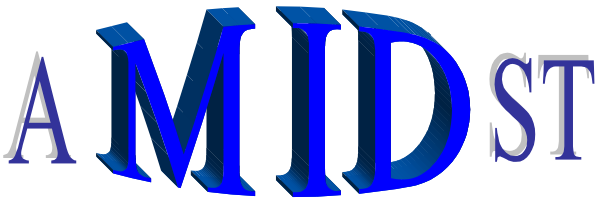

Deliverable 2.1.4: Mapping Aspects to Components

REFERENCE	:	AMIDST/WP2/N006/V05
DATE OF ISSUE	:	23 December 1999
ACCESS RIGHTS	:	public
STATUS	:	draft
EDITOR	:	L. Bergmans
AUTHORS	:	L. Bergmans, M. Aksit, B. Tekinerdogan

SYNOPSIS

This document defines a representation of aspects in the component model. Such a representation requires modeling the available (primitive) components, defining the composition mechanism, and representing aspects as enhancements of components.

Note 1: This document is a draft final version; it will be replaced soon by the final version, which incorporates a number of detailed changes and additions.

Note 2: Note: This document will be published in M. Aksit (ed.), “Software Architectures and Component Technology”, Kluwer Academic Publishers, 2000.

Document History

DATE	VERSION	MODIFICATION
23 December 1999	5	draft final version

Abstract

This document defines a representation of aspects in the component model. Such a representation requires modeling the available (primitive) components, defining the composition mechanism, and representing aspects as enhancements of components.

Table of Contents

1. Introduction

Applying the object-oriented paradigm for the development of large and complex software systems offers several advantages, of which increased extensibility and reusability are the most prominent ones. The object-oriented model is also quite suitable for modeling concurrent systems. However, it appears that extensibility and reusability of concurrent applications is far from trivial. The problems that arise, the so-called inheritance *anomalies* or *crosscutting* aspects have been extensively studied in the literature.

As a solution to the synchronization reuse problems, we present the composition-filters approach. Composition filters can express synchronization constraints and operations on objects as modular extensions. In addition, the composition-filters approach is able to express various different aspects in a reusable manner.

In this document we briefly explain the composition filters approach, demonstrate its expressive power through a number of examples and show that composition filters do not suffer from the inheritance anomalies.

2. Object-Oriented Composition Strategies

When designing object-oriented systems, the software engineer has to choose between two major strategies of implementing composition. We will illustrate this with the simplest possible case of reusing an existing class *A* in a new class *B* (this means composing the behavior of *A* and the –newly defined– behavior of *B*). This is illustrated in the following figure:

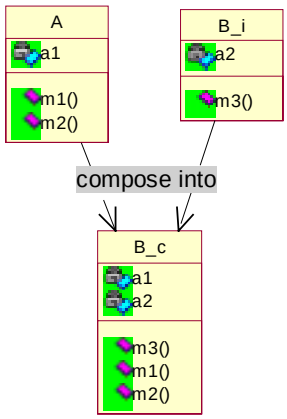

Figure 1. Illustrating the composition of newly defined behavior in class B *with reused behavior in class* A.

The two strategies for implementing composition are:

Aggregation based composition (also called 'black-box reuse'): this means that class *B* will employ an aggregation ('part-of') relationship with class *A*, and reuse the behavior that is visible on the interface of class *A* through message invocations only. If class *B* wants to mimic the behavior of class *A*, this requires the redirection of methods from the interface of class *B* to class *A* (typically be redefining the appropriate methods in class *B*).

Inheritance based composition (also called 'white-box reuse'): this means that class *B* will define an *inheritance* relation to class *A*, by which all the behavior (methods and instance variables of class *A*) automatically become available in class *B*.

The following figure illustrate these two strategies:

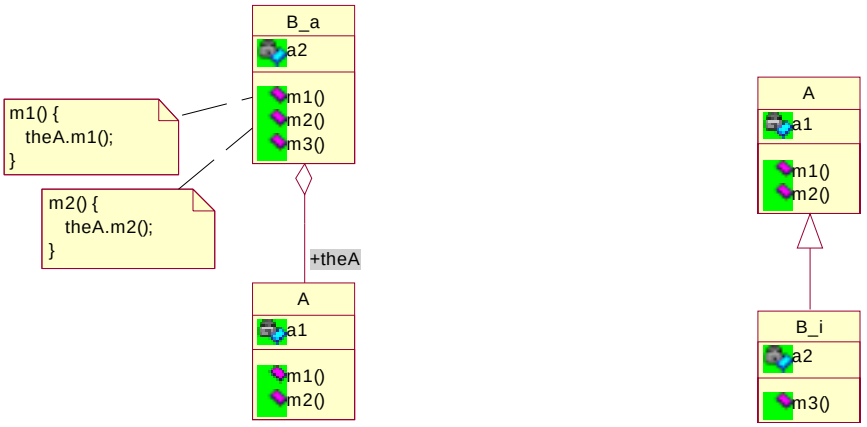

Figure 2. (a) aggregation-based composition (b) inheritance-based composition

Both of the approaches have advantages and disadvantages, we summarize the most important ones:

Aggregation-based:
- It is more robust because the reuse relation depends only on the well-defined invocation interface of the reused class
- Supports dynamic reconfiguration.
- Explicit message forwarding is required to achieve reuse of interfaces; this requires adding many extra methods, and lacks the notion of 'self'

Inheritance:
- Because the subclasses have more access and as a result more dependencies to the reused classes,
- Automatic (cascaded) updates of an interface to the classes that reuse it; e.g. when adding a new method to the interface, all inheriting classes automatically receive that
- Is static, cannot be adapted at run-time

3. Composition Anomalies

An *anomaly* means (Merriam-Webster): “deviation from the common rule; irregularity” or “something different, abnormal, peculiar, or not easily classified“. With *composition anomaly* we refer to the cases where a conceptually sound composition causes undesired effects that can only be solved in ways that lead to reduced maintainability.

We can define composition anomalies with more detail as follows:

- Given two classes C_a and C_b and a *composition scheme* S, where the composition of C_a and C_b into C_c under composition scheme *S* (in short-hand: C_c=S(C_a, C_b)) is sound at the conceptual level:

- A *composition anomaly* occurs if the composition “S(C_a, C_b)” does not result in the desired C_c: either because the composition is undefined, or because the resulting behaviour is not correct: C_c must be a valid class, the semantics of C_c must sound and intuitively correct and the semantics of C_c must be a logical result of the composition.

Circumvention of a composition anomaly (if possible) typically requires either:

- A modification of C_a or C_b (highly undesirable or even impossible).

- Additional ‘glue code’ to be added to C_c, usually in the form of redefinitions of methods already specified in C_a or C_b.

- A combination of these.

The term *composition anomaly* is analogous to the term *inheritance anomaly*, as was coined by Matsuoka et.al. in [Matsuoka 90, 93] to denote the more specific case where the embedding of synchronization code in classes caused serious problems when trying to reuse and extend such code, especially through inheritance mechanisms. In those cases, it typically appeared that the problems could be patched by overriding in a subclass substantial parts of the methods defined by a superclass. We refer to [Matsuoka 90] and [Bergmans 94] for extensive analysis of these problems. One of the crucial conclusions from this work was that these problems where language-dependent; i.e. they were related to the chosen synchronization scheme, and its composition semantics.

In this document, we will illustrate some composition anomalies through examples. We show that the conventional object-oriented composition scheme model cannot cope with these examples, but exhibits composition anomalies in these cases. We then look at the composition filters approach to show how this avoids the composition anomaly in these cases.

4. Example problem and Anomalies

In the following sections, we illustrate a number of *composition anomalies* in the conventional object-oriented model through an example problem.

4.1 Setting the Stage: Email example

Consider a simple mail system, which consists of classes *Originator*, *Email*, *MailDelivery* and *Receiver*. The following figure shows the class diagram of this simplified system, focusing on the details of class *Email*:

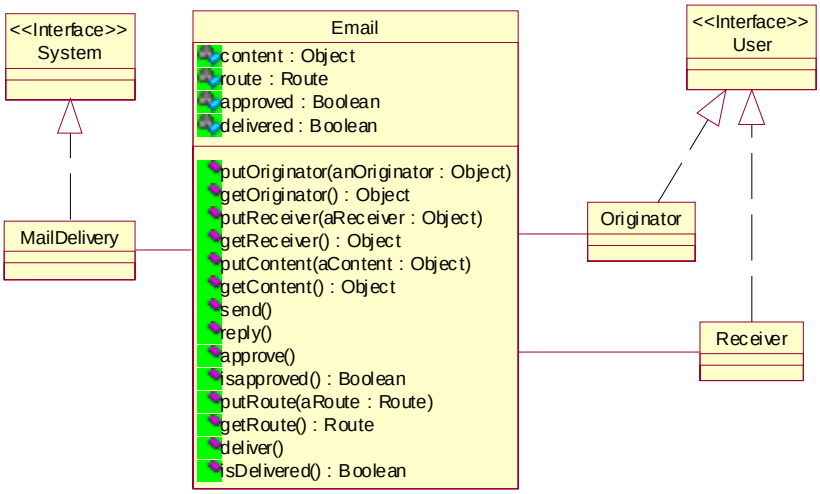

Figure 3. The interface methods of class EMail.

Class *Email* represents the electronic messages sent in this system and provides methods for defining, delivering and reading mails. For example, the methods *putOriginator(), getOriginator(), putReceiver(), getReceiver(), putContents()* and *getContents()* are used to write and read the attributes 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 to send a reply message. In this text, *Email* will be used as the base class for developing various kinds of email objects.

To illustrate a number of composition anomalies, class *Email* will be extended several times, each time adding new behavior and/or constraints. Most extensions deal with the management of so-called *multiple views*: the situation where a single class offers a different interface according to the perspective that is looked from. The different perspectives may include different clients or a different state of the system. Multiple views have been explained among others in [Aksit ecoop 92]. For instance the work on rôles by Reenskaug and others [Tryve Reenskaug rôle book] is related to this topic.

The following figure provides an overview of the change cases we address and the classes

involved[1]. Each change case defines an extended or modified version of class *Email* or one of its refinements; it requires the composition of an existing class with a certain new behavior. The point of 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.

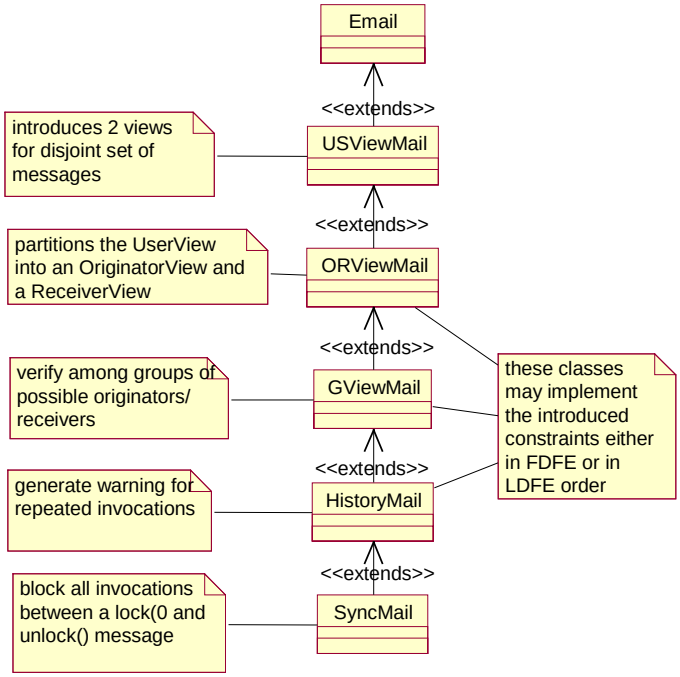

Figure 4. Overview of change cases and classes.

Note that no assumptions are made on the implementation of the <<extends>> relations in the above diagram. In the discussion to follow both inheritance-based and aggregation-based composition techniques will be discussed.

4.2 Adding Multiple Views

4.2.1 Class USViewMail

Given a working and tested implementation of class *Email*, assume that we want to introduce a version of class *Email* with a certain constraint. Like in a real-world postal mail or e-mail system, we want to restrict the possible access to email objects: the current implementations of the various classes introduced above, and particularly that of class *Email*, do not restrict access to e.g. the contents of a mail at all. We would like to make sure that at the programming level, the possible accesses of a mail object by other objects are constrained. Obviously, this must be achieved by refining the *Email* object only, independent of the implementations of other objects.

For this purpose, we extend class *Email* to *USViewMail* (User/System-View) and restrict

[1] The FDFE and LDFE orderings that are mentioned in the figure stand for “First Defined, First Enforced” and “Last Defined, First Enforced” ordering of applying constraints. This is discussed in more detail in Section 4.3.3.

access to its methods based on the type 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 defined for the methods *getOriginator(), getReceiver(), isApproved(), getRoute()* and *isDelivered()*.

We will assume that the identity of the client object (the sender of the message) can be obtained[2]. As discussed in Section 2, there are mainly two strategies to compose an existing implementation with new behavior in the conventional object model: aggregation-based and inheritance-based composition.

4.2.2 Aggregation-based Composition

In our example, 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()*[3]. Each method that requires a view constraint to be enforced must start with some code that implements this constraint. In this case the methods have already been implemented in class *Email*, so this implementation is then reused by invoking the corresponding method in the encapsulated *Email* object.

For example method *putOriginator()*, which is subject to the 'User' view can be implemented as follows:

```
USViewMail::putOriginator(Object anOriginator) {
   if self.userView()  //returns boolean indicating if view applies
   then imp.putOriginator(anOriginator)
   else self.viewError(); }
```

Alternatively, a shorter implementation might be:

```
USViewMail::putOriginator(Object anOriginator) {
   self.userView(); //generates exception if client is not a user
   imp.putOriginator(anOriginator); //invoke original impl. }
```

The following class diagram shows how aggregation based implementation of multiple views can be done:

[2] 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.

[3] Since the view checking will occur in several places, and duplicated code should be avoided.

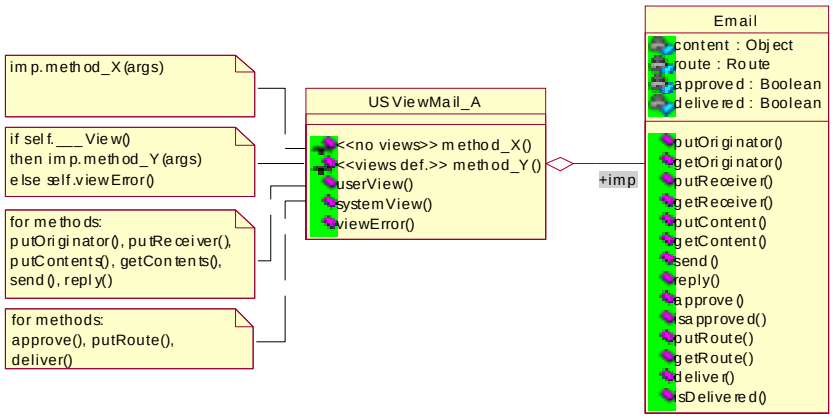

Figure 5. Aggregation-based implementation of multiple views.

Notice that in this implementation strategy, all the methods have to be declared and implemented by class *USViewMail_A*, even those methods that do not require any view enforcement (since the redirection of these methods to the *imp* class *Email* must be implemented for all methods).

4.2.3 Inheritance-based Composition

In the case of inheritance-based composition, view checking is again implemented at the start of each view-method, and reuse is realized though *super* calls. Here, only the methods with views have to be redefined; other methods can be inherited as they are from the super class(es). We show an implementation for method *putOriginator()* again:

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

The class diagram is shown in the following figure:

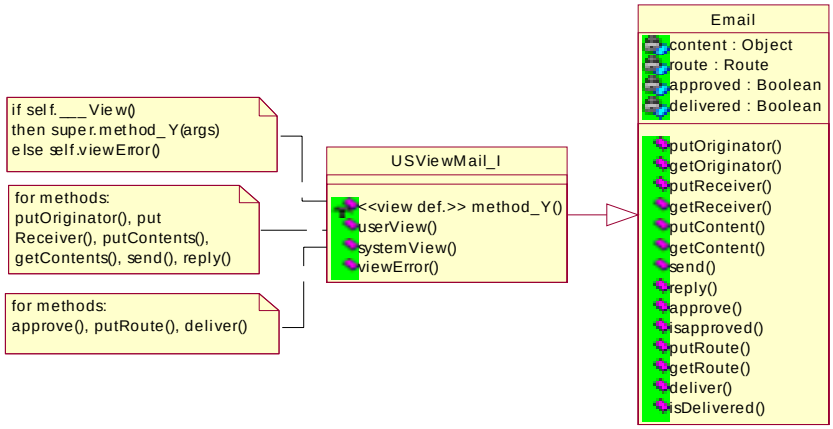

Figure 6. Inheritance-based composition of multiple views.

4.2.4 Evaluation

In case of aggregation-based composition, *USViewMail2* implements 16 methods[4]. Among these, 9 methods implement view checking and forwarding (see Figure 5), 5 methods are used for forwarding only, and 2 methods implement the views. The inheritance-based implementation requires 11 methods. Here, 9 methods implement view checking and super class calls (see Figure 6), and 2 methods implement the views[5]. Ideally, we would implement the two view implementation methods and a mapping between those methods and the methods to which they apply (i.e. can be prefixed). The following table summarizes these numbers:

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

4.3 View Partitioning

4.3.1 Class ORViewMail

Assume that class ORViewMail partitions the *user view* into *Originator* and *Receiver views*. Only the client of *originator* type can invoke the methods *putOriginator*, *putReceiver*, *putContent* and *send*. The client of *receiver* type is allowed to invoke the method *reply*. For other methods, the restrictions defined by USViewMail apply.

Again, this class can be implemented using aggregation or inheritance-based reuse.

4.3.2 Black-Box Strategy

In the example, in case of aggregation-based reuse, the aggregated object is an instance of class USViewMail. In the inheritance-based reuse approach, class ORViewMail inherits from class USViewMail.

4.3.3 Order of Constraint Enforcement

USViewMail and ORViewMail both enforce views on some methods. There are two ways how this ordering can be realized:

1. First the *originator* and *receiver views* then the *user* and *system views*. We call this ordering ‘last-defined-first-enforced’ (LDFE).
2. First the *user* and *system views* and then the *originator* and *receiver views*. We call this ordering ‘first-defined-first-enforced’ (FDFE).

[4] The exact number of methods depends on the language used.

[5] We ignore the method *viewError()*.

Implementation of LDFE ordering is relatively simple because object-oriented models naturally support it. In the aggregation-based implementation, after verifying the constraints, requests are forwarded to the aggregated objects. In the inheritance-based reuse, verified requests are forwarded to the super classes through super calls. However, both reuse mechanisms require a considerable number of re-implementations. Similar to class USViewMail, in the aggregation-based implementation, ORViewMail implements 16 methods. The inheritance-based implementation requires 7 methods. Here 5 methods are for originator and receiver view checking and 2 methods implement the *originator* and *receiver* views.

The aggregation-based implementation of FDFE ordering is somewhat more complicated, because it requires reordering of the aggregate structures. Consider the code shown in Figure 7. If the sender of the message is of *user* and *originator* type, the message *putOriginator* is forwarded to the aggregated object *imp*, otherwise the error method *viewError* is invoked. Here, the method *userView* will be unnecessarily invoked twice, first by the ORViewMail object and then by the USViewMail object. If a multiple invocation is not desired, then the aggregate structure must be reorganized. The aggregated objects must be reconfigured as interface objects and vice versa. This reconfiguration can be a rather complex operation and may require additional method definitions, such as *retrieve*, *store* and *configure*. The methods *retrieve* and *store* can be used to read and write the aggregated object, respectively. The method *configure* is responsible to establish the desired aggregate structure. We assume that the FDFE ordering requires at least 3 additional methods for reconfiguring the aggregation structure, resulting in total 19 method implementations.

```
ORViewMail::putOriginator(anOriginator)
   if imp.userView
   then if  self.orginatorView
        then imp.putOriginator(anOriginator)
        else imp.viewError;
```

Figure 7. Aggregation-based reuse of putOriginator *in FDFE implementation.*

The inheritance-based implementation of FDFE ordering requires redefinition of the call patterns. Nevertheless, the total number of required methods remains as 7. Consider the implementation of the method *putOriginator* of class ORViewMail. Notice that here first *userView* and then *originatorView* are verified:

```
ORViewMail::putOriginator(anOriginator)
  If self.userView then
      If self.orginatorView then
         super.putOriginator(anOriginator)
      else self.viewError;
```

Figure 8. Inheritance-based reuse of putOriginator in FDFE implementation.

4.4 View Extension

4.4.1 Class GViewMail

In the next example, we reuse ORViewMail in GViewMail by extending the views 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.

4.4.2 aggregation-based LDFE

In case of the aggregation-based LDFE reuse, the implementation of class GViewMail is similar to the one shown in Figure 7. In total 16 methods have to be implemented: 5 methods are used for view checking, 9 methods are used for forwarding messages only, and 2 methods implement the views.

4.4.3 inheritance-based LDFE

In case of the inheritance-based LDFE reuse, the methods *originatorView* and *receiverView* of ORViewMail can be re-implemented in GViewMail as *group originator* and *receiver* views, respectively. Here, the method *putOriginator* can be inherited from class ORViewMail, and therefore it is not necessary to declare it in class GViewMail. The *self.originatorView* call in the method *putOriginator* will then refer to *originatorView* implemented in GViewMail. Only 2 methods are required for re-implementing the views.

4.4.4 agregation-based FDFE

The agregation-based FDFE implementation requires in total 19 methods. Among these, 3 methods are used to configure the aggregation structure.

4.4.5 inheritance-based FDFE

In the inheritance-based FDFE implementation, because of the required changes in call patterns, the method *putOriginator* must be redefined in GViewMail. Namely, view checking must be realized in the reverse order, first the views of USViewMail and last group views must be verified. In total 7 methods are required: 5 methods are used for view checking and 2 methods implement the views.

4.5 Advanced Conditions: History Information

4.5.1 Class HistoryMail

Assume that class HistoryMail extends class GViewMail with a history view. If a method is invoked more than once for the same mail object, a warning message is generated.

4.5.2 LDFE

Figure 9 shows an aggregation-based LDFE ordering of the method *putOriginator*. It is estimated that both the aggregation and inheritance based implementations require 15 methods. 14 methods of Email have to be re-implemented for call administration, plus the method *single.* This method accepts a name as an argument, and returns *true* if the name, which corresponds to a method, has not been used before on the mail object.

```
HistoryMail::putOriginator(anOriginator)
   if self.single('putOriginator')
   then imp.putOriginator(anOriginator)
   else self.giveAWarning;
```

Figure 9. Aggregation-based LDFE ordering of putOriginator *in class* HistoryMail.

4.5.3 FDFE

It is estimated that the aggregation and inheritance based FDFE orderings will require 18 and 15 methods, respectively. The additional 3 methods for the aggregation-based reuse are required for reconfiguring the aggregate structures.

4.6 Adding Synchronization

4.6.1 Class SyncMail

Consider, for example, class SyncMail, which inherits from HistoryMail. This class provides 2 additional operations called *lock* and *unlock*. If the method *lock* is invoked, then all the messages are delayed until the invocation of the method *unlock*.

We can utilize a *semaphore* to delay and activate messages. In the aggregation-based reuse, the semaphore can be implemented at the interface object. An inheritance-based implementation of LDFE ordering is shown in Figure 10.

```
SyncMail::putOriginator(anOriginator)
   if self.locked then sema.wait;
   super.putOriginator(anOriginator)
```

Figure 10. Inheritance-based LDFE ordering of putOriginator *in class SyncMail.*

Both the aggregation and inheritance based LDFE implementations require in total 17 method definitions. Here, 14 methods are overridden for semaphore implementation, 2 methods are required for *lock* and *unlock* operations, and 1 method is used for implementing the semaphore.

4.6.2 FDFE

The aggregation-based implementation of FDFE ordering requires 20 methods. Here, three additional methods are needed for reconfiguring the aggregate structure. The inheritance-based reuse requires 17 methods.

4.7 Evaluation and Requirements

In the previous section we introduced a set of classes which are derived from each other. Class Email is used as a base class and defines 14 methods. USView mail illustrates that a considerable number of methods of EMail have to be re-implemented if two views are enforced on 9 methods. Class ORViewMail shows that view partitioning requires re-implementation of the corresponding methods. In addition, if the view enforcement is applied from the most general to specific views (FDFE ordering), the aggregation-based reuse becomes problematic due to the encapsulated objects; this requires a complete reconfiguration of the aggregated objects. Class GViewMail illustrates that the inheritance-based reuse may be advantageous with respect to the aggregation-based reuse, if only the implementation of views is changed. However, if the views are verified in FDFE ordering, then the methods with views have to be redefined, because the call patterns to the super classes have to be modified. Class HistoryMail shows that demanding a history information requires modification to all methods. Similarly, SyncMail illustrates that adding a simple synchronization constraint like locking causes redefinition of all the methods.

Despite of all these composability problems, the object-oriented model has many useful features. In order to cope with the problems, however, the current object-oriented languages must be enhanced. Since more than one problem can be experienced for the same object, multiple enhancements must be specified independent from each other.

5. The Composition-Filters Model

We will now investigate *natural* solutions to the composability problems. Assume for example that we want to take a picture of a flower, which is too close to our camera, and the ambient light is not suitable for the film. As a result, the camera cannot provide a satisfactory picture. In other words, the camera cannot express this image; this is an example of a modeling problem. A cost-effective way to solve this problem is enhancing the camera using two extensions: a lens to sharpen the picture and a color filter to filter out the unwanted light effects. These are called modular extensions because the expression power of the camera is enhanced without changing its basic structure. The lens and filter can be used together because their functionality is orthogonal to each other.

The expression power of the object-oriented model can be enhanced similar to the photo camera example. Independent extensions can be used to effect the incoming messages without modifying the basic object-oriented model. This is illustrated by Figure 11.

A photo camera with a standard lens is a metaphor for the conventional object-oriented model. A photo camera with a set of extensions is analogous to the composition-filters model. The claim here is that the expression power of the conventional object-oriented model can be improved through modular and orthogonal extensions rather than building increasingly complex object structures.

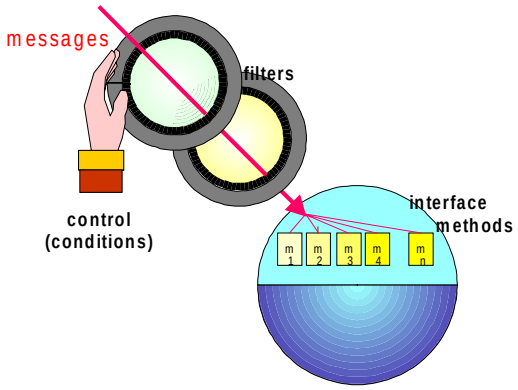

Figure 11. Enhancing objects with modular and orthogonal extensions.

Each message that arrives at an object is subject to evaluation and manipulation by the filters of that object. In this section, we will briefly introduce how composition-filters can help in reusing components without unnecessary re-definitions. Composition-filters can be attached to objects defined in current object-oriented programming languages such as Smalltalk and Java without modifying these languages.

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. The reified message has to pass the filters in the set, until it is discarded or dispatched. Dispatching means that the message is activated or delegated to another object. Each filter can either accept or reject a message. The semantics associated with acceptance or rejection depend on the type of the filter.

6. Composition Filters Approach to the mail problem

6.1 Multiple Views

In Figure 12, the filter specification of class USViewMail is shown. Class USViewMail has two attached (input) filters. The filter USView is an instance of an Error filter. If an error filter accepts the received message, then it is forwarded to the following filter. Otherwise an exception will be generated. The filter Execute is an instance of Dispatch filter. If a dispatch filter accepts the received message, then the message is executed.

The conditions *userView* and *systemView* are Boolean methods defined by class USViewMail. If *userView* is true, then the messages *putOriginator, putReceiver, putContent, getContent, send* and *reply* are accepted by the error filter. 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 constant *true*.

```
USViewMail
   mail: Email;
inputfilters
   USView: Error =
      {userView =>{putOriginator, putReceiver,
         putContent, getContent, send, reply},
      systemView => {approve, putRoute,  deliver},
      true => {getOriginator, getReceiver, isApproved, getRoute, isDelivered};
Execute: Dispatch = { true=> {inner.*, mail.*}};
```

Figure 12. Composition-filters extension of USViewMail.

The specification “inner.*” and “mail.*” means that the dispatch filter accepts all the methods declared by class USViewMail and Email. The pseudo-variable *inner* refers to instance of USViewMail.

Since filters are fully separated from the class, they can be reused separately. For example, the programmers can implement the above mentioned classes in any object-oriented language without attaching filters. Filters can be stacked and attached to any of these classes, whenever necessary. This allows the programmer to implement both LDFE and FDFE ordering strategies. Note that the composition-filters implementation of USViewMail requires only 3 new method definitions: These are 2 view implementations and 1 composition-filters specification.

6.2 View Partitioning

In the following the filter extension for class ORViewMail is given:

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

Figure 13. Composition-filters extension of ORViewMail.

If the view *origView* is true, the messages *putOriginator*, *putReceiver*, *putContent*, *getContent* and *send* are accepted. These messages will then be dispatched to object *mail* 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. The operator "~>" means all messages are accepted except the specified one. The composition-filters implementation of ORViewMail requires only 3 new method definitions. These are the implementation of views and the filter specification.

6.3 View Extension

The composition-filters implementation of Class GViewMail does not require any specific filter definition. Since conditions are methods, they can be inherited from class ORViewMail. However, in GviewMail, these methods must be re-defined as group originators and receivers.

6.4 Advanced Conditions: History Information

Consider now class HistoryMail with its filter extension:

```
count: Meta = { [*] inner.count };
execute: Dispatch = { true=> {inner.*, mail.*}};
```

Figure 14. Composition-filters extension of HistoryMail.

The Meta filter is used to reify a message. If the received message matches, in this specification it always matches ([*]), it is reified and converted to a new message with the original message as an argument of the new message. This new message is then passed to the method *count*. This method reads the attributes of the original message. In this case it reads the method name used in the original call. After that, if the same request has been invoked before the current message, it gives a warning signal and converts the message back to its original form. The dispatch filter then executes it. A more detailed information about Meta-filters can be found in [Aksit et al. 93]. The composition-filters implementation of HistoryMail requires only 2 new methods: a filter specification and the method count.

6.5 Adding Synchronization

Finally, class SyncViewMail has the following filter specification:

```
queue: Wait = {locked => unlock, unlocked => *};
execute: Dispatch = {true=> {inner.*, mail.*}};
```

Figure 15. Composition-filters extension of SyncViewMail.

If the condition *locked* is true, then only an *unlock* message matches the filter. If the condition is *unlocked*, then any message matches the filter. If a wait filter matches a message, then the message is forwarded to the next filter. Otherwise it is queued until the message can be accepted. Note that the composition-filters implementation here requires only 3 new methods. These are the methods *locked* and *unlock* and the filter specification.

7. Conclusion

7.1 Summary of CF Principles

The composition-filters approach aims to enhance the expression power and reusability 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.
2. Instances of a filter class can be created and attached to a class defined in various languages such as Smalltalk and C++. Filter classes are referred to as filter classes or filters, and the later as language classes or classes. Some filters may demand certain features from the language environment such as concurrency and/or real-time scheduling.
3. An instance of a filter class can be defined and attached to a class by using the filter interface definition language. A minimal filter interface definition consists of a class name and an **inputfilters** clause[6]. In Figure 16, *SyncStack* is the class name, and *sync* and *disp* are instances of filter classes *Wait* and *Dispatch*, respectively:

```
class SyncStack interface
inputfilters
   sync:Wait={NonEmpty=>pop, True=>*\pop };
     // specifies synchronization constraints
   disp:Dispatch={ coll.* };
     // provides all the  methods of  orderedcollection
end;
```

Figure 16. Minimal filter interface definition.

4. A filter instance can be initialized using a filter expression. The following expression is used to initialize sync: "sync:Wait= {{NonEmpty=>pop, True=>*\pop};". The second filter expression "disp:Dispatch={ coll.* };" is used to initialize disp. These are declarative specifications in that they do not make any assumptions about how they can be implemented7.
5. If the stack is empty, the condition NonEmpty will be false, therefore a request to the method pop will be blocked. This is expressed by the first filter element of filter sync. In the second filter element, the expression "*\pop" is used to indicate that all messages are acceptable excluding message pop. Thus, messages push, at, remove and size will always pass this filter, as they are associated with the condition True. The expression "disp:Dispatch={ coll.* };" means that disp delegates all the received messages to object coll.
6. The synchronization and delegation operations are based on a filter message manipulation process. If a filter is attached to a class, and if an object is created from that class, then the attached filter may manipulate the messages received[8] by the object. A message manipulation operation may change the implicit attributes of the received message. The implicit attributes are typically the identities of the receiver and the sender objects, the

[6] There are also output filters. For simplicity we do not discuss these filters here.

[7] A filter can be implemented in various ways, for example, as a run-time entity by using message reflection, or as an in-lined code, by using compilation techniques.

[8] In case of output filters, messages sent from the object are manipulated.

name of the method to be invoked, and zero or more arguments. The language environment may add extra attributes to the message, such as real-time constraint values. In the expression "disp:Dispatch={ coll.* };", *coll* is an instance of class *OrderedCollection*. Here, the received message is manipulated by replacing the identity of the receiver with the identity of *coll*, and the *self-variable* with the identity of the current instance of class *SyncStack*. *Dispatch* implements a *true delegation* mechanism as defined by Lieberman []. This requires that the delegating object (here instance of *SyncStack*), must be always referable by the delegated object (here *coll* of *OrderedCollection*), through a pseudo variable such as *self*. To distinguish from the inheritance-based *self-reference*, we introduce a new pseudo variable called *server*, which refers to the delegating object. The detailed description of filter manipulation operations are given in [].

7. Typical manipulation operations are matching and/or substituting. For example, if the condition *NonEmpty* is true, the first filter matches the message with a selector *pop*. If the condition is false, and/or selector is not *pop*, then this filter matches any selector except *pop*.
8. A filter specification may depend on the state of its object. For example, in the first filter specification, the condition *NonEmpty* is true if there are one or more elements in the stack.
9. A filter expression is composed of one or more filter elements. These elements can be combined using logical operators such as CONDITIONAL OR, CONDITIONAL AND, and EXCLUSION. Here, 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. The character "\" is an exclusion operation. A CONDITIONAL AND operation can be implemented by cascading filters, using the ";" sign in the filter definition language. For example, in class *SyncMail* in Section 5, the filters *Wait* and *Dispatch* are composed together in a CONDITIONAL AND manner.
10. A filter specification refers to the parameters of the received messages only. It does not make any assumption about other filters. A filter may, however, refer to the conditions of its object, which can be accessible through the interface operations of the object.
11. A filter expression may also refer to object's interface variables, and some other external variables. For example, the expression disp:Dispatch={ coll.* } refers to the interface variable *coll* of class *OrderedCollection*. This mechanism is used for behavior composition, such as delegation. If a message is delegated to an interface object (here *coll*), the encapsulating object (here instance of *SyncStack*), inherits the interface behavior of the interface object (here instance of *OrderedCollection*) through the delegation mechanism.
12. Filter classes adopt similar initialization syntax. They differ from each other in how they react to the manipulated messages. For example, when a message is accepted by an instance of filter class *Wait*, the message passes to the next filter. If, however, the evaluation is not successful, the message remains in the queue until it fulfils the condition of one of the filter elements. Requests to methods of an object can be synchronised by associating messages with specific conditions that implement specific synchronisation conditions. When a message is accepted by an instance of filter class *Dispatch*, the message is delegated to specified object. If, however, the evaluation is not successful, the message passes to the next filter.
13. For type checking purposes, the filter interface definition language may require additional declarations.
14. Programmers may introduce new filters, provided they fulfil the conditions[9].

[9] In our current implementation of Sina, implementing a new filter class requires sub-classing the language class *Filter* and overriding several operations. The filter compiler recognizes the newly introduced filter classes automatically.

7.2 Evaluation

From the perspective of reusability, the conventional object-oriented model performs unsatisfactorily. The examples show that reusing components using aggregation and inheritance mechanisms may not always be successful, if objects implement concerns like multiple views, history information and synchronization. The aggregation-based reuse requires 94 and 106 method implementations, for LDFE and FDFE orderings, respectively. The inheritance-based reuse performs better, but cannot implement dynamically changing behavior easily. For both LDFE and FDFE orderings, the inheritance-based reuse requires 66 method implementations. In this example, the composition-filters extension requires only 27 implementations. The composition-filters clearly perform better, since they avoid unnecessary method re-definitions. Besides, filters are largely language independent and therefore can be attached to objects implemented in various different languages.

8. References

[Aksit 96] M. Aksit, Separation and Composition of Concerns, ACM Computing Surveys 28A(4), December 1996, http://www.acm.org/surveys/1996/.

[Aksit et al 93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, Abstracting Object-Interactions Using Composition-Filters, In object-based distributed processing, R. Guerraoui, O. Nierstrasz and M. Riveill (eds), LNCS, Springer-Verlag, 1993, pp 152-184.

[Matsuoka 90] S. Matsuoka, K. Wakita & A. Yonezawa, Synchronization Constraints with Inheritance: What is Not Possible- So What is?, Tokyo University, Internal Report, 1990

[Matsuoka 93] S. Matsuoka & A. Yonezawa, Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 19