Transaction Management in EJBs: Better Separation of Concerns With AOP

Johan Fabry*
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussel, Belgium
Johan.Fabry@vub.ac.be

March 8, 2004

1 Introduction

The long-term goal of our research is to enhance transaction management in multi-tier distributed systems through the use of higher-level, semantical information and advanced transaction models. As a part of this research, we have performed an evaluation of transaction management in Enterprise JavaBeans, and found it lacking in multiple respects.

Most relevant for this workshop is the bad separation of concerns, as we will show below. In the worst cases, code handling the concern of transaction management can be split in three distinct locations. We feel it would be better to centralize this code into one location, by using a well-defined aspect. Our current work is the implementation of such an aspect, which we introduce here.

This paper first details how Enterprise JavaBeans aims for container-based separation of the transaction concern, and discusses how this concern remains split. Second, we describe our proposed transaction management aspect for EJBs, which aims to integrate this split concern into one. Third, we conclude and project our future work.

2 Container-Based Tx Management in EJBs

Enterprise JavaBeans (EJB) [8] is a well-known and widely used Java component architecture for middleware applications, which, among other services, provides for Container-based Transaction management. The EJB Object, which defines business code, lets its' Container manage transactions by declaring transactional properties for each method in a separate file: the deployment descriptor.

2.1 Declarative Transaction Management

In EJB, transaction management is provided by the container, and this behavior is usually determined by the transaction attributes set in the EJB's deployment

*Author funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) in the context of the CoDAMoS project.

descriptor. This is called *declarative transaction management* [5] and is an instance of Container-based separation of concerns, since the Container will perform transaction management.

In the deployment descriptor, the Bean declaratively states its transaction requirements for every business method. For example, `Required` states that the method must be executed within the scope of a transaction, and if needed, a new transaction will start, while `Mandatory` states that the method invocation will fail if not executed within a transaction scope.

Because all calls to an EJB Object are mediated by the Container, this Container can now transparently start and end transactions. If a transaction is created upon execution of a method, the Container will commit or rollback this transaction when the method ends. The decision to rollback the transaction is primarily based on exceptions thrown. If the method (or a nested method called by this method) throws a *system exception*: a `RuntimeException`, a `RemoteException`, or a subclass of these exceptions, the transaction will be rolled back, and the method will throw a `TransactionRolledbackException`. Also, a transactional method can mark itself for rollback by calling the `setRollbackOnly()` method on the Container, and determine its rollback flag by calling the `getRollbackOnly()` method on the Container. When a transactional method that is marked for rollback ends, the transaction will be rolled back but no exception will be thrown.

Declarative transaction management is said to achieve a greater separation of concerns [5] and is said to allows the transactional behavior of the EJB to be modified without needing to change the implementation of the business logic [8] . Therefore, the same Bean should be able to be reused over different applications with different transactional requirements.

However, while declarative transaction management is a promising evolution in transaction management, this new concept, as implemented in EJB, has a number of significant drawbacks. One of these drawbacks is especially relevant here: the incorrect separation of concerns.

2.2 Separation of Concerns?

As said above, declarative transaction management is said to achieve a greater separation of concerns, resulting in cleaner business method code. However, if we further investigate how declarative transaction management is currently implemented in EJB, we see that this is not really the case.

Consider commits and rollbacks: the decision to commit or rollback a transaction is made primarily based on exceptions thrown during method execution. If a system exception is thrown, the transaction will be rolled back when the method ends, if not, the transaction will be committed. Also, the method may call the `getRollbackOnly()` and `setRollbackOnly()` methods on the Container to obtain and set the rollback flag.

However, to call the above container methods, or to manually throw a system exception breaks separation of concerns. Since the method now contains code whose concern is to handle a section of the transaction, transaction management is not cleanly separated out. In other words: to cleanly use declarative transaction management, the method may never get or set its' rollback flag. Manipulating this flag not only taints the method with the transaction management concern, but also splits this concern in two disjunct parts.

Furthermore, consider what should be done in case of a rollback. Conceptually, handling of the rollbacks of transactions is a part of the concern of transaction management. Therefore, to have a clean separation, such error-handling code should also be defined when stating the methods' transaction requirements. Sadly, in EJB this is not the case. When a transaction is rolled back due to a system exception, the method will throw a `TransactionRolledbackException` to the caller of the method, and when a transaction is rolled back due to the use of `setRollbackOnly()` the caller will not be informed of this in any way. So, when using the `setRollbackOnly()`, the method needs to additionally signal this to the caller by either returning an 'error' value or throwing an exception.

The above implies that handling a rollback can only be done from within the method callers' code. Conceptually, this means not only that the caller is now tainted with the error-handling part of transaction management, but also that transaction management is split up in three disjunct sections: the transaction declaration, manipulation of the rollback flag and a-posteriori handling of rollbacks. Furthermore, since the callers to the transactional method need to specify the error-handling code, this may lead to code duplication if there are multiple callers to the method.

In other words, declarative transaction management, as currently implemented in EJB, only provides a clean separation of concerns in the most trivial cases. This is when a transaction never rollbacks: no transactional system errors may occur, deadlocks may not be broken through a rollback, and the application itself may not decide to perform a rollback. In all non-trivial cases, declarative transaction management, as currently implemented in EJB, provides a worse separation of concerns than traditional transaction demarcation. This is because the transaction concern is forcefully split up in three distinct parts, in different sections of the application, whereas in traditional transaction demarcation these three parts are contained within one location: the implementation of the transactional method.

3 Toward a Comprehensive Aspect for Tx Management of EJBs

We feel it would be better to separate out transaction management of EJBs into one complete section, instead of three disjunct sections. To do this, we propose defining transaction management in one, comprehensive aspect.

There is a body of existing work on defining transaction management as an AspectJ [1] aspect, either stand-alone by Kienzle and Gerraoui [4] or as a by-product of specifying a persistence aspect by Soares et al.[7] and also by Rashid and Chitchyan [6]. However, each approach not only is unrelated to EJBs but also falls short of our proposed comprehensive aspect. Briefly put, the work of Kienzle and Gerraoui [4] is inadequate with regard to exception-handling, as stated by the authors themselves. The work of Soares et al. [7] has been deemed as too application-specific [6], and the work of Rashid and Chitchyan [6] omits the handling of rollbacks.

We have started work on defining a more comprehensive aspect, using the technique of *logic meta-programming* to write our custom aspect weaver [9]. The use of LMP for AOSD is not new: LMP has, among others, been used to define

domain-specific aspects [2], and to argue for more expressive crosscut languages [3].

An integral part of our transaction management solution is a custom-built transaction monitor. Woven code uses traditional transaction demarcation calls to our transaction monitor, instead of using the Container's transaction monitor. This is because, in future work, we want to provide extended transaction capabilities which are not available using the standard transaction monitor, as defined in the EJB standard.

3.1 Declaring transactional methods

Our weaver uses logic programming to reason about the method code, and can easily detect some of the methods' properties, which helps in crosscut definition. For example: if a method M calls getters and setters of an entity bean, it makes sense to make M transactional. Furthermore, if M only calls getters, we can mark the transaction as read-only[1]. Note that, at this time, we do not perform any recursive analysis: methods called by M are not investigated for getters and setters.

This automatic detection of transactional methods is the default behavior of our weaver: all classes within a given package are investigated, and transaction demarcation code is inserted for all methods that should be transactional. This demarcation code also includes exception handling, equal to the standard EJB behavior, i.e. the transaction is rolled-back in case of a system exception. The woven code now behaves as if all transactional methods were declared as `RequiresNew` in the EJB's deployment descriptor.

The weaver can also be used in a more conventional manner, by specifying which methods should be made transactional in a separate `transactions` aspect file. In such a file, for a given bean, the method signatures are listed and postfixed either with `new` or `none`, indicating whether a new transaction should be started or the method is not transactional. For the parameter list of method signature, the `*` wildcard may be used, indicating applicability regardless of parameter types. Also, default behavior for a bean can be set to be either `new`, `none`, or `detect`, this last signifying automatic transaction detection. An example transactions aspect is below:

```
transactions CounterBean
{
   increment(int count) new;
   get(*) none;
   default detect;
}
```

Note that we can also use the weaver to automatically generate these aspect declarations, effectively explicitizing the information implicit in the code.

3.2 Adding Exception Handlers for Rollbacks

However, as we have said above, we want tot go further than this; our transaction aspect also centralizes exception handling for transaction-related excep-

[1] This information can be used by the transaction monitor at run-time to optimize throughput

tions. Indeed, we can define exception handlers for methods, by simply appending a number of `catch` blocks, containing java code, to the transactional declaration of the method. Within this code, the transaction can be rolled back by simply calling a `txRollback()` method. If the transactional method throws an exception that is not caught by these handlers, or the handlers do not call the `txRollback()` method, the transaction will commit when the method ends.

Lastly, we add an extra feature which is not available in EJB transaction management: restarting a transaction from an exception handler. When restarting, the transaction rolls back and the method is restarted (by re-calling the method), which implies the creation of a new transaction. Transaction restart is indicated by using a `catch` block with as body the `restart` keyword.

An example aspect definition containing these kinds of exception handlers is given below:

```
transactions CounterBean
{
   increment(int count) new catch (RuntimeException ex)
         {txRollback(); ex.printStackTrace(); System.exit(1)}
      catch (RemoteException ex) restart;
   get(*) none;
   default detect catch(Exception ex)
         { txRollback(); ex.printStackTrace(); System.exit(1)};
}
```

This effectively centralizes transaction declaration and handling of rollbacks due to exceptions in one location, and avoids unnecessary code duplication (for exception handling) in callers of the Beans' methods. One last element that is missing in this centralized transaction aspect, is the use of `setRollbackOnly()` to set the rollback flag. This occurs when, somewhere within the execution of the method, some heuristic determines that the transaction should be rolled back. At this time, we do not yet support this use of heuristics, we consider it as future work.

While it may seem unnecessary to have rollback handlers in the aspect when not being able to declare heuristics that will trigger a rollback, this is not the case. Significant causes for a rollback can already be found in this setup: transactional system errors may occur and deadlocks may be broken though a rollback. For these cases, our system is arguably better than the EJB implementation: instead of transaction declaration and rollback handling separated in two places, these are now integrated into one.

4 Conclusion and Future Work

This paper started with a discussion of Container-Based Transaction Management, as currently implemented by enterprise JavaBeans. Bean methods declare their transaction properties in the deployment descriptor, and the Bean Container automatically starts and ends transactions if required. Transactions are rolled back if the method ends in a system exception, or if the method set the rollback flag of the transactions.

Sadly, handling of exceptions and rollbacks produces a bad separation of concerns: code concerned with transaction management is now not localized

in one place but in three places. First we have declaration of the transaction properties in the deployment descriptor, second we have determining of rollbacks in the bean itself and third handling of rollbacks in the beans' callers.

After discussing transaction management in EJB's, we proposed a better solution with regard to separation of concerns. We have shown our current work, which can detect the need for transactional methods, and integrates transaction handling and the handling of rollbacks in one aspect. Heuristically determining a rollback within the method code is not yet supported. However, the current incarnation already has its merits because it is useful for handling exceptions due to e.g. network outages, forced rollbacks due to deadlocks, and so on.

Also, as a result of explicitly treating rollback handling, we considered default strategies for handling a rollback, and have already implemented a transaction restart. This leads us into future work: we are further exploring handling of rollbacks in the context of advanced transaction models, such as the use of compensating transactions.

Lastly, as future work, we will investigate how we can integrate the 'missing' concern part: rollback heuristics, into the transaction aspect.

5 Acknowledgments

Thanks to Thomas Cleenewerck and Jessie Dedecker for proof-reading and Theo DHondt for supporting this research.

References

[1] The AspectJ project. http://eclipse.org/aspectj.

[2] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 110–127. Springer Verlag, 2002.

[3] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[4] J. Kienzle and R. Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag.

[5] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, third edition, 2001.

[6] A. Rashid and R. Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[7] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM.

[8] Sun Microsystems. Enterprise JavaBeans specification. http://java.sun.com/products/ejb/docs.html.

[9] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.