History-based Approach for Detecting Modularity Defects in Aspect Oriented Software

Hanene Cherait and Nora Bounour
LISCO Laboratory, BadjiMokhtar – Annaba University P.O. Box 12, 23000 Annaba, Algeria
E-mail: {hanene_cherait, nora_bounour}@yahoo.fr

Keywords: modularity defects, aspect oriented programming, crosscutting concerns, frequent itemset mining, logical coupling, refactoring.

Received: May 4, 2014

The evolution of Aspect oriented (AO) software would degrade and modify its structure and its modularity. In this scenario, one of the main problems is to evaluate the modularity of the system, is the evolved AO software still has a good modularity or not? Unfortunately, this research area is not explored yet. This paper presents a history-based approach that detects modularity defects in evolved AO software. It is a two-step automated approach: 1) in the first step, it applies data mining over an AO software repository in order to detect logical couplings among its entities. It analyses fine-grained logical couplings between AO software entities as indicated by common changes. 2) These last are then analysed to detect modularity defects in the AO software system. The approach focuses on the evaluation of an AO system's modularity and points out potential enhancements to get a more stable one. We provide a prototype implementation to evaluate our approach in a case study, where modularity defects are detected in 22 releases of three well-known AspectJ systems: Contract4J, Health-Watcher and Mobile-Media. The results show that the approach is able to detect logical couplings among aspect entities, as well as modularity defects that are not easily (or not) detectable using static source code analysis.

Povzetek: Članek se ukvarja z zaznavanjem defektnosti modulov med evolucijo programov.

1 Introduction

Aspect-oriented programming (AOP) [11] allows a developer to modularize a crosscutting concern's implementation by introducing a new kind of module called "Aspect". This last encapsulates crosscutting concerns and thus improves modularity, understandability, and evolvability of the code. As any software system, AO systems are continuously modified and increase in size and complexity. After many enhancements and other evolution activities, the AO software modularity can be violated, and modifications become hard to do. The insufficient modularity of crosscutting concerns complicates AO software evolution and reduces crosscutting concern reusability. Therefore, methods and techniques are needed to detect modularity defects in AO software, in order to improve its decomposition and enhance its modularity.

To detect modularity defects in AO software, we need to understand the relationships among entities that belong to software aspects, more specifically, to the crosscutting concerns of the system. However, many works [2, 5, 20] have proved with empirical evidence the ripple effects in AO software i.e. changes are propagated to unrelated entities in the program. So, it is difficult to detect modularity defects in AO software through a static analysis of the source code (e.g. [26]). In reality, two crosscutting concerns that are supposed to be independent statically may frequently change together.

This paper presents a history-based approach to detect modularity defects in AO software. It consists of two main steps: first, an AO software repository is mined to detect logical couplings between the aspect's entities of the system—how entities actually change together. In our approach, we don't detect coupled aspects only. But, we can extract the aspect entities related to this coupling. Second, the resulted logical couplings are analysed to detect modularity defects. We identify modularity defects by external logical couplings i.e. if two entities always change together to accommodate modification requests, but they belong to two independent aspects; we consider this as a modularity defect. These last can be used to improve the AO software modularity in order to prevent it from decay. For example, the detected defects could be removed or minimized by using appropriated refactorings to change the AO software decomposition.

The rest of the paper is organized as follows: in the next section we give the background used in this paper. We describe our approach in section 3; where we present the relationship between logical couplings and modularity defects, and how this relationship can be used to detect modularity defects in AO software. The tool chain is presented in section 4. Our approach is applied on a case study in section 5. Section 6 summarizes related work. Finally, section 7 closes with conclusions and future work.

2 Background

In this section, we first introduce definitions of important concepts related to our proposal. Then, we give a brief description of the AspectJ language.

2.1 Modularity defects

The IEEE Standard Glossary of Software Engineering Terminology (IEEE, 1990) defines modularity as "the degree to which a software system is composed of discrete components such that a change to one component has minimal impact on other components". So, it allows each part to be modified, substituted or deleted with minimal impact on the rest of the system. Modularity has been playing a pervasive role in the context of software development and evolution. It can be considered a fundamental engineering principle as it allows:

- to develop different parts of the same system by distinct people;
- to test systems in a simultaneous fashion;
- to substitute or repair defective parts of a system without affecting with other parts;
- to reuse existing parts in different contexts; and
- to restrict change propagation.

In reality, however, during software evolution two modules that are supposed to be independent may always change together, due to unwanted side effects caused by quick and dirty implementation [24]. When such couplings exist, the software can deviate from its designed modular structure, which is called a *modularity defect (violation).* Such modularity defects could cause modularity decay over time and may require expensive system-wide refactorings. Detecting and fixing modularity defects make programs easier to understand and to evolve.

2.2 Logical coupling

Semantically coupled software entities may not structurally depend on each other i.e. different entities of a software system may be related to each other although this relationship is not easily detectable in the software source code. When different entities of a software system change together (as the system evolves) their common behavior is referred to as logical coupling [7]. Recently, researchers have used revision histories to more effectively identify semantically coupled components by checking how components historically change together [9, 10].

Logical couplings detection extract interesting dependencies between software entities that is not possible with the analysis of a single version. So, based on the historical data we can detect logical couplings between the entities of a software system. In this last, two entities are coupled whenever a change in an entity A implies a change in another entity B—one says that B depends on A.

The logical couplings have been used for different purposes: to identify hidden architectural dependencies, to point developers to possible places that need change, or to use them as change predictors. In our context, we use such dependencies to evaluate the modularity of an AO software system.

2.3 AspectJ

AOP is a new paradigm introduced by Kiczales et al. [11] that provides separation of crosscutting concerns. It modularizes the crosscutting concerns in a clear-cut fashion, yielding a system architecture that is easier to implement, and to evolve. With AOP, a program is composed with a set of aspects, and a base code describing the core modules. An *aspect weaver*, which is a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called *weaving* [12].

AO languages offer abstractions for the implementation of crosscutting concerns whose modularization cannot be achieved by using traditional programming languages. During the last decade, a considerable number of AO languages have been introduced. AspectJ [12] has been the pioneer of the AO languages, and it is still one of the most relevant frameworks supporting the AOP methodology. For the remaining of this paper, we will use AspectJ as our target language, although the observations made are also valid for other currently available AspectJ-like languages. AspectJ defines two types of crosscutting: *dynamic* crosscutting and *static* crosscutting.

Dynamic crosscutting: is the weaving of new behaviour into the execution of a program using: *join point*, *pointcut* and *advice.* We briefly introduce each of these constructs as follows:

- **Join Point:** denotes points at which crosscutting code can be executed. The join point is a well-defined "point" in the dynamic execution flow of an application. For instance, in object oriented languages, join points may refer to passing messages and writing on instance variables.
- **Pointcut:** is a program element that picks out join points and exposes data from the execution context of those join points. The pointcut language of AspectJ offers a set of *primitive pointcut designators*, like call specifying method call or get/set specifying field access. These primitive pointcut designators can be combined using logical operators (and "&&", or "||", not "!").
- **Advice:** represents a program module which is to be executed at the designated join points. There are three types of advices *before*, *after* and *around*, which correspond to the program modules to be executed prior, after or instead of the designated events, respectively. It is defined in terms of pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut.

Static crosscutting: is the weaving of modifications into the static structure—the classes, interfaces, and aspects—of the system. By itself, it does not modify the system behavior, but it operates over the static structure of type hierarchies. AspectJ provides inter-type member declarations (introductions) and other declare forms. It makes static changes to the modules of the system, for example, we can add a method or field to a class.

Finally, an Aspect is a modular unit designed to implement a crosscutting concern. It contains the code that

expresses the weaving rules for both dynamic and static crosscutting. An aspect may also incorporate member variables, methods, etc., just like a normal class Java.

3 Our approach

3.1 Basic idea

To understand better our contribution, it is important to define clearly the relationship between logical coupling detection and modularity defects. In this section, we present the utility of logical couplings in the detection of software modularity defects. And, we explain how this idea can be used in the context of AO software.

There is a strong correlation between modularity defects and logical couplings. Some modularity defects are not easily detectable by static or dynamic software analysis. Fluri et al.'s [7] study shows that a large number of change coupling relationships are not entailed by structural dependencies.

Extracting logical couplings and analysing them can help in detecting modularity defects in a software system. The basic idea is that we can distinguish two types of logical couplings, as depicted in Figure 1: internal and external logical couplings. A logical coupling is an internal logical coupling, if it relates two entities that belong to the same module in the software decomposition. On the other hand, an external logical coupling relates two entities that belong to two different (independent) software modules.

The last type is the most important in our context; because the existence of external logical couplings in a software system presents possible modularity defects in that system. Two modules that are supposed to be changed independently are changed together i.e. a change in an entity that belong to a specific module will necessitate changes in other(s) entity(s) that belong to other module(s). So, we call such logical couplings "negative logical couplings" or "modularity defects".

In AOP, the crosscutting concerns are modularized by identifying a clear role for each one in the system, implementing each role in its own module, and loosely coupling each module to only a limited number of other modules [12]. Unfortunately, these systems need to evolve continually in order to cope with ever-changing software requirements. Empirical results show that AO software is not immune from the negative side effects of software evolution [2, 5, 20]. This fact harms the modularity of the AO program, hinders the concerns encapsulation and reduces the aspect reusability. To overcome this problem is a hot topic.

Our research question is: how we can detect efficiently the modularity defects in AO software? To this end, we use the idea described above to achieve our goal. In this context, software modules (Figure 1) are the crosscutting concerns (Aspects) of the system, and the modularity defects are considered as external logical

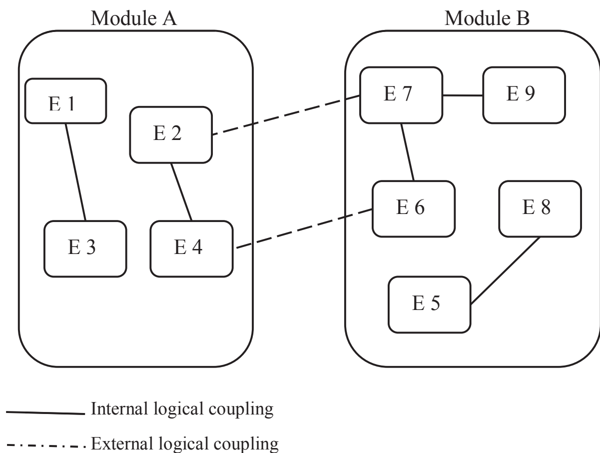

Figure 1: Types of logical couplings.

couplings among these aspects. So a modularity defect in AO software can be defined as follows:

Definition of a modularity defect: let A and B two independent aspects. A modularity defect (x, y) is a logical coupling between the two entities x and y, where $x \in A$ and $y \in B$.

To resume up, just how well does the AO software system evolution justify its best modularity? The existence of modularity defects (external logical couplings) in an AO system shows that the separation of crosscutting concerns (modularity) into that system is violated. The coupled crosscutting concerns are candidates for restructuring or refactoring. Here, the detected modularity defects are used to guide improvement efforts; in order to get a more stable decomposition with very little dependencies i.e. an ideal situation would allow changing each crosscutting concern independently of the others. This is very useful to reconstruct a best modularization for the AO software system and a good reusability of their crosscutting concerns.

3.2 Approach overview

The purpose of this paper is to present an approach to uncover modularity defects in AO software by analysing its evolution history. As depicted in Figure 2, our approach consists of two complementary steps, which form an integrated approach for detecting modularity defects: 1) An AO software repository is mined to detect logical couplings between the software entities that belong to the different aspects of the system; 2) The resulted logical couplings are then analysed according to the AO software decomposition to detect and locate modularity defects. If two entities x and y are frequently changed together, and they belong to two independent aspects A and B respectively, so the logical dependency (x,y) represents a modularity defect. The main purpose of such modularity defects is to evaluate how modular an AO application is, and to guide improvement efforts i.e. these couplings can be used to guide the software developer during restructuring and refactoring tasks.

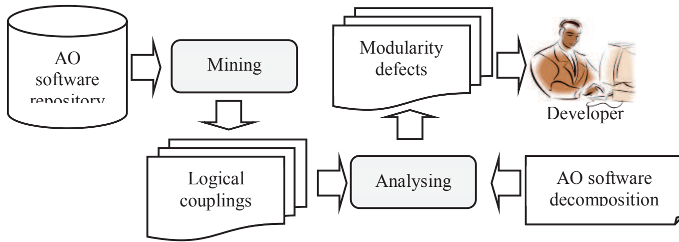

Figure 2: Modularity defect detection.

So, it is to the developer to examine the corresponding code (the entities that are related to a modularity defect) to improve it and enhance the AO software modularisation.

3.3 Logical coupling detection

In this step, a syntactic analysis of the Aspects source code is performed, such that additions and modifications of aspect's entities can be recorded. So, the source data for the mining will constitutes of the different building blocks of the software aspects: fields, methods, pointcuts, advices, and introductions. With our Mining approach, we address the following questions: 1) what are the coupled aspect entities in the AO system? and 2) what are the strengths of these couplings?

3.3.1 Coupled aspect entities

As we are mining for entities that are frequently changed together, it seems natural to use the technique called frequent itemset mining, which is able to discover interesting relations in a database. Our mining approach follows these steps: it acquires aspects data from a repository and transforms them into change-sets, which consist of the names of the entities added or modified in each transaction. Filtering may help in avoiding irrelevant data at this stage. We aim to track and mine software entities belonging to aspects, so we do not take into account base code entities. We focus on the logical couplings among crosscutting concerns only. So, we keep in every transaction only the aspect entities of an AO system (not base code entities). These are then processed using the Apriori frequent itemset mining algorithm [1].

Let $E = \{e_1, e_2, \dots, e_n\}$ be a set of aspect's entities i.e. entities that belong to the different aspects of the AO software, and $X \subseteq E$ an entity-set. We define repository R as a set of transactions: $R = \{t_1, t_2, \dots, t_m\}$, where $t_i = \{e_{i1}, e_{i2}, \dots, e_{ik}\}$ and $e_{ij} \in E$. Also, let $s(X)$ be the set of transactions that contain entity-set X, formally $s(X) = \{Y \in R | Y \supseteq X\}$. Finally, the support of an entity-set X is the fraction of transactions in the repository that contain X: $support(X) = \frac{|s(X)|}{|R|}$. Then X is called a logical coupling when its support is higher than a given minimum support: $support(X) \geq minsupport$.

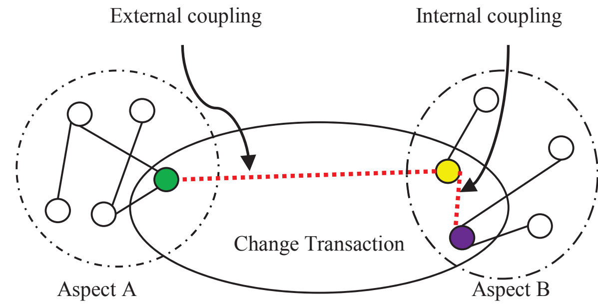

Figure 3: Analysing logical couplings.

3.3.2 Strength of a logical coupling

The strength of a logical coupling is considered as the support of this logical coupling. So, the strength of a logical coupling $\{e_1, \dots, e_n\}$ where each e_i is an aspect entity, is measured by support which is the number (or percentage) of transactions containing the entities $e_1, \dots, e_n$.

3.4 Modularity defect detection

The logical couplings extracted in the above step, are then analysed to detect modularity defects in the AO software system. This analysis is based on the structural decomposition of the AO software as crosscutting concerns (Aspects). In this step, the detected logical couplings are classed into two categories: internal and external logical couplings. As depicted in Figure 3, we define internal coupling as a dependency between two entities that belong to the same Aspect. The couplings between entities of an Aspect and any other entities that belong to other aspects are considered as external couplings.

These external logical couplings are considered as possible defects in the AO software modularity. Formally, a set of external logical couplings ELC is defined as ELC $= \{(e_i, e_j) | e_i \in A, e_j \in B\}$, where A and B are two independent aspects. So, the set of modularity defects MD in an AO program P is defined as:

$$MD(P) = \sum_{i=1}^{|ELC|} ELC_i$$

Since modularity defects are logical couplings, each modularity defect has a strength/support value (the number of transactions that contain the external logical coupling). So, we can say that (x, y) is a modularity defect that occurred once, (y, z) is a modularity defect that occurred twice, and so on.

3.5 Discussion

Using the detected logical couplings between aspect entities, we can deduce the coupled crosscutting concerns (aspects) in the AO system. As depicted in Figure 4, if two aspect entities e_1 and e_2 that belong to the independent aspects A_1 and A_2 respectively (modularity defect). Then, we deduce automatically that A_1 and A_2 are coupled aspects.

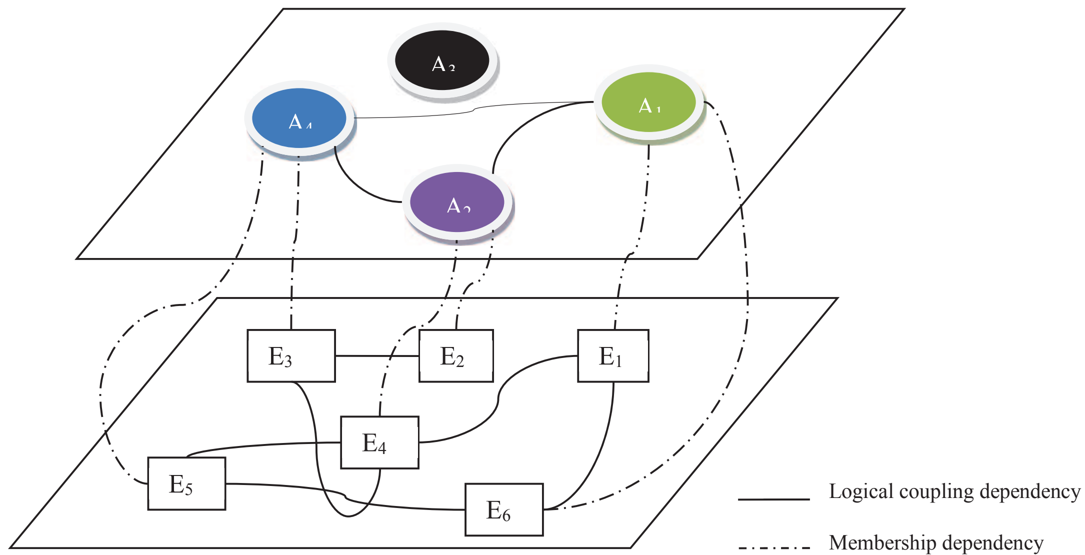

Figure 4: Coupled aspects.

The detected modularity defects can be used to assist restructuring and refactoring tasks. If some aspects change at the same time very often over several releases, they can be used to point to candidates for refactoring.

On the other hand, our results can be used to evaluate the AO software modularity. We can for example define a modularity measure using the detected logical couplings (it can be equal to the number of no coupled aspects, devised by the total number of aspects). Based on this measure we can evaluate the AO system modularity. This measure can be used later to compare many implementations of AO software systems. So, we can answer interesting questions as: Is the AO program P more modularized then the AO program P'? Is the implementation of the crosscutting concern C in program P is much more encapsulated than in program P'? If we detect crosscutting concerns (aspects) that have no coupling to any other crosscutting concerns, these can be a perfect reusable crosscutting concerns.

4 Tool chain

This section describes the tool-chain with which we identify modularity defects in AO programs written in AspectJ [12]. This last is a well-established AOP language. As depicted in Figure 5, the overall process is performed using three main tools:

The AspectJML Tool: an existing open source proposed by Melo Junior and Mendonça [13]. It is an XML-based markup language for representing source code written in AspectJ. The AspectJ source code is converted in XML (eXtended Markup Language) format [21] through the power of AspectJML. This XML-based representation is then used by the other tools in the tool-chain.

The Mining Tool: We have implemented this tool to extract logical couplings from the AspectJ repository. First, this tool takes change transactions from the repository and filters them to keep just the changed entities belonging to the aspects of the system (not base

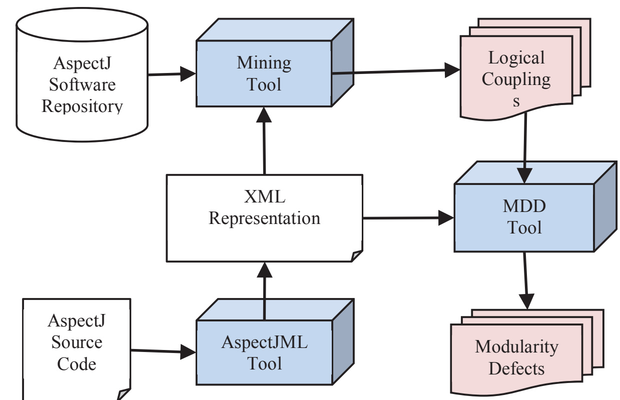

Figure 5: Tool chain.

code entities) in every transaction. Then, every entity in the transactions is replaced by its identifier. This last is extracted from the XML-based representation of the AspectJ source code. So, the tool gets change transactions of entity identifiers and organizing them in a single XML document. Finally, the transactions are mined using the Apriori algorithm.

Here, we used the XQuery implementation of the Apriori algorithm proposed by Wan and Dobbie [22]. The output of this tool is the logical couplings in the AspectJ source code that have a support higher than a specific threshold (min support). Every logical coupling is a set of entity identifiers.

The MDD Tool: a Modularity Defect Detection tool is implemented to filter the logical couplings obtained by the Mining tool. Here, the tool extracts modularity defects by eliminating internal logical couplings. It uses the XML-based representation of the source code to test if the entities that belong to a specific logical coupling are existing in the same aspect, or in different aspects using their identifiers. The results present possible modularity defects.

5 Case study

In order to assess the feasibility and correctness of our approach, this case study uses 22 releases of three well-known AspectJ programs available as open source. These systems were selected because they are rich in kinds of crosscutting concerns. Also they are used as case study in different research works [4, 5, 8, 14].

Table 1 describes these systems. It gives the number of versions and aspects of each software system. The first one, called Contract4J, it supports "Design by Contract" programming in Java. We considered the 5 releases of Contract4J in our study. The second is a product line for deriving applications that manipulate photos, videos and music on mobile devices called Mobile Media [6]. We selected its 7 releases in this experimentation. The last system called Health Watcher [19]; is a real Web-based information system that allows citizens to register complaints about health issues in public and heath care institutions to investigate and take the required actions. We selected the 10 releases of Health Watcher in our study.

After the application of our approach on these systems, we find many internal logical couplings as: “frequently changing a pointcut involves changing its related advices”, “changing a field, involves changing the methods that use this field”, etc. Many modularity defects (external logical couplings) are detected also. Table 2 presents the detected coupled aspects in each system. For each coupled aspects, it gives the number of detected modularity defects (external logical couplings). Besides, it gives the support of each coupling. Here the support is the average of the supports of the related modularity defects i.e. the support of a logical dependency between two Aspects A and B is the sum of supports of their related modularity defects, devised by the number of such modularity defects.

In the program Contract4J we have detected that the aspects *ConstructorBoundaryConditions* and *MethodBoundaryConditions* are tightly coupled with 7 modularity defects. Here the coupled low-level entities with the higher support for these aspects are: the method *doTest* in the aspect *ConstructorBoundaryConditions* and the method *doBeforeTest* in the aspect *MethodBoundaryConditions*.

Software	#versions	#Aspects
Contract4J	5	8—14
Mobile Media	7	4—42
Health Watcher	10	11—23

Table 1: Subject programs.

The aspect *ConstructorBoundaryConditions* is also coupled with the aspect *UsageEnforcement* through 3 modularity defects. The modularity defect with the higher support here is between: the advice applied after the pointcut *postCtor* in the aspect *ConstructorBoundaryConditions* and the pointcuts *preNotInContract*, *postNotInContract*, and *invarNotInContract* that belong to the aspect *UsageEnforcement*.

In the Mobile Media program, we have detected much more coupled aspects than those detected in the Contract4J program. The aspects *DataModelAspectEH* and *UtilAspectEH* are coupled via 2 modularity defects: the pointcuts *loadMediaDataFromRMS* and *readMediaAsByteArray* belonging to *DataModelAspectEH* and *UtilAspectEH* respectively are frequently changed together. Also, the pointcuts *getMedias* and *getBytesFromMediaInfo* are tightly coupled.

We have also detected that the aspect *SortingAspect* is coupled with 3 other aspects, which restricts its evolvability and reusability. It is coupled with the aspect *FavouritesAspect* via 4 modularity defects. Besides, it is coupled with the aspects *ControllerAspectE* and *CopyPhotoAspect* through one modularity defect. The most frequent detected modularity defects here are of the type pointcut duplications. For instance, the pointcuts *handleCommandAction* and *appendMedias* are duplicated in the aspects *FavouritesAspect* and *SortingAspect*. Besides, the pointcut *showImage* is defined in the aspects *ControllerAspectEH* and *SortingAspect*. So any modification in such pointcuts implies changes in many aspects.

Application	Coupled aspects	#Modularity defects	Support
Contract4J	ConstructorBoundaryConditions MethodBoundaryConditions	7	0,6
	ConstructorBoundaryConditions UsageEnforcement	3	0,6
Mobile Media	DataModelAspectEH UtilAspectEH	2	0,4
	FavouritesAspect SortingAspect	4	0,4
	ControllerAspectEH SortingAspect	1	0,2
	CopyPhotoAspect SortingAspect	1	0,2
Health Watcher	—	—	—

Table 2: Detected modularity defects.

Finally, in the Health Watcher application we have detected many internal logical couplings, but we do not detect serious modularity defects in that system (except of a few external logical couplings which are detected once). So, in contrast to the above applications (Contract4J and Mobile Media), we can say that Health Watcher has a good modularization, and their crosscutting concerns (Aspects) are good reusable modules.

6 Related work

This section of the paper presents related works discussing the benefits of our proposal in contrast to the other ones. Our work involves the following research areas:

AO software analysis: Existing approaches for detecting dependencies among AO software generally use static analysis [15, 17, 25, 26]. Such approaches are mainly based on an instruction-level to analyse the evolution of an AO software system: the source code is analysed and source code slicing is used to perform change impact analysis. We may say that such code-based approaches reveal syntactic dependencies and what we are really interested in is logical dependencies among AO software concerns. On the other hand, the information is derived using analysis of textual software artefacts that are found in a single version of the software. In contrast, our approach is based on an empirical observation of AO system structural modifications. We treat the whole evolution history to detect the modularity defects.

Mining AO software repositories: There are many approaches and techniques for detecting logical couplings in OO software [9, 10]. These works prove that such historical analysis is often able to capture couplings among software entities that cannot be captured by static and dynamic analysis. But this research area still not enough explored for AO software. Few works are dedicated to mine AO software repositories. For instance, Qian et al. [16] treat the detection of change patterns in AspectJ programs. They analyse the successive versions of an AspectJ program, and then decompose their differences into a set of atomic changes. Finally, they employ the Apriori data mining algorithm to generate the most frequent item-sets. In [3], we have also detected change patterns in AspectJ software by Mining a rewriting rule-based repository. In this paper, our goal is different, as we aim at identifying logical couplings between the aspect entities instead of change patterns.

Detecting software modularity defects: Many works prove the benefits of analysing the OO software evolution history for assessing its modularity. In [18] the authors state that to improve current modularity views, it is important to investigate the impact of design decisions concerning modularity in other dimensions, as the evolutionary view. They propose the ModularityCheck tool to assess package modularity using co-change clusters, which are sets of classes that usually changed together in the past.

Wong et al. [23, 24] presented CLIO, a tool that detects and locates modularity violations. CLIO compares how components should co-change according to the modular structure and how components usually co-change retrieving information from version history. A modularity violation is detected when two components usually change together but they belong to different modules, which are supposed to evolve independently. We use the same idea to detect modularity defects in AO software. However, these works extract couplings at a file level; in contrast, we detect logical couplings at entity level. Our detected fine-grained logical couplings can be very useful for restructuring and refactoring tasks.

7 Conclusion

Unintended modularity defects of AO software may not be easily detectable by static or dynamic analysis techniques, but could cause modularity decay and bad separation of crosscutting concerns. To detect such modularity defects, we suggested a history-based approach based on the logical couplings in the AO software system.

Our approach applies frequent itemset mining over an AO software repository in order to detect logical couplings among its entities. The extracted logical couplings are then analysed to detect modularity defects in the AO software system. Many case studies are experimented to demonstrate the feasibility of our approach. The results show that the approach is able to detect logical couplings among aspect entities, as well as modularity defects. The approach leads naturally to an evaluation of AO system's modularity. The results of our approach can be used for reducing the dependencies between AO software Aspects and consequently promoting its modularity.

The same idea can be used for detecting other types of AO software defects. For example, we can analyse the AO software evolution history for detecting bad smells, anti-patterns, etc.

References

[1] R. Agrawal, and R. Srikant (1994). Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, Proceedings of 20th International Conference on Very Large Data Bases, Santiago, Chile, pp. 487–499.

[2] R.T. Alexander, J. M. Bieman, and A. A. Andrews (2004). Towards the Systematic Testing of Aspect-Oriented Programs. Report CS-04-105, Colorado State University, Fort Collins-USA.

[3] H. Cherait, and N. Bounour (2014). Detecting Change Patterns in Aspect Oriented Software Evolution: Rule-based Repository Analysis. International Journal of Software Engineering and Its Applications (IJSEIA), Vol. 8, No. 1, pp. 247-266.

[4] R. Dyer, H. Rajan, and Y. Cai (2012). An Exploratory Study of the Design Impact of Language Features for Aspect-oriented Interfaces. In Proceedings of AOSD'12, Potsdam, Germany.

[5] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L.

Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado (2010). An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs. In Proceedings of ICSE '10, Cape Town, South Africa, ACM press, pp. 65 – 74.

[6] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas (2008). Evolving software product lines with aspects: an empirical study on design stability. In Proceedings of ICSE'08.

[7] B. Fluri, H. C. Gall, and M. Pinzger (2005). Fine-grained analysis of change couplings. In Proceedings of 5th WICSA'05, pp. 66–74.

[8] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dósea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid (2007). On the impact of aspectual decompositions on design stability: An empirical study. In Proceedings of ECOOP, pp. 176–200.

[9] A. E. Hassan (2008). The road ahead for mining software repositories. In Frontiers of Software Maintenance, pp. 48–57.

[10] H. Kagdi, M. L Collard, and J. I. Maletic (2007). A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 19, No. 2, pp. 77-131.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In Proceedings of 11th European Conference on Object-Oriented Programming, Springer-Verlag, LNCS Vol. 1241, pp. 220–242.

[12] R. Laddad (2003). AspectJ in Action: Pratical Aspect-Oriented Programming. Manning Pubblications Company.

[13] L. S. Melo Junior, and N. C. Mendonça (2005). AspectJML: A Markup Language for AspectJ. In Proceedings of the 2nd Brazilian Workshop on Aspect Oriented Software Development, Uberlândia, MG, Brazil.

[14] A. C. Neto, M. Ribeiro, M. Dosea, R. Bonifacio, P. Borba, and S. Soares (2007). Semantic Dependencies and Modularity of Aspect-Oriented Software. In Proceeding of the First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07).

[15] E. K. Piveta , M. Hecht , M. S. Pimenta , and R. T. Price (2006). Detecting bad smells in AspectJ. Journal of Universal Computer Science.

[16] Y. Qian, S. Zhang and Z. Qi (2008). Mining Change Patterns in AspectJ Software Evolution. In Proceedings of the International Conference on Computer Science and Software Engineering, pp. 108-111.

[17] M. Rinard, A. Salcianu, and S. Bugrara (2004). A classification system and analysis for aspect-oriented programs. In Proceedings of FSE'04, pp. 147–158.

[18] L. L. Silva, D. Félix, M. T. Valente, M. de A. Maia (2014). ModularityCheck: A Tool for Assessing Modularity using Co-Change Clusters. In Proceedings of the Brazilian Conference on Software: Theory and Practice (CBSoft'14) - Tool Session.

[19] S. Soares, E. Laureano, and P. Borba (2002). Implementing distribution and persistence aspects with AspectJ. In Proceedings of the 17th OOPSLA.

[20] F. Steimann (2006). The Paradoxical Success of Aspect-Oriented Programming. In Proceedings of OOPSLA'06, pp. 481-497.

[21] J. Suzukiand, and Y. Yamamoto (1998). Managing the software design documents with xml. In Proceedings of the 16th annual international conference on Computer documentation, ACM Press: New York, pp. 127-136.

[22] J. W. W. Wan, and G. Dobbie (2003). Extracting Association Rules from XML Documents using XQuery. In Proceedings of WIDM'03, New Orleans, Louisiana, USA, pp. 94-97.

[23] S. Wong, Y. Cai, and M. Dalton (2009). Detecting Design Defects Caused by Design Rule Violations. Report DU-CS-09-04, Drexel University.

[24] S. Wong, Y. Cai, M. Kim, and M. Dalton (2011). Detecting software modularity violations. In Proceedings of 33rd International Conference on Software Engineering (ICSE'11), pp. 411–420.

[25] G. Xu, and A. Rountev (2008). AJANA: A General Framework for Source-Code-Level Interprocedural Dataflow Analysis of AspectJ Software. In Proceedings of AOSD'08, Brussels, Belgium.

[26] J. Zhao (2002). Change Impact Analysis for Aspect-Oriented Software Evolution. In Proceedings of the 5th International Workshop on Principles of Software Evolution, Orlando, Florida, pp. 108-112.