1. ABOUT THE ASPECTS OF ASPECT-ORIENTED PROGRAMMING

It is clear that developing high quality software systems is a difficult task. Since aspect-orientation is a means to tackle difficult problems, we have to find out the important aspects of aspect-oriented programming.

It looks like that the aspect-oriented community agrees at least on two aspects: aspect-definition and aspect weaving. The fundamental question is, of course, what are the aspects and how to weave them?

2. THE GLOBAL PICTURE

It is a common practice to decompose software development activities into various phases. Typically, these are Requirement Specification, Domain Analysis, Architecture Definition, Design, Implementation and Maintenance. Requirement Specification looks at the problem from the perspective of the user, Domain Analysis aims at finding background information related to the problem being solved, Architecture Definition tries to find out the essential structures in the system, Design is concerned with how to solve the problem, Implementation puts the design into an executable code, and finally, Maintenance tries to cope with the necessary changes after the deployment of the system.

These phases are defined based on the concerns of the software engineer, and managing these are considered essential for reducing the maintenance costs. Since, these concerns have a major impact on the final structure and quality of software, they must be recognized as aspects. It is generally known that going from one phase to another is a difficult step. The so-called seamless design principle, generally a property attributed to object-oriented software development, aims at a smooth transition from one phase to another. Any practical designer, however, soon realizes that object-oriented development is not as seamless as it is claimed. Going one phase to another is actually an aspect weaving process.

3. THE IMPLEMENTATION PHASE

Let us now look at the implementation phase in the software life-cycle. The implementation phase is determined by the current processing and compilation techniques. The most important feature of this phase is that basically the implementation medium is deterministic. Uncertainty is only allowed in a very limited set of primitive mechanisms (like guarded commands) and in transparent features (like process scheduling).

Adaptability is an important quality factor of software systems. Incremental adaptability means coping with changing requirements without modifying the previously defined software components. The conventional object-oriented model supports adaptability through composition, encapsulation, message passing and inheritance mechanisms. Despite its
intended behavior, if a language mechanism fails in realizing an incremental transition from one implementation to another, then it means there is an anomaly in the definition of that mechanism. Obviously, there are also other important quality factors, such as robustness, expressiveness and reusability.

The question remains: What are the relations between aspects and aspect weaving process and the programming language mechanisms? This has to be answered within the context of proving a high quality software.

The answer is quite simple if we consider software development as a problem solving activity. Given a problem, if a solution for the problem exists, that solution determines the aspects and aspect weaving features. So the issue is developing solutions, which are expressible (realizable), adaptable and reusable. To be able to provide these quality factors, we have to find out canonical models for the solution techniques, and within these models, we must identify the aspects and how these aspects interact with each other (weaving). So clearly, the answer is not in the domain of programming, but in the Requirement Specification and Domain Analysis phases.

It is, in principle, possible to define an "ideal solution" for a given problem, if the ideal solution for the problem exists. Construct a canonical model of the selected technique, define a computation model for this model, and finally define incremental adaptability mechanisms for this model (In JPDC July 96 issue, for example, we presented such models for synchronization and real-time problems). Of course, the reality is more difficult than suggested. Apart from finding the right canonical models, the corresponding computation models must be mapped to the current realization techniques (compilers and languages). In addition, since more than one problem can exist within a given application, more than one type of computation model must be composed together.

Due to the difficulty of the problem, and by considering the economics as the major aspect, mapping the solution techniques to the conventional object-oriented languages can be weighed more feasible by many practitioners. The question we have to answer is, how much it does it cost not to apply the aspect-oriented principles. Or in other words, are there economical aspect-oriented solutions for today’s problems?

3.1. HOW GOOD IS THE CONVENTIONAL OBJECT-ORIENTED MODEL

3.1.1. Multiple Views

Example 1: Class EMail
Consider a simple mail system which consists of classes Originator, Email, MailDelivery and Receiver. As an example, the interface methods of class EMail is shown in the following:

```java
putOriginator(anOriginator);
getOriginator(anOriginator)
putReceiver(aReceiver);
getReceiver(aReceiver);
putContent(aContent);
getContent;
send;
```
The interface methods of class EMail

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, getContents are used to write and read the attributes of a mail object. The methods putRoute, getRoute, deliver, isDelivered are used by class MailDelivery while delivering the messages from originators to receivers. The method reply is used to send a reply message.

Example 2: Class USViewMail

Now assume that like in the ordinary postal mail system, we want to restrict accesses to the email objects. We therefore extend class EMail to USViewMail by restricting the accesses to its methods based on the type of the client object. If the client is of User type, 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 System type. No restrictions are defined for the methods getOriginator, getReceiver, isApproved, getRoute and isDelivered.

The immediate problem here is that how to get the identity of the client object? In languages like C++ there is no direct way for obtaining the identity of the client object. This means that the aspects User and System Views may not be directly expressed by some languages. This is a lack of aspect expression problem.

Now assume that the identity of the client object is available. There are two possible implementations in the conventional object model: composition based and inheritance based.

In the composition-based implementation, the interface object implements the view checking method. The aggregated object implements the method. For example, the method putOriginator can be implemented as follows:

```python
putOriginator (anOriginator)
    if self.userView then imp.putOriginator(anOriginator)
    else self.viewError;
```

Here imp is the name of the aggregated object.

In the inheritance based approach, view checking is implemented within a method, and executions are realized though super calls:

```python
putOriginator (anOriginator)
    if imp.userView then super.putOriginator(anOriginator)
    else self.viewError;
```
Example 3: Class ORViewMail

As the next example, we partition the User View into Originator and Receiver views. The methods putOriginator, putReceiver, putContent and send can only be invoked by the client of Originator type. The client of Receiver type is allowed to invoke the method reply. For the other methods, the restrictions defined by USViewMail applies.

Again, this class can be implemented using composition or inheritance based structures. In the composition based implementation, the aggregated object is an instance of class USViewMail. In the inheritance based implementation, class ORViewMail inherits from class USViewMail.

There are two ways how the view checking can be ordered: (a) first the Originator and Receiver views then the User and System Views or (b) first the User and System views and then the Originator Receiver views.

Example 4: class GViewMail

In the next example, we extend the ORViewMail to GViewMail by extending the views to a group of originators and receivers. This may be required, for example, for offices where more than one person is responsible of sending and receiving mails. Again, there are two ways how the view checking can be ordered. First group originator and receiver views, then originator and receiver views and then user or system views. This ordering we term as "last defined first enforced". The reverse order, however, may be considered more logical (or correct).

Other multiple view examples:

Assume that class W2Mail extends the GVMail class with a history view. If the user method is invoked twice for the same mail object a warning message has to be generated.

Class SecMail extends W2Mail by composting two separately defined views. The first type of view is taken from class W2Mail. The second type of view is the security view and defined for the mail content. Every mail data is classified with security layers. A client object is only allowed to execute the methods of SecureMail if the security level of the client object is equal or higher than the security level of the mail data.

3.1.2. Other Aspects

Of course, one may require many more aspects for the mail objects. For example, class SyMail represents a mail object which can be locked or unlocked. If the method locked is invoked, then all the invocations are delayed until the invocation of the method unlocked.

Class DyMail changes the implementation of the method send based on the type of the mail data. If the mail is of ASCII type then, a simple send protocol is used. If the content is of video type, then the video protocol is used, and so on.

3.2. Evaluation
The assessment of conventional object-oriented languages is presented by the following table:

Here, the first column indicates the classes as defined in the previous section. In principle, the order of the rows (evolution order) has no affect to the number of method definitions, except for the first row. Here, the second column indicates the minimal number of required methods. This number is derived from the total number of additional aspects and plus one for the weaver. The third column (A) indicates the required number of methods if the checking order is based on rule "last defined first enforced". The fourth column (B) indicates the required number of methods if the checking order is based on "first defined first enforced. The meaning of the symbols are defined as follows:

- i: implementation, f: forwarding a request to the aggregated object, v: implementation of a view, c: methods necessary to re-configure the aggregated object. These are put, get and reconfigure methods (combination of Decorator and Bridge patterns). a: bookkeeping method such as counting the number of invocations, s: security level checking, q: a delay mechanism such as a semaphore, t: testing an attribute.

<table>
<thead>
<tr>
<th>class Name</th>
<th>Min #</th>
<th>A: # of methods</th>
<th>B: # of methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USViewMail(comp)</td>
<td></td>
<td>+16 = 9i + 5f + 2v</td>
<td>not applicable</td>
</tr>
<tr>
<td>USViewMail(inh)</td>
<td>+ 3</td>
<td>+11 = 9i + 2v</td>
<td>not applicable</td>
</tr>
<tr>
<td>ORViewMail(comp)</td>
<td></td>
<td>+16 = 9i + 5f + 2v</td>
<td>+19 = 9i + 5f + 2v + 3c</td>
</tr>
<tr>
<td>ORViewMail(inh)</td>
<td>+ 3</td>
<td>+11 = 9i + 2v</td>
<td>+ 7 = 5i + 2v</td>
</tr>
<tr>
<td>GViewMail(comp)</td>
<td></td>
<td>+16 = 5i + 9f + 2v</td>
<td>+19 = 5i + 9f + 2v + 3c</td>
</tr>
<tr>
<td>GViewMail(inh)</td>
<td>+ 2</td>
<td>+2 = 2v</td>
<td>7 = 5i + 2v</td>
</tr>
<tr>
<td>W2Mail(comp)</td>
<td></td>
<td>+15 = 14i + 1a</td>
<td>+18 = 14i + 1a + 3c</td>
</tr>
<tr>
<td>W2Mail(inh)</td>
<td>+ 2</td>
<td>+15 = 14i + 1a</td>
<td>+15 = 14i + 1a</td>
</tr>
<tr>
<td>SecMail(comp)</td>
<td></td>
<td>+15 = 6i + 8f + 1s</td>
<td>+18 = 6i + 8f + 1s +3c</td>
</tr>
<tr>
<td>SecMail(inh)</td>
<td>+ 2</td>
<td>+7 = 6i + 1s</td>
<td>+7 = 6i + 1s</td>
</tr>
<tr>
<td>SyViewMail(comp)</td>
<td></td>
<td>+17 = 14i + 2i + 1q</td>
<td>+20 = 14i +2i+1q + 3c</td>
</tr>
<tr>
<td>SyViewMail(inh)</td>
<td>+ 3</td>
<td>+17 = 14i + 2i + 1q</td>
<td>+17 = 14i + 2i + 1q</td>
</tr>
<tr>
<td>DyViewMail(comp)</td>
<td></td>
<td>+17 = 1i + 13f + 3t</td>
<td>+20 = 1i +13f + 3t +3c</td>
</tr>
<tr>
<td>DyViewMail(inh)</td>
<td>+ 4</td>
<td>+4 = 1i + 3t</td>
<td>+4 = 1i + 3t</td>
</tr>
</tbody>
</table>

Table: Evaluation of conventional object-oriented languages with respect
to the example.

In the ideal case, the total number of required methods is 33. In the composition-based implementation, the total number of required methods is 126 and 144, for cases A and B, respectively. In the inheritance-based implementation, the total number of required methods is 77 and 82, for cases A and B, respectively. The inheritance-based approach requires less methods because method forwarding is not necessary (inheritance provides transitive reuse) and polymorphic overriding is possible. For example, class GViewMail can be easily derived from ORViewMail by redefining the methods for checking the originator and receiver views. However, the inheritance based approach cannot support dynamic evolution. Obviously, inheritance and compositional techniques can be combined. For example, redirected methods can be inherited from a common super class. The methods necessary for adapting the object model can be inherited as well. The total number of methods in case of B can then be reduced to 109.

3.2. THE COMPOSITION-FILTERS APPROACH

In the composition-filters approach, the basic behavior is implemented by using any programming language, and the additional aspects are defined in the filters.

In the following, the filters of class USViewMail are shown:

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

The filter `checkview` if an instance of Error filter. If a received message is accepted by an error filter, then it is forwarded to the following filter.

The filter `execute` is an instance of Dispatch filter. If the received message is accepted by a dispatch filter, then it is executed. The conditions `userView` and `systemView` are Boolean methods defined by class USViewMail. If `userView` is true, then the methods `putOriginator, putReceiver, putContent, getContent, send` and `reply` are accepted by the error filter. Similarly, the methods `approve, putRoute` and `deliver` are only accepted if `systemView` returns true. The remaining methods are not restricted by the error filter. The dispatch filter accepts all the methods declared by class USViewMail (through inner) and Email.

This composition filter specification can be attached to class USViewMail implemented in any object-oriented language.

Attaching is the weaving operation. Since filters provide a declarative specification, this attachment can be considered as a static weaving. The filters of class USViewMail, for example, can be fully integrated (in-lined) with the implementation of USViewMail.
Since the semantics of these filters can be defined as message transformations, filtering can also be considered as a dynamic weaving process (a smart compiler may weave a filter partially or fully statically).

Filters are fully separated from the class definitions (this is not possible in the language Sina), and therefore, 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 "last defined first enforced" and "first defined first enforced" strategies. However, once a filter is attached to an object, it cannot be separated from it at run-time (at least in the current implementation. However, reflective techniques can make this possible). If a run-time change is required, then methods must be provided to reconfigure the object compositions (Combination of Decorator and Bridge patterns).

ORViewMail:

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

Here, the operator "~>" means all messages are accepted except the specified one.

GViewMail:

Class GViewMail can be implemented by redefining the view checking methods for a group of originators and receivers.

W2Mail:

count: Meta = { [*] count.input };
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 ("[*]"), it is reified and concerted to a new message with the original message as an argument of the new message. This new message is then passed to the object count. This object can read the attributes of the original message. In this case it reads the selector of 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 (by invoking the fire method) back to its original form. It is then executed by the dispatch filter.

SecMail:

document.secureFilter;
execute: Dispatch = { true => { inner.*, mail.* } };

Here, secureFilter is the name of the filter defined at the interface of the internal mail object document. All the received messages first have to pass through this filter. If they succeed (which means the security-level of clients are accepted) then they are forwarded further and checked by the other filters implementing the views.

SyViewMail:

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

If a message is accepted by a wait filter, then it is forwarded to the next filter. Otherwise it is queued until the message can be accepted.

DyViewMail:

select: Dispatch = {videoType=> videoPr.send, asciiType=> ascii.send,
  imageType=> imagePr.send,
  true ~> send;
execute: Dispatch = {true=> {inner.*, mail.*}};

3.3. EVALUATION OF THE COMPOSITION-FILTERS APPROACH

If the filters are statically attached, then the composition-filters implementation of the scenario performs as good as the ideal case. If a filter implementation is considered equivalent to a method implementation, then the complete scenario can be implemented using 33 methods. However, if the intention is to dynamically reconfigure objects, then 3 additional methods must be defined to put, get and configure the internal objects.

Are all the problems then solved? Filters only provide aspect definition and weaving at the language level. There are, however, many design level concerns. For example, one may define alternative implementations for the same scenario. In some cases, it may be desired to change the order of view enforcement, and in another case, this could result in unnecessary overhead. All these concerns can be explicitly programmed, but this will make software hard to maintain. Therefore, it would be better to define design-level aspects and design-level weaving processes.

Important characteristics of design level aspects is that mostly they are based on uncertain factors, they are conflicting, context-dependent and non-deterministic. Therefore, new aspect definition and weaving mechanisms must be defined. Currently, we are experimenting with design level aspects and context-dependent fuzzy design rules. The workshop 10 of ECOOP'97 is somewhat devoted to aspect-oriented design.

4. Conclusions

Aspect-oriented programming must be considered in a broader context. The structure of a software system has to be based on the results of Requirement Specification and Domain Analysis phases. Ideally, aspects and aspect weaving process have to be derived from the canonical models of solution techniques. Architecture definition and design phases have
to deal with conflicting and ambiguous aspects.

Generally, implementation phase is based on unambiguous and context free solutions.

From the perspective of adaptability and reusability, the conventional object-oriented model performs unsatisfactorily. Especially, multiple views, synchronization and conditionally changes behavior cannot be implemented well. Inheritance based solutions perform better, but however, they cannot implement dynamically changing behavior. The conventional object-oriented model requires 3 to 5 times more method implementations than the ideal case. The composition-filters model provides almost an ideal solution. However, the composition-filters model is not capable of expressing aspects and weaving process at the design-level. Therefore, new techniques must be defined for design-level aspects and aspect weaving processes.