

Mining and Classification of Diverse Crosscutting Concerns

Muhammad Usman Bhatti
CRI, Université de Paris 1 Sorbonne, France
muhammad.bhatti@malix.univ-paris1.fr

Stéphane Ducasse
INRIA, Lille Nord Europe, France
stephane.ducasse@inria.fr

ABSTRACT

Crosscutting concerns appear in software system due to the inherent inadequacy of OOP mechanisms to capture them in suitable encapsulating units. This results in scattered and tangled code. One more form of scattering and tangling may result from the absence of OOP abstractions for domain entities of a software. These non-encapsulated domain entities end up scattered and tangled, appearing as crosscutting concerns in code. Aspect mining techniques automate the task of search for possible aspects in the code and falsely attribute all the crosscutting code to aspects even when these scattered concerns point to the absence of a domain abstraction. This paper discusses the application of aspect mining in the presence crosscutting code originating from the absence of aspects and OOP abstractions. A roadmap of a possible solution is provided to distinguish these two types of code scattering.

Keywords

Reverse Engineering, Crosscutting Concerns, Aspect Mining

1. INTRODUCTION

The aspect oriented paradigm introduces new mechanisms to capture crosscutting concerns appearing due to the shortcoming of OOP mechanisms to capture such concerns and provides joinpoint, pointcut and advice mechanisms to encapsulate these concerns in a well-modularized fashion [6]. Aspect mining activity aims at automatically locating crosscutting concerns in the code and refactoring them to the constructs provided by AOP mechanisms for their better modularity.

Software development activity in enterprises is confronted with time to market and budget limitations, hence industrial software often lacks an elementary software design. Such absence of lack of design for software results in scattered and tangling of code related to non-encapsulated domain entities, resulting in crosscutting concerns. In such scenarios, aspect mining can result in a great number of false positives

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LATE Linking Aspect Technology and Evolution Workshop, April 1st, Brussels, Belgium

Copyright 2008 ACM 978-1-60558-147-7/08/04 ...\$5.00.

which may mislead a developer towards the overkill of aspects although the identified concerns are mainly pointing to the absence of certain OOP abstractions rather than AOP mechanisms.

We are working on an industrial system, developed in C#. Assigned with the task to improve the reusability of the the software, it was observed that the system lacks an elementary object-oriented design and there is a manifestation of crosscutting code due to missing abstractions for domain entities. In such scenarios, aspect mining techniques falsely attribute the crosscutting code to the absence of aspects.

This paper presents results of the two aspect mining techniques and examines diverse crosscutting concerns that are discovered, including those appearing due to the absence of abstraction of domain entities. Therefore, this paper examines the occurrence of diverse crosscutting concerns and an improvement in the classification provided by the existing techniques. In the later part, we provide a roadmap for the classification of diverse crosscutting concerns through the analysis of the usage of domain entities related data.

This paper is organized as follows: Section 2 presents our case study software along with a few OO metrics to demonstrate its characteristics. Section 3 provides an insight into the results provided by two aspect mining techniques that we used to mine aspects in our case study software. Section 4 provides an overview of our approach. Section 5 concludes the paper.

2. BLOOD ANALYSIS APPLICATION

The case study software is used to drive the machines for the analysis of patients for blood diseases. For the sake of precision and clarity, we shall only be talking about the software layer 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, raw and interpreted results, patient data, and quality control, are the key features implemented at this layer.

Table 1 below shows some of the software quality metrics for our case study business entity component. Lines Of Code (LOC) tallies executable instructions. Number Of Methods (NOM) and number Of Attributes (NOA) 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 [4].

Table 1 communicates some facts about the business en-

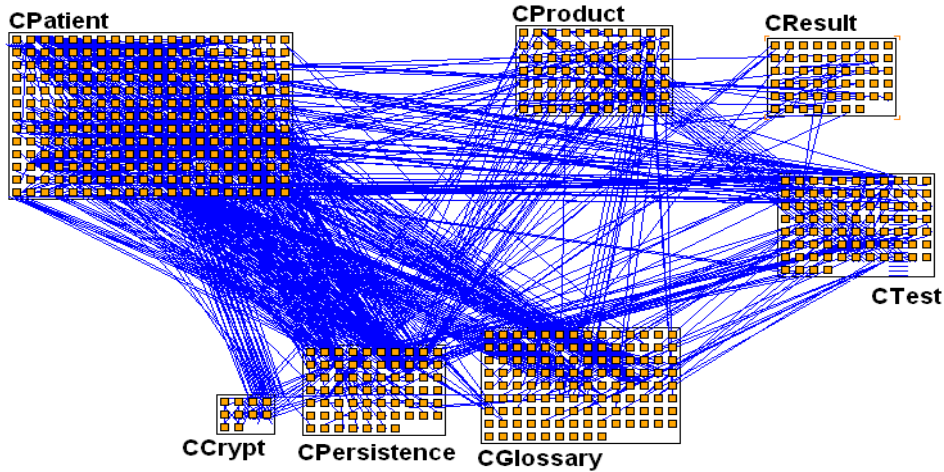


Figure 1: Inter-Class and Intra-Class Method Invocations

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
CResult	1652	52	13	1	0.85
CPersistence	1325	67	29	2	0.97
CGlossary	1010	121	5	1	0.80
CCrypt	267	10	4	1	0.32

tity 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 remarked that certain domain entities such as quality control and raw results do not have associated classes in the code (CResult class in the table contains the functionality to calculate interpreted results). Figure 1 demonstrates interclass and intraclass interactions between classes of the subsystem through method calls (a snapshot taken using MOOSE reverse engineering environment [8]). We term this type of code as *procedural object-oriented code* because of absence of OO design. We observed that 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 procedural object-oriented code raises the problem of the qualification of aspects which are identified by aspect mining. In the following we employ two aspect mining techniques to identify crosscutting concerns in procedural OO code.

3. ASPECT MINING RESULTS

Aspect mining techniques search for crosscutting concern candidates that can be refactored into aspects to enhance the modularity and changeability of programs [5]. In order to search for possible crosscutting concerns, we used two aspect mining techniques namely, Aspect Browser [3] and FAN-in metric [7]. In the following we present results associated to

the aspect mining techniques that we employed:

3.1 FAN-in Results

The principle behind the use of FAN-in as support 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 [7]. Since there were no tools computing FAN-in in C#, we developed our own tool based on 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 (pertaining to crosscutting concerns) identified by our tool and their FAN-in value along with their possible classification.

Table 2: Application methods and associated FAN-in values

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

Although, crosscutting concerns indicated the presence of

scattered code, almost 63% of the results pertained to the 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 (pertaining to the absence of aspects and non-abstracted domain entities) 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 with the usage of the application data to distinguish the type of the behavior being invoked and the type of logic the invoked method provides to its caller.

3.2 Aspect Browser Tool

The Aspect Browser tool is based on the assumption that crosscutting concerns can be discovered by identifying recurrent textual-pattern and lexical tokens. Aspect Browser calculates two metrics for the source code in question: Redundant lines and Most Common identifiers. The two metrics sometimes carry useless information such as comments and the keywords like “new” and “return”. A manual effort is required to filter and aggregate information relate to the identified concerns.

Table 3 demonstrates the concerns that are mined using the tool and the frequency of the occurrence of artifacts related to each concern along with a possible classification. Different lines and identifiers referring to the same concern in Aspect Browser are aggregated under the name of the referred concern in the table. The frequency of the concern is calculated by adding up the two metrics presented by the tool for each concern’s related lexical tokens and discarding the repetitive entries. Below we present a comparison of the results from the two tools.

Table 3: Crosscutting Concerns and their Frequency

Concern	Frequency	Possible Classification
PhysicalMeasures	37	Domain
Trace	40	Aspect
Events	91	Aspect
Singleton	118	Aspect
Glossary	222	Domain
Quality Control	240	Domain
Analysis	280	Domain
Transaction	300	Aspect
Product	350	Domain
Calibration	550	Domain
Patient	555	Domain
RawResult	750	Domain

3.3 Comparison of Results

The first and the foremost thing to be remarked in the results of the two is the presence of domain entities as crosscutting candidate. This is because of the lack of elementary design and scattered artifacts are pointing towards the components of various domain entities. Scattered artifacts identified by Aspect Browser are much higher in frequency than those identified by the FAN-in tool and also contain those identified by the FAN-in tool. This is because FAN-in detects crosscutting concerns through methods calls while

Aspect Browser searches lexical tokens. The results of the FAN-in are more precise. Aspect Browser lexical tokens may sometimes contain large number of false positives due to inclusion of comments, name similarities and the present of language-constructs.

A through investigation of the crosscutting candidates reveals that the application subsystem consists of two kinds of scattering: Data scattering and behavioral scattering. Aspect Browser is apt for detecting data scattering and FAN-in metric for behavioral scattering. We analyzed each of them and describe them below:

Data Scattering.

The absence of domain entity abstractions in our application caused two types of data components to appear in a subsystem: objects representing database tables and global enumerated types. Access and manipulation of this data is controlled by a set of methods. This separation of data and associated behavior is manifested in Figure 2. Hence, code related to following 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.

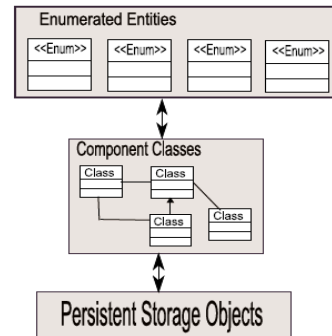


Figure 2: Separation of Data and Behavior

Behavioral Scattering.

Behavioral scattering means that two distinct behaviors are composed together in a single abstraction. In our component classes, this usually happens in the form of method calls, hence indicated by abnormal high FAN-in. Following are the scenarios for behavioral scattering to occur:

- 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 *divulged* into several client classes of the entity. This causes the client

classes to perpetually call the provider-logic, causing a high FAN-in value for logic-provider methods.

- A method may provide key or central information. For example, a method always passes through the patient data to get associated results and since result logic used is quite often, this results in access to patient information from all the client locations.
- 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.

Hence in the absence of elementary OO design crosscutting “seeds” identified by the FAN-in tool consist of the following types of methods.

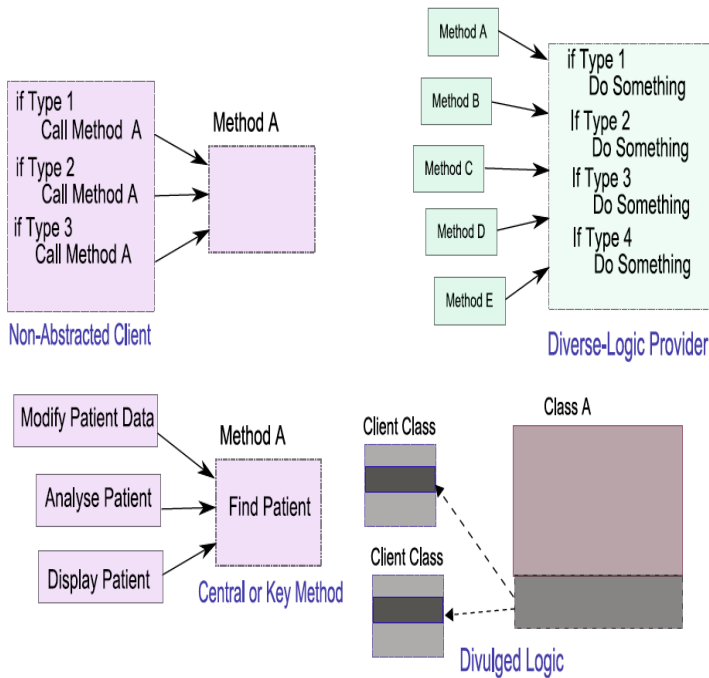


Figure 3: Classification of Crosscutting Seeds

- **Non-Abstracted Client.** A method being called from non-abstracted, duplicated code corresponding to domain entity subtypes or from a method providing multifarious behavior as demonstrated in Figure 3. This is a case of missing use of polymorphism [1].
- **Diverse-Logic Provider.** A method providing logic for the same entity subtypes, as depicted in Figure 3. This occurs because of the absence of common code or template behavior in a parent class method.
- **Central Method.** A method may provide central or key information such as patient search function which is used from various other methods.
- **Divulged-Logic.** This lack of abstraction and encapsulation happens when the behavior of an object is not

defined in a specific class but spread into client classes as shown in Figure 3.

- **Utility Method.** A method providing utility functions such as conversion parameters.
- **Aspects.** A method providing scattered technical behavior such as transaction, logging, and exception handling.

In the following section, we present a roadmap for an approach for the classification of diverse crosscutting concerns by incorporating information extracted from the use of variables representing domain entities.

4. TOWARDS A CLASSIFICATION APPROACH

To identify crosscutting concerns appearing due to missing domain entity abstractions, we define a model based on application data usage and resolution of associated behavior. We make the hypothesis that the data is placed far from its corresponding behavior due to the absence of associated domain classes, which causes crosscutting concerns. This means that in order to extricate the subset of crosscutting concerns appearing due to absence of abstraction for domain entities, we need to identify the data and its associated behavior. This approach helps in noise removal from aspect mining results.

4.1 Domain Entity Model

As previously mentioned, the data in our case study mainly consists of the representation of various domain entities and variables for global state and entity subtypes. 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. This is done by resolving the data accessed by each method. 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.

4.2 Aspect Model

As defined earlier, it is assumed that crosscutting concerns also appear due to the absence of appropriate OOP mechanisms to compose two intersecting behaviors in a non-recurrent way. It is noted for our subsystem that interleaving and composition of distinct behavior has been performed through method calls because of the application of refactorings proposed to encapsulate clone code in methods [2]. Hence, we base our crosscutting identification model on the FAN-in metric [7] (*i.e.*, the higher the number of calls to a method, the more the chances are for it being a crosscutting concern). It seems reasonable because of the fact that a comprehensive amount of aspect mining techniques search for the occurrence of scattered and tangled method calls to detect crosscutting behavior [5]. But we only consider those methods which do not directly or indirectly relate to domain entities.

4.3 Algorithm for Concern Classification

Once all the domain entity methods are ascertained, a simple algorithm is defined to distinguish various crosscutting concerns discovered in the subsystem by the FAN-in tool. 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.

4.4 Results for Classification Approach

Table 4: 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

The results for the crosscutting concern classification are presented in Table 4. 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.

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. In addition it corresponds well with the established aspect candidates described in literature such as tracing, exception handling and transactions. Therefore, use of domain data is useful for the classification of crosscutting concerns.

At first, huge number of “GetGlossaryValue” method calls to the glossary concern may indicate that its classification as domain entity is a false positive of the approach and that aspects may provide a better encapsulation for such a scattered concern. But a profound look at this concern reveals that the glossary actually replaces the absence of various classes (types and subtypes) representing domain entities. Glossary can easily be removed by introducing the appropriate classes (types) for each domain entity and using *typeof* operations to store their type.

5. CONCLUSION AND FUTURE WORK

Two types of crosscutting concerns are actually identified by aspect mining techniques: lack of OO design and shortcoming of OO mechanisms to cleanly encapsulate some concerns. The former results from the scattering of data and its associated behavior while the latter is an aftermath of the

recurrent composition of two distinct behaviors. The paper describes an experience of aspect mining activity with two different tools and their results. It turns out that crosscutting concerns are identified with the two tools we used without any distinction for the origin of crosscutting behavior. We described an approach to distinguish two types of crosscutting concerns by usage of domain entity data by program methods. Our is the first approach to distinguish crosscutting concerns springing from different origins. We aim to study and understand the efficacy of the simultaneous application of OO restructuring and aspect refactoring to remove crosscutting concerns.

6. REFERENCES

- [1] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [3] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, 3, 2000.
- [4] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
- [5] 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.
- [6] G. Kiczales. *Aspect-oriented programming*. *ACM Computing Survey*, 28(4es):154, 1996.
- [7] M. Marin, A. v. Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.
- [8] M. Meyer, T. Gırba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis’06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.