

# Replication as an Aspect

Position paper submitted to the ECOOP'98 workshop on AOP

Johan Fabry (johanf@wendy.vub.ac.be)

May 20, 1998

## Abstract

We have performed an experiment in writing an aspect language for replication. This experiment has shown that although at first sight replication could easily be described as an aspect, a number of problems crop up making full separation of concerns hard, if not impossible.

## 1 Introduction

In the context of our graduation thesis [1], we have constructed an AOP extension for replication to Java. The base algorithm of the program using replication is implemented in a variant of Java, the replication aspect and the error-handling aspects are specified in two separate aspect languages.

In this experience report, we will discuss how replication can be seen as an aspect, and explain the two major problems: naming of replicas and initialization of replicas. Two solutions to these problems are addressed, which do not achieve complete separation of concerns. It remains an open question whether it is possible to find a solution which achieves full separation.

From this experience, we derive the following question which we would like to see discussed in the workshop: "Do we always need to achieve full separation of concerns?" In some cases this might not even be possible, in other cases the work required to achieve this might not outweigh eventual benefits for the programmer.

We will now briefly introduce the topic of replication, after which we will analyze what will be replicated, how this can be achieved and how possible errors will be handled. Finally, the problems of naming and initialization will be discussed, and our solutions are presented. These solutions do achieve a high degree of separation, but do not achieve full separation of concerns.

## 2 Replication

An important problem in distributed systems is sharing of data. The different computers which make up a distributed system must be able to work with the same data to achieve a common goal. A number of solutions for sharing the data exist, one of which is replication. In *replication* a number of servers contain copies of the data, and clients can access this data through the network. Whenever changes are made to the data on one server, this server will ensure that these changes are propagated to the data on the other servers, keeping the data in a consistent state.

We will use the term *replicamanager* to describe the server process which manages the accesses to the shared data and which ensures the data consistency. The different processes running on client computers and which access the data will be called *clients*.

## 2.1 Replication of What?

To determine how we should approach replication, it is crucial to first analyze what will be replicated. Because we are working in an OO language, the maxim ‘everything is an object’ seems to give the answer: what we are replicating is an object. But an object is more than just data, objects also contain behavior. Nevertheless, for the moment, we have chosen to replicate data only, but without ruling out the possibility to also replicate the objects’ behaviour later on. So what needs to be replicated is merely the data contained in the object, which is contained in its instance variables (also known as its *fields*). Note that these fields may either be Java primitive types or objects. The Java primitive types can be considered purely as data; they can either be read or written. In case the variables contain objects (which we will call *sub-objects*), the sub-object’s methods could be executed.

Method calls on sub-objects will either modify the sub-objects state or will not modify the state. If the state is modified, these changes will have to be propagated to the other replicamanager. Therefore the replicamanager will have to be aware of the possible effect of method executions on sub-objects to ensure the data can be kept consistent. At the moment we have chosen not to allow this, and changes made to sub-objects by methods of these sub-objects are lost. To avoid this, the sub-object can still be copied to a temp object on the client, the changes made on this temp object, and the temp object saved back to the replicamanager.

So should the programmer want to replicate fields which contain objects, she should regard replicated sub-objects purely as data. This data can be copied locally, and if needed, changed locally and written to the replica. If the sub-objects will be used and changed frequently, the programmer should replicate these sub-objects and not keep them as replicated sub-objects of a replicated object.

We can now define a *replicated object* as an object containing a number of fields which will be replicated. A *replica* for an object is the collection of the *replicated fields* of an object. The collection of all replicas on the replicamanager for a given replicated object is the replicated objects’ *replicagroup*.

## 2.2 How to Replicate

Replicating an object is replicating a number of its fields, which we treat as primitive types. These variables can either be read from, or assigned to. Since these fields will no longer be contained within the object, but in its replicagroup, access to the fields will have to be changed into accesses of the fields on the appropriate replicas within the replicagroup.

In a number of cases, it might not be necessary to replicate all fields of an object. In these cases, replicating only the fields which need to be replicated will speed up the system.

To provide this greater control over replicated objects, a means must be provided to specify which fields of the object must be replicated and which fields must not. This will be the role of the aspect language; using the aspect language, a programmer will be able to specify which fields of a class must be replicated.

This defines the first task for the weaver: the weaver must use the aspect language to determine which fields of a class must be replicated, and for these fields it must modify all reads and assigns into reads and writes on the appropriate replicagroup. The weaver must also create the classes of the replicas which will make up the replicagroup.

## 2.3 Error-Handling

In most cases when using replication, a program will have a network link to its replicamanager, instead of a ‘regular’ association of an object with another object within the same program. This

means that interactions between the client and the replicamanager will usually have to occur over the network. An important factor in these network accesses is that they can fail, and throw an exception.

Clearly, catching the exceptions should not be done in the base algorithm, because it should not be aware of the replication aspect. We could let the aspect weaver provide only a default error handler, but this is clearly not a good solution. The severity of being unable to access the replicas will be different for different types of replicated data. The programmer will want to specify different exception handlers for different kinds of replicated data, so she can take appropriate action. These actions will also depend on what kind of application the programmer is writing. So providing only one, or a limited number of default error handlers, should be avoided.

The straightforward solution would be to add to the replication aspect language a construct which allows the programmer to specify exception handlers for the network accesses. However, error handling could itself be considered as an aspect, since it is a special-purpose concern in the code, and error-handling code tends to be interspersed throughout all the code of the base algorithm. Therefore it makes sense to specify the error-handling in its own separate aspect language. At the moment only the errors caused by the replication aspect can be handled by our error-handling aspect language, but it should be straightforward to later add other sources of errors to the error-handling aspect language.

Let us now concentrate on when these errors can occur. Since the only accesses to the replicated data are either reads or writes, the only moments at which errors can occur seem to be at reads or at writes of the data. This overlooks one type of error: the error which might occur when the client first tries to access the replicamanager to locate the correct replicagroup. So at least these three types of errors need to be handled, errors when first accessing the replicamanager, errors while reading the data and errors while writing the data.

Whenever these errors occur, the actual source of the error can be one of many; the network might be faulty, the replicamanager may be inoperative, . . . . In Java different kinds of exceptions are thrown for these errors, so it should be possible to specify an error-handler for each kind of exception which might be thrown.

To cover the combination of kinds of operations which might throw an exception, and the kind of exceptions thrown, the error-handling aspect language should enable the programmer to specify an exception handler, based on a type of exception, for each kind of operation (read, write and contact).

The task of the weaver would then be to add these exception handlers to the code responsible for accessing the replicas.

## 2.4 Naming and Location

In a system where multiple replicagroups are active, a replicagroup must be identified uniquely to be able to access it. Some sort of ‘naming service’ must allow the accesses of replicated fields to be executed on the correct replicagroup based on a name given to this replicagroup.

### 2.4.1 Example

Consider the following example: a Counter class, containing an integer field `count` and a method `add()` which adds 1 to `count` needs to be replicated, to e.g. be able to count the number of bottles of beer produced by different production lines in a beer factory. One Counter object would be replicated and each production line would increase the replicated Counter object whenever a bottle of beer is produced. Now suppose the Counter should not count the overall number of bottles produced, but a different Counter should exist for each type of beer (say Pils, Kriek and

Witbier)<sup>1</sup>. How will the production line be able to distinguish between the different counters? This must be possible to make sure each line only updates the product counter of its own kind of beer, and not of some other kind. In other words, lines producing Pils should not increase the counter for Kriek or Witbier and vice-versa.

One possible solution would be to create a different Counter class for each type of product, say Pils\_Counter, Kriek\_Counter, and Witbier\_Counter. This is not a workable solution in many cases, because of its inflexibility. Each time a new kind of beer is added to the range of products manufactured by the production lines, the counter program will need to be partially reprogrammed, by implementing a new Beer\_Counter.

A better solution would be to allow different instantiations of the Counter class on the replica-manager, where each instantiation represents a different kind of beer. However, how can we now distinguish the different Counter replicated objects? How will the different production lines know which counter to increase?

To be able to distinguish the counters, each counter should have an unique identifier, say the type of beer it is counting. Using this unique identifier, the client can ensure it will interact with the correct replicated Counter object.

The need for a unique identifier is not unique to replication, all forms of data sharing in distributed systems need to be able to uniquely identify a part of the data, so it can be correctly read or written. The algorithm which establishes the correct identity based on a given ‘name’ is usually called the *naming service*. Providing the name for a replica is also known as ‘naming’ the replica.

## 2.4.2 Naming

As we have seen in our example, the identity of a replicagroup cannot be determined per class, because different objects of the same class may be replicated at the same time, each representing a different replica.

So providing the name of the replicagroup for an object and finding it must be done at run-time. The algorithm which provides the name must know what the replicated object will be used for, so it can provide the name of the correct replicagroup. The only algorithm which ‘knows’ for which purpose the replicated object is created, is the base algorithm. In our example, only the production lines will know what kind of beer they are producing.

This leads us to require that naming the replicagroup has to be performed by the base algorithm, because only this algorithm knows what the replicated object will be needed for. All the objects used by different clients for the same purpose will then have the same replicagroup, ensuring correct data sharing. In our example, production lines producing Pils should share the counter for Pils amongst themselves, production lines for Kriek should share the counter for Kriek, and lines for Witbier should share the Witbier counter. To achieve this, each production line producing Pils, will increase the Counter named “Pils”, the lines producing Kriek will increase the counter “Kriek”, and the lines producing Witbier will increase “Witbier”.

However naming the replicagroup from the base algorithm has serious consequences. When this is required, the base algorithm will need to concern itself with elements of the replication aspect. Base algorithm and aspect are no longer separated, which is clearly not desirable.

Solutions which contain no explicit naming can be imagined, but cannot prevent the need for a replicated object to be associated with a replicagroup. The association could be based on the class of the replicated object, but this excludes the possibility to have different objects of the class to be replicated.

---

<sup>1</sup>For more information on different kinds of Belgian beers, we refer to the Belgian Beer Degustation taking place after the workshop.

We have chosen not to develop this approach, and we keep our original requirement: The base algorithm will need to specify a name for each replicated object when it is instantiated. This name will be used to associate the replicated object with a replicagroup.

### 2.4.3 Location of Replicagroups

To perform accesses to the replicagroup, not only must the group be named, but also a network location for the replicamanager to be used for this group must be given. The location of the correct replicamanager depends on the location of the program in the network. It is not advisable to hard-code this location into the program, as it would lead to a separate build for each copy of the program. Each copy needs to be able to determine at run-time which replicamanager it should use. The easiest way for this would be to provide a (list of) replicamanagers in a configuration file which can easily be changed for each copy, without requiring a rebuild.

### 2.4.4 Conclusion

We have now defined a third task for the weaver: The weaver must ensure that whenever a replicated object is instantiated, it is associated with the named replicagroup for the object, and with the correct replicamanager, all according to a configuration file.

## 2.5 Initalization

One important factor which can easily be overlooked in replication is initialization of the replicas. Whenever a replica is created, it will have to be initialized to certain values.

The case for initialization of the replica is similar to naming the replicagroup, which we have discussed in 2.4. Initialization of the replica should be done from the base algorithm, because only the base algorithm ‘knows’ what the replica will be used for, and thus knows the relevant initial values. This is what happens in the objects’ constructor: the initial values for fields of the object are set, according to what the object will be used for.

Now consider when the initial values should be written to the replica. Clearly this should only happen immediately after it has been created, certainly not after the replica has been used by a client. However, because replicas are accessed in a transparent fashion by the base algorithm, it is not aware of their existence. Therefore, the algorithm cannot write the initial values to the replica only when it just has been created. Instead, every time a client will instantiate a replicated object, the initial values for its fields will be written to the replica, essentially re-initializing it (implying a re-initialization for all clients, which is probably not desired).

To avoid this unwanted re-initialization, initializing replicated objects must be considered with some care. There are two possible options to perform this initialization: The first option would be to create a special initialization program, which explicitly initializes the replica. The second option would be to let the program check if the replica contains appropriate values and, if not, assign correct values to the replicated fields.

Unfortunately, this need to re-think initializers for the replicated fields implies that the base algorithm will again need to handle an element of the replication aspect. A possible solution would seem to be to introduce the following convention: the initializers in the replicated fields’ declarations and the assignments in the objects’ constructors are initializers for the replica, and need only be executed when the replicagroup is created. This implies that initialization of replicated objects should respect this convention, so the programmer will need to consider the replication aspect whenever she is working on the replicated object, i.e. the base algorithm.

So in either case, how the replicated object is initialized should be considered with great care. Therefore we have chosen not to use this convention, and to always perform assignments in the

constructors of replicated objects.

### 3 Conclusion

We have created two aspect languages, one for replication and one for error-handling, and built an aspect weaver for these languages. While creating these languages, we encountered two problems which did not allow us to reach full separation of concerns: naming and initialization.

To ensure clients connect themselves to the correct replicagroup, a naming service must be implemented. Because only the base algorithm knows to which replicagroup accesses must be made, naming the replicagroup must be performed from the base algorithm. Also, for correct initialization, the base algorithm needs to be re-examined to ensure replicas are not re-intitalized whenever a client creates a connection to the replicagroup.

Because of these two elements, we do not have full separation of concerns. This leads to the research question wether it is indeed possible to obtain a full separation of the replication concern. We also fear replication will not be an isolated case, and other concerns may also not be separated fully from the base algorithm. Further research in this area must be undertaken to determine which concerns cannot be fully separated, and to which degree they can be separated.

Not only for these cases, but for all concerns, we suggest a trade-off be made between the work needed to ensure a larger degree of separation and the gains which would result for the programmer.

A similar problem which emerged while working on our aspect languages is the orthogonality of these languages, which has also been recognised as a topic for futher research in the ECOOP'97 AOP workshop [2]. We have seen that our two aspect languages are not completly orthogonal: our error-handling aspect language contains elements specific for the replication aspect. Again the question is wether it is indeed possible to obtain full orthogonality between any two aspect languages and if not, to which degree they can be made orthogonal.

Preferably, we would like to have some general criteria to determine wether full separation and/or orthogonality can be achieved, and if not, to which degree it can.

### Acknowledgements

Thanks to Kim Mens, for suggesting me to write this paper and for helping me while I was writing it.

### References

- [1] Johan Fabry. A framework for replication of objects using aspect-oriented programming. Graduation thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica, 1998.
- [2] Kim Mens, Cristina Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-oriented programming workshop report. In *Ecoop'97 Workshop Reader*. Springer Verlag, 1997.