

# Mining Aspects

Neil Loughran, Awais Rashid

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

**Abstract.** The mining of existing assets is an important concern for software developers and organisations. The availability of tools which allow fine grained queries to be performed on these assets will improve productivity by allowing developers to locate and adapt assets efficiently. Mining of assets can take part at many different stages throughout the software development lifecycle. Typical assets for mining could include program code, designs, system architectures, specifications, algorithms and the like. Aspects are particularly suitable for mining as they allow system wide concerns such as synchronisation, logging, debugging and the like to be encapsulated into a single construct, making them ideal candidates for reuse. This paper discusses the increasing demand for tools that support the mining of existing assets with a special focus on Aspect-Oriented Software Development (AOSD).

## 1 Introduction

This paper discusses the increasing demand for tools that support the mining of existing assets with a special focus on Aspect-Oriented Software Development (AOSD).

As software systems increase in size and become more complex, the need for quality software to be produced within cost and schedule constraints has become a prime concern for software developers and organisations. The reuse of existing assets has been demonstrated to greatly reduce software development costs, as well as improve productivity and quality [1]. Being able to store assets has other benefits such as being able to store domain knowledge, reuse context, test cases and results, and other verification and validation data.

The term ‘mining existing assets’ generally refers to the locating of useful information from an organisation’s asset base for reuse in new applications. Typical candidates for mining include program code, designs, system architectures, specifications, algorithms and the like. Repositories for the storage of these assets can play a central role in the development of new systems. However, whilst having a solid core base of these assets is important, navigation, storage and retrieval in a meaningful manner is not a simple task due to the underlying data representations involved. If it takes too long to locate an asset then the benefits of reuse are not being fully utilised. Various approaches (in increasing complexity) in use range from the storage of keywords (meta-data) along with the asset to the mapping of an asset to a suitable representation within the asset repository (this usually requires mapping the asset to its representation in the underlying database model).

In this paper, we focus on the mining of one particular category of assets: the aspects. This is of particular interest due to the following reasons:

Aspects seek to solve some general architectural problems which are common from system to system in a modularised way. Hence, effective storage and mining support can provide a higher degree of reuse.

The crosscutting nature of aspects dictates that an effective mining approach is not only able to find suitable aspects but also assist in determining the context of their original use and influence within the new context.

Effective asset mining can help trace an aspect and, therefore, its variations (to suit the new application context) and their impact throughout the software life cycle. While this is a feature desirable for all assets, the crosscutting nature of aspects makes such traceability critical for effective reuse and change impact determination.

There is an increasing interest from the software engineering community in this new paradigm. Therefore, it is only natural that support for asset mining be extended to this new paradigm.

The next section further motivates the need for aspect mining. Section 3 discusses the nature of tool support required and some initial prototypes for the purpose. Note that the prototypes, at present, only support mining of code level aspects due to the less well-defined nature of aspects at earlier development stages. However, the discussion in this paper is based on the view that aspect mining applies to aspects at all development stages. Also, as discussed in section 3.4, the approach used in the prototypes also forms a suitable candidate for mining early aspects.

## 2 Need for Aspect Mining

AOSD is a new software development paradigm which employs special abstractions, the aspects, to modularise concerns that cut across other parts of a system. Examples of such crosscutting concerns include synchronisation, debugging, logging, memory management, security and persistence. The modularisation of such concerns with AOSD techniques renders them ideal candidates for reuse. 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. 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 [2] 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. In case of variation of requirements between the two application contexts it is essential that this variation is traced from early on in the life cycle through to the implementation and evolution stages.

It has been said that design and testing account for a major portion (in some cases as high as 80%) of the system costs [3]. Effective aspect mining can provide more effective reuse of existing assets. Besides, a significant number of tests used in the verification of these assets can be reused. As mentioned earlier the crosscutting nature of aspects makes such verification and validation an expensive activity. Aspect storage and mining can help reduce these costs.

As pointed out in [4] 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.

The possibility of mining the asset base 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. 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 capabilities required for that particular software product.

### 3 Tool Support

A basic premise for any mining tool is that it must be able to locate the desired assets efficiently. If the underlying data structures are too complex and the database is populated with a very large number of assets then the mining operation could take too long to perform. Similarly, if the underlying data representation is too simple then the potential benefit of fine-grained querying is minimal resulting in too many records (most of them undesirable) being retrieved.

There are a number of approaches to consider for the development of a framework to support the mining of aspects at different stages of the software development lifecycle. We consider three of these approaches in some detail. These include:

- *Direct* aspect storage coupled with meta-data
- Mapping of aspect anatomy to the database model
- Hybrid approach

#### 3.1 *Direct* Aspect Storage Coupled with Meta-data

This approach entails storing the aspect as a binary/character object along with some useful meta-data that describes the object in a coarse grained fashion. The user is then able to query this meta-data in order to retrieve the aspects. Fig. 1 outlines a possible implementation on a requirements aspect.

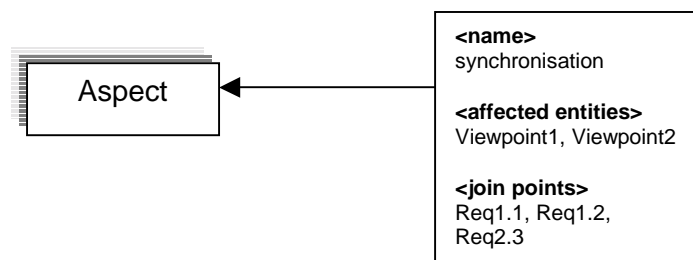


Fig. 1. Requirements aspect with associated meta-data representation

Advantages:

- easy to implement;

- fast and efficient.

Disadvantages:

- loss of aspect representation in the database;
- needs effective capturing of meta-data and its evolution as the aspect changes in line with different application contexts;
- fine-grained variation and traceability is severely constrained by loss of aspect representation;
- query complexity constrained by meta-data;
- aspect constrained for use in the particular AOSD approach used to develop it.

### 3.2 Mapping of Aspect Anatomy to the Database Model

This approach involves fine-grained mapping of an aspect to the underlying database model e.g. an object model or a relational model. The user is then able to query properties of an aspect in a flexible fashion and, hence, perform complex fine-grained queries. Simple examples of such queries include finding join points influenced by an aspect or tracing a fine-grained change in a specification aspect to its corresponding design and implementation aspects (at a fine-granularity). An aspect mapped to the relational database model (simplified for the purpose of this paper) is displayed in fig 2.

A side benefit of this approach is that the aspect storage structure might relate to one particular AOSD technique but upon retrieval the aspect may be transformed for use in a different AOSD mechanism. [5, 6], for example, propose algorithms to map aspects in AspectJ to Hyper/J [7] and vice versa. Similarly, queries based on a different AOSD technique can be transformed to comply with the aspect representations in the database. As AOSD techniques at earlier development stages mature, we envisage that such algorithms will be developed for mapping between them and exploited during retrieval of aspects at these stages.

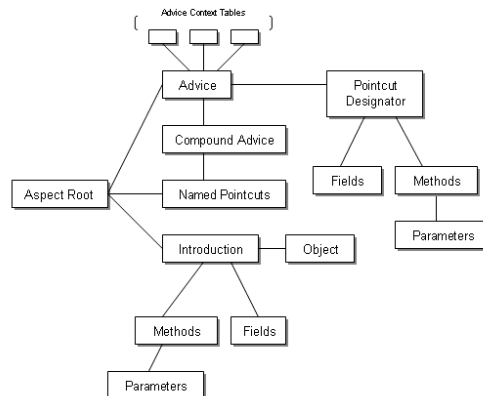


Fig. 2. Simplified model of aspect mapped to relational schema

Advantages:

- aspect representation is retained in the database;
- fine-grained querying possible;

- resilient to changes in the aspect as there is no associated meta-data;
- can be relatively easy to adapt to other AOSD techniques.

Disadvantages:

- complex to implement;
- difficulty in mapping certain aspectual constructs such as combinational logic;
- potentially resource intensive.

### 3.3 Hybrid Approach

Perhaps the best option would be a hybrid of the previously mentioned approaches. The hybrid solution could involve only storing the most important parts of an aspect's anatomy into a database together with the complete aspect itself as a binary or character object.

Advantages:

- a reasonable degree of fine-grained querying possible;
- some aspect representation retained in database;
- relatively easy to implement;
- good efficiency.

Disadvantages:

- some restrictions on the types of querying possible;
- aspects constrained to a particular AOSD technique.

### 3.4 Existing prototypes

To date we have developed two prototypes to store, retrieve and mine AspectJ aspects. One of these prototypes, PersAJ, is based on an object database [8] while the other employs a relational database. Our choice of aspects has been constrained to the implementation level due to the lack of well-defined semantics of aspects at earlier development stages.

We have chosen to employ the approach based on mapping the aspect anatomy to the database model as it provides greater flexibility in terms of aspect mining (fine-grained queries are possible) and evolution of the persistent aspect structure and retrieval mechanisms in line with the continuously evolving nature of AOSD techniques [9]. During our experiments with code level aspects we have found fine-grained mapping and querying of aspects to be useful in determining their original context and their influence and effectiveness in the application context in which they are to be reused. We have also found that the fine-grained approach makes variations an easier task. We are of the view that while aspects at earlier stages mature it will be best to use a similar mechanism for storage and mining of these early aspects. Once, the semantics and representation of aspects at the various development stages have stabilised it will be more efficient to move to the hybrid approach.

## 4 Conclusion

It has been demonstrated with this paper that the mining of aspects will become an important concern where reuse is required. Being able to recover the desired aspects accurately and efficiently for reuse or in order to create new ones will have a beneficial effect on the software community in terms of time to market, and it remains to be seen what other benefits and solutions AOSD will bring. Certainly AOSD's ability to localise crosscutting concerns to single constructs can benefit the software community by effectively making such concerns easier to read and alter in response to changing requirements.

Having the ability to run very fine-grained queries may sound good in practice but it would have to be up to the end user to judge if the benefits of that approach outweigh its demerits. It has been said that the reuse of small grained software assets such as subroutines or small programs is rarely economical with the most useful assets for mining being those which make up large patterns of interoperation throughout architecture. However, the latter could certainly describe the functionality of any given aspect construct.

The table in fig 3 summarises the attributes which the different data modelling approaches possess.

Approach	Direct Storage with Meta-data	Relational Database Mapping	Hybrid
<b>Complexity of Implementation</b>	Simple	Complex	Simple to Moderate
<b>Query Flexibility</b>	Low	Very High	Moderate
<b>Performance</b>	High	Dependent on Query Complexity	Moderate to High
<b>Typical Aspect Candidates</b>	Requirements Aspects Design Documentation Architectural Models	Aspectual Code Patterns	Requirements Aspects Design Documentation Architectural Models Aspectual Code Patterns

**Fig. 3.** Comparison of existing approaches

Perhaps the best solution and compromise would be the use of a hybrid data structure where efficiency and some ability for fine-grained queries co-exist and the structures are not too complex to design and implement. The decision as to which data modelling technique best serves the aspect mining needs of the user ultimately depends upon the resources, context and reuse policies in place within the organisation.

## References:

1. Cusumano, M. *The Software Factory: A Historical Interpretation*, IEEE Software (March 1989) pp. 23-30:
2. Xerox PARC, USA, *AspectJ Home Page*, <http://aspectj.org/>
3. Cusumano, M., *Japans Software Factories*, Oxford University Press, 1991:
4. Clarke, S. and Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. *International Conference on Software Engineering (ICSE)*, 2001:
5. 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:
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. IBM Research, *Hyperspaces*, <http://www.research.ibm.com/hyperspace/>
8. 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;
9. Rashid, A. *Weaving Aspects in a Persistent Environment*, ACM SIGPLAN Notices, Feb. 2002