Relational Database Support for Aspect-Oriented Programming

Awais Rashid, Neil Loughran

Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
awais@comp.lancs.ac.uk
loughran@comp.lancs.ac.uk

Abstract. Code repositories play a central role in the reuse and mining of existing assets when engineering large, complex software systems. It is, therefore, essential that database support be extended to new programming paradigms as and when they emerge. This paper proposes an approach to support the storage, reuse and mining of *aspects* - constructs used in Aspect-Oriented Programming (AOP) to separate crosscutting concerns - in AspectJ (an aspect language for Java) using a relational database. The approach is based on mapping an aspect's anatomy to the relational model hence allowing fine-grained queries to be composed. This results in greater flexibility during search and retrieval in contrast with most existing code repositories which store the code as BLObs complemented by meta-data about the code.

1 Introduction

The mining and reuse of existing assets is a core activity in the engineering of large, complex software systems and a driving factor in the reduction of their costs and time to market. Code repositories are at the nucleus of this activity making it possible to reuse existing program modules, adapt them to new problem contexts and, mine and extract parts of different modules reusable in the development of a new one. It is, therefore, essential that support for storage and retrieval of program modules in code repositories remains in step with the emergence and adoption of new programming paradigms. This trend can already be observed in several proposals on the need for component repositories to support component-based development e.g. [19] [26].

Aspect-Oriented Programming (AOP) [9] is a new programming paradigm which aims at separating concerns which cut across parts of a system. Examples of such crosscutting concerns include code handling synchronisation, debugging, security, resource sharing, distribution and memory management. It is not possible to encapsulate such code within a single program module (e.g. a class) using conventional decomposition mechanisms (which result in tangled representations). With AOP crosscutting code is modularised using special constructs known as *aspects*. This promotes localisation of changes hence reducing development, maintenance and evolution costs.

Previously we have developed reification-based models for storage and retrieval of aspects in object databases [16] and explored composition of aspects with objects in

such an environment [18]. The focus of this paper is on the use of relational database systems to support storage and retrieval, and hence mining and reuse, of aspects. Our choice is driven by the major share of the database market claimed by relational systems and, therefore, significant amount of existing investment in such systems on part of software development organisations. Furthermore, since aspects capture system-wide properties their storage as an asset makes it possible to query program modules in an asset repository based on their *global* properties. Our proposed approach is based on mapping the aspect structure to relational tables which are used to store the aspects. This is in contrast with the more generally used approach to building code repositories where code is stored in BLObs (Binary Large Objects) or CLObs (Character Large Objects) together with meta-data which is used to retrieve or mine the code. The reasons behind our choice are as follows:

- The conventional approach results in loss of actual code representation hence constraining queries and, consequently the mining process to the meta-data. If the information required by the mining process cannot be derived from the meta-data it is not possible to run arbitrary queries against the binary code. The queries become too complex in case of CLObs and are hard to formulate due to the lack of a formal structure e.g. a relational schema. Explicitly capturing the code structure of an aspect makes it possible to run fine-grained queries offering more flexible aspect mining capabilities. Fine-grained queries on the code representation are particularly useful to support extracting parts of different existing modules (aspects in our case) to be reused in the development of a new one.
- Specification of meta-data for aspects is a difficult task due to the their crosscutting nature.
- There are a number of AOP techniques available ranging from linguistic mechanisms [1] [2] to filter-based approaches [3] through to traversal-based [14] and multi-dimensional approaches [12] [17]. Each approach supports a conventional base programming language e.g. AspectJ [1] is an aspect language for Java. If aspects are stored as BLObs they are limited to use within the particular AOP approach they are implemented in. However, if their structure is captured it is possible to use mapping algorithms (e.g. [4] [6]) to reuse or adapt them for a project employing a different AOP approach. This, in turn, provides an opportunity to evolve the relational schema into a *common persistent aspect representation* [16] complemented by algorithms for mapping the persistent representation to aspect structures in different AOP techniques.

In this paper we focus on mapping AspectJ aspects to relational tables and execution of fine-grained queries on the stored aspects. AspectJ has been chosen because it has the largest user base (over 2500 downloads per month) among the available AOP techniques. Note that while some algorithms for mapping AspectJ aspects to some other techniques exist (e.g. [4] [6]) we do not discuss such mapping in this paper nor do we discuss the derivation of a common persistent aspect representation. These will form the subject of a future paper.

The next section provides an overview of AOP in AspectJ. The discussion, based on AspectJ1.0, focuses on the anatomy of aspects in the language. Section 3 describes the mapping of this anatomy to a relational schema and the rationale behind various design choices. Section 4 shows a fine-grained SQL query using a GUI tool

developed to support the aspect repository. Section 5 discusses some related work while section 6 concludes the paper and identifies directions for future work.

2 Aspect-Oriented Programming

As mentioned earlier AOP aims at modularising concerns which are otherwise spread across the system. Fig. 1(a) shows how code for synchronisation and debugging is spread across multiple classes in an OO language such as Java or C++. Inheritance might be perceived as a solution to such code tangling because it makes it possible to encapsulate a crosscutting concern (e.g. synchronisation) in a superclass. An inheritance-based solution, however, simply shifts the code tangling problem to a different dimension: extensive invocation and overriding of superclass methods in subclasses [21]. This not only increases the maintenance overhead but also gives rise to the *fragile base class problem* [15]. Similarly, while patterns [10] can help to deal with such crosscutting code by providing guidelines for a good structure, they are not available or suitable for all cases and mostly provide only partial solutions to the code tangling problem [20].
Fig. 1(b) illustrates how the scenario in fig. 1 (a) is addressed using AOP. The classes are designed and coded separately from the code that crosscuts them. Aspects encapsulate the crosscutting code. The links between aspects and classes are maintained by means of special reference points known as *join points*. An *aspect weaver* is used to compose the aspects and classes with respect to the join points. This composition may be carried out statically at compile time or dynamically at run time [13].

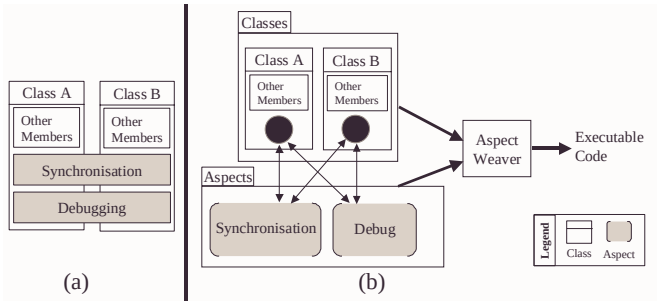


Fig. 1 (a) Crosscutting concerns in an OO language (b) Separation of crosscutting concerns using AOP

There have been several approaches to AOP developed for a variety of languages from a range of paradigms (OO, functional, procedural). They all share the common goal of providing an improved separation of crosscutting concerns. One of the leading AOP approaches is AspectJ [1] developed at Xerox PARC. The environment offers an aspect language to formulate the aspect code separately from Java class code, a weaver and additional development support. All of the examples and data structures contained in this paper relate to AspectJ 1.0.

2.1 Anatomy of an Aspect in AspectJ

Fig. 2 shows the anatomy of an aspect in AspectJ. Join points are nodes in a simple object call graph at run-time i.e. points at which an object receives a method call or has its fields referenced. Join points in AspectJ, therefore, include (among others) method/constructor calls and executions, field get and set, and exception handler execution.

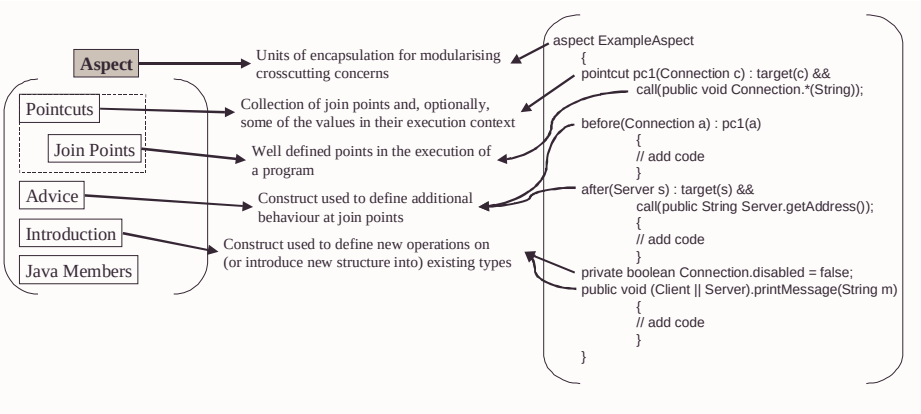


Fig. 2 Anatomy of an AspectJ aspect and an example aspect

Fig. 2 also shows an example aspect written in AspectJ. The aspect:

- creates a named pointcut called *pc1* which attaches itself to *call*s on public methods which return void and accept single String arguments in Connection objects;
- creates a *before* advice which specifies behaviour to be executed before the pointcut defined by *pc1* is executed;
- creates an *after* advice which specifies behaviour to be executed after calls to the getAddress method in Server objects have completed execution. Note that an implicit pointcut exists in this case (as opposed to an explicit named pointcut such as *pc1*);
- introduces a private boolean variable called disabled into Connection objects and sets it to false;
- introduces a public method called printMessage into Client and Server objects.

3 Aspect to Relational Mapping

From the anatomy shown in fig. 2 we can identify three main constructs in an aspect (ignoring ordinary Java members): named pointcuts, advices and introductions. As discussed in section 2 an advice may or may not be related to a named pointcut; an advice can be defined independently of a named pointcut in which case an implicit

pointcut comes into existence. This leads us to the definition of our initial high-level mapping of an aspect structure to a relational schema. This mapping is shown in fig. 3.

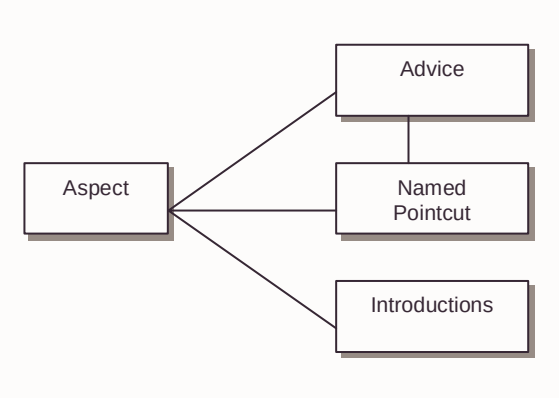


Fig. 3 High-level aspect table structure prior to further decomposition

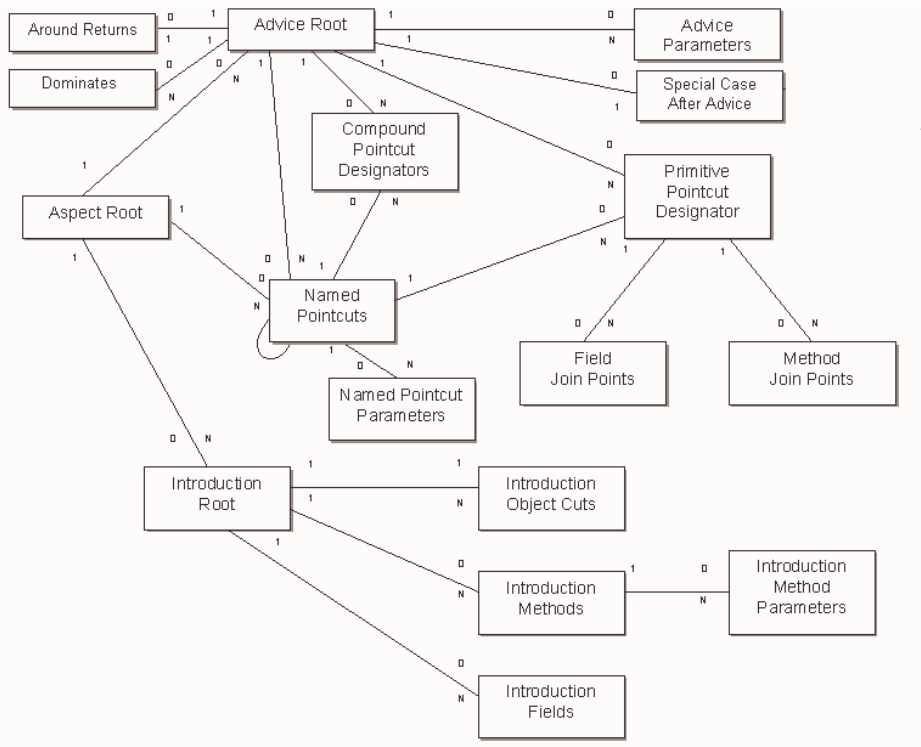


Fig. 4 Fine-grained mapping of aspect anatomy to the relational schema

The next step in the mapping is further decomposition of our initial high-level relational schema. This is essential for a number of reasons:

- We aim to achieve a decomposition compliant with Boyce-Codd normal form hence adhering to established good design practices for relational database systems.
- As discussed earlier detailed mapping of an aspect structure to a relational model makes it possible to run fine-grained queries which are particularly useful to support extracting parts of different existing aspects to be reused in the development of a new one. In addition, it makes it possible to use mapping algorithms to reuse the aspect in an AOP technique other than the one used to develop it.
- AOP is a new paradigm and, like all other AOP techniques, AspectJ is continuously evolving. Having a fine-grained mapping makes it easier to evolve the relational schema in step with changes in AspectJ. Our previous experience with evolving persistent aspect stores for AspectJ based on object databases [18] has shown that the evolving nature of AOP techniques is a very important consideration in the design.

The various tables and their relationships in our fine-grained mapping are shown in fig. 4 while the detailed structure of each table is shown in fig. 5.

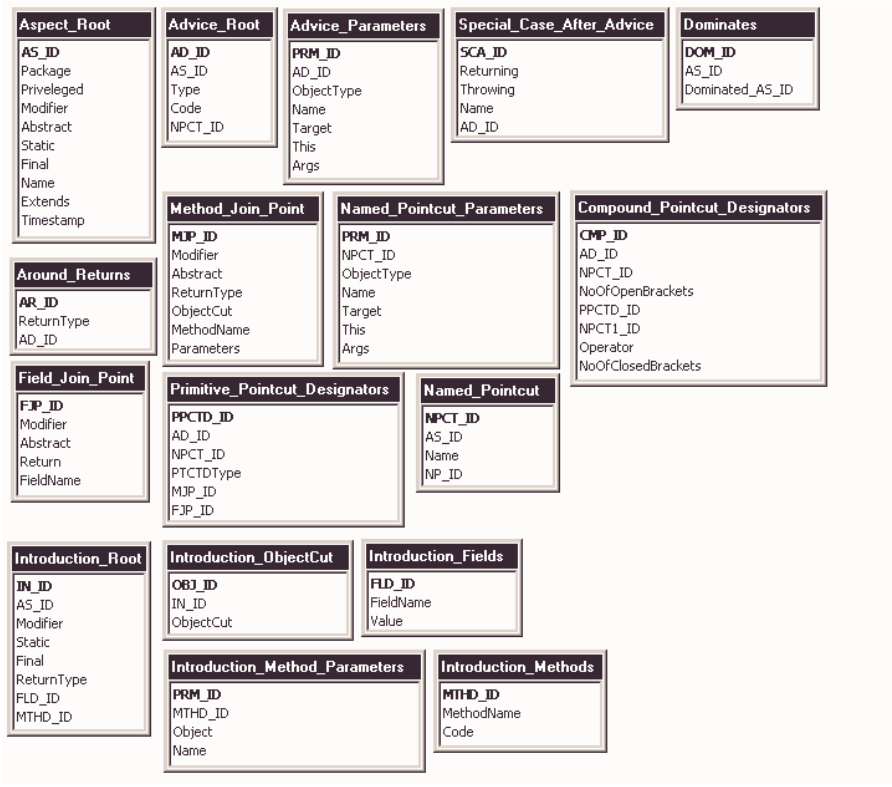


Fig. 5 Detailed structure of each relational table in the mapping

The data to be stored is obtained through an API developed for the AspectJ 1.0 parser. The API supports traversal through the structure of an aspect and obtaining detailed information about its components (advices, pointcuts, etc.).

3.1 Mapping Example

We discuss the structure of the various tables and rationale behind some of the design choices by discussing the mapping of our example aspect from fig. 2 to the relational schema. When showing the mapping we omit the tables and the attributes not used in our example aspect.

The *Aspect_Root* table captures the information that relates to the aspect at the top-most level e.g. the Java *package* the aspect belongs to, whether it is *privileged* to access private members of classes in the package and other aspects that it might *extend*. This table also provides an opportunity to capture some additional meta-data about the aspect e.g. a *timestamp* showing when the aspect was created. Note that the inheritance relationship (*extends*) is captured within the *Aspect_Root* table while the relationship with other aspects over whose behaviour an aspect has a higher execution priority (*dominates*) is captured separately in the *Dominates* table. From the viewpoint of the aspect specifying the relationships, these are both one-to-many relationships. However, they are stored differently because of the following reasons:

- An additional *Extends* table will be redundant as the link in the *Aspect_Root* table already represents the *many* edge of the relationship i.e. there are links from all the sub-aspects to the super-aspect but not vice versa. Storage of a new aspect does not require additional operations as no references in its super-aspect record need to be updated.
- Capturing the *dominates* relationship within the *Aspect_Root* table will in fact equate to capturing the *dominated_by* relationship. This will mean that when a new aspect is stored in the database all the *dominated_by* links will need to be updated. It is more efficient to store the relationship with reference to the aspect it was specified in.
- While *dominates* might appear a one-to-many relationship from the viewpoint of the aspect in which it is specified, in fact it can be an inverse many-to-one relationship from the viewpoint of the aspect *dominated*. This is because multiple aspects can specify a relationship to *dominate* the same aspect.

ASPECT Root		
AS ID	**Name**	**Timestamp**
1	ExampleAspect	07 Mar. 2002 14:45

The *Advice_Root* table captures the advice type (before, after, etc.) and the code to be executed at the join points it operates on. It also relates to the aspect which defines the advice through the *Aspect ID*. If the advice uses a named pointcut the *NPCT_ID* attribute is used to relate to the appropriate pointcut in the *Named_Pointcut* table. Note that although advices and named pointcuts are strongly related they have been modelled separately due to the following reasons:

- Named pointcuts do not need to model the advice type (before, after, etc.). This is because the advices depend on the pointcut but not vice versa.

- Named pointcuts can also reference other named pointcuts[1] (hence the recursive link in fig. 4).

Parameters for the advice and the named pointcut usage in the advice are mapped using the *NamedPointcut_Parameters* and *Advice_Parameters* tables.

ADVICE Root				
AD ID	AS ID	AdviceType	Code	NPCT ID
1	1	before	// some code	1
2	1	after	// some code	<null>

Named Pointcut		
NPCT ID	AS ID	Name
1	1	pc1

NamedPointcut Parameters				
PRM ID	NPCT ID	ObjectType	Name	Target
1	1	Connection	c	c

Advice Parameters				
PRM ID	AD ID	ObjectType	Name	Target
1	1	Server	s	s
2	2	Connection	a	a

The *Primitive_Pointcut_Designators* table captures single join points and relates them to the appropriate named pointcut or advice operating upon them. The *Compound_Pointcut_Designators* table captures the compound pointcuts formed from the primitive ones and the combinational logic involved. The mapping of compound pointcuts and capturing of the combinational logic was one of the main challenges encountered in the database design. This is because it is not possible to perceive the complexity of a compound pointcut or its combinational logic as these depend on the preferences and programming styles of individual developers. In our current design this problem has been addressed by storing details of opening and closing brackets in such a statement in attributes within the *Compound_Pointcut_Designators* table. While this is not the most elegant of designs and leads to highly syntactic mapping, it can store even the most complex of statements effectively and cater for diverse programming styles. We plan to work on a neater yet effective design of this feature in the future.

Our example aspect only uses primitive pointcut designators.

Primitive Pointcut Designators				
PPCTD ID	AD ID	NPCT ID	PTCTD Type	MJP ID
1	<null>	1	call	1
2	1	<null>	call	2

The *Method_Join_Point* and *Field_Join_Point* tables capture the details of each join point i.e. the method and field signature. Our example aspect only specifies behaviour for method join points.

Method_Join_Point					
MJP ID	Modifier	ReturnType	ObjectCut	MethodName	Parameters
1	public	void	Connection	*	String
2	public	String	Server	getAddress	<null>

[1] Note that named pointcuts can also be referenced by implicit pointcuts.

The *Introduction_Root* table captures general information about the introduced fields and methods e.g. modifiers, type or return type and relates these to the *Introduction_Fields* and *Introduction_Methods* tables which capture the details of the introduced fields and methods. The *Introduction_Method_Parameters* table stores information about parameters of introduced methods while the *Introduction_ObjectCuts* table captures the object types into which the methods and fields are introduced.

Introduction_Root					
IN_ID	**AS_ID**	**Modifier**	**ReturnType**	**FLD_ID**	**MTHD_ID**
1	1	private	boolean	1	<null>
2	1	public	void	<null>	1

Introduction_Fields		
FLD_ID	**FieldName**	**Value**
1	disabled	false

Introduction_Methods		
MTHD_ID	**MethodName**	**Code**
1	printMessage	// some code

Introduction Method Parameters			
PRM_ID	**MTHD_ID**	**ObjectType**	**Name**
1	1	String	m

Introduction ObjectCuts		
OBJ_ID	**IN_ID**	**ObjectCut**
1	1	Connection
2	2	Client
3	2	Server

4 Querying Support Mechanism

A GUI tool has been developed to support the querying and retrieval of aspects from the repository. As discussed below the queries from the GUI tool are translated to SQL for processing against the repository. Consequently, an expression builder can be used to bypass the GUI and support an experienced developer in forming his/her own queries. Another key feature of the tool is support for extracting advice and introductions alone as well as whole aspects.

To demonstrate the querying support we use a simple query that a user might put together. The query is as follows:

Retrieve an aspect which employs:

- *an **after** advice that takes in **Server** object types as a **parameter**;*
- *uses a **call** pointcut designator on methods called **getAddress**;*
- ***introduces public** methods on **Client** objects*

Figures 6 (a) through (c) show the steps involved using the GUI.

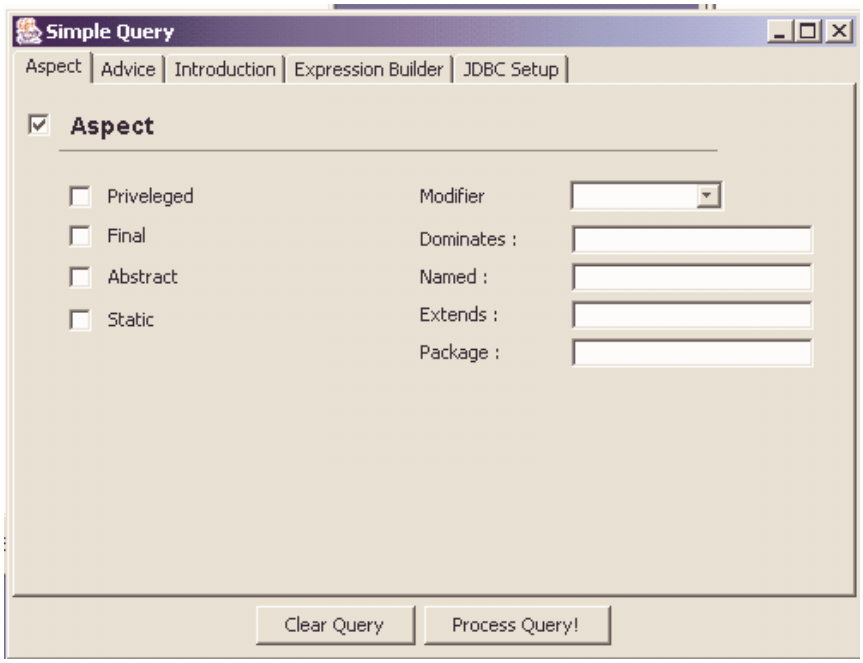


Fig. 6(a) Example query step 1

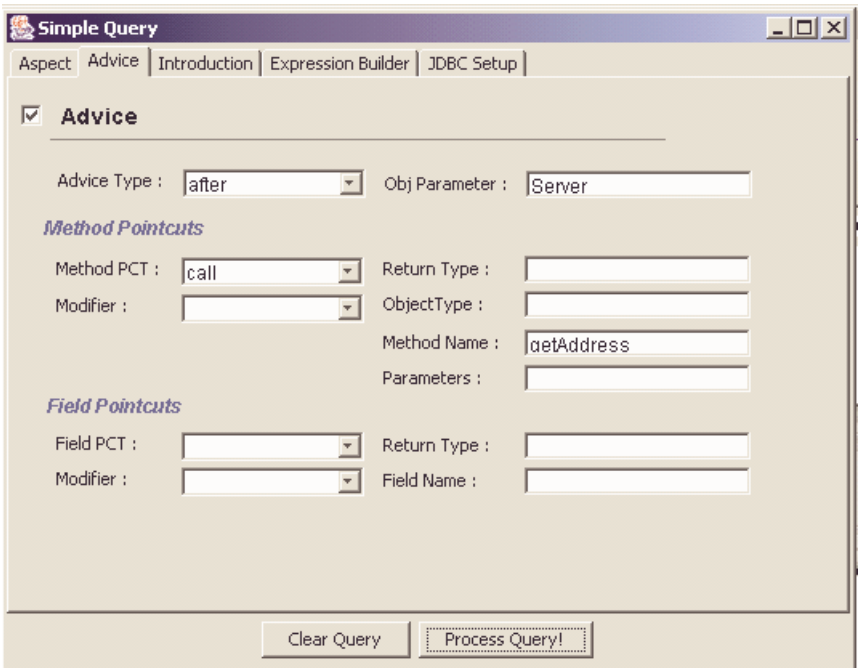


Fig. 6(b) Example query step 2

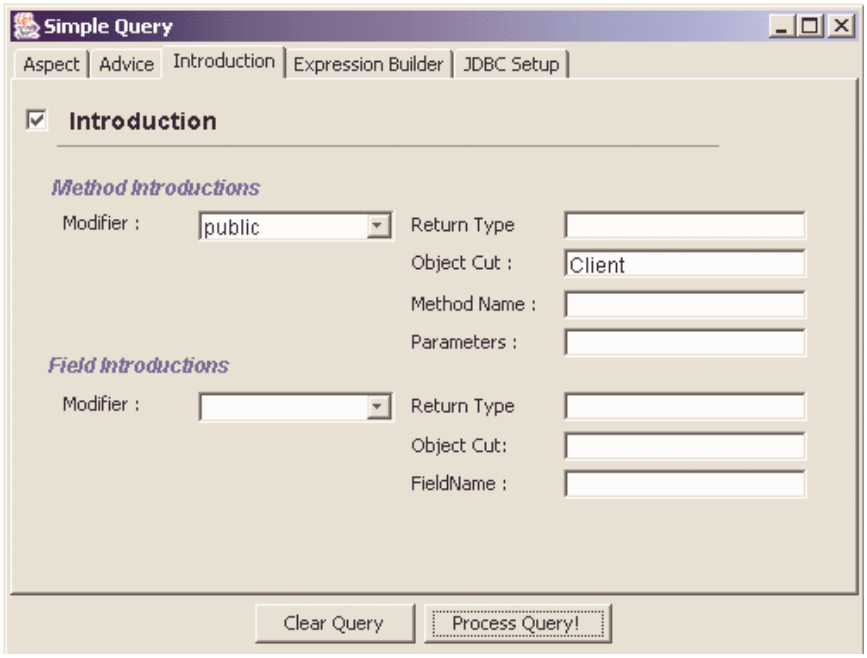


Fig. 6(c) Example query step 3

4.1 Query Processing and Translation to SQL

The data provided by the user is extracted from the GUI and appropriate **SELECT**, **FROM** and **WHERE** clauses are generated. The retrieval of the aspect code from the database is a two step operation:

1. Assemble and execute the SQL using the GUI as previously described in order to return the key ID's in the result set.
2. Using the key ID's obtained in the first step create a number of SQL queries to retrieve result sets in order to recreate the aspect/advice/introduction.

In the case of our query the SQL clauses generated for step 1 are as follows. The Aspect ID obtained can then be used to create other SQL queries to retrieve appropriate parts of the aspect in order to recreate it.

SELECT clause:

```
Aspect_Root.AS_ID
```

FROM clause:
In the example query the following tables were determined to be used in this clause:

```
Aspect_Root
Advice_Root
Advice_Parameters
Primitive_Pointcut_Designators
Method_Join_Point
Introduction_Root
Introduction_Method
Introduction_ObjectCuts
```

WHERE clause:
The relevant keys are joined and the data entered/selected in the GUI added to the **WHERE** string as follows:

```
Aspect_Root.AS_ID = Advice_Root.AS_ID AND
Aspect_Root.AS_ID = Introduction_Root.AS_ID AND
Advice_Root.AD_ID = Advice_Parameters.AD_ID AND
Advice_Root.AD_ID = Primitive_Pointcut_Designators.AD_ID AND
Primitive_Pointcut_Designators.MJP_ID = Method_Join_Point.MJP_ID
AND
Introduction_Root.IN_ID = Introduction_Object_Cuts.IN_ID AND
Introduction_Root.MTHD_ID = Introduction_Methods.MTHD_ID AND
Advice_Root.AdviceType = 'after' AND
Advice_Parameters.ObjectType = 'Server' AND
Primitive_Pointcut_Designators.PTCTD_Type = 'call' AND
Method_Join_Point.MethodName = 'getAddress' AND
Intrduction_ObjectCuts.ObjectCut = 'Client'
```

5 Related Work

As is the case with the natural progression of all paradigms, research on aspect-orientation is now progressing from programming to earlier stages such as design [5] [24] [25] and requirements [8] [22]. Similarly, the role of aspects in persistent environments has also been explored and vice versa. Most of the pioneering work in this area has been carried out at Lancaster University, UK. [20] describes the extension of object-oriented databases with the notion of aspects. [12] [17] [21] [23] discuss the role of AOP to support customisability and evolution in object-oriented database systems. [16] proposes PersAJ, a prototype to make AspectJ aspects persistent using an OODBMS. The approach is based on taking the crosscutting nature of persistence into account. An aspect is used to separate the persistence approach from the persistent aspects themselves. This makes the persistence model independent of a particular AOP approach because the persistent aspects do not encapsulate the knowledge about their storage structure (which largely depends on the particular AOP approach being employed). It also localises the changes resulting from the evolution of the aspect language or the aspect structure making maintenance and modifications to the persistence model inexpensive [18]. A detailed description of

weaving aspects in a persistent environment can be found in [18]. All the above approaches focus on AOP and object databases. The approach proposed in this paper employs a relational database to support storage and retrieval of aspects. This forms a contribution towards identifying mapping of an aspect to the relational model and exploiting the significant investment in relational database systems on part of a large number of organisations.

[11] describes a component repository supporting aspect-based indexing and querying of components. Components can be retrieved for reuse on the basis of services they provide or require. The approach proposed in this paper can also be used to retrieve objects on the basis of aspects which cut across them. It also supports fine-grained querying of aspects themselves hence making it possible to identify their application context. This supports validation of aspects in their reuse context. Fine-grained querying also makes it possible to retrieve parts of different aspects for reuse in the formulation of a new one.

[19] identifies evaluation, integration, context, process, quality and evolution risks as life cycle aspects in a component-based development cycle; these risks cut across the various development stages. A component repository is used at the nucleus of the risk management process devised. The approach proposed in this paper can also help in risk reduction as it makes it possible to validate aspects in their reuse context.

6 Conclusions and Future Work

This paper has proposed an approach for storing aspects in a relational database system. The novelty of the work is in the provision of aspect persistence in relational asset repositories, and improved support for reuse through querying and retrieval of program modules on the basis of their global properties (described by aspects). In addition, by having aspects available which have specified tasks and attributes that can be used system wide we can reuse them with little or no alteration in other systems. As pointed out in [5] several aspects follow easily recognisable patterns and so can be adapted relatively easily to new contexts. An example of this would be the observer pattern or a simple tracing aspect which logs the instantiations and calls exhibited by a system during testing. The existence of such reusable (and adaptable) patterns further strengthens the argument for aspect storage and mining. However, for such variation and reuse to be effective it is essential that not only suitable aspects are retrieved but also support is available to help determine their original context of use and their influence within the new application context. This is critical as with mechanisms such as pattern matching employed in techniques such as AspectJ it is possible that reuse will result in matching modules to which the aspect's behaviour should not apply or ignoring modules to which it should. Inherent support for fine-grained querying in the proposed approach makes it possible to identify an aspect's original application context and its influence in the reuse context.

The possibility of mining the asset repository for existing aspects and adapting them to new or slightly different application contexts at lower effort and cost brings exciting possibilities to support variation within software product lines [7]. Software product lines are concerned with the creation of families of products, with each

product sharing a common set of features. For instance a common asset in a software product line could be a networking module that all the products in the family share. The variation point in this instance might be the network protocols which that particular module is capable of. The aspect could simply weave in the various networking capabilities required for the particular software product. Our work in the future will focus on use of aspect repositories to support such variations. We also plan to develop an improved design for capturing compound pointcuts. We will also explore the development of mapping algorithms and a common persistent aspect representation to underpin aspect reuse. Another natural future direction is the use of the work presented in this paper and our previous work on object databases in this context as a basis for building aspect stores based on object-relational database systems. We are currently considering some commercial object-relational systems for the purpose.

Acknowledgement: This work is partly support by UK Engineering and Physical Sciences Research Council Grant GR/R08612. The authors also thank Songuel Ballikaya for developing the API for the AspectJ 1.0 parser.

References

1. Xerox PARC, USA, *AspectJ Home Page*, http://aspectj.org/
2. Hirschfeld, R., *AspectS Home Page*, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/
3. Bergmans, L. and Aksit, M., *Composing Crosscutting Concerns using Composition Filters.* Communications of the ACM, 2001, **44**(10).
4. Chavez, C., Garcia, A.F., and Lucena, C.J.P. *Some Insights on the Use of AspectJ and Hyper/J. Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster University, UK*, 2001: Cooperative Systems Engineering Group, Technical Report, CSEG/03/01.
5. Clarke, S. and Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. *International Conference on Software Engineering (ICSE)*, 2001.
6. Clarke, S. and Walker, R.J. *Mapping Composition Patterns to AspectJ and Hyper/J. ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering*, 2001.
7. Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*. 2002: Addison-Wesley.
8. *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop at AOSD Conference*, 2002.
9. Elrad, T., Filman, R., and Bader, A., *Theme Section on Aspect-Oriented Programming.* Communications of ACM, 2001, **44**(10).
10. Gamma, E., et al., *Design Patterns - Elements of Reusable Object-Oriented Software.* 1995: Addison Wesley.
11. Grundy, J.C. *Storage and retrieval of Software Components using Aspects. Australasian Computer Science Conference*, 2000: IEEE Computer Society Press: 95-103.
12. IBM Research, *Hyperspaces*, http://www.research.ibm.com/hyperspace/
13. Kiczales, G., et al. *Aspect-Oriented Programming. ECOOP*, 1997: Springer-Verlag, Lecture Notes in Computer Science 1241.

14. Mezini, M. and Lieberherr, K.J. *Adaptive Plug-and-Play Components for Evolutionary Software Development*. *OOPSLA*, 1998: ACM, SIGPLAN Notices 33(10): 97-116.
15. Mikhajlov, L. and Sekerinski, E. *A Study of The Fragile Base Class Problem*. *ECOOP*, 1998: Springer-Verlag, Lecture Notes in Computer Science 1445: 355-382.
16. Rashid, A. *On to Aspect Persistence*. *2nd International Symposium on Generative and Component-based Software Engineering (GCSE)*, 2000: Springer-Verlag, Lecture Notes in Computer Science 2177: 26-36.
17. Rashid, A. *A Hybrid Approach to Separation of Concerns: The Story of SADES*. *3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection)*, 2001: Springer-Verlag, Lecture Notes in Computer Science 2192: 231-249.
18. Rashid, A., *Weaving Aspects in a Persistent Environment.* ACM SIGPLAN Notices, Feb. 2002.
19. Rashid, A. and Kotonya, G. *Risk Management in Component-Based Development: A Separation of Concerns Perspective*. *ECOOP Workshop on Advanced Separation of Concerns (ECOOP Workshop Reader)*, 2001: Springer-Verlag, Lecture Notes in Computer Science.
20. Rashid, A. and Pulvermueller, E. *From Object-Oriented to Aspect-Oriented Databases*. *11th International Conference on Database and Expert Systems Applications (DEXA)*, 2000: Springer-Verlag, Lecture Notes in Computer Science 1873: 125-134.
21. Rashid, A. and Sawyer, P., *Aspect-Orientation and Database Systems: An Effective Customisation Approach.* IEE Proceedings - Software, 2001, **148**(5): p. 156-164.
22. Rashid, A., et al. *Early Aspects: A Model for Aspect-Oriented Requirements Engineering*. *IEEE Joint International Requirements Engineering Conference (Accepted for Publication)*, 2002.
23. Rashid, A., Sawyer, P., and Pulvermueller, E. *A Flexible Approach for Instance Adaptation during Class Versioning*. *ECOOP 2000 Symposium on Objects and Databases*, 2000: Springer-Verlag, Lecture Notes in Computer Science 1944: 101-113.
24. Suzuki, J. and Yamamoto, Y. *Extending UML with Aspects: Aspect Support in the Design Phase*. *3rd AOP Workshop held in conjunction with ECOOP '99*, 1999.
25. Tarr, P.L., et al. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. *International Conference on Software Engineering (ICSE)*, 1999: ACM: 107-119.
26. Weyuker, E.J., *Testing Component-Based Software: A Cautionary Tale.* IEEE Software, 1998, **15**(5): p. 54-59.