

Component Composability Issues in Object-Oriented Programming

Mehmet Aksit
Bedir Tekinerdogan

University of Twente

Building software from reusable components is considered important in reducing development costs. Object-oriented languages such as C++, Smalltalk, and Java, however, are not capable of expressing certain aspects of applications in a composable way. Software engineers may experience difficulties in composing applications from components, for example if components implement code for multiple views, dynamic inheritance, and synchronization [1]. If these aspects have to be programmed, then object-oriented languages may require a considerable amount of redefinition although this may not be intuitively necessary. To solve the composability problems, languages must be enhanced modularly without losing their basic characteristics. In addition, since more than one problem can be experienced for the same object, enhancements must be independent from each other. We have extended the conventional object-oriented model using the concept of composition-filters. Composition-filters can be attached to objects expressed for example in Smalltalk and Java. A number of different filter types have been defined, each addressing a certain concern. This paper first illustrates some practical problems and then introduces composition-filters solutions to overcome these problems.

The conventional object-oriented model is considered unsatisfactory with respect to reusability of software components. Current reuse strategies by aggregation or inheritance require an overload of method re-definitions. As a consequence, OO languages have to be extended without sacrificing their useful features. The concept of composition-filter is an language independent enhancement to the OO model; it avoids unnecessary method re-definition when components are reused.

Examples

In the following sections, we illustrate some component composability problems by using a number of classes. Although intuitively unnecessary, if an existing class say A, cannot be reused by a new class, say B, without modifying the implementation of class A, then this is termed as a composability problem.

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 Figure 1.

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 delivering the messages from originators to receivers. The method *reply* is used to send a reply message. In this article, EMail will be used as the base class for developing various kinds of email objects.

```

class Email interface
putOriginator(anOriginator);
getOriginator returns anOriginator;
putReceiver(aReceiver);
getReceiver returns aReceiver;
putContent(aContent);
getContent returns aContent;
send;
reply;
approve;
isApproved returns Boolean;
putRoute(aRoute);
getRoute returns aRoute;
deliver;
isDelivered returns Boolean;

```

Figure 1: The interface methods of class EMail.

Class USViewMail

Now assume that like in a postal mail system, we want to restrict accesses to 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 the 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 the system type. No restrictions are defined for the methods *getOriginator*, *getReceiver*, *isApproved*, *getRoute*, and *isDelivered*.

Now assume that the identity of the client object is available. There are mainly two possible ways of reuse in the conventional object model: aggregation-based and inheritance-based. In our example, in case of aggregation-based reuse, the interface object implements the view checking operation. The aggregated object implements the method to be executed. For example, the method *putOriginator* can be implemented as shown in Figure 2.

```

USViewMail::putOriginator(anOriginator)
if self.userView then
    imp.putOriginator(anOriginator)
else self.viewError;

```

Figure 2: Aggregation-based reuse of *putOriginator*.

In Figure 2, if the sender of the message is of the user type, the message *putOriginator* is forwarded to the aggregated object *imp*, otherwise the error method *viewError* is invoked. Here, *imp* is an instance of Email. Notice that in the aggregation-based reuse, all the methods have to be declared at the interface of USViewMail, even though some methods do not require any view enforcement.

In the inheritance-based reuse, view checking is implemented within a method, and reuse is realized through super class calls. Here, only the methods with views have to be redefined; other methods can be inherited from the super classes.

```

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

```

Figure 3: Inheritance-based reuse of *putOriginator*.

In case of the aggregation-based reuse, USViewMail implements 16 methods¹. Among these, nine methods implement view checking and forwarding (see Figure 2), five methods are used for forwarding only, and two methods implement the views. The inheritance-based implementation requires 11 methods. Here, nine methods implement view checking and super class calls (see Figure 3), and two methods implement the views.

Class ORViewMail

Assume that class ORViewMail partitions the user view into originator and receiver views. Only the client of originator type is allowed to 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. In the example, in case of aggregation-based reuse, the aggregated object is an instance of class USViewMail. In the inheritance-based reuse, class ORViewMail inherits from class USViewMail. 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, and then the user and system views. This ordering is termed as last-defined-first-enforced (LDFE). (2) First the user and system views, and then the originator and receiver views. This ordering is termed as first-defined-first-enforced (FDFE).

Implementation of LDFE ordering is relatively simple because object-oriented models naturally support it. In the aggregation-based reuse, 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 reuse, ORViewMail implements 16 methods. The inheritance-based reuse requires seven methods. Here, five methods are for originator and receiver view checking and two methods implement the originator and receiver views.

1. The exact number of methods depends on the language used.

The aggregation-based implementation of FDFE ordering is somewhat more complicated, because it requires reordering of the aggregate structures. Consider the code as shown in Figure 4. 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 three additional methods for reconfiguring the aggregation structure, resulting in total 19 method implementations.

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

Figure 4: 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 seven. Consider the implementation of the method *putOriginator* of class ORViewMail (see Figure 5). Notice that here first *userView* and then *originatorView* are verified.

```
ORViewMail::putOriginator(anOriginator)
  if self.userView then
    if self.originatorView then
      super.putOriginator(anOriginator)
    else self.viewError;
```

Figure 5: Inheritance-based reuse of *putOriginator* in FDFE implementation.

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. In case of the aggregation-based reuse, the implementation of class GViewMail is similar to the one shown in Figure 4. A total of 16 methods have to be imple-

mented: five methods are used for view checking, nine methods are used for forwarding messages only, and two methods implement the views.

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 two methods are required for re-implementing the views. The aggregation-based FDFE implementation requires in total 19 methods. Among these, three methods are used to configure the aggregation structure.

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 seven methods are required: five methods are used for view checking and two methods implement the views.

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.

Figure 6 shows a 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(fputOriginatorF) then
    imp.putOriginator(anOriginator)
  else self.giveAWarning;
```

Figure 6: Aggregation-based LDFE ordering of *putOriginator* in class HistoryMail.

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

Class SyncMail

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

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 7.

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

Figure 7: 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, two methods are required for lock and unlock operations, and one method is used for implementing the semaphore. 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.

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. US-View mail illustrates that a considerable number of methods of EMail have to be re-implemented if two views are enforced on nine methods. Class OR-ViewMail 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 re-definition of all the methods.

Despite of all these composability problems, the ob-

ject-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.

The Composition-Filters Approach

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 8.

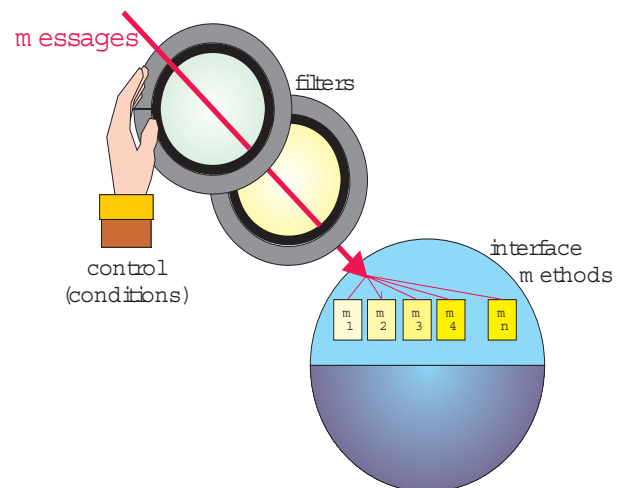


Figure 8: Enhancing objects with modular and orthogonal extensions.

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-orient-

ed model can be improved through modular and orthogonal extensions rather than building increasingly complex object structures.

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.

In Figure 9, the filter specification of class USViewMail is shown.

```
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 9: Composition-filters extension of USViewMail.

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 five methods are not restricted by the error filter, because the condition is specified as constant true.

The specification 'inner.*' and 'mail.*' means that the dispatch filter accepts all the methods declared by class USViewMail and Email. The pseudo-vari-

ble *inner* refers to an 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 three new method definitions; these are two view implementations and one composition-filters specification.

In Figure 10, 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 10: 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 three new method definitions. These are the implementation of views and the filter specification.

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.

Consider now class HistoryMail with its filter extension as shown in Figure 11.

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

Figure 11: 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 [2]. The composition-filters implementation of History-Mail requires only two new methods: a filter specification and the method *count*.

Finally, class *SyncViewMail* has the filter specification as shown in Figure 12.

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

Figure 12: 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 three new methods. These are the methods *locked* and *unlock* and the filter specification.

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.

References

- [1] M. Aksit, *Separation and Composition of Concerns*, ACM Computing Surveys 28A(4), December 1996, <http://www.acm.org/surveys/1996/>.
- [2] 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.



Dr.ir. Mehmet Aksit holds an M.Sc. degree from Eindhoven University of Technology and a Ph.D. degree from the University of Twente. Currently, he is an associate professor at the University of Twente. He is the leader of the Twente Research and Education on Software Engineering (TRESE) project.

Ir. Bedir Tekinerdogan is currently a Ph.D. student within the TRESE project. His research is related with the development of adaptable OO models and methods for large scale complex software systems.