Comparing Three Aspect Mining Techniques

Fernando Sérgio Barbosa

Faculdade de Engenharia da Universidade do Porto
fsergio@est.ipcb.pt

Abstract. Even in well designed software systems there are some concerns that are spread over many units. Aspect Oriented Programming is a new programming paradigm that enables modularisation of these crosscutting concerns as aspects. Some crosscutting concerns are obvious to spot, but many others are not. In large systems or frameworks, identifying them is a hard task that needs the help of tools. The identification of such crosscutting concerns is called aspect mining. In this paper we compare three aspect mining techniques: Fan In analysis, Dynamic analysis and Clone detection. We compared them using a common target: the JHotDraw framework. We then discuss each of the techniques strengths and weaknesses by comparing the results. Finally we analyse the possibilities of combining the techniques to achieve better results.

Keywords: aspect mining, aspect oriented programming, fan in analysis, dynamic analysis, clone detection

1. Introduction

The tyranny of the dominant decomposition [1] states that there are always some concerns, called crosscutting concerns, that, even in well decomposed systems, will not neatly fall in any of the units thus being spread over many of them.

Aspect Oriented Programming (AOP) [2] is an emerging programming paradigm with primitives that allow modularisation of the so-called cross-cutting concerns, removing much of the code scattering and tangling, placing them in aspects. Because of this programmers seek to use AOP in their programs. But, as with migrating to a new technology, they have to deal with a large base of installed artefacts based on the old technology and must keep on using it or refactor [3] it to aspects.

While some crosscutting concerns are plain obvious, like the traditional ones referred in AOP literature as logging, persistence, security, memory management, etc, others are not so obvious and require a much more detailed look at the source code. In large systems or frameworks this is a hard task that not all programmers, if any, wishes to do. This calls for the help of tools that can identify possible aspects from the source code. This aspect finding activity is called aspect mining [4]. Current tools allow for semi-automatic identification for aspects only, but in the future this can be automated. If automatic aspect mining is used together with refactoring techniques then the conversion of OO code into AO code also becomes automatic [5].

In this paper we compare three aspect mining techniques: Fan In analysis [6], Dynamic analysis [7] and Clone Detection [8]. We compared them using a common

target: the JHotDraw framework. We do not aim to decide which of the techniques is best but to identify their strengths and weaknesses. A best/worst categorization would need well established criteria for good aspect modularisation and that is not yet available. Because of this we limit ourselves to a qualitative comparison.

Another objective of this paper is to see if the combination of the techniques can improve the results or if they overlap. Either way it is good to know: if they overlap we can discard one of them, if they complement each other then we can combine them to achieve better results. Our conclusions support the fact that the three techniques are combinable as they have different strengths and weakness.

This paper is structured as follows: In Section 2 we present the three aspect mining techniques and briefly present the tools used for the comparison. The experiment is presented in Section 3 as well as the results of the experiment. In Section 4 the comparison of the three techniques is made based on the results from Section 3. Section 5 presents related work and Section 6 concludes the paper.

2. The Three Aspect Mining Techniques

In this paper we will compare the following 3 aspect mining techniques: fan-in analysis [6], dynamic analysis [7], and clone detection [8]. These are not the only proposed aspect mining techniques but they are supported by publicly available tools. The respective tools are briefly presented within each technique.

2.1. Fan-In Analysis

The fan-in of a method *M* is defined as the number of calls to method *M* made from other methods [6]. Because of polymorphism, one method call can affect the fan-in of several other methods. A call to method *M* contributes to the fan-in of all methods refined by *M* as well as to all methods that are refining *M [9]*. The more places the method is called from the more likely it is that the method implements a crosscutting concern so the amount of calls (fan-in) is a good measure for the importance and scattering of the discovered concern.

The analysis follows three consecutive steps: (1) Automatic computation of the fan-in metric for all the methods in the targeted source code. The result is stored as a set of "method-callers" structures that can be sorted by fan-in value. (2) Filtering of the results of the first step, by restricting the set of methods to those having a fan-in above a certain threshold; filtering getters and setters from this restricted set. Get/Setters on static fields are not eliminated because these can be used in the Singleton design pattern; filtering utility methods, like toString( ), collections manipulation methods, etc. (3) Analysis of the remaining set of methods. The elements considered at this step are the callers and the call sites, the method's name and implementation, and the comments in the source code.

A tool that supports Fan In is FINT[1]. FINT is implemented as an Eclipse[2] plug-in [10]. The current implementation of FINT includes three source code analysis

[1] http://swerl.tudelft.nl/bin/view/AMR/FINT

techniques to identify crosscutting concerns: Fan-in analysis, Grouped calls analysis and Redirections finder. Since this paper only concentrates in the first one the others were disabled.

The results of the analysis are displayed in the Fan-in analysis view as a tree structure of callee-callers elements, sorted by name or fan-in value. From this view the user can inspect the source code of each of the displayed elements and apply the already mentioned filters to restrict the elements to the most relevant candidates.

2.2. Dynamic Analysis

The technique of Formal Concept Analysis (FCA) is fairly simple [11]. Starting from a (potentially large) set of elements and properties of those elements, FCA determines maximal groups of elements and properties, called concepts.

FCA is used for aspect mining according to the following procedure: Execution traces are obtained by running an instrumented version of the program under analysis for a set of use cases. The execution traces associated with the use cases are the objects, while the executed class methods are the attributes. In the resulting concept lattice, the concepts specific of each use case are located, when existing. The use case specific concepts are those labelled by at least one trace for some use case (i.e. the concept contains at least one specific property) while the concepts with zero or more properties as labels are regarded as generic concepts. When the methods that label one concept crosscut the principal decomposition, a candidate aspect is determined. More specifically, a concept is a candidate aspect if:

- scattering: more than one class contributes to the functionality associated with the given concept (i.e., the methods labelling the concept belong to more than one class);
- tangling: the class itself addresses more than one concern (i.e., appears in more than one use case specific concept).

The first condition alone is typically not sufficient to identify crosscutting concerns, since it is possible that a given functionality is allocated to several modularised units without being tangled with other functionalities. In fact, it might be decomposed into sub-functionalities, each assigned to a distinct module. It is only when the modules specifically involved in a functionality contribute to other functionalities as well that crosscutting is detected, hinting for a candidate aspect.

Dynamo[3] is a tool for the identification of aspects in existing Java classes by means of a dynamic code analysis [7]. Execution traces are generated for the use cases that exercise the main functionalities of a given application. The relationship between execution traces and executed computational units is subjected to concept analysis. In the resulting lattice, potential aspects are detected by determining the use case specific concepts and examining their specific computational units.

---

[2] http://www.eclipse.org/

[3] http://star.itc.it/dynamo/

2.3. Clone detection techniques

Clone detection techniques aim at finding duplicated code, which may have been adapted slightly from the original. Several clone detection techniques have been described and implemented [8]: Text-based techniques attempt to detect identical or similar lines of code. Token-based techniques use tokens as a basis for clone detection. AST-based techniques build an Abstract Syntax Tree which is searched for similar subtrees. PDG-based approaches construct a Program Dependence Graphs (PDGs) that contains information such as control and data flow of the program. Metrics-based techniques calculate, for each fragment of a program, a number of metrics which are used to find similar fragments. Information Retrieval-based methods exploits semantic similarities present in the source code.

Clone detection techniques are promising in aspect mining due to two likely causes: First, by definition, scattered code is not well modularised so developers are unable to reuse concern implementations through the language module mechanism. Therefore, they are forced to write the same code over and over again, typically resulting in a practice of copying existing code and adapting it slightly to their needs. Second, they may use particular coding conventions and idioms to implement superimposed functionality, i.e., functionality that should be implemented in the same way everywhere in the application. This is even more so with the adoption of patterns [12], where similar code is found in various implementations of a pattern.

CCFinder[4] is a multilanguage clone detector [13]. CCFinder makes a token sequence from the input code through a lexical analyser and applies a, language specific, rule-based transformation to the sequence. The purpose is to transform code portions in a regular form to detect clone code portions that have different syntax but have similar meaning. Representing a source code as a token sequence enables the detection of clones with different line structures, which cannot be detected by line-by-line algorithm. After the detection the user has several browsing capabilities.

3. Running the Experience

As a common ground for comparison of the three aspect mining techniques the JHotDraw[5], v5.4b, framework was used. JHotDraw is a Java GUI framework for technical and structured graphics that has been developed as a "design exercise". Its design relies heavily on design patterns [12]. JHotDraw's original authors have been Erich Gamma and Thomas Eggenschwiler. JHotDraw is considered well designed and so has been used in both aspect mining [9] and aspect refactorings efforts [14].

3.1. Fan In Experience

The setting up of the experience with FINT is easy. All we had to do was create a project in Eclipse and run the plug-in. The results came in about 1,5 seconds. The

[4] http://www.ccfinder.net/index.html
[5] http://www.jhotdraw.org/

results comprised 479 filtered methods (with all filters on). After applying a threshold of 10 to the fan in metric the final result gave 120 methods that had to be analysed.

For each method two steps were taken: first we inspected the method's body to get a perception of its functionality, second we analysed the call sites in order to get the context of the methods usage. After this manual examination 45 methods were considered seeds for possible aspects. This doesn't mean that 45 aspects were found as many methods contributed to the same functionality. For example an instantiation of the Composite pattern [12] had 10 methods associated with it. But some concerns were associated with a single method. An example is util.Undoable.isRedoable() that indicates a consistent behaviour because every redo action must perform a isRedoable() test before execution.

A summary of the crosscutting concerns found is presented in Table 1. The grouping of the consistent behaviour and contract enforcement is done for it is sometimes hard to distinguish between the two.

Table 1. Summary of the results from the Fan In experiment

Description	methods	Concern type
When a tool has finished execution method toolDone() must be called	3	Consistent Behaviour
Border decorator	1	Decorator
Composite pattern in Figure hierarchy	10	Composite
FigureChangedListener	7	Observer
After execution of commands/tools the view must be checked for damages	4	Consistent Behaviour
After execution of commands (specially redos) the selection must be cleared	2	Consistent Behaviour
Every tool class must call the superclass constructor	1	Consistent Behaviour
Tool activation and deactivation	2	Consistent Behaviour
Every tool calls super.mouseDown()within their own mouseDown()	1	Consistent Behaviour
many figures use readInt, writeInt and such methods to write to a stream	6	Persistence
Before doing a redo the method isRedoable() must be called	1	Consistent Behaviour
Every undo operation must call its superclass undo()	1	Consistent Behaviour
Every UndoableAdapter calls its superclass constructor	1	Consistent Behaviour
UndoableComand uses the decorator pattern	1	Decorator

From the experience we could see that Fan In analysis can detect crosscutting concerns that are implemented in one of three ways: (1) The crosscutting concern is implemented by a single method that is called from several call sites, as is the case with the consistent behaviour where the method has to be called before some other execution. (2) The crosscutting concern is implemented by a single method but that method is used from several call sites for the same basic functionality. An example is persistence where methods to read/write several types of data are used by most classes that implement a figure. (3) When several methods contribute to a crosscutting concern it is likely that the crosscutting concern is derived from a pattern superimposed role. Examples of these patterns are Composite (10 methods) or Observer (7 methods).

Fan In analysis helped to find some crosscutting concerns by "mere chance". An example is the Decorator pattern [12] that was found because the methods from the decorator class had a high Fan In. If we limited the examination to the call sites such concern would be undiscovered because they were addressing the class main concern. The method code, however, clearly indicated that the class was a decorator. If it were not for the fact that the methods had high Fan In AND the names of identifiers and methods clearly indicated the purpose this pattern would be unnoticed.

3.2. Dynamic Analysis Experience

This was the most difficult experience of all. Before the analysis could be done the code had to be "instrumented". This meant that every ".java" file class had to be copied to a ".oj" file. Afterwards every file had to be edited and, for every class, (inners classes too) some code had to be inserted. Every file should also add an import clause for the dynamo package. This work could be done automatically by a tool but there wasn't, unfortunately, none available. We had to write a piece of code to do the work (the idea of doing it manually was refused). In the end those ".oj" files had to be compiled with a special compiler (OpenJava[6]) and the resulting ".java" files had also to be compiled. The OpenJava compilation is not fast, taking in JHotDraw more than 5 hours in a 2.4 Ghz, Pentium IV with 512MB. This "instrumentation" has only to be performed once and the tests are done with the instrumented version.

After the successful instrumentation of the framework some use cases were run. The time spent (more than 1 hour) running the use cases was considerable. The concept lattice generated had more than 1500 nodes. This amount of information cannot be dealt with manually so the aspect mining tool of Dynamo was used.

Table 2. Summary of the results from the Dynamic Analisys experiment

Description of concern	Concepts	Methods
Undo	2	36
Bring to front	1	3
Send to back	1	3
Connect text	1	18
Persistence	1	30
Handle manipulation	4	60
Figure Observer	4	11
Command executability	1	25
Connecting figures	1	55
Manage figures outside drawing	1	2
Get Attribute	1	2
Set Attribute	1	2
Managing view rectangle	1	2
Visitor	1	1

A summary of the concerns found is presented in Table 2. As can be seen from Table 2 several concepts contribute to the same concern. This is the case for the handle manipulation concern in which four concepts contribute to that concern.

3.3. Clone Detection Experience

CCFinder is an easy to use tool. It is enough to specify which files to analyse. The analysis was done in 15.3 seconds. To that time it must be added the time to calculate some metrics that were needed latter, in about 20 seconds.

The first result included 433 clone sets. We then filtered the sets with a population less than 4, reducing the clone sets to 79. These 79 clone sets were manually inspected. We selected the clone set to inspect and the tool showed the various places where the clone was reproduced. It must be mentioned that some clones do overlap,

[6] http://www.csg.is.titech.ac.jp/openjava/

i.e. a larger code clone may include a smaller code clone. Some clones are also similar to others, and because the tool doesn't take in account identifiers names some clones have unrelated code (false clones). These clones were removed.

After inspection 46 clones were discarded leaving a final 33 clone sets. Much like Fan In analysis a clone set does not have a one to one relation with a crosscutting concern. This is the case with the Undo concern that has 14 clone sets associated with its implementation. Some clones do represent a single crosscutting concerns as for example the clone that shows that tools must update the view after a mouseDown( ).

A summary of the concerns found is presented in Table 3.

Table 3. Summary of the results from the Clone Detection experiment

Description	Clone sets	Concern type
Tools must update the view after a mouseDown( )	1	Consistent behaviour
1 subject roles in Observer pattern: Command Listener and Tool Listener	4	Observer (2x)
Undo	14	Undo
Reading and writing data	1	Persistence
1 observer role and 3 subject roles in FigureListener	2	Observer
2 subject roles of ViewListener	1	Observer
1 subject roles of Figure Listener and 2 subject roles of DesktopListener	1	Observer (2x)
Handle manipulation	1	Handles
Event dispatcher + undo	1	undo + event dispatcher
Connecting figures	1	Connecting figures
5 subjects roles of FigureListener	1	Observer
Before executing commands isExecutable() must be called	1	Consistent behaviour
Every handle checks owner's display box	1	Consistent behaviour
3 instances of decorator	1	Decorator
Many commands have to perform isExecutableWithView	1	Consistent behaviour
1 ViewChangeListener subject role + 2 DesktopListener subject roles	1	Observer

From the experience we could see that Clone Detection analysis can detect crosscutting concerns that are implemented in one of three ways: (1) the code that deals with the crosscutting concern is implemented in every class that addresses the crosscutting concern. This is the case with the consistent behaviour where every class must perform a call to a specific method. (2) The code that deals with a particular crosscutting concern is implemented by several classes with minor variations, as is the case with the undo concern. (3) The code is part of a superimposed role from the participation in a pattern. This is the case with the various instances of the Observer pattern where every class must implement addObserver() like methods which are very similar. It must be said that the detection of some patterns depends on the number of the classes that participate. For example the detection of the Observer pattern was possible because there were several subjects. If it was only one subject none would be detected because only one instance of addObserver( ) method existed. The way each observer reacts upon an event depends on the observer itself and so the code is unique, which explains why only one observer was found.

It must be stressed that sometimes the clone itself didn't provide enough information to achieve a conclusion, but examining the surroundings of the clones for a wider vision was enough to detect some crosscutting. The most expressing one is undo: while several clone sets contributed to this concern it was when the surroundings where examined that we found that every class implementing undo had a UndoActivity as inner class. This clearly indicated Undo as a crosscutting concern.

4. Comparing the techniques

For the comparison of the techniques we selected some concerns that were detected by all the techniques and some that were discovered by two or one of the techniques. For each concern we discuss why the various techniques failed/succeed to identify it.

4.1. Concerns Used in Comparison

Observer

Every technique succeed to identify this concern, even if they didn't find all instances of the Observer pattern. The Fan In and Dynamic analysis identified only one while the clone detection technique discovered four. Note however that Clone Detection only identifies subjects (with exception of one observer), this is because the Observer pattern uses very similar code between its instantiations, namely to manage observers. The Fan In succeeds in the FigureChangedListener because there are many observers thus the methods to register in the subject surpasses the Fan In threshold. But because there are few DesktopListener and ViewChangeListeners those methods didn't pass the Fan In threshold and it failed to identify them.

Composite

Only the Fan In analysis succeeded to identify this concern. Clone detection failed because there are few instances of the pattern so clone population didn't arise above the threshold. Dynamic analysis failed to identify it due to the fact that every trace uses the composite pattern because every drawing is a composite figure and so this is exercised in every use case.

Undo

Both Clone Detection and Dynamic analysis detected this concern, but Fan In failed. Even though Fan In detected that all redo must do a isExecutable() this falls in the category of consistent behaviour. It can be argued that detecting this behaviour one could suspect that an undo concern is present so Fan In detects it too, but that is the annalist deduction/perception/experience that enables it and not a direct consequence of the Fan In. Clone Detection succeed because every undo activity has similar code (most notorious are the undo and redo methods of an inner class UndoActivity). Dynamic analysis detected it because undo is considered a use case and so is exercised in the application.

Handle Manipulation

Both Clone Detection and Dynamic analysis detected this concern, but Fan In failed. Handle manipulation is used in the use cases that manipulates figures so the Dynamic analysis detects it fairly well. Clone Detection succeeded because much of the handles code is similar. Fan In failed because there are many methods related to the handle manipulation but each is called from very few points.

Consistent behaviour

Every technique identified this type of concern but they identified it in very different ways and not all the consistent behaviour was discovered by all the

techniques. Namely Dynamic analysis only captured the command executability concern. The Fan In was the one that discovered more cases (9). Clone detection also discovered a few (4). This is clearly the goal of the Fan In method were a specific concern is done by a method that is called from several places in the code. Clone detection also checks this because some of the code is very similar (often involving an if statement with a method invocation followed by a return statement). But since this code is usually very short Clone Detection can fail to capture it because the clone has to be bigger than a minimum length in size. To capture this code the minimum clone length must be small but this in turn increases the size of the clone sets found and the noise generated is much greater (for example in one of the experiments every for was considered a clone). Dynamic analysis failed to detect most of the concerns because they are exercised in almost every use case.

Bring to Front/Send to back

Only Dynamic analysis detected this concern. Fan In failed because the methods dealing with it are not called from many places as it is a rather specific concern. Clone detection failed because the concerns code is specific to two classes and not repeated.

Table 4. Summary of the comparison between the tree techniques with the selected concerns.

Concern	Fan In Analysis	Dynamic Analysis	Clone Detection
Observer	+	+	++
Composite	+		
Undo		+	++
Handle Manipulation		++	+
Consistent Behaviour	++ (9)	+ (1)	+ (4)
Bring to Front/Send to back		+	

Table 4 summarizes the previous discussion. In Table 4, concerns that are discovered by a technique are marked +, if they are not discovered they are not marked. If a technique was more efficient than others in discovering a concern it is marked ++. This way we can see the strengths and weaknesses of the techniques.

4.2. Limitations of the techniques

Comparing the techniques in a common ground enables us to identify their strengths and limitations. While the strengths are outlined in the techniques presentation area their limitations are uncovered in the experiment. Next we present the limitations found for each technique.

Fan IN

Mainly addresses crosscutting concerns that are largely scattered and have a significant impact on the modularity of the system. More: it depends on the correct modularisation of those concerns in methods. If the code is not placed in methods but “copy/pasted” (unfortunelly this is not uncommon) the technique fails completely. This means that concerns with a small code footprint and thus with low fan-in values, will be omitted. For example, as debated before, the identification of Observer design pattern instances is dependent on the number of classes implementing the observer

role. The number of observer classes will determine the number of calls to the registration method in the subject role. A collateral effect is the anticipated unsuitability of the technique for analysing small case studies.

Dynamic Analysis
Among the known limitations of this technique, the two most important ones are that it is partial (i.e., not all methods involved in an aspect are retrieved) and it can determine only aspects that can be discriminated by different execution scenarios (e.g., aspects that are exercised in every program execution cannot be detected). Additionally, it does not deal with code that cannot be executed (e.g., code that is part of a larger framework, but is not used in a specific application).

Clone Detection
This technique addresses code that is similar in crosscutting concerns, but if a concern is handled differently by those who implement it the technique fails. Even one of the best results obtained by the technique depends on the amount of code, as is the case with the Observer patterns of which it encountered several instantiations (mainly subjects). If there was only one instance of this pattern then Clone Detection would fail to discover it. The undo concern is another example: if few classes implemented the undo or didn't follow a common usage it would be undetected.

4.3. Combining the techniques

Overall, fan-in analysis and dynamic analysis show largely complementary result sets. This is an expected result [9], since the first technique focuses on identifying those methods that are called at multiple (scattered) places. However, when a method is called multiple times in a system, it is likely to occur in most the execution traces, so that no specific use case can be defined to isolate the associated functionality.

Clone detection is also a very good technique capable of complementing the other two, specially in the case of superimposed roles that have similar implementation. As can be seen from Table 4, the best results may be obtained from the combination of the three techniques and not by a single technique alone. The example of the Observer pattern also suggests that Clone Detection technique could be used together with Fan In to achieve better results. The Fan In identified the observers roles while Clone Detection identified the subject role. This is also the case with the badly written modularity (using copy/paste instead of method calling) where Fan In would fail but Clone Detection would succeed. Using Dynamic analysis would bring to the scene the concerns that are not detected by the other techniques.

5. Related work

Ceccato et al compare three aspect mining techniques: Fan In analysis, Identifier analysis and Dynamic analysis and provides some thoughts in combining them [9]. We added a, minor, extension to their work using a different technique and our results are somewhat different. This is explained by the fact that stating that a given concern

is a crosscutting concern and not a class specific concern can vary. Such an example is the move figure concern identified as a crosscutting concern in [9] and as a class concern by us. Others include the detection of the Command pattern that we dismissed based on the assumption that the Fan In technique does not directly identify this concern. The identified method didn't address a crosscutting concern and the call sites reflected that, but its implementation suggested that the Command pattern was being used, mostly by the identifiers names. To be fair in the comparison we marked that concern as not identified. Identical decisions were made in the Dynamic analysis. The same authors give a full description on how to combine the techniques in [15].

Bruntnik et al compare three clone detection techniques in aspect mining [8]. They evaluated the suitability of the three techniques for identifying crosscutting concern code. The evaluation considers token, AST, and PDG-based clone detection techniques and provides a quantitative comparison of their suitability. In their case study they manually identify five crosscutting concerns and evaluate to what extent the crosscutting concern code is matched by the three clone detection techniques.

Breu suggests to enhance DynAMiT, a aspect mining tool based on dynamic analysis, with static information and generating the traces using call pointcuts [16].

Cojocar and Serban propose several criteria to be used in comparing aspect mining techniques [17]. But most criteria take into account the detected aspects versus the number of existing aspects. This means that the comparison must be made using a well known and previously aspectized code. Such a code base is not yet available but some steps are being taken in order to completely refactor the JHotDraw framework into an aspect oriented framework named AJHotDraw [14].

6. Conclusion

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate crosscutting concerns, a problem known as aspect mining. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity.

In this paper we presented three aspect mining techniques and compared them using a common target. Each of the techniques has its strengths and weaknesses. Fan In deals better with concerns that are implemented in methods that are called from the various places dealing with the concern. It cannot handle concerns that are not much scattered or that are not implemented with methods calls but with in site code. Dynamic analysis deals better with concerns that are associated with some of the application use cases but fails to discover the concerns that are common to all use cases and also fails to detect unused concerns making it rather unsuitable to detect concerns in a framework for example. Clone Detection finds the crosscutting concerns that are implemented via replicated code but fails to address those concerns that have different code for each place that deals with it, as was the case with the observer role in the Observer pattern.

The combination of the techniques would benefit the results because each technique has its strengths were the others have their weakness.

References

1. Tarr, P., Ossher, H., Harrison, W., Sutton, J. S. M.: N degrees of separation: Multi-dimensional separation of concerns. Proceedings of the 21st international conference on Software engineering, (1999) 107-119
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. Aspect-oriented programming. In Proceedings of ECOOP 1997, Finland, (1997) 220–242.
3. Monteiro, M.P., Fernandes, J. M.: Towards a Catalogue of Aspect-Oriented Refactorings. In proceedings of AOSD 2005, Chicago, USA (2005) 111-122
4. Kellens, A., Mens, K., Tonella, P.: A Survey of Automated Code-level Aspect Mining Techniques. Transactions on Aspect-Oriented Software Development, Special Issue on Software Evolution, 2007, to appear.
5. Deursen, A., Marin, M., Moonen, L.: Aspect mining and refactoring. In Proc. of the 1st International Workshop on REFactoring: Achievements, Challenges, Effects. (2003)
6. Marin, M., Deursen, A., Moonen, L.: Identifying aspects using fan-in analysis. In Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004). (2004)
7. Tonella, P. , Ceccato, M.: Aspect mining through the formal concept analysis of execution traces. In Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004), Delft, The Netherlands (2004).
8. Bruntink, M. Deursen, A. Engelen, R. Tourwé, T., On the use of clone detection for identifying crosscutting concern code, IEEE Transactions on Software Engineering, Vol. 31, No. 10, (2005)
9. Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T.: A qualitative comparison of three aspect mining techniques. In Proceedings on the 13th International Workshop on Program Comprehension, (2005) 13–22
10. Marin, M., Moonen, L. Deursen, A.V.: FINT: Tool Support for Aspect Mining, 13th Working Conference on Reverse Engineering (WCRE 2006), (2006) 299-300,
11. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag, (1999)
12. Gamma, E., Helm, R., Johnson, R, Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995).
13. Kamiya, T., Kusumoto, S., Inoue K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," IEEE Transactions in Software Engineering, vol. 28, no. 7, pp. 654-670, (2002-7).
14. Deursen, A.V., Marin, M., Moonen, L.: AJHotDraw: A Showcase for Refactoring to Aspects. In Proceedings of the Workshop on Linking Aspects and Evolution (LATE05). 4th International Conference on Aspect-Oriented Programming, (2005)
15. Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwé, T.: Applying and combining three different aspect Mining Techniques, Software Quality Control, v.14 n.3, (2006), 209-231
16. Breu, S.: Extending Dynamic Aspect Mining with Static Information, Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), (2005) 57-65
17. Cojocar, G. S., Şerban, G. 2007. On some criteria for comparing aspect mining techniques. In Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution (Canada,. LATE '07). (2007)