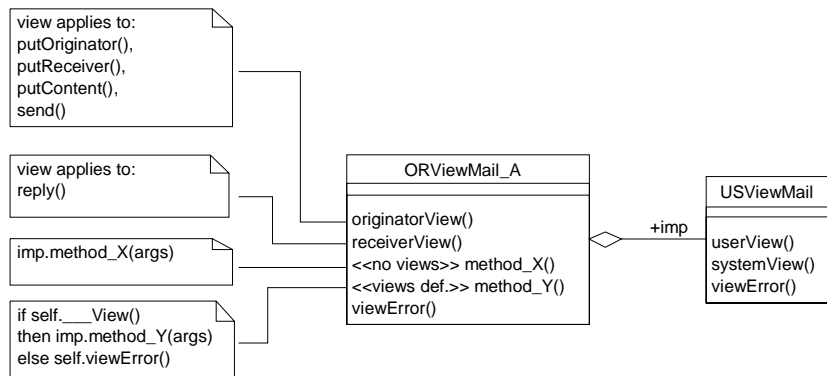


Figure 4: Aggregation-based composition of view partitioning

Again, this specification can be implemented using aggregation or inheritance-based composition. In the example, in case of aggregation-based composition, the aggregated object is an instance of class *USViewMail*. The implementation is along the same lines as for class *USViewMail*, as shown in Figure 4. This means that all the methods for which the additional

constraints defined by *originatorView()* and *receiverView()* apply, must be redefined to add this new view constraint. All other methods that must appear on the interface of class *ORViewMail* have to be defined and redirected as well.

In the inheritance-based composition approach, class *ORViewMail* inherits from class *USViewMail*. This requires redefining all the methods that are subject to the newly defined *originatorView()* and *receiverView()*. All other methods are inherited from class *USViewMail*. This implementation is shown in the following figure:

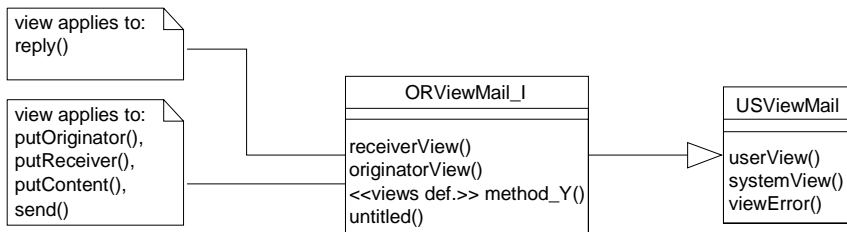


Figure 5: Inheritance-based composition of view partitioning

We now summarize the number of method (re-)definitions required for the view partitioning. Ideally, we only have to define the two views, and specify the methods upon which these apply (rather than embedding the view checking inside each method implementation). In the aggregation-based case, for each reused method that must be visible on the interface (i.e. 14 methods), a redirecting method must be created. For all of these methods upon which the views apply, also the view checking must be embedded in these method implementations. In addition, two new methods defining the *originator* and *receiver*-view must be created. In the inheritance-based case, the two new views must be implemented, and all the methods that require one of these views (respectively 1 and 4) must be redefined as well. The following table shows these numbers:

Table 2: Evaluation of composition anomalies in *ORViewMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2+1 view mapping
Aggregation	16 (+2 view redirection methods)
Inheritance	7

Note that in the aggregation-based approach, the view definitions of *userView* and *systemView* are not directly available for class *ORViewMail*.

### 2.3 View Extension

In the next example, we extend the *originatorView()* and *receiverView()* that are defined in class *ORViewMail* so that they apply to a *group of originators* and *receivers*. This may be required, for example, in offices where more than one person is responsible for sending and receiving mails.

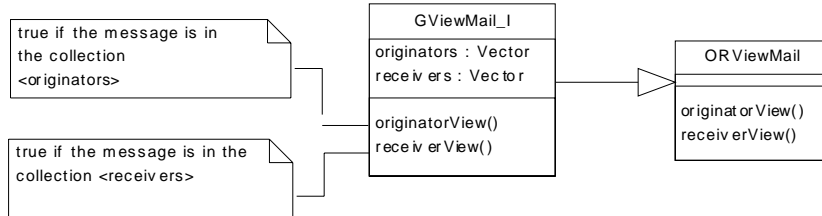


Figure 6: Inheritance-based composition of view extension

In inheritance-based composition, the methods *originatorView()* and *receiverView()* of class *ORViewMail* are overridden in *GViewMail* to define *group originator* and *receiver* views, respectively. All other methods can be inherited from class *ORViewMail*. A call to, for example, *originatorView()* in method *ORViewMail::send()*, will then refer to the *originatorView()* implemented in *GViewMail*. In this way, only 2 methods are required for re-implementing the views (we assume that *UserView()* and *SystemView()* need not be re-implemented). *Figure* shows inheritance-based composition.

In aggregation-based composition, the implementation of class *GViewMail* is analogous to that of *ORViewMail* as shown in Figure 6. The two methods that implement the views are redefined, the methods that are subject to a view must include the checks to these new view methods<sup>3</sup>, and the other reused methods require a simple redirection implementation.

Table 3: Evaluation of composition anomalies in GviewMail

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2
Aggregation	16
Inheritance	2

The inheritance-based approach behaves in this case ideal: only the new view conditions require additional method definitions. In the aggregation-based approach, in total 16 methods have to be implemented: 5 methods are

<sup>3</sup> This is because in the aggregation-based case we do not have the equivalence of dynamic binding through a *self* pseudo-variable.

used for view checking, 9 methods are used for forwarding messages only, and 2 methods implement the views. Table 3 shows these numbers.

## 2.4 History Sensitive Behavior

Assume that we want to introduce the following refinement: a class *HistoryMail*, which adds a view that does not depend on the client object, but upon historical information about the invocations on class *HistoryMail*. If the same method is invoked twice or more in a row for the same mail object, a warning (error) message must be generated, and the method is not executed. Assume that this constraint applies to the methods *send()*, *reply()*, and *deliver()*.

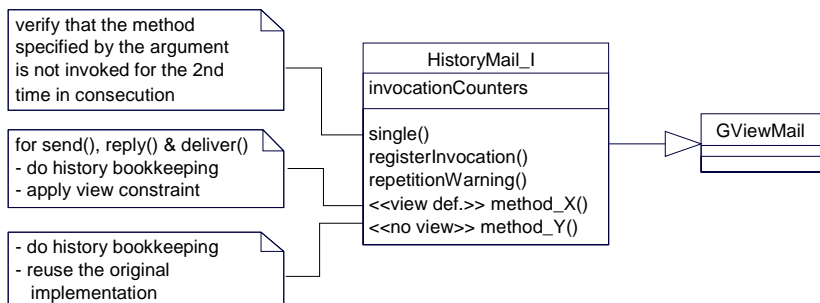


Figure 7: Inheritance-based composition of history sensitive behavior.

Figure 7 shows inheritance-based composition of the new history-sensitive behavior with class *GViewMail*. The realization of this behavior involves two conceptually different problems: the first is to collect the relevant history information. This requires bookkeeping of all the method invocations, including those that have no constraint defined upon them. In other words, bookkeeping of invocations is an aspect that applies to all the methods of an object and affects seemingly unrelated parts of the class. The bookkeeping requires redefinition of all the reused methods, for example as illustrated by the following pseudo-code for the inheritance-based strategy:

```

HistoryMail::methodY(<args>) // only add bookkeeping
    self.registerInvocation('methodY');
    return super.methodY(<args>);
  
```

The second issue is to enforce the constraint upon the three selected methods. This requires –a different– redefinition of all the (three) methods

that are subject to the history-sensitive behavior, for instance as shown in the following pseudo-code example:

```
HistoryMail::methodX(<args>)
  // history-sensitive; add bookkeeping and verify view
  self.registerInvocation('methodX');
  if self.single('methodX')
  then return super.methodX(<args>)
  else self.repetitionWarning();
```

The aggregation-based solution is almost identical; all the method implementations are as shown above, except that in each place where “*super.methodX()*” is written in the inheritance-based case, the aggregation-based case will have “*imp.methodX()*” instead. Both cases require 3 new methods (*single()*, *registerInvocation()* and *repetitionWarning()*) and a total of 14 methods to be redefined (that excludes all view and bookkeeping methods defined by the reused classes).

Extension with history-sensitive behavior introduces two new aspects to the class:

- *History bookkeeping aspect*: this aspect *crosscuts* all the methods of the class: it requires the addition of (a call to) bookkeeping code to all the methods. Thus one can image a composition scheme (and language model) that requires only the definition of the bookkeeping (as in the method *registerInvocation()*) and a specification that states that this definition applies to all methods of the class, including the inherited methods and –most likely– the methods that will be introduced in subclasses.
- *Constraint behavior aspect*: for the three methods *send()*, *reply()*, and *deliver()*, the constraints have to be specified (i.e. they are not executed twice in a row, and then an error is generated). Ideally we would like to specify this constraint only once and then simply assign it to the three methods.

The following table shows the number of definitions the above solutions require. Since all the solutions require the history bookkeeping implemented by all methods, the number of methods that are redefined are the same in all solutions.

Table 4: Evaluation of composition anomalies in *HistoryMail*

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	2 + 2 mappings
Aggregation	3 new +14 redef.
Inheritance	3 new +14 redef.

## 2.5 Adding Synchronization to Multiple Classes

Our final example deals with the extension of the mail example with a code necessary to synchronize concurrent threads. It is different from the previous extensions in that we want to extend multiple classes: *HistoryMail*, *MailDelivery*, and *Receiver*. However, all of these classes need to be extended with the same logical feature: the ability to *lock* all operations such that they are halted until an *unlock* operation has been called. We introduce class *SyncMailSystem*, from which we want to reuse the synchronization specification and 2 additional operations called *lock* and *unlock*. If the method *lock* is invoked, then *all* subsequent messages are delayed until the invocation of the method *unlock*.

To illustrate a possible implementation we use *semaphores*, one of the simplest mechanisms to delay and activate threads<sup>4</sup>. A first question is how to extend the three classes: the approach taken so far in this chapter is to create specialized classes. In this case this means the introduction of three new classes, namely *SyncMail*, *SyncMailDelivery* and *SyncReceiver*. Because these three classes require similar extensions, we focus on one class, i.e. *SyncMail*: the same changes have to be repeated for the other two classes.

*SyncMail* must reuse behavior from both *HistoryMail* and *SyncMailSystem*. With inheritance, this requires support for multiple inheritance, ‘multiple aggregation’ is equivalent to repeated aggregation and requires no special language support. Because all the (reused) methods of *SyncMail* are affected, aggregation-based and inheritance-based composition are further largely identical: each method that is visible on the interface must be extended to start with some code that verifies the locking state and acts accordingly, for example:

```
SyncMail::methodX()          // any method of SyncMail
    if impSyncMS.isLocked() then impSyncMS.wait();
    return impHM.methodX();
    // call original method from instance of HistoryMail
```

<sup>4</sup> For the sake of simplicity, we will ignore possible concurrency conflicts: this could also be integrated with the locking code, but we assume they are handled by separate mechanisms.

This implementation calls the method *isLocked()* upon an instance of class *SyncMailSystem* (i.e. *impSyncMS*) to request the locking state. If the state is ‘locked’, the thread will be blocked by calling *wait()*, otherwise the original method is executed by an instance of class *HistoryMail*.

The implementation of *SyncMail* as illustrated in the previous section requires in total 16 method definitions. Here, 14 methods are overridden to add synchronization constraints, 2 methods are required to forward the *lock()* and *unlock()* operations. In the case of multiple inheritance, forwarding methods is not needed, which reduces the number of methods to be defined with 2.

Table 5: Evaluation of composition anomalies when adding synchronization.

Composition Scheme	# Method (re-)definitions
Ideal/Conceptual	4 new +1 mapping
Aggregation (multiple)	4 new + 24 redef. + 6 forw. = 34
Inheritance (multiple)	4 new + 24 redef. = 28

In addition, this must be repeated for classes *SyncMailDelivery* and *SyncReceiver*, if these have each 5 methods on their interface, the total number of defined methods is  $14+2 + (5+2) + (5+2) + 4 = 34$ . The final number 4 refers to the number of new methods defined by class *SyncMailSystem*. Table 5 shows the number of definitions that are required for the various solutions.

All the previous evolution steps have illustrated the need to apply certain behavior repeatedly within other methods of the same class. This repetition characteristic is also referred to as *crosscutting* [10]. One of the distinctions in this case is that, in addition, behavior needs to be repeated within other classes. In other words, the synchronization crosscuts methods in multiple classes.

## 2.6 Overall Evaluation of Change Cases

The following table provides an overview of the number of method definitions for each of the classes and each of the two reuse strategies (i.e. aggregation respectively inheritance). The first column gives (an estimation of) the number of definitions that are at least needed to realize the concern that is to be introduced by each of the classes; i.e. in the ideal case. If the same behavior is to be applied to multiple methods, this may be expressed by defining a simple mapping between the definition of that behavior and the methods; we separately add the number of such mappings.

Table 6: An overview of the Mail change case and resulting anomalies

Extension (class)	Ideal #defs	Aggregation		Inheritance	
		#defs	#anom	#defs	#anom
Email	14	14	0	14	0
USViewMail	2+1	16	14-1	11	9-1
ORViewMail	2+1	18	16-1	7	5-1
GviewMail	2	16	14	2	0
HistoryMail	2+2	17	15-2	17	15-2
SyncMail	4+1	34	30-1	28	24-1
Totals	26+5	115	89-5	79	53-5
		31	115	84	79

The table shows the number of definitions (`#defs`) and ‘the number of anomalies’ (`#anom`) for each of the strategies. The number of anomalies is equal to the number of superfluous definitions, as calculated by subtracting the number of definitions required in the ideal case from the actual number of definitions required.

We can conclude that from the perspective of reusability, the conventional object-oriented model –at least in the given example case– performs unsatisfactorily. The examples show that reusing components through aggregation and inheritance mechanisms may not always be successful, if objects implement concerns like multiple views, history information and synchronization. An important characteristic of the presented problems is that they involve crosscutting behavior.

The aggregation-based change case requires 84 superfluous (re-) definitions. Inheritance-based reuse performs better (‘only’ 48 superfluous definitions), but cannot implement dynamically changing reuse relations.

Despite of all these composability problems, the object-oriented model has many useful features. For example, the change case we presented shows that each of the versions of the mail system can be adequately realized; it is the evolution between versions that cannot be dealt with satisfactorily. For this and other –more practical– reasons, we believe that to cope with the evolution problems, we should enhance *current* object-oriented languages, rather than replacing them.

### 3. THE COMPOSITION FILTERS MODEL

#### 3.1 Basic Structure of Composition Filters Objects

The composition filters (CF) model is a modular extension to the 'conventional' object model as adopted e.g. by Java, C++ and Smalltalk. The behavior of an object can be substantially affected and enhanced through the manipulation of incoming and outgoing messages only. To do so, in the CF model, a layer called the *interface part* is introduced. The resulting model and its components are shown in Figure 8.

The most significant components in the CF model are the *input filters* and *output filters*. Each individual filter specifies a particular manipulation of messages. Various filter types are available for different types of manipulations. The filters together compose the behavior of the object, possibly in terms of other objects. These other objects can be either *internal objects* or *external objects*. Internal objects are encapsulated within the composition filter object whereas external objects remain outside the composition filters object, such as globals or shared objects. The behavior of the object is a composition of the behavior of its internal and external objects.

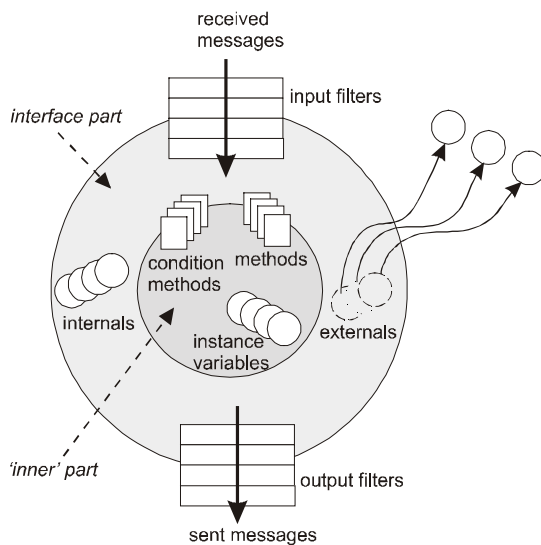


Figure 8: The components of the composition-filters model

In addition, –part of– the behavior of the object can be implemented by the 'inner' object, which is therefore also referred to as the *implementation part*. Any conventional object-oriented programming language, such as Java,

C++ or Smalltalk<sup>5</sup> can implement the inner object: the interface part is a modular extension to the inner object.

### 3.2 The Principle of Message Filtering

We will explain the basic mechanism of message filtering with the aid of Figure 9. The discussion focuses on input filters, but output filters work in exactly the same manner. The main difference is that output filters deal with messages sent by the object instead of received messages.

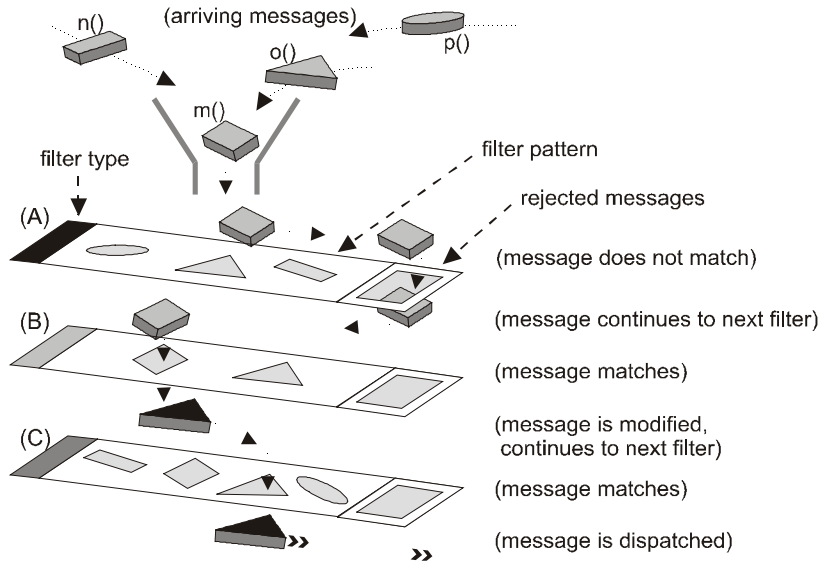


Figure 9: An intuitive schema of message filtering.

To understand the schema the following should be kept in mind: filters are defined in an ordered set. A message that is received by an object is first reified, i.e. a first-class representation of the message is created<sup>6</sup>. The reified message has to pass the filters in the set, until it is discarded or can be dispatched. Dispatching means that the message is activated again, for example to start the execution of a method body, or to be delegated to another object. Each filter can either accept or reject a message. The semantics associated with acceptance and rejection depend on the type of the filter.

<sup>5</sup> Implementations of composition filters have been built as extensions for each of these languages in the past [9, 11, 14].

<sup>6</sup> Composition filters thus apply a form of message reflection [8].

Figure 9 visualizes the processing of messages by three filters, A, B and C. An object can receive a variety of messages, in the figure exemplified by  $m()$ ,  $n()$ ,  $o()$  and  $p()$ . All received messages are subject to manipulation by all successive filters. Each filter tries to match messages based on a specific pattern. All filters for defining these patterns use a common syntax. The matching process can be defined in terms of message properties, but may also depend on the current state of the object.

We follow the message  $m()$  as it passes through the filters. In Figure 9, message  $m()$  does not match with the pattern defined by filter (A). Thus, this filter rejects the message. In this example, the rejected message is simply passed on to the next filter.

The message will then be evaluated by filter (B). The pattern that is defined by this filter matches with the message. This is referred to as *acceptance* of the message by the filter. This initiates a particular action that depends on the filter type: the message may be manipulated and modified, other side effects might take place as well. In the example of filter (B), the message is modified (designated in the figure by its changed shape and color), and then passed on to the next filter.

For the last filter in the example, filter (C), the pattern also matches the message. The acceptance of the message in this case causes the message to be dispatched, for example to a local method of the object. The message itself contains information that determines how it should be dispatched (i.e. the target object and the message selector).

In general, every filter set should contain a filter that causes messages to be dispatched, as this is the only means to trigger the execution of a method. For output filters, dispatching means that the message is submitted to the target object. Note that also in this case, upon the reception of this message by the target object, the message must first pass the input filters of the target object.

### 3.3 Filter Specifications

In this section, we briefly introduce the syntax and intuitive semantics of filter specifications. A single filter specification consists of the following elements:

```
<name>:<filter-type>={<filter-elem>,<filter-elem>, ... };
```

This can be interpreted as the declaration of a filter instance of `<filter-type>` with the name `<filter-name>`, which is initialized with an expression that contains a number of filter elements that are separated by comma's<sup>7</sup>.

Filter elements take the following form<sup>8</sup>:

```
<condition> => [<match-target>.<match-sel>]<target>.<sel>
```

In this element, the condition can be seen as a guard that enables the rest of the element. The default condition is the *True* condition, which always enables the element. On the right hand side, matching (between the square brackets) and substitution (the rightmost pair) with the target and/or the selector of a message takes place. The 'implication' operator '='>' has a counterpart, expressed as '~>', which means that if the condition is satisfied and the message does match on the right-hand side, the filter element will reject the message.

As a simple example, consider the following filter expression that defines extension of class *Mail*, by inheriting from *Mail* and adding a number of new methods (the first filter element is redundant, for illustrative purposes only):

```
inh:Dispatch={True=>[outer.m]inner.m, inner.*, superObj.*};
```

The first element is always enabled by the condition *True*, then continues to select only the messages with selector *m* that have been sent to the interface of this object (*outer*). If this matches, the target of the message is replaced with *inner* and the message selector with *m* (which is redundant since this was already the case). The second element has no explicit condition, in which case the default condition *True* is assumed. As a result it will match with any message that is defined by (i.e. is in the signature of) *inner*, and in that case substitute *inner* as the target of the message. The third element has again the default condition *True*, and will thus match with any message in the signature of—the class of—*superObj*, and in that case substitute *superObj* as the target of the message.

If the message matches any of the filter elements, the resulting (modified) message will be dispatched, i.e. the method defined by the *selector* of the message is to be executed upon the object defined by the *target* of the message. If the target is *inner*, this causes direct method execution. But if the target is another composition filters object, the message is delegated to that object, and will start by evaluating the filters of that object.

<sup>7</sup> Actually, the comma's represent just one particular composition operator (best described as a conditional OR). Other operators have been discussed previously and may be defined and implemented in the future.

<sup>8</sup> Actually, both the left- and the right-hand side of the '='>' can consist of a set of the respective elements.

### 3.4 The Superimposition Mechanism

In this section we briefly introduce the superimposition mechanism of the composition filters model. This is an extension to the model that we have presented so far: Figure 10 shows a refined version of Figure 8 with a few new elements. In the place of the set of input filters in Figure 8, now a box with a number of *instantiations* of filters is shown: these sets of filters are defined elsewhere (in the same or other objects). This is exemplified by the gray *filter definitions* in the figure.

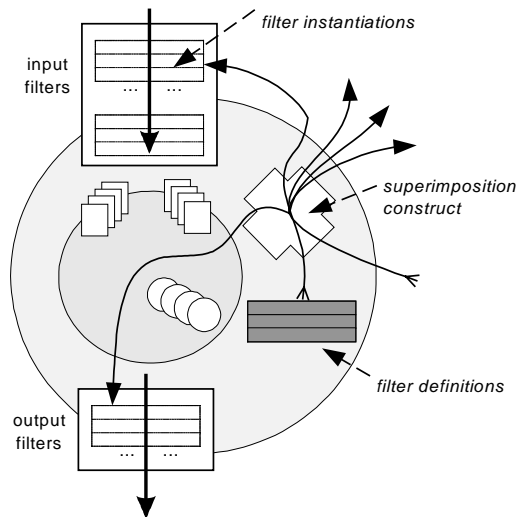


Figure 10: Superimposition mechanism in the composition filters model.

The most important part is the *superimposition* construct: this is a part of a class definition that maps the filtersets (from this object or others) onto a set of instances (possibly including instances of the current class). This means that the filterset is added (on top of) the existing filterset(s). The superimposition construct can also be applied to superimpose other elements of the interface part, such as internals, externals and conditions. For more details on the superimposition mechanism we refer to [5].

### 3.5 Summary of CF Principles

The composition-filters approach aims to enhance the expression power and maintainability of objects. Filters are 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 filters, 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 [14], Smalltalk [11] and C++ [9]. This may occur and change dynamically (at run-time). However, dynamic changes may degrade understandability, correctness and implementation optimizations and therefore must be realized with care.
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>9</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 composed 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.

<sup>9</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 [11]), or as in-lined code, by adopting compilation and optimisation techniques (e.g. in [14]).

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.

## 4. COMPOSITION FILTERS APPROACH TO THE MAIL PROBLEM

### 4.1 Multiple Views

The composition filters version of class *USViewMail* has two attached (input) filters. The filter *USView*, which is an instance of an *Error* filter, expresses multiple views. If an *Error* filter accepts the received message, then it is forwarded to the following filter. Otherwise an exception is generated. The filter *execute* is an instance of a *Dispatch* filter. If a *Dispatch* filter accepts the received message, then the message is executed. The interface (filter) definition of this class can be written as follows:

```
inputfilters
USView : Error =
  { UserView    => {putOriginator, putReceiver,
                  putContent, getContent, send, reply},
    SystemView => {approve, putRoute, deliver},
    True       => {getOriginator, getReceiver,
                  isApproved, getRoute, isDelivered} };
execute : Dispatch = { inner.*, mail.* };
```

The conditions *UserView* and *SystemView* are Boolean methods defined by class *USViewMail*. If *UserView* is true, then the *Error* filter accepts the messages *putOriginator*, *putReceiver*, *putContent*, *getContent*, *send* and *reply*. Similarly, the messages *approve*, *putRoute* and *deliver* are only accepted if *SystemView* returns true. The remaining 5 methods are not restricted by the *Error* filter, because the condition is specified as *true*.

The specifications “inner.\*” and “mail.\*” in the *Dispatch* filter mean that the filter accepts *all* (cf. wildcards) the methods declared by class *USViewMail* and the class of the internal mail object: *Email*. The pseudo-variable *inner* refers to the inner part of the current instance of *USViewMail*.

Since filters are separated and largely independent from the class, they can be reused separately. For example, software engineers can implement the core functionality of the classes mentioned above in any object-oriented language without attaching filters. Filters can later be stacked and attached to

any of these classes, whenever necessary. Note that the composition-filters implementation of *USViewMail* requires only 3 new definitions: 2 view implementations (the conditions) and 1 composition-filters specification (*USView*) to solve the view problem<sup>10</sup>.

## 4.2 View Partitioning

The following filter definitions are required to realize class *ORViewMail* using composition filters:

```
ORView:Error =
  { origView => {putOriginator, putReceiver, putContent,
                getContent, send},
    recView   => reply,
    true      ~-> {putOriginator, putReceiver, putContent,
                  getContent, send, reply} }
execute: Dispatch = { inner.*, mail.* };
```

If the condition *origView* is true, the *Originator* view is valid and the messages *putOriginator*, *putReceiver*, *putContent*, *getContent* and *send* are accepted. These messages will then be dispatched to object *mail*, an instance of class *USViewMail*. If *USViewMail* is also extended with filters, the accepted message will pass through the filters of *USViewMail* object as well. The condition *recView* is used to enforce the *receiver* view; if this condition is true, the *reply* messages are accepted by the filter. The operator “~>” in the last part of this filter means that if the condition is true (which is always the case in this example), all messages are accepted except the specified ones. The effect of this is that all these other messages will always pass the filter, regardless of the actual view that applies. The composition-filters implementation of *ORViewMail* requires only 3 new definitions. These are the implementation of 2 views by conditions and the *ORView* filter specification.

## 4.3 View Extension

The composition-filters implementation of class *GViewMail* does not require any specific filter definition. Since conditions are methods, they can be reused from class *ORViewMail* or overridden if necessary. In *GViewMail*, these conditions can be redefined to check for groups of originators and receivers.

<sup>10</sup> We do not count the *Dispatch* filter because it is only used to express inheritance, something that we did not count as a separate definition in the examples of the conventional object model either.

## 4.4 History Sensitive Behavior

Consider the following definition of filters for class *HistoryMail* :

```
check : Meta = { [*]inner.verify }
bookkeeping : Meta = { [*]inner.count };
execute: Dispatch = { True=>{inner.*, mail.*} };
```

The *Meta* filter is used to reify a message. If the received message matches—in this specification it always matches because of the wildcard "[\*]"—, it is reified. The resulting object is sent as the argument of a newly created message, with a target and selector as specified by the second part of the filter element.

In this case, the first *Meta* filter sends the reified message to the *inner* object, executing the method *verify*, which verifies repeated execution and generates a warning whenever appropriate. The second *Meta* filter sends the reified messages to the inner *bookkeeping* method, which performs the actual bookkeeping of the last executed message.

More detailed information about Meta-filters can be found in [2]. The composition-filters implementation of *HistoryMail* requires 4 new definitions: two filter specifications, and the methods *count* and *verify*.

## 4.5 Adding Synchronization to Multiple Classes

The problem of adding synchronization can be split in two issues: the first is how to specify synchronization, the second is how to attach this crosscutting specification to the three classes involved (*HistoryMail*, *MailDelivery*, and *Receiver*).

Using composition filters, we can express synchronization by a filter of type *Wait*; filters of this type perform synchronization of messages by queuing all messages as long as they cannot match with any of the filter elements. Locking can be expressed with the following filter definition:

```
queue: Wait = { True=>unlock, Unlocked =>* };
```

The message *unlock()* will always match at the first element, and is thus never blocked. If the condition *Unlocked* is true, then any message matches and will proceed to the next filter, otherwise all messages—except *unlock()*—will be queued until the condition *Unlocked* does become true.

Instead of creating three new subclasses, we can create a single class *SyncMail*, which contains all the new definitions and a superimposition specification to attach the necessary synchronization specifications to classes *HistoryMail*, *MailDelivery* and *Receiver* as well. The following specification shows how to superimposes the filterset *locking* (which contains the *queue*

filter of type *Wait* that we showed above) upon all the instances of classes *HistoryMail*, *MailDelivery* and *Receiver*:

```

superimposition
  selectors
  lockables={*=HistoryMail, *=MailDelivery, *=Receiver};
  filtersets
  lockables <- locking;

```

Superimposition specifications consists of two distinct parts: first one or more *selectors* are declared; each selector expression defines a set of instances. The second part uses the selector identifiers to superimpose filtersets (or internals, externals, conditions or methods) upon a certain set of instances as designated by the selectors.

This composition-filters implementation requires 6 new definitions. These are the filterset locking (with only a *Wait* filter specification), the condition *unlocked*, the two methods *lock()* and *unlock()*, the definition of the selector *lockables* and the superimposition of the filterset *locking*. In the case of an ideal or minimum number of definitions, the selector and superimposition could be merged<sup>11</sup>.

## 4.6 Evaluation

In order to compare the composition filters approach with the conventional object model, we have counted the number of (method, filter or condition) specifications in each of the different change cases. They are shown in Table 7, together with the results from Table 6 that show the results for inheritance and aggregation.

Table 7: The Mail change cases and resulting anomalies including composition filters

Extension (class)	Ideal	Aggregation		Inheritance		Compos. filters	
	#defs	#defs	#anom	#defs	#anom	#defs	#anom
Email	14	14	0	14	0	14	0
USViewMail	3	16	13	11	8	3	0
ORViewMail	3	18	15	7	4	3	0
GviewMail	2	16	14	2	0	2	0
HistoryMail	4	17	13	17	13	4	0
SyncMail	5	34	29	28	23	6	1
Total definitions	30	115	<b>84</b>	79	<b>48</b>	32	<b>1</b>

This table is different from Table 6 only in the last two columns, where the results for composition filters are shown. It appears that the composition filters implementations are 'ideal' in the sense of the amount of redefinitions

<sup>11</sup> But note that this separation is made for reasons of modularity and adaptability.

that are required to implement these change cases. In short, this is due to the appropriate granularity and modularity that composition filters allow. The next section discusses the contributing factors in more detail.

## 5. CONCLUSION

In this chapter we have illustrated the limitations of the two most widely used forms of object composition in object-oriented design and programming: inheritance and aggregation. The change case of the mail system illustrates some of the problems in the evolution of object-oriented software. Each change case consists of the addition or refinement of a single aspect –or concern– to existing classes.

We have assumed the following requirements for dealing with evolving software:

- Modularity: the newly introduced aspect must be modeled as a separate entity of development and reuse.
- No modifications to the existing classes are allowed (this is partly implied by the previous item).
- Avoid code replication, because of the maintenance problems this brings (it also requires extra work).

The change case of the *Mail* example has introduced the following features (generalizations of these problems have been added between brackets):

- adding multiple views (i.e. dynamically adding constraints to groups of methods)
- view partitioning (or in general, state partitioning)
- view extension (i.e. condition refinement)
- history sensitive behavior (administering executions plus state dependent constraints)
- adding synchronization (adding special execution semantics to groups of methods in multiple classes)

Through these examples, we have illustrated that both inheritance and aggregation cannot *adequately* express certain cases of evolving software. This is apparent by looking at the number of definitions that were required: for inheritance 79, and for aggregation 115. In these cases 48 respectively 84 definitions were unnecessary from an 'ideal', i.e. conceptual point of view. These superfluous definitions are a serious maintenance problem.

In section 3 we briefly introduced the composition filters model: a modular extension to the object-oriented model that allows for composing new classes from (a) the visible behavior of existing classes and (b) well-

defined semantic actions. The latter semantic actions are defined by the various available *filter types*, and may for example express synchronization, exceptions, message reification and message dispatch.

We have shown in section 4 how the composition filters model can be used to support software evolution as illustrated by the Mail system change case. For the given example, almost no superfluous definitions were required to implement it. The following characteristics of the composition-filters model contribute most to this result.

- The composition mechanism of composition filters merges most of the benefits of both the aggregation-based approach and the inheritance-based approach.
- Composition filters provide abstractions (conditions) for expressing states (which can also be used to express views or constraints).
- Composition filters provide abstractions for mapping these states plus a certain behavior to one or more elements in the interface (signature) of an object. This realizes an important form of crosscutting of behavior over methods.
- Using the *Meta* filter, meta-level state such as history information can be obtained and managed in a straightforward way.
- The superimposition construct allows to specify a certain behavior (e.g. as a filter) in one place, and apply that behavior to multiple locations (classes). This expresses crosscutting across multiple classes.

One may wonder whether the example in this chapter suits the abilities of the composition filters model particularly well. Although there is an obvious match, the examples can be easily generalized (as described earlier in this section) to a very wide range of problems. The important benefits of the model lie in the composition mechanism, which is applicable to arbitrary domains.

## ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

## 6. REFERENCES

1. M. Akşit, L. Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92 Conference Proceedings, LNCS 615, Springer-Verlag, 1992, pp. 372-395.
2. M. Akşit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, *Abstracting Object-Interactions Using Composition-Filters*, In: *Object-based distributed processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds), LNCS, Springer-Verlag, 1993, pp 152-184.
3. M. Akşit, J. Bosch, W. van der Sterren, L. Bergmans, *Real-Time Specification Inheritance Anomalies and Real-Time Filters*, ECOOP'94 Conference Proceedings, LNCS 821, Springer-Verlag, 1994, pp. 386-407.
4. L. Bergmans. *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994
5. L. Bergmans and M. M. Akşit, *Composing Crosscutting Concerns using Composition Filters*, Communications of the ACM, Vol. 44, No. 10, (to appear) October 2001
6. S. de Bruijn, *Composable Objects with Multiple Views and Layering*, MSc. thesis, Dept. of Computer Science, University of Twente, March 1998
7. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
8. J. Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 317-326
9. M. Glandrup, *Extending C++ using the concepts of Composition Filters*, MSc. thesis, Dept. of Computer Science, University of Twente, November 1995
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241, June 1997.
11. P. Koopmans, *On the design and realization of the Sina compiler*, MSc. thesis, Dept. of Computer Science, University of Twente, August 1995
12. S. Matsuoka, K. Wakita & A. Yonezawa, *Synchronization Constraints with Inheritance: What is Not Possible- So What is?*, Tokyo University, Internal Report, 1990
13. S. Matsuoka & A. Yonezawa, *Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in *Research Directions in Concurrent Object-Oriented Programming*, (eds.) Agha, Wegner & Yonezawa, MIT Press, April 1993, pp. 107-150
14. 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.