Aspect Mining in Procedural Object-Oriented Code

Muhammad Usman BHATTI[1] Stéphane DUCASSE[2] Awais RASHID[3]
[1]*CRI, Université de Paris 1 Sorbonne, France*
[2]*INRIA - Lille Nord Europe, France*
[3]*Computing Department, Lancaster University, UK*
muhammad.bhatti@malix.univ-paris1.fr, stephane.ducasse@inria.fr, awais@comp.lancs.ac.uk

Abstract

Although object-oriented programming promotes reusable and well factored entity decomposition, industrial software often shows traces of lack of object-oriented design and procedural thinking. This results in domain entity scattered and tangled code. This is often true in data intensive applications. Aspect mining techniques search for various patterns of scattered and tangled code pertaining to crosscutting concerns. However, in the presence of non-abstracted domain logic, the crosscutting concerns identified are inaccurately related to aspects since lack of OO abstraction introduces false positives. This paper identifies the difficulty of identifying crosscutting concerns in systems lacking elementary object-oriented structure. It presents an approach classifying various crosscutting concerns. We report our experience on an industrial software system.

1 Introduction

The problem of scattered code has principally been treated in the domain of aspect mining [9, 12]. The underlying assumption in most of the existing work in this domain is that scattered and tangled code in object-oriented systems only originates from the tyranny of dominant decomposition [10], while other system artifacts have been adequately encapsulated in their associated abstractions (namely classes) resulting from an object-oriented analysis. Unfortunately, this is not always the case in existing industrial software systems as most of them are developed under budget limitation and time to market, therefore the design quality is often deteriorated [5]. This is often the case in data intensive applications. Often there are missing object-oriented abstractions for domain entities, which end up as crosscutting concerns scattered and tangled in other classes of the system. Therefore, traditional aspect mining techniques are inadequate in presence of *non-abstracted domain logic* as they wrongly associate lack of object-oriented structure with aspects. This occurs because, in such systems, scattered code is often not a sign of missing aspects but missing object-oriented abstractions. Thus this situation introduces noise during aspect mining. Therefore, there is a need to complement the existing works on aspect mining with approaches identifying abstractions in object-oriented code.

We coined the term *procedural object-oriented* code for the code which lacks an overall object-oriented design but nonetheless has been developed using state of the art object-oriented languages. Henceforth, we introduce an approach for the identification of diverse crosscutting concerns present in procedural object-oriented code. The approach presented in this paper identifies different crosscutting concerns present in a software system: aspects as well as non-abstracted domain logic. Crosscutting concerns pertaining to non-abstracted domain entities are identified based on their data application usage.

This paper is organized as follow: Section 2 describes the context and gives an overview of the case study that is used in the paper. Section 3 describes the technique used to mine crosscutting aspects and its limits in presence of procedural object-oriented code. Section 4 presents our approach for the identification and classification of concerns. It presents the results obtained on the case study presented before. Section 6 talks about the related work and Section 7 concludes the paper.

2 A Case Study : Blood Analysis Application

To illustrate the shortcomings of aspect mining techniques in presence of procedural object-oriented code and our approach to classify crosscutting concerns, we use an industrial case study: a software system driving machines performing blood disease analyses.

For the sake of precision and clarity, we shall only be talking about the software subsystem that manages the functional entities and processes, and operates with the database

layer to manage the relevant data. Certain core functionalities, such as blood analysis data, reagents used by the machines, results and patient data are the key features implemented at this layer. Every test is performed on patient data and the results of the tests are then stored in persistent storage system. The persistent storage system is extensively used to record all the business objects, machines activities, test traceability information, machine products, and machine maintenance information. Quality control is performed on the machines with plasma samples for which the results are known beforehand, to determine the reliability of machine components. In addition to quality control, a machine is calibrated with the predetermined plasma samples so that raw results can be interpreted in different units according to the needs of the biologist or doctor for easy interpretation.

Table 1 below shows some of the software quality metrics for our case study business entity subsystem. Lines Of Code (LOC) tallies all lines of executable code in the system. Number Of Methods (NOM) and number Of Attributes (NOA) metrics indicate respectively the total level of functionality supported and the amount of data maintained by the class. Depth Of Inheritance (DIT) indicates the level of inheritance a class has. And finally, Lack of Cohesion Of Methods (LCOM) indicates the cohesion of class constituents by examining the number of disjoint sets of the methods accessing similar instance variables; lower values indicates better cohesiveness [8].

Table 1. Case Study Metrics

Class Name	LOC	NOM	NOA	DIT	LCOM
CPatient	11,462	260	9	1	0.85
CTest	2792	81	13	1	0.72
CProduct	2552	77	6	1	0.72
CResults	1652	52	13	1	0.85
CPersistency	1325	67	29	2	0.97
CGlossary	1010	121	5	1	0.80

Table 1 communicates some facts about the business entity layer: There is a clear lack of hierarchical structure and presence of huge service component classes lacking cohesion, with large number of methods. It can also be noted that certain domain entities such as quality control, calibration, and raw results do not have associated classes in the code (CResult class in the table contains the functionality to calculate interpreted results). These are signs of as *procedural object-oriented code*. Procedural object-oriented code results in *crosscutting concerns* both due to absence of domain abstractions and limitation of OOP mechanisms to encapsulate certain concerns cleanly. The presence of such a procedural object-oriented code raises the problem of the distinction of aspects which are identified by aspect mining techniques. In the following we employ one aspect mining technique to examine the crosscutting concerns identified in procedural OO code.

3 Evaluating the FAN-in Aspect Mining Technique

Aspect mining techniques automate the identification of scattered code in software systems by producing a list of crosscutting concern candidates. These can be identifiers and redundant lines of code, code clones, metrics, etc. [9]. We employed FAN-in technique for mining crosscutting concerns in our case study software [12]. The principle of FAN-in for concern identification is to discover all methods which are called frequently because crosscutting concerns may reside in calls to methods that address a different concern than that of the caller [3].

Since there were no tools computing FAN-in in C#, we developed our own tool based on the bytecode analysis. This tool looks for method calls to all the methods defined in the application classes and lists those with values higher than the filtering threshold given by the user for the degree of their scattering *i.e.,* FAN-in metric. Table 2 shows the crosscutting candidate methods for FAN-in ≥ 10 (threshold for crosscutting candidates as described in [3]).

Table 2. Application methods and associated FAN-in values

Method	FAN-in
UpdatePhysicalMeasures	10
CreateResultCalibration	10
NewMeasureCalibration	10
SearchProductIndex	10
SearchCalib	13
SearchPatient	17
PublishException	19
ReadMesureCalib	22
Trace	24
SearchProduct	26
SearchTestData	29
DecryptData	35
ReadRawResults	41
PublishEvent	96
ValidateTransaction	89
GetGlossaryValue	127
GetInstance	101

Although, crosscutting concerns indicated the presence of scattered code, a good amount of the results is related to methods pointing to domain entities because of the non-abstracted domain logic (See Table 2).

Hence, it shows that the FAN-in metric can identify different types of crosscutting concerns, at the same time due

to the absence of aspects and lack of object-oriented abstractions, but without distinguishing them. This is because there is no inherent way while analyzing method calls to ascertain the origin of crosscutting concerns. The FAN-in metric provides us a hint about scattering and tangling but this information needs to be complemented as we present with our approach in the next section.

4 Concern Classification

Our approach to classify crosscutting concerns incorporates information extracted from the use of variables representing domain entities. With the use of application data we distinguish the type of behavior being invoked and the type of logic the invoked method provides to its caller. This section describes the approach illustrated in Figure 1, and is organized as follows: Section 4.1 describes data and behavioral scattering and Section 4.2 defines a model for our concern classification approach. Section 4.3 describes the assignment of various methods to domain entities related concerns, and Section 4.4 describes the algorithm for the classification of concerns.

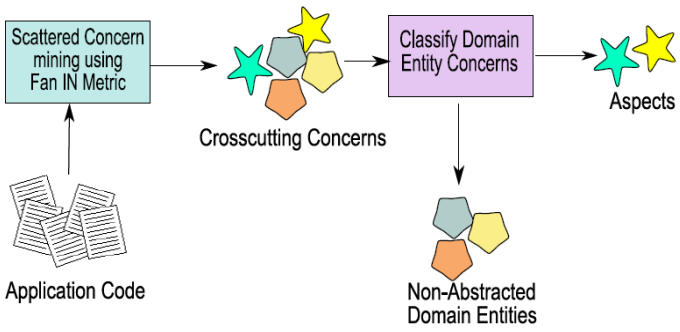

Figure 1. Concern Classification Approach

4.1 Data and Behavioral Scattering

As we show in the previous section, FAN-in aspect mining technique identifies candidates that are only due to a simple lack of object-oriented abstraction. To be able to identify such false positives, we make the hypothesis that data is placed far from its corresponding behavior due to absence of associated domain classes [5]. Therefore, we need to identify the data and its associated behavior. Once this identified, we will be able to remove noise from aspect mining candidates. Let us analyze two forms of code scattering.

Data Scattering. The absence of domain entity abstractions causes two types of data components to appear: entities representing database tables and global enumerated types. The access to this data is performed through their accessors from the methods implementing their behavior (See the pattern Move Behavior close to Data [5]). Hence, code accessing such data causes crosscutting concerns to appear:

- **Global enumerated type accesses.** Dispersed accesses to global enumerated types representing the states and object types of various entities (such as patient, test, tube types, etc.) in diverse methods of classes present in the system.
- **Direct access to persistent data.** Reading and writing of persistent storage entities stored in the database without any particular classes associated to them. For example, there is no class encapsulating the operations performed on patient tubes.

Behavioral Scattering. Behavioral scattering means that multiple distinct behaviors are composed together in a single abstraction. This usually happens in the form of method calls, hence indicated by abnormal high FAN-in. In our case study this situation was frequent. Following are the scenarios for behavioral scattering occurrence:

- Since the required data is away from its behavior, therefore one behavior perpetually calls the other one to get its particular data. This results in high FAN-in value for data providers.
- Lack of a proper encapsulation for a behavior related to an entity and the behavior is spread into several client of the entity. This causes the client classes to frequently call the provider-logic, causing a high FAN-in value for logic-provider methods.
- Lastly, behavioral scattering occurs because a particular concern is impossible to be encapsulated in a particular abstraction using traditional OO techniques hence resulting in scattered behavioral composition of the crosscutting calls in the client locations such as caching and logging operations in our software system.

4.2 Model for Concern Classification

To identify crosscutting concerns appearing due to missing domain entity abstractions, we define a model based on data usage representation and identification of associated behavior. This model takes into account all the methods and data components present in the application. We define M as a set of all methods in components under analysis. T is defined as a set of all entities representing persistent storage units, in our case database tables, and V is defined as a set of all global state variables representing the states and various types associated to domain entities in T.

Domain Entity Model. For the domain entity we represent the state: global variables and database accesses. In the case of the case study the state mainly consists of the representation of various domain entities in the form of persistent storage (*i.e.,* a database) and global state and type variables. In the case of direct access to database elements, we use the mapping between domain entities and the database data to determine the methods accessing the states. The key point is that at the end of this step we have a clear identification of which methods access each state and this independently of the way the application is developed.

In the case study, database tables had a clear one-to-one association with the domain entities. Moreover, the system states and entity types represented by global enumerated types also have a clear one-to-one association with the domain entities. The one-to-one association between the domain entities and the above-mentioned data components *i.e.,* variables and database tables, is utilized to determine the methods related to each domain entity.

We define E as the set of all domain entities that are implemented by the application subsystem. Entity $e \in E$ consists of table $t(e)$ and variable $v(e)$ related to the associated domain entity e *i.e.,* $entity(e) = t(e) \vee v(e)$.

Hence, all methods in M accessing directly or indirectly domain entity-related data e are classified as implementing the concern related to the domain entity it accesses. In the example of Figure 2, methods of class A and class B access data "D" of class C either directly or through accessors hence they are identified as implementing concern relating to the entity "D".

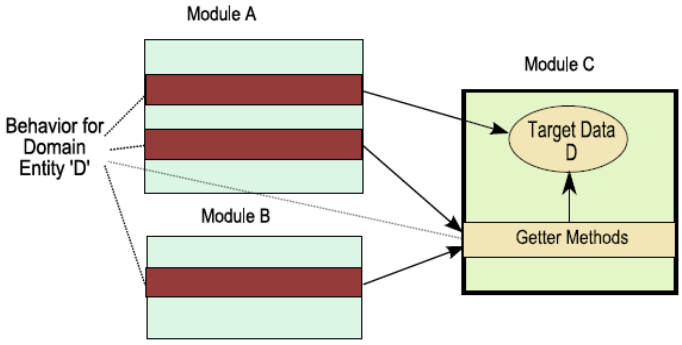

Figure 2. Domain Entity Concern Identification

Aspect Model. As defined earlier, it is assumed that crosscutting concerns also appear due to the absence of appropriate OOP mechanisms to interleave two intersecting behaviors in a non-recurrent way. Hence, we base our crosscutting identification model on the FAN-in metric [12] (*i.e.,* the higher the number of calls to a method, the more the chances are for it being a crosscutting concern). This hypothesis is reasonable since there is a consequent amount of aspect mining techniques that search for the occurrences of scattered and tangled method calls to detect crosscutting behavior [9]. But we only consider those methods which do not directly or indirectly relate to domain entities. In general, such methods are invoked by those methods which are associated to domain entities as depicted in Figure 3. In the following, we introduce a model to ascertain that methods implementing domain entity related concerns are identified.

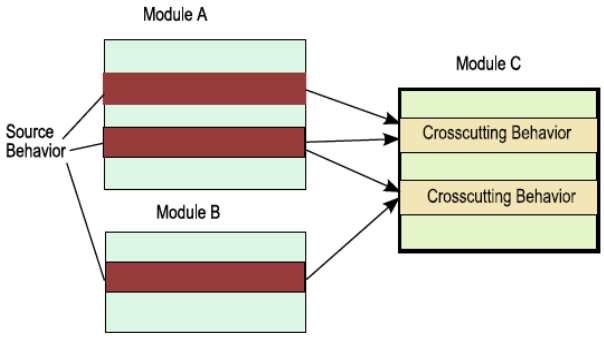

Figure 3. Aspect Identification

4.3 Domain Entity Concern Assignment

To classify methods related to domain entities, we define following primary properties:

- m *reads* t means that m directly reads from the object representation of table $t \in T$
- m *writes* t means that m directly writes to the object representation of table $t \in T$
- m *reads* v means that m directly reads from the variable $v \in V$
- m *calls* n means that m calls another method n
- m *accesses* t means that m directly reads from or writes to the object representation of table $t \in T$ (*i.e.,* *accesses* = *reads* $\cup$ *writes*)

From these properties, we define the following derived properties for a concern c:

- m *implements* c related to domain entity e if m *accesses* $t(e)$ or m *reads* $v(e)$ *i.e.,*

$implements(m, c) = \{m \subseteq M | \forall e \in E : m$ *accesses* $t(e) \vee m$ *reads* $v(e)\}$

- Method n implements a concern c if n calls another method m and m *implements* c pertaining to domain entity e

$implement(n, c)$ = $\{n \subseteq M | \forall m \in M : n$ *calls* $m \wedge implements(m, c)\}$

We do not consider the classes during the concern identification because they do not mean much in term of coherent abstraction: As stated earlier, these are half-baked objects without clearly identified abstraction.

4.4 Algorithm for Concern Classification

We now define a simple algorithm to sort the various crosscutting concerns candidates identified using the FAN-in. The algorithm works as follow: All the crosscutting methods having a threshold value higher than f are added to the set crosscutting seeds. Each method is then examined to implement concerns related to the domain entities. Once the domain entity related methods have been marked, all the methods which are marked as crosscutting seeds and have not been marked as related to domain entities are crosscutting concerns.

$\{M\} \leftarrow \forall$ *Methods*
  Test all m *in* $\{M\}$ *for a FAN-in metric* f
  if $fanin(m) > f$
    $\{CCSeeds\} \leftarrow m$
  $\forall m \in \{M\}$ = *Iterate over Instructions of m*
  if $implements(m, c)$
    $concern \leftarrow m$
    $M \leftarrow M/m$ {*Remove* m *from* M}
  if $n \in \{M \cap CCSeeds\}$
    $\{Aspects\} \leftarrow n$

4.5 Results for Classification Approach

Using the algorithm described above, we implemented a tool to classify concerns related to domain entities and aspects. The results for the crosscutting concern classification are presented in Table 3. First two columns are those methods discovered as crosscutting candidates by the FAN-in tool and their corresponding FAN-in metric. In addition, the last column indicates concern classification performed by our tool.

We checked manually the results in the code and we found that the results produced are close to the classification that we have produced manually. Glossary concern appeared due to the absence of hierarchal structure and can be removed by the introduction of types of sub-types for various domain entities. In addition, identified aspects corresponds well with the established aspect candidates described in literature such as tracing, exception handling and transactions. Therefore, we conclude that the use of domain data is useful for the classification of crosscutting concerns.

Table 3. Algorithm Results

Method	FAN-in	Classification
UpdatePhysicalMeasures	10	Domain Entity
CreateResultCalibration	10	Domain Entity
NewMeasureCalibration	10	Domain Entity
SearchProductIndex	10	Domain Entity
SearchCalib	13	Domain Entity
SearchPatient	17	Domain Entity
PublishException	19	Aspect
ReadMesureCalib	22	Domain Entity
Trace	24	Aspect
SearchProduct	26	Domain Entity
SearchTestData	29	Domain Entity
DecryptData	35	Aspect
ReadRawResults	41	Domain Entity
PublishEvent	96	Aspect
ValidateTransaction	89	Aspect
GetGlossaryValue	127	Domain Entity
GetInstance	101	Aspect

5 Discussion

For the concern extraction activity, we do not consider it necessary to include statement-level local information (temporary variable accesses) because they are not directly representing main state accesses: A higher level abstraction of the program is more useful [13]. However, in our case study, classes were non-cohesive units that do not represent useful information for concern extraction. Thus, we include only methods and global variables.

One of the limitation of our work is the fact that it bases on the model of method invocation and FAN-in metric which assumes that there is a minimum of behavioral encapsulation in the form of methods and these methods represent a well-defined, crisp functionality. In situations where there is a haphazard, extensive scattering (*i.e.,* methods do not have sharp focus and data components do not have accessors) this approach will not produce any meaningful crosscutting candidates. Crosscutting can also occur in the form of code idioms to give rise to code clones [2], which fan-in wouldn't detect and hence possible combination of clones classes and domain entity data has to be combined to adapt the approach. We also suppose that programs generally represent domain entities through well defined, succinct global variables, which help to relate methods to concerns. In the absence of such variables, a manual effort is required to associate methods with concerns.

The validation that we performed is encouraging. However, we are aware that our case study while using industrial code was a data intensive software system. We plan to validate our approach on other kind of software showing clear lack of object-oriented design. We believe that our approach is suitable for software systems other than data intensive ones as soon as they show signs of poor object-

oriented design. But this will have to be proved.

6 Related Work

Aspect mining techniques automate the process of aspect discovery and propose to their user one or more aspect candidates based on code lexical information, static or dynamic analysis of an application [9]. FAN-in analysis determines the scattering of a concerns in program code by identifying methods that are called too frequently within program code [12]. Lexical analysis provides a hint about crosscutting concerns by the analysis of program tokens – either through aggregation of tokens, types or through Formal Concept Analysis of the tokens [7, 3]. Clone detection technique has been applied to an industrial C application in order to evaluate their effectiveness in finding the crosscutting concerns present in legacy software [2]. An aspect mining technique named DynAMiT (Dynamic Aspect Mining Tool) [1] has been proposed which analyzes program traces reflecting the run-time behavior of a system in search of recurring execution patterns. Tonella and Ceccato apply concept analysis [3] to analyze how execution traces are related to class methods and identify related methods as crosscutting concerns. All of the above mentioned aspect mining techniques do not take into account the crosscutting concerns originating from the absence of object-oriented design. Hence, our algorithm can be used to improve the existing techniques to distinguish diverse crosscutting concerns.

Concern identification and interaction through manual feature selection tool has been presented in [11]. Eaddy *et al.* [6] have presented a manual approach for concern identification and concern assignment. Features are located in procedural code by interactively searching for artifacts contributing to the implementation of a feature [4]. FEAT allows the user to interactively build Concern Graphs for object oriented programs [13]. These approaches remain nonetheless concern exploration tools and require their users to classify various identified concerns.

7 Conclusion and Future Work

Crosscutting concerns may appear due to non-abstracted domain logic as well as due to the shortcomings of object-oriented mechanisms to capture inherent crosscutting of concerns. Aspect mining techniques are capable of identifying diverse crosscutting concerns but are not capable to distinguish between them. In this paper, crosscutting concerns are originating from non-abstracted domain logic are identified according to their association to domain entities. The outcome of the approach is quite promising for automatic concern identification and their classification. To the best of our knowledge, the approach presented in the paper is the first one towards the distinction of diverse crosscutting concerns present in a software subsystem originating from the lack of elementary object-oriented design and absence of aspects. We validated our approach on an industrial application. However this approach has only been validated on a data-intensive system. This approach needs to be tested with further case studies, preferably involving processing-intensive systems, in order to better evaluate the results and refine the presented model.

References

[1] S. Breu and J. Krinke. Aspect mining using event traces. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[2] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[3] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *13th International Workshop on Program Comprehension (IWPC)*, pages 13–22. IEEE CS, 2005.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, pages 241–249. IEEE Computer Society Press, 2000.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] M. Eaddy, A. Aho, and G. C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[7] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, 3, 2000.

[8] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[9] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[10] G. Kiczales. *Aspect-oriented programming*. *ACM Computing Survey*, 28(4es):154, 1996.

[11] A. Lai and G. Murphy. The structure of features in java code: An exploratory investigation, 1999.

[12] M. Marin, L. Moonen, and A. van Deursen. Fint: Tool support for aspect mining. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 299–300, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.