A Pragramatic Approch to Mine Aspects 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 scattered and tangled code related to domain entities because their data and the associated behavior do not share the same abstraction.

Aspect mining techniques search for various patterns of scattered and tangled code pertaining to crosscutting concerns and associated them with aspects. However, in the presence of non-abstracted domain logic, the crosscutting concerns identified are inaccurately related to aspects even when they are indicating lack of OO abstraction.

This paper identifies the difficulty to mine aspects in object-oriented software systems suffering from procedural thinking. It presents an approach based on FAN-in analysis and aspect related software metrics to distinguish real *aspects from simple lack of object-oriented decomposition. We present a metric-based evaluation of the crosscutting concerns and a new metric called Spread-out to calculate the spread of crosscutting concerns. The approach presented in the paper has been implemented and validated on industrial case studies.*

1 Introduction

Software reusability is highly desirable and can directly be associated to modular continuity [18], which suggests that a small change in a problem specification triggers a change of just one module, or a small number of modules. Yet it happens that software developed, even using object-oriented paradigm, is full of scattered code across various classes, hindering software reuse and incurring high maintenance costs. The problem of scattered code has principally been treated in the domain of aspect mining [13, 17]. The underlying assumption in most of the existing work in this domain is that scattered and tangled code in object-oriented systems origins only from the tyranny of dominant decomposition [14], 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 [7]. 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, aspect mining techniques to discover crosscutting concerns present in a software system are inadequate in the presence of *non-abstracted domain logic* as they wrongly associate the lack of object-oriented structure with aspects. This occurs because 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. Class hierarchy reengineering, object identification and software refactoring do not target the occurrence of scattered and tangled code phenomenon due to missing abstractions. Secondly, they do not take into account those concerns that can benefit from the usage of AOP mechanisms. Aspect mining techniques, as aforementioned do not distinguish amongst various crosscutting concerns those resulting from the absent OO abstractions.

Most of the existing publications in the domain of software restructuring for object-oriented software target can be divided into four categories:

- Class hierarchy reengineering through the usage of attributes and methods of classes making up the system and grouping those which are used together [19, 4, 8,

23, 1].

- Software refactoring through the *manual* identification of *small* design problems within class hierarchies and provides various heuristics for their rectification [10, 7].

- Identification of classes and objects in legacy code through the identification of common usage of data by various methods [22].

- Aspect mining to refactor crosscutting concerns in aspects [13, 17, 5].

Henceforth, we introduce an approach for the identification of diverse crosscutting concerns present in the system. The approach presented in this paper identifies and groups 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 and extracted from through their usage of application data. We also present a metrics-based evaluation of all the crosscutting concerns identified in our case study software, in order to quantify and study their scattering in software classes. A new metric called *spread-out* is introduced to quantify the divulgence of diverse crosscutting concerns.

The contribution of the paper are

- the identification of aspect mining limits in presence of industrial object-oriented code showing strong signs of procedural thinking,

- the use of FAN-in to distinguish real aspects from lack of object-oriented design,

- the definition of a new metric to qualify the scattering of cross-cutting concerns.

This paper is organized as follow: Section 2 describes the context and overview of our case study software. Section 3 discusses the results of the aspect mining technique we used to mine crosscutting concerns and the occurrence of domain entities in the list of crosscutting concerns. Section 4 presents our approach for the identification and classification of concerns and the results that are obtained with the application of the approach to our case study software. Section 5 presents the scattering of concerns through the light of concern scattering metrics proposed in the literature and one new metric to quantify concern spread. Section 6 discusses the results obtained from our approach and that of concern scattering metrics. Section 7 talks about the related work and Section 8 concludes the paper.

2 A Case Study: a Blood Analysis Application

The case study software is used to drive the machines for the analysis of patients for blood diseases. The machine is loaded with a sample of patient plasma and reagents (products) for chemical reactions. The machine performs the analysis and raw results are calculated and converted to interpreted results for their easier interpretation by doctors and medical staff.

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. This is one of the software subsystems that is replicated on each new machine and it is thus beneficial to analyze and improve its design. 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 [12].

Table 1. Case Study Metrics

Component 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 remarked 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). We term this type of code as *procedural object-oriented code*. Procedural object-oriented code consists of so called *half-balked* objects: objects encompassing other objects. For example, the class CPatient, in addition to its core functionality, contains logic for raw result calculation and calibration related functionality. 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 one aspect mining technique to examine the crosscutting concerns identified in procedural OO code.

3 Evaluating FAN-in Aspect Mining Technique

Aspect mining techniques automate the task of the identification of scattered code in software systems by enlisting a set of corresponding crosscutting concern "candidates". These can be identifiers and redundant lines of code, code clones, metrics, etc. [13]. We employed FAN-in technique for mining crosscutting concerns in our case study software [17]. The principle of 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 [5].

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 [5]).

Although, crosscutting concerns indicated the presence of scattered code, a good amount 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.

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

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

4 Concern Classification

It is important to identify and distinguish the various kinds of crosscutting concerns to propose an adequate remedy. For this purpose, we propose an identification for those crosscutting concerns appearing due to lack of abstraction for domain entities. 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.

4.1 Data and Behavioral Scattering

We make the hypothesis that data is placed far from its corresponding behavior due absence of associated domain classes resulting in crosscutting concerns. This means that

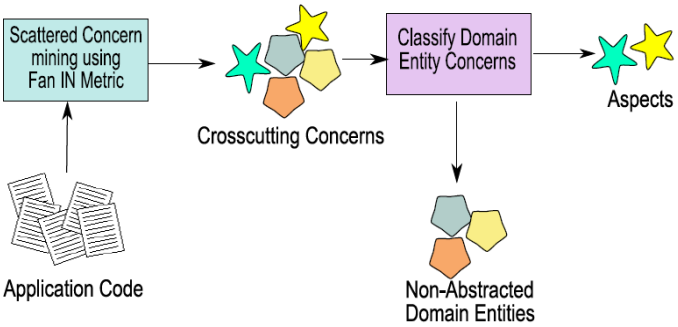

Figure 1. Concern Classification Approach

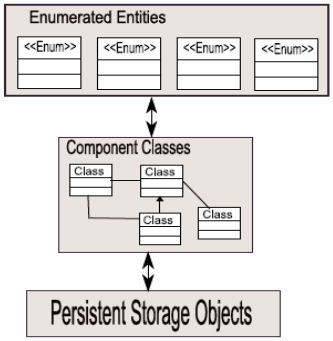

Figure 2. Separation of Data and Behavior

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 once disentangled from the overall set of crosscutting concerns removes the noise from aspect mining candidates.

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. The access to this data is performed through their accessors from the methods implementing their behavior (See the pattern Move Behavior close to Data [7]). This dissociation 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.

Behavioral Scattering. Behavioral scattering means that two distinct behaviors are composed together in a single abstraction. This happened very frequently in our subsystem. 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.

4.2 Model for Concern Classification

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. This model takes into account all the methods and data components present in the application's subsystem. We define M as a set of all methods in component being analyzed. T is defined as a set of all objects 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. As previously mentioned, the data in our case study 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 our case, database tables have clear one-to-one association

with the domain entities. Secondly, 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. This is done by resolving the data accessed by each method.

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 3, 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".

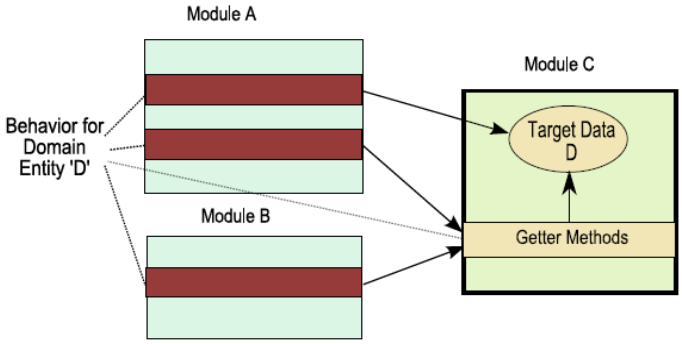

Figure 3. Domain Entity Concern Identification

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. 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 [10]. Hence, we base our crosscutting identification model on the FAN-in metric [17] (*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 [13]. 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 4. In the following, a model is introduced to ascertain methods implementing domain entity related concerns are identified

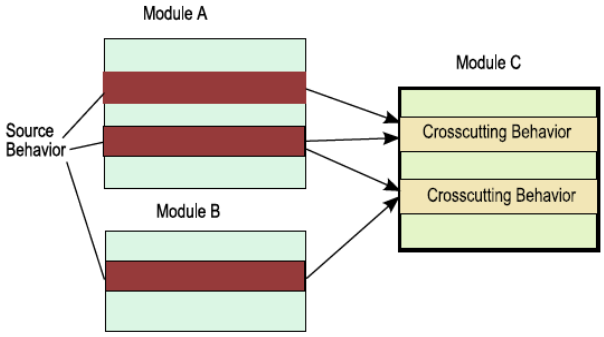

Figure 4. Aspect Identification

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 any sharp focus.

4.4 Algorithm for Concern Classification

We now define a simple algorithm 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.

$\{M\} \leftarrow \forall$ *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 $n \in \{M \cap CCSeeds\}$
$\{CC\} \leftarrow n$

4.5 Results for Classification Approach

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

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.

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 more profound look at this concern reveals that the glossary actually replaces the absence of various classes (types) representing domain entities. Glossary takes enumerated types and their associated values for domain entity types as input and stores them as concatenated string in database tables that corresponds to the domain entity type indicated by the value. This happens because their is no provision to store enumerated types directly into the database. Glossary can easily be removed by introducing the appropriate classes (types) and sub-classes (sub-types) for each domain entity and using *typeof* operations to store their type into the database.

Now that we show that the use of domain data was a good indicator for classifying concerns, we want to go a step further: the concern classification tags all methods related to domain entities and hence helps distinguishing different crosscutting concerns. It is interesting to understand and examine the concerns identified so far using scattering metrics proposed in the literature. This activity may provide useful information such as the scattering patterns and extent of scattering of crosscutting concerns. We want to see if this is possible to verify the results from the classification approach defined earlier. This would also help understand the alignment of non-abstracted domain logic and aspects with respect to the class boundaries as they are defined in our subsystem. In the next section, various crosscutting concerns of our subsystem are studied through the light of concern scattering metrics.

5 Scattering Metrics of Crosscutting Concerns

To understand the scattering of the different crosscutting concerns, we examine them using concern scattering quantification metrics [15, 9]. Since these metrics were not defined in the context of procedural object-oriented code we adapted them using the model we presented in Section 4.3

Three metrics quantify the dispersion of various artifacts of a program in application classes: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO) and concern diffusion over Lines of Code (CDLOC) [15]. Since we tag methods with concerns for our approach, hence we employ the CDO metric to understand the scattering of crosscutting concerns in our subsystem. Concerns diffusion over operations counts the number

of methods whose main purpose is to contribute to the implementation of a concern and the number of other methods. Hence CDO for a concern c is:

$CDO(c)$ = *number of implements(m,c)*

Eaddy et al. argued that CDO and other metrics from the same suite only discern the presence of scattering but not their extent [9]. Concern extent is defined by two metrics: Degree of Scattering (DOS) and Degree of Focus (DOF) [9]. However, these metrics are calculated by manually marking each line of code for a particular concern. The manual concern calculation is a laborious task for large software systems. We redefine the Concentration (CONC) and DOS metrics defined in [9] to include only methods, instead of lines of code, to calculate these metrics for the crosscutting concerns of our subsystem. We relate the concentration to our model in the way described below.

$$CONC(c,t) = \frac{\textit{Number of } implement(m,c) \textit{ in class } t}{\textit{Total } implement(m,c)}$$

$$DOS(c) = 1 - \frac{|T| \sum_t^T (CONC(c,t) - 1/T)^2}{|T| - 1}$$

Hence, CONC calculates the ratio of the number of methods implementing a particular concern in a class to the total number of methods contributing to the implementation of the concern: higher the concentration, the more a concern is concentrated in a component. DOS is a measure of the variance of the concentration of a concern over all components with respect to the worst case (*i.e.,*, when the concern is equally scattered across all classes). DOS values can lie between 0 and 1: 0 indicates that a concern is completely localized whereas 1 indicates uniform distribution of a concern over all classes.

To show the spread of each concern over other classes, we introduce a metric called *Spread-Out*. It represents the ratio of the operations, associated or contributing to a given concern, located outside the main component implementing the logic for the concern. For example, Spread-Out for the Patient Records concern identifies the ratio of the methods implementing Patient Records outside the class CPatient. This helps us discern the dispersion for that concern over other subsystem classes. Note that there are some concerns which do not have a dedicated class associated with them; in such a case we consider the class containing the largest number of operations associated to that concern as its home location. We define $spread\text{-}out(c)$ of a concern c as follow:

$$spread\text{-}out(c) = \frac{implement(m,c) \textit{ outside principal class}}{\textit{Total } implement(m,c)}$$

The values for the scattering metrics (CDO, DOS and spread-out) for various concerns of our case study are provided in Table 4 (ded. class indicates if there is a dedicated class for the given concern).

Figure 5 shows the distribution for the Concern Diffusion over Operations metrics (CDO). We can spot that the entities which do not have a dedicated class have higher CDO such as Calibration and Raw Results. On the contrary, Patient Tube, even in the absence of a dedicated class, has a lower CDO because it represents a smaller concern with small number of associated operations and interpreted results have higher CDO even in the presence of a dedicated class. Consequently, we deduce from the graph that CDO is a relative metric which is useful to compare concerns of the same size and with the same number of associated operations. Patient Tube data and Tracing are relatively small concerns and hence, in the presence of large concerns, such as Calibration and Transactions, they tend to present smaller CDO, and hence, it is an error to infer that they are well-encapsulated. Secondly, it doesn't indicate the extent of encapsulation for a concerns *i.e.,* whether the number of operations contributing to a concern are located inside the class implementing the concern or outside it.

Figure 6 shows the comparison between the DOS and Spread-out metrics; we see that all the concerns are scat-

Table 4. Concern Scattering Results

Concern	Ded. Class	CDO	DOS	Spread-out
Raw Results	No	147	0.31	0.12
Patient Records	Yes	99	0.50	0.25
Quality Control	No	39	0.55	0.21
Calibration	No	234	0.52	0.25
Analysis	No	129	0.26	0.09
Interpreted Results	Yes	228	0.68	0.43
Patient Tube	No	54	0.78	0.57
Glossary	Yes	254	0.73	0.50
Transactions	Yes	94	0.58	0.94
Tracing	Yes	26	0.47	0.93
Singleton	Yes	108	0.54	0.94
Events	Yes	110	0.57	0.88

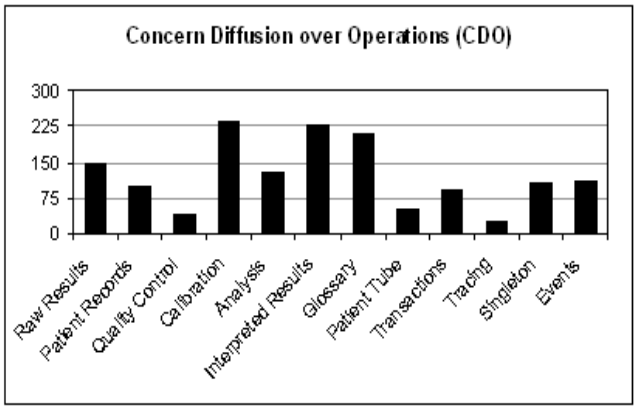

Figure 5. Concern Diffusion over Operations for Concerns

tered with varying degrees. Spread-out and DOS metrics also better depict the degree of scattering of the Patient Tube vis-à-vis Raw Results than CDO metric because the size of the concerns is normalized. DOS, in general demonstrates the average scattering of various concerns over classes, and all concerns have ~ 0.50 DOS. This depicts the terrible state of encapsulation for various concerns and hence provides a measure of scattering of concerns and their tangling with other concerns. Spread-out demonstrates that although some of the concerns do not have an associated class, they still are concentrated in one of the subsystem classes, for example Analysis and Raw Results.

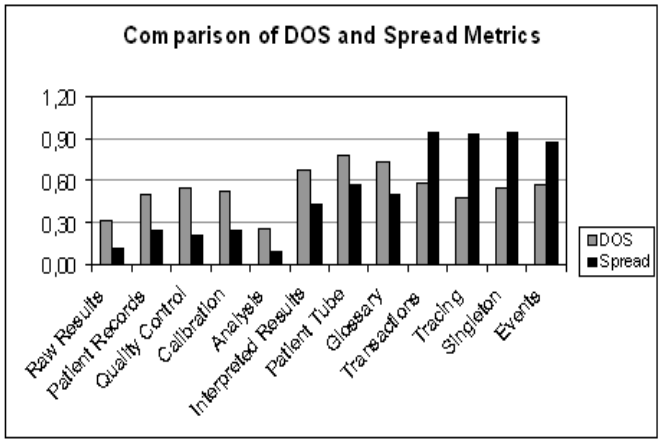

Figure 6. Comparison of DOS and Spread-out

Regarding the Spread-out values in Figure 6, it demonstrates that first that there are two kinds of Spreading-out: one part of the crosscutting concerns such as Raw Results, Patient Records, Analysis presents a low Spreading-out value. The other part of the crosscutting concerns has a high spreading-out value such as Transactions, Tracing, Singleton. This indicates two kinds of different situations which correspond well with our classification of domain entities and aspects: domain entities have lower Spread-out. On the other hand, Tracing, Singleton instances, Events and Transactions represent typical examples of aspects and their Spread-out value is much higher. Consequently, Spread-out helps better identify the cause of crosscutting concerns than DOS and CDO: concerns exhibiting Spread-out values of more than 0.8 over the other components can be considered as aspects. And it is this anomalously extensive spread of such crosscutting concerns over other classes that one wants to capture in aspects for their improved modularity.

6 Discussion

It turns out that our hypothesis about the data and behavior scattering in procedural object-oriented system and crosscutting concerns in general turns out to be quite substantiative because of the precision of the resulting methods that appeared in the two categories. That is, most of the domain entity-related methods that were marked as crosscutting concerns have been distinguished as occurring due to lack of elementary OO design.

It is striking to see the Spread-out of Singleton instances which is nearly equals to 94%. This high value of spread-out occurs because the code represents a procedural thinking therefore huge classes are written and accessed in a procedural manner. Secondly, since most of the class mostly implement unrelated concerns, the scattered concerns are composed recurrently through singleton instances, causing their spread in client classes.

.

About the information used. For the concern extraction activity, we do not consider it necessary to include statement-level local information because there may be certain elements such as temporary variable assignments which may not be required: A higher level abstraction of the program is more useful [21]. However, in our case classes are 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, in such cases 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 [3], 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 with concerns. In the absence of such variables, a manual effort is required to associate methods with concerns. In general, automatically identifying and classifying crosscutting concerns out of completely unstructured code is near to impossible, if not impossible because in such cases there is no anchor point to start the automatic search for possible candidates. In such scenarios, Dynamic Analysis techniques can be helpful to locate crosscutting features and concerns.

7 Related Work

Aspect mining techniques automate the process of aspect discovery and propose their user one or more aspect candidates based on lexical information of the code, and static or dynamic analysis of an application [13]. FAN-in analysis determines the scattering of a concerns in program code by identifying methods that are called too frequently within program code [17]. 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 [11, 5]. 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 [3]. An aspect mining technique named DynAMiT (Dynamic Aspect Mining Tool) [2] 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 [5] to analyze how execution traces are related to class methods and identify related methods as a crosscutting concerns. All of the above mentioned aspect mining techniques do not take into account the crosscutting concerns originating from the absence of OO design. Hence, our algorithm can be used to augment the existing techniques for distinguishing diverse crosscutting concerns.

Concern identification and interaction through manual feature selection tool and a first set of metrics for feature scattering have been presented in [16]. Primitive metrics spread, tangle, and density are provided for feature concentration in diverse program files. Wang et. al. undertook a study similar to ours to compute, using dynamic analysis of code, various features implemented in code and calculate the relationship of these features with program components through disparity, concentration and dedication metrics [24]. Eaddy et. al. have presented a manual approach for concern identification and concern assignment and presents two concerns quantification metrics derived from [24]: degree of scattering and degree of focus [9]. The proposed approach promotes manual identification of concerns and doesn't mention crosscutting concerns arising due to non-abstracted domain entities. Garcia et. al have also provided a set of metrics to measure the scattering of crosscutting concerns: CDC, CDO, and CDLoc [15]. The assignment of code artifacts to concerns is manual. An experience of marking concerns in two programs with a tool SPOTLight has been presented in [20]. This tool serves to associated code snippets with various concerns and concern markings of the two developers have been compared by analyzing the overlap of the lines of code of their respective concerns. The work identifies useful guidelines for concern identification, nonetheless manually. Features are located in procedural code by interactively searching for artifacts contributing to the implementation of a feature [6]. A search graph for a feature is constructed and consists of global variables and functions related to the feature. Feature Analysis and Exploration Tool (FEAT) allows the user to interactively build Concern Graphs for object oriented programs. Concern Graphs consist of object oriented artifacts such as classes, methods, attributes and method calls implementing a concern [21]. Search Graphs and Concern Graphs help in program comprehension through exploration of code, and mining of concerns and features requires user input.

8 Conclusion and Future Work

Software reuse is hindered by the presence of scattered code resulting due to the lack of knowledge of object-oriented techniques and owing to the absence of appropriate decomposition mechanisms. Crosscutting concerns may appear due to non-abstracted domain logic as well as due to the shortcomings of OO 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. We have also evaluated the existing metrics to quantify the concern scattering and introduced a new metric called Spread-out metric to support results from the classification approach. The metric helps discern the amount of behavior of a concern that has been divulged in client classes. 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 OO design and absence of aspects. We validated our approach on an industrial application, still this approach needs to be tested with further case studies in order to better evaluate the results and Spread-out metric. Secondly, finding domain-related aspects remains to be seen with this approach and its verification with metrics.

References

[1] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 62–71. IEEE Computer Society, Mar. 2005.

[2] 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.

[3] 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.

[4] E. Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, Dec. 1994.

[5] 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.

[6] 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.

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

[8] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications)*, pages 251–267. ACM Press, 1996.

[9] 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.

[10] 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.

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

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

[13] A. Kellens and K. Mens. A survey of aspect mining tools and techniques. Technical report, UCL, Belgium, June 2005.

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

[15] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

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

[17] 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.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[19] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications)*, pages 235–250. ACM Press, 1996.

[20] M. Revelle, T. Broadbent, and D. Coppit. Understanding concerns in software: Insights gained from two case studies. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.

[21] 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.

[22] H. A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. In *Proceedings of ASE '97 (12th International Conference on Automated Software Engineering)*, pages 210–218. IEEE, IEEE Computer Society Press, Nov. 1997.

[23] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[24] E. Wong, S. Gokhale, and J. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.