

# Refactoring Transactions: A Case Study

Irum Godil  
Department of Computer Science,  
University of Toronto  
Toronto, Ontario  
CANADA M5S 2E4  
416-946-8669  
irum@cs.utoronto.ca

## ABSTRACT

Various attempts have been made in studying whether or not it is possible to refactor transactions and persistence as aspects. The general and the more common opinion on refactoring transactions has been negative. In this paper we aim to present our findings on this topic of research. Our experiment involved refactoring of transactions in the HSQL Database application using AspectJ, a general purpose aspect-oriented extension to Java. Our results have been pretty optimistic and we have been successful in refactoring out code for the Atomicity, Isolation and Durability concerns of Transactions. Although we do face some challenges in making transaction code fully transparent to the database system, we argue that this is a deficiency of certain features in AspectJ, which any code base with such design would face; and is not specific to Transaction refactoring. We do not find aspectizing transactions different than any other concern aspectization.

## Keywords

Transactions, Aspect Oriented Programming, Refactoring

## 1. INTRODUCTION

Aspect Oriented programming aims at modularizing crosscutting concerns. Logging, tracing, synchronization etc. are usually cited as common concerns for which Aspect Oriented Programming (AOP) has been useful in greatly modularizing code bases. With the rapid growth and adaption of the AOP technology it is imperative to apply it to crosscutting concerns of various categories in order to understand both its potential and its limitations.

Transaction refactoring is one such area of research which has gained a lot of popularity among AOP critiques and users. The experimentation done in refactoring transactions as presented by Kienzle and Guerraoui [4] on the Optima Framework has been extremely negative. Driver and Clarke [5] who aimed at studying whether evolution can be enhanced using AspectJ, also arrived at pessimistic results for transaction refactoring. Some other researchers Rashid and Chitchyan [7] and Soares et. al [6] specifically refactored persistence as aspects and were optimistic in their conclusions. For both Rashid [7] and Soares [6] transaction control is part of persistence implementation and hence their conclusions negate the results of [4] and [5].

The goal of our study was to refactor a database application system to explore whether or not transactions can or cannot be refactored; and to study which of the two opinions should be given greater weight. Our refactoring standards were not based on specific code metrics such as evolution; but rather to apply our prior experience of refactoring to the Transactional crosscutting

concerns and observe if this refactoring was any different than other concern refactorings. We also aimed to measure various code metrics before and after refactoring to study how much complexity, size, code-cohesion and coupling changed after refactoring.

To this end we present our analysis of our case study in this paper below. We chose HSQL [1] as our database application to refactor transactions. We first discovered all points of transaction support and crosscutting areas in the application and then applied AspectJ refactoring on the system. Based on the refactoring of the code base, our general conclusion was that refactoring transaction is no different than refactoring other concerns; and that it is very much possible to separate transaction interfaces and semantics from the application. Though, as presented in Section 6 below, we observe certain limitations of AspectJ which prevent us from fully making transactions transparent from the core code base; we cite this as a general AspectJ limitation and nothing specific to transactional support. Thus, our overall conclusion is positive for both transactional refactoring and AspectJ's potential in achieving modularization of crosscutting concerns.

In the following, section 2 provides an overview of the HSQL application used as the basis for this discussion. Section 3 presents transaction support features in the HSQL system. Section 4 presents crosscutting of transactional features and implementation of aspects for transactions. Section 5 presents metrics and evaluations of our refactoring results. Section 6 presents our observations of transaction refactoring in the light of earlier research papers and our own observations. Section 7 concludes the paper and outlines future work areas.

## 2. HSQL OVERVIEW

We used the HSQL Database system for refactoring transactions [1]. HSQL is a SQL relational database engine written 100% in Java. It offers a small, fast database engine which offers both in-memory and disk-based tables. HSQL runs in both stand-alone and client-server mode.

HSQL supports in-memory, cached and text tables. It provides full transactional support by providing atomicity via committing and rollbacking transactions, consistency, isolation at the READ\_UNCOMMITTED level and durability of data by storing it on disk. HSQL provides 95% JDBC interface support with full metadata and batch statement functionality.

In our study we were mainly interested in the transactional support features of the application. In addition to main classes that provide transactional support, it was also necessary to understand the various connection modes i.e. Server/Stand-alone, the different table types, existence of various files for a database,

support for savepoints and nested transactions and classification of features that are possible only in persistent mode vs. present in both persistent and transient mode. In the following sections we present some details of these features under the various aspect categories.

### 3. TRANSACTION SUPPORT

In this section we describe the support for the Transactional ACID properties in the HSQL System.

#### 3.1 Atomicity

Atomicity is supported in the HSQL System by supporting the following features:

##### 3.1.1 Commit-Rollback

HSQL provides support for “committing” and “rollbacking” transactions. It is possible to treat each statement as a single transaction in which case the statement is committed right after executing. This status is known as “autocommit”. Also, it is possible to group statements together as a transaction and then explicitly invoke “commit” or “rollback” in which case the whole group is committed or rolled back.

In addition to API support for “commit” and “rollback” in the system, atomicity is supported by keeping a “TransactionList” to store the group of statements included in the Transaction. There is support for whether the session is “isAutoCommit” by default or not and configuring the auto-commit status.

##### 3.1.2 Savepoints

A savepoint is a point in a transaction that serves as the target for partial database rollback. A savepoint marks a particular point in the execution of a transaction [8, 685].

##### 3.1.3 Nested Transactions

HSQL supports nested transactions. Nested transactions allow the transaction designer to design a complex transaction from top-down. A transaction is decomposed into subtransactions in a functionality appropriate way. Subtransactions control their commit/abort decisions; however the overall rollback/commit depends on the parent transaction. If the parent rollsback, even if a subtransaction has committed, the entire transaction rollsback [8, 691].

##### 3.1.4 Error Message Support

Support for error messages relating to Atomicity features is present in the system.

#### 3.2 Consistency

From [8, 671] we have: “Keep in mind that producing consistent transactions is the sole responsibility of the application programmer. The remainder of the transaction processing system takes consistency as a given and provides atomicity, isolation and durability – the properties needed to ensure that concurrent execution of consistent transactions preserves the relationship between the state of the database and the state of the enterprise in spite of failures”.

This claim was verified by the developers of the HSQL Database System. Also, our own manual inspection of the code base did not reveal any specific places for consistency support. Hence we did

not find any consistency support in the system and did not refactor any code for this transactional property.

#### 3.3 Isolation

HSQL supports Isolation at the “READ\_UNCOMMITTED” level. This means that at this level dirty reads i.e. reading uncommitted transactions, is possible. Isolation is supported by the following features:

##### 3.3.1 Locking Row in Use

In HSQL a row that is inserted or updated by an uncommitted transaction is locked to be modified by another transaction. Upon commit, the row can be accessed by other transactions. HSQL achieves this by keeping a list of rows that are in use by a specific transaction and when a row is accessed by another transaction it is first verified that the row is not already in use.

HSQL also only allows multiple connections from within the same executing instance to access the same database. Different executing instances cannot access the same database. In this way the isolation support provided by HSQL is limited to only one executing instance of the application.

##### 3.3.2 Setting and Getting Isolation Level

The HSQL System provides support for getting and setting the isolation level.

#### 3.4 Durability

In HSQL Durability is supported in the following ways:

##### 3.4.1 Database Modes

HSQL allows connection to the Database in various modes including creating database in memory, persistent database and running the database as a server. We refactor the mode relating to the creation of persistent database.

##### 3.4.2 Logging and Scripting

HSQL supports durability by logging all the transactions to a log file when the database is in use. Once the database is shutdown all the operations from the log file are written to a “.script” file. Thus durability is supported by logging and scripting features.

##### 3.4.3 Persistent Table Types

HSQL supports various table types. Out of these table types the “Text” and the “Cache” type are only supported for persistent databases. Thus, we remove their support from the core since once durability is refactored these will no longer be supported.

##### 3.4.4 Locking Support

In the case of persistent database case, the database is locked when one application is accessing the database. Since this feature is not supported for in-memory databases, this feature is removed.

##### 3.4.5 Error Message Support

Support for error messages relating to Durability features is present in the system

### 4. TRANSACTION ASPECTS

In this section we describe the crosscutting nature of the transaction concerns and the aspect implementation details.

## 4.1 Crosscutting Areas of Core

All three features of transaction support i.e. Atomicity, Isolation and Durability crosscut through various classes and packages of the core. The crosscutting of these features is shown in figure 1.

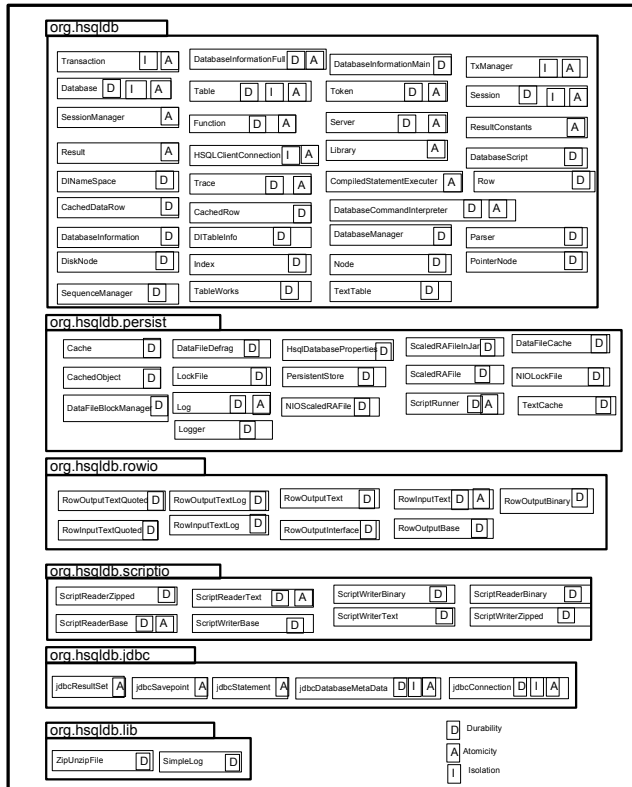


Figure 1. Crosscutting of Concerns in HSQL

## 4.2 Aspect Implementation

For each of the three concerns we added a <concern>.weave package where we refactored all the code relating purely to that concern. Thus we have three packages namely, aspects.durability.weave, aspects.atomicity.weave and aspects.isolation.weave. These packages contain one aspect per class that exhibits crosscutting.

In addition certain classes were completely related to the concern; if the class did not access any protected variables those classes were completely refactored into the “weave” package. If the class did access non-public fields then the entire class body was copied into a new aspect for that class in the package.

In cases where the same class exhibited crosscutting for more than one concern there were some common pointcuts that were accessed by more than one aspect. In order to avoid code-duplication, we introduced another package called “aspects” which implemented abstract-super aspects for each such class. The aspects include common pointcuts definitions, and are extended by the child-aspects. An example of the DatabaseSuperAspect is shown in figure 2.

Also, in the case of Atomicity and Isolation, both the aspects required declaration of the two classes Transaction and TxManager and each implemented its own aspect method and field introductions. Also, a piece of Database class code for

initialization of TxManager is common to both the concerns. Thus we added a new package called “aspects.atomicityisolation-Common.weave” where we have placed the common code.

```
public abstract aspect DatabaseSuperAspect
extends CommonCallSuper {

protected pointcut tar(Database d); target(d)
protected pointcut thisDB(Database d): this(d)

protected pointcut logOps(): execution(void logOperations())

protected pointcut inreopen(): withincode(void reopen())
protected pointcut inClose(int closeMode): args(closeMode) && withincode(void close(int))
protected pointcut inClose_withoutargs(): withincode(void close(int))
protected pointcut inDropIndex(Session session): args(session, ..) &&
withincode(void dropIndex(Session, String, String, boolean))
protected pointcut inDropIndex_noArgs(): withincode(void dropIndex(..))

protected pointcut inConstr(): withincode(Database.new(String,String,String,HsqlProperties))
}
```

Figure 2. DatabaseSuper Aspect - Example of a Super Aspect

## 4.3 Aspect Convolution

In the refactoring process we made two observations. Firstly, we noted that when two or more transaction concerns crosscut the same method, and we have refactored the crosscutting concern as an around-advice around the entire method in both separate concerns; then we need to provide a configuration with an around-advice around the method which covers all the crosscutting concerns. In such a case, we have individual aspects with around advices for the method and an aspect for around advice in case both the concerns are present. An example of such a case is shown in figure 3 for the Function class. We show the original Function.getValue() function, with underlined lines for crosscutting code. Then we present the individual Durability and Atomicity aspects and finally the combined aspect which also declares “precedence” for itself over the other two aspects.

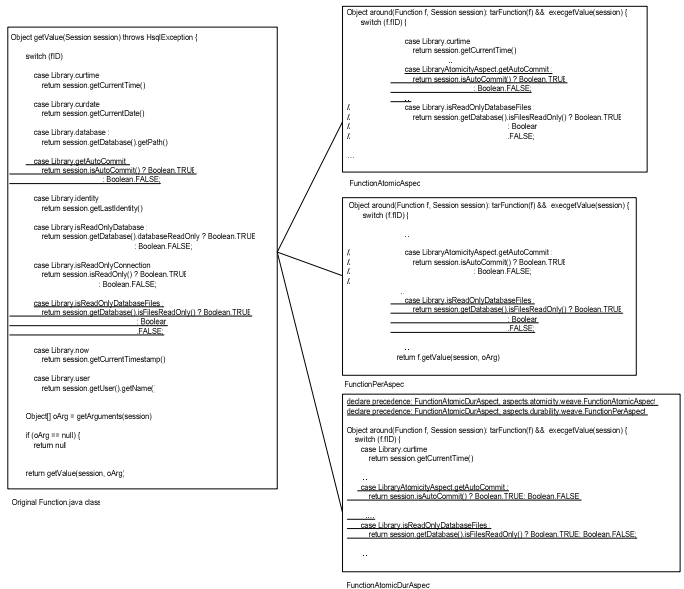


Figure 3. Atomicity/Durability Common Method Crosscutting

In the case of HSQL we found 3 such cases. The summary of these cases is presented in Table 1.

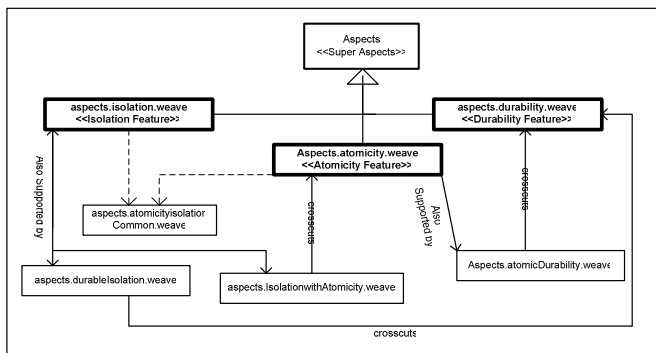
Another point of convolution is that certain code statements of one concern crosscut into methods or components of another

concern. For instance, the Log class which is applicable only for the Durability concern has a piece of code in a method relevant to Atomicity concern. Thus, we need to be able to provide configurations that weave additional code into aspects. The summary of these cases is presented in Table 1.

**Table 1. Aspect Convolution Description**

Package	Convolution Description
atomicDurability	This package has 10 aspects. 1. In three of the aspects we provide a combinational around advice for functions in classes jdbcDatabaseMetaData, Function and DatabaseCommandInterpreter, which have around advices for methods that support both atomicity and durability 2. There are components and methods related to durability which have Atomicity specific code. This crosscutting code is put into the remaining seven aspects
durableisolation	This package has one aspect namely, TableAspect. It implements isolation specific code as an advice that is weaved into the Table class method specific to Durability
IsolationwithAtomicity	This package contains 2 aspects, which implements advices that crosscut into Atomicity specific code to support Isolation

In figure 4 below we summarize the relationship between the various aspect packages used.



**Figure 4. Aspects Implementation Relationship**

#### 4.4 Aspect Anatomy

In this section we present an anatomy for the aspect packages. We summarize on the number of kind of pointcuts that are used in all the packages. Specifically, we used execution, within, call, set field, target and this pointcuts. We also include number of field and method introductions. We summarize the numbers in Table 2.

**Table 2. Metrics for Pointcut Types**

Package	intro	exec	within	call	set	target	this
aspects.atomicity.weave	49	35	8	16	1	2	4
aspects.durability.weave	89	77	58	40	5	7	6
aspects.isolation.weave	13	7	0	0	0	1	1
aspects	0	4	30	5		8	6
aspects.atomicDurability.weave	0	8	9	7	1	2	2
aspects.atomicityisolation.Common.weave	1	0	0	1	0	0	0
aspects.durableisolation.weave	0	1	0	0	0	0	0
aspects.IsolationwithAtomicity.weave	0	1	2	1	0	0	1
<b>Total</b>	152	133	107	70	7	20	20

We also calculate the different kinds of advices used in implementing the aspects. We summarize the number of around, before, after, after returning and after throwing advices in each package in Table 3.

**Table 3. Metrics for Advice Types**

Package	Around	Before	After	After return	After throw
aspects.atomicity.weave	33	9	10	1	0
aspects.durability.weave	117	18	24	4	0
aspects.isolation.weave	6	1	1	0	0
aspects	0	0	0	0	0
aspects.atomicDurability.weave	7	3	9	0	0
aspects.atomicityisolation.Common.weave	0	1	0	0	0
aspects.durableisolation.weave	1	0	0	0	0
aspects.IsolationwithAtomicity.weave	1	0	1	0	0
<b>Total</b>	165	32	54	5	0

## 5. EVALUATION METRICS

For the purposes of evaluation we have gathered various metrics on the original and refactored code. Our suite of metrics includes separation of concerns, coupling, cohesion and size metrics [2]. The following is a summary of the results gathered

### 5.1 SoC Metrics

We have used the following SOC Metrics:

#### 5.1.1 Concern Diffusion over Components (CDC):

From [2], CDC counts the number of primary components whose main purpose is to contribute to the implementation of a concern. It also counts the number of components that access the primary components by using them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods.

This metric was calculated manually for the three transactional properties: Atomicity, Isolation and Durability. The data is presented in figure 5. We see that the number of components have increased in the Aspect-Oriented version for each of the three concerns. This is an expected result since for each concern that crosscuts a class we have added an aspect for implementing the crosscutting code and in addition there are super abstract aspects defining common pointcuts used by more than one aspect.

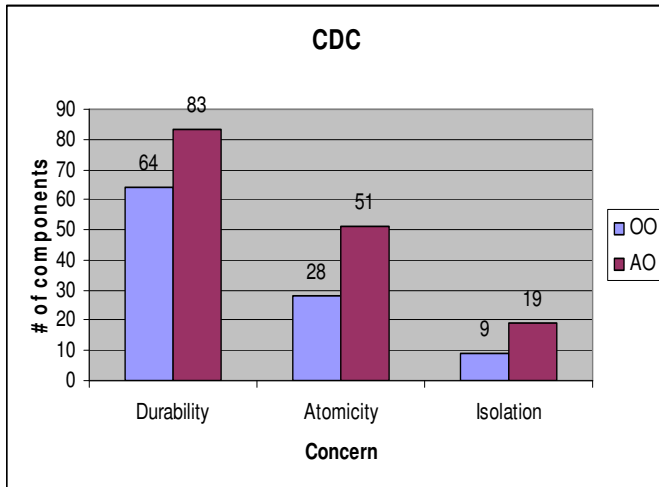


Figure 5. CDC Metric Chart

### 5.1.2 Concern Diffusion over Operations (CDO):

From [2], CDO counts the number of primary operations whose main purpose is to contribute to the implementation of a concern. In addition, it counts the number of methods and advices that access any primary component by calling their methods or using them in formal parameters, return types, throws declarations and local variables.

In the OO version, we simply counted the number of operations purely supporting the concern and the operations where the concern was crosscutting. In the AO version, we counted the aspect declarations, advices and helper methods that were introduced for pointcut definition purposes. The results are presented in figure 6. We also present a detailed breakdown of OO and AO operation type count to measure where the difference is occurring. We see in the aspect oriented case there are 56 helper methods for the Durability concern, 22 helpers for Atomicity and 4 helpers for the Isolation concern. The number of methods purely supporting the concern in the OO case are moved as aspect declarations and hence the two numbers in both the cases are equivalent. We see a discrepancy between the number of methods that are crosscutted vs. the number of advices introduced in the Durability concern. This is because in some cases more than one location of the same method had crosscutting code which result in greater than one advice for one method. Secondly, in some cases we needed some auxiliary advices to capture method parameters or local variables which were used in the original crosscutting code.

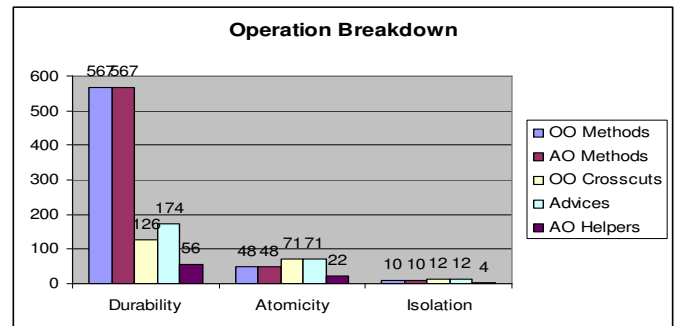
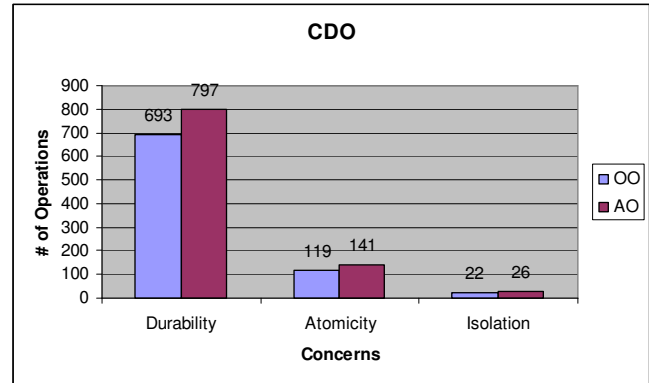


Figure 6. CDO and Operation Types Metric Charts

## 5.2 Coupling and Cohesion

The coupling and lack of cohesion metrics for the entire application were measured automatically [3], before refactoring and for the refactored core. Coupling was additionally manually calculated for aspects using the definition in [2]. Also, cohesion and coupling for refactored classes was measured automatically. The result is presented in figure 7. We see that both coupling and cohesion have reduced for the refactored versions. This is a positive result since coupling has gone down and cohesion is a measure of lack of cohesion, which implies that cohesion has increased.

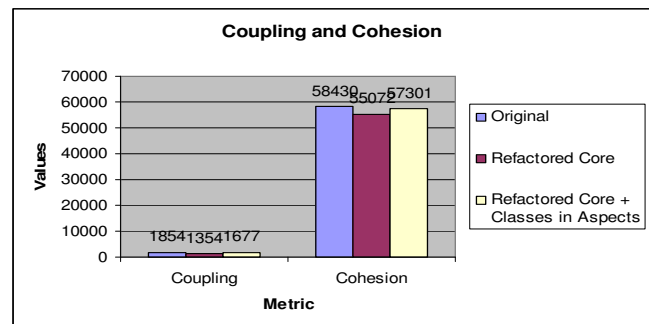


Figure 7. Coupling and Cohesion Metric Chart

In figure 8 below we present the result of the aspect coupling of the packages used for all the concerns. This metric was calculated manually.

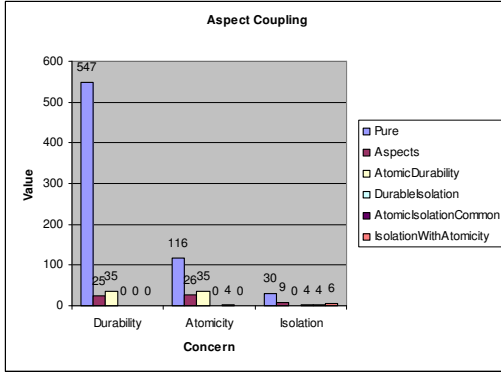


Figure 8. Aspect Coupling Manually Calculated by Packages

### 5.3 Size Metrics

The following size metrics were measured for the application. The charts are shown in figure 9.

#### 5.3.1 Vocabulary Size (VS):

VS counts the number of system components i.e. the number of classes and aspects in the system. This number is lower in refactored core but greater in the total refactored system than the original system. This is because certain components have been completely removed from the core, and in the refactored system those components plus new aspects have been added.

#### 5.3.2 Lines of Code (LOC):

This counts the lines of code.

#### 5.3.3 Weighted Operations per Component (WOC):

The metric measures the complexity of a component in terms of its operations.

We measured the LOC and WOC metrics for the original code, refactored core and only classes in the refactored version.

The number for both LOC and WOC is lower in both the refactored core and the core plus refactored classes. This is because a lot of crosscutting code has been moved into aspects which are not counted here.

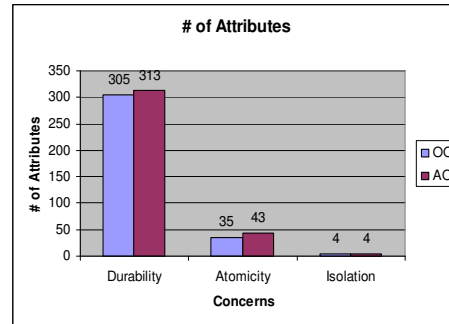
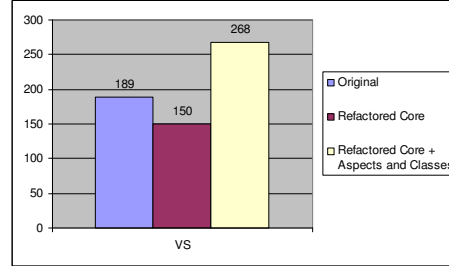
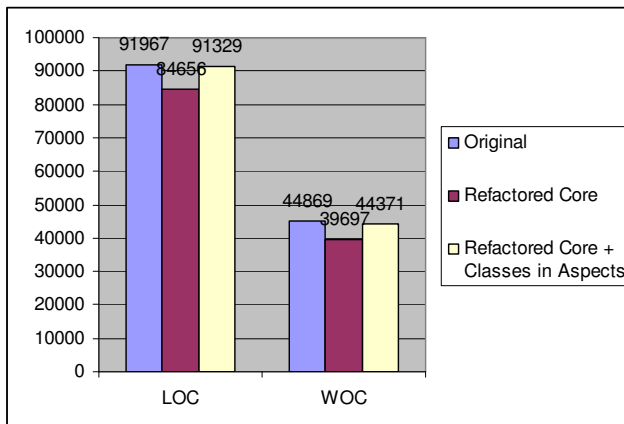


Figure 9. Size Metrics Charts

#### 5.3.4 Number of Attributes (NOA):

Here we measured the number of attributes relating to the specific transactional property before and after the refactoring. In the aspect oriented version, we also measured the helper fields that were added for refactoring purposes. The number of attributes for the Durability and Atomicity aspects is greater for the aspect-oriented version due to introduction of helper fields in various aspects to store function parameters or local variables of crosscutting code.

### 5.4 Metric Conclusions

We have presented various metrics above. Although in most cases i.e. CDC, CDO, size metrics the numbers are higher for the AO case as compared to the OO case, these numbers in themselves do not give any evidence whether transaction refactoring has been possible or not. The reason is that no matter what concern we were to refactor we would have to introduce new aspects, super aspects, helper methods and local variables. So the presence of these factors do not signal an increased complexity. What one could perhaps conclude is that AspectJ refactoring can lead to increased number of components, attributes and methods; but on the other hand when weighted against modularity and configurations that are achieved using AspectJ; this is not necessarily a drawback.

## 6. TRANSACTION SEPARATION NOTES

As mentioned in the introduction of the paper, the goal of this project was to study whether or not it is possible to refactor Transaction support using Aspect oriented programming or not. To this end various studies have been done. Kienzle and Guerraoui [4] studied transaction refactoring based on the Optima Framework. Their conclusion is that aspectizing Transaction semantics is impossible, aspectizing transaction interface leads to confusing code and aspectizing transaction mechanism leads to only physical separation and not semantic separation. Driver and Clarke [5] aim at studying if evolution of a system can be

enhanced using AspectJ. They conclude that in the case of Transactions the effect on evolvability is not positive. Their AspectJ implementation of transactions is not very simple and extra hooks in the core code had to be added for refactoring support. However, they conclude that given sufficient knowledge of the transaction concern, the ease with which it can be modified is enhanced.

Contrary to the above negative transaction refactoring results, Soares et. all [6], and Rashid and Chitchyan [7] conclude that persistence can be aspectized and not only that but a reusable framework for the same can be created.

As presented above we have been successful in refactoring code relating to transaction support in HSQL into aspects; however, we explore here whether or not this refactoring has really achieved its goal i.e. that of separating transaction support from the application or not. During our refactoring we made the following two important observations:

## 6.1 Presence of Transaction Related Signatures in Core

AspectJ allows us to refactor only the code base for which the source code is available to us; and not from imported libraries. One disadvantage of this feature is that various transactional interfaces are defined in libraries which are not part of the HSQL code base. For instance the HSQL class `jdbcConnection` implements the `java.sql.Connection` class; which defines various transactional methods e.g. `commit`, `rollback` etc. These methods from the `jdbcConnection` class could not be moved as introductions into aspects, as the class would no longer be implementing its interface. Thus, the empty method body had to be left in the core and an “around” advice had to be created to support the interface. The list of all such instances where certain traces had to be left in the core are presented in Table 5.

The presence of these signatures count against the complete removal of transaction support from the HSQL core. Another impact for existence of these signatures is that their existence allows tests/applications to be written on top that let someone define `con.commit()` and `con.rollback()` functions. One can define applications which use these APIs, but since the methods are empty they can mislead the developer to believe that actual commit and rollback is happening when in reality nothing occurs.

**Table 5: Methods/Fields that could not be removed from Core**

Transactional Feature	Class	Number of Methods/Fields
Isolation	<code>jdbcConnection</code>	2 methods
Isolation	<code>jdbcMetaDatabase</code>	2 methods
Atomicity	<code>jdbcConnection</code>	10 methods
Atomicity	<code>jdbcMetaDatabase</code>	8 methods
Atomicity	<code>jdbcStatement</code>	1 methods
Atomicity	<code>jdbcResultSet</code>	2 constants

We notice there are a total of 25 transactional traces left in the system core. One work around to remove this would be to remove the “implements <interface>” part from the core and “declare

parents” in the aspects. The problem with this approach is that other parts of HSQL application rely on these classes to be implementing the respective interfaces for polymorphism support. Thus, this approach is not feasible.

## 6.2 Removal of Non-Transactional Functionality

One other observation that was made is that in order to remove durability support we removed support for scripting commands to the file; since durability in HSQL is obtained via scripting commands to the `.script` file. Though this feature was successfully refactored, removal of this feature from the core prevents scripts of files to be created for transient databases. In HSQL it is possible to create script files for memory databases, and in the quest for separate persistent database support from transient databases, we have deprived the transient databases of an essential feature in the core case.

## 6.3 Observations

Other than the two above observations, nothing peculiar in terms of increased complexity, adding hooks etc. was observed. What we can hence conclude of the first observation is that this is again not something specific to transactional refactoring; but would be generally the case whenever an interface is implemented in the core and the code for the interface is not available for refactoring. The second observation does create a deficiency for the in-memory databases; but it could perhaps be overcome by moving all the scripting related functionality to a separate aspect package and including it with the core base. Such a refactoring has not been done, but it could be studied whether this is possible or not.

One may argue that introduction of helper methods is adding hooks in the core code for refactoring support as mentioned in [5]. However, once again the helper methods are not specific to transaction refactoring, since in any case when suitable pointcuts are not available one has to add helper methods.

Thus, given the fact that we have refactored all the transactional properties of Atomicity, Isolation and Durability; and given the fact that the above two concerns are not really transactional specific concerns, we conclude that unlike the Optima study [4], transactional refactoring is possible; and any issues encountered are AspectJ specific and not transactional specific concerns.

## 7. CONCLUSION

This paper has presented our experience in aspectizing transactions in the HSQL Database application. Our general aim was to explore whether such a refactoring is fully possible or not and to negate or affirm the views of other researchers of this area. Our conclusion, despite the high size metrics and the fact that not all interfaces could be fully removed, is still positive. We are able to configure a HSQL version with and without the transactional concerns and this has led to increased overall modularization of the code base.

Our future work will focus on verifying whether the scripting feature issue presented in Section 6.2 can be configured out of the transactional aspects so that it can be weaved into in-memory database as well as persistent ones separately. Additionally, it would be an interesting study to build a database application from scratch with refactoring of transactions in mind and noting down

the differences between the two scenarios. It will also be worthwhile to refactor transactions from other database applications to further affirm our results or to notice any differences attached with specific database applications for transaction refactorings.

## 8. REFERENCES

- [1] SourceForge.net. HsqlDb. At <http://hsqldb.sourceforge.net/>
- [2] Cláudio Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena & Arndt von Staa  
*On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework* XVII Brazilian Symposium on Software Engineering, Manaus, Brazil, October 2003, 19-34.
- [3] Borland TogetherSoft website. URL:  
<http://www.togethersoft.com>
- [4] J. Kienzle and R. Guerraoui, *AOP Does it Make Sense? The Case of Concurrency and Failures*, ECOOP, 2002, SpringerVerlag, LNCS 2374, 37-61
- [5] C. Driver and Siobhan Clarke, *Distributed Systems Development: Can we Enhance Evolution by using AspectJ*, 2004
- [6] S. Soares, E. Laureano, and P. Borba, *Implementing distribution and persistence aspects with AspectJ*, OOPSLA, 2002, ACM Press, 174-190
- [7] A. Rashid and R. Chitchyan, *Persistence as an Aspect*, 2nd International Conference on Aspect-Oriented Software Development. ACM. 120-129
- [8] P. Lewis, A. Bernstein, M. Kifer. *Databases and transaction processing : an application-oriented approach*. Addison-Wesley, 2002.