Refactoring object oriented software: cross-cutting concerns identification and isolation

Luca Cavallaro
DEI, Politecnico di Milano
cavallaro@elet.polimi.it

Matteo Miraz
DEI, Politecnico di Milano
miraz@elet.polimi.it

ABSTRACT

The capability of dividing a big problem in smaller, independent and more tractable units is crucial for the success of the project. Object oriented decomposition uses to work well to manage decomposition in a software project. Altough there are some concerns that overcome the object oriented software decomposition schema. These concerns are called cross-cutting concerns.
Being able to identify and isolate cross-cutting concerns, present in an object oriented software, can improve the software readability and maintainability. The identification of these concerns is called aspect mining. The mined cross-cutting concerns can be implemented as isolated compilation units, called aspects. Aspect oriented languages were born as object oriented extension to implement isolation of concerns.
In this work we present the problem of cross-cutting concerns and their identification and isolation in an existing object oriented software.

1. THE MODULARIZATION PROBLEM

The problem to properly modularize software is an important issue to be considered when building large projects. A proper modularization impacts code redability, prevents the introduction of bugs, due to loss of control on code complexity, and improves the possibility of introducing adaptive and perfective changes in the project.
To improve the modularization of large projects the object oriented paradigm was introduced. According to this paradigm a program has to be modularized folowing a functional decomposition. Each functionality has to be captured by a *class*. A class should hide the implementation of the functionality and should make the functionality usable by exposing some public interface.
The functional decomposition usually works properly, but for some concerns. These are called *cross-cutting concerns*, since they cross-cut the program structure. Such functionalities are not capturable in a single class and, implementing them following a functional decomposition, implies to spread the implementation on several classes. This results in a tangled code, which is difficult to be read and maintained. Moreover using these classes in the project body implies to scatter, all over the project code, calls to their methods. This makes the system difficult to maintain, since changing some method in one of this classes may imply modifying all the points in the system calling that method. An example of how cross-cutting concerns can impact a system code is reported in figure 1. In this figure two cross-cutting concerns are shown in black and gray. These concerns are scattered through classes `A`, `B` and `C`, making the code difficult to understand and maintain.

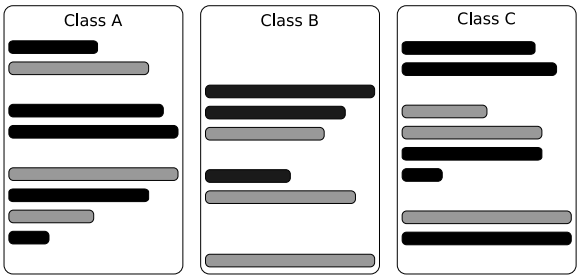

Figure 1: Effect of cross-cutting concerns.

A better modularization for cross-cutting concerns can be achieved through aspect oriented programming [1]. This programming paradigm is an extension of object oriented paradigm to achieve a better isolation of functionalities that cross-cuts the program. Each cross-cutting concern can be implemented in a stand alone file, called *aspect*. Each aspect contains some *pieces of advice* that implement the given functionality. Each advice is associated with some points in the base system code, called *join points*. At run time the code of each advice is executed when those joint points it is associated to, are reached in the program.
Since aspect oriented programming is a relatively new technique, the ability of individuating and isolating cross-cutting concerns into aspects is crucial for refactoring object oriented programs into aspect oriented systems. Such an ability is achieved with far from trivial techniques based on the individuation of common patterns that cross-cutting concerns, embedded in an object oriented code, usually show.

The rest of this paper is so structured: section 2 gives more details on the problem of cross-cutting concerns, section 3

talks about the techniques to find, in an object oriented software, pieces of code candidates to be moved into aspects, section 4 introduces one of the most well known aspect oriented languages, section 5 introduces some techniques to refactor into aspects pieces of object oriented code, marked during a previous aspect mining phase, finally section 6 reports some considerations on aspect oriented programming and its actual applicability.

2. CROSS-CUTTING CONCERNS

One of the main principles of the design of modern software systems is the *separation of concerns*:

> *Separation of concerns allows us to deal with different aspects of a problem, so that we can concentrate on each individually.* [2]

Anyway present languages and methodologies are not able to cope well with all aspects of a problem and force the developer to choose a prevalent decomposition of its problem and let other aspects to cross-cut the system.
For example, suppose that we have to create an application for managing online courses. Designing the application probably we will create classes for teachers, students, and courses. Some problems will appear with requirements like security (e.g. only the teacher of a course can change its description). Probably could create classes for managing those security issues, but we will have to *spread* calls to these functionality among the whole system, as shown in Figure 2.

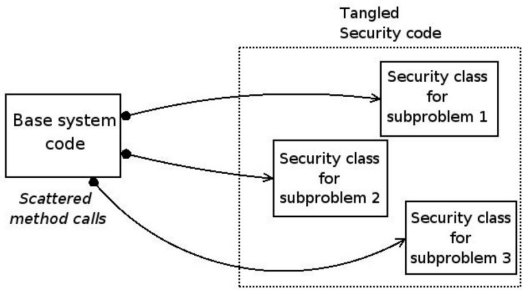

Figure 2: Example of a cross-cutting concern

In [3] the authors study this phenomena, and call it "*the tyranny of the dominant decomposition*": even if a software system is well decomposed into modular units, some functionality will always cross-cut that decomposition. Like the security concern of the example, those requirements cannot be cleanly captured inside one single module with the present modularization techniques.
When a developer has to maintain a cross-cutting concern, he has to localize the code that implements it. Since it may be scattered through different modules, it is possible that he has to inspect different modules. Anyway each piece of code that implements the concern may contain also code that implements other requirements, so the interesting code is tangled with code of other concerns. As a result, the presence of cross-cutting concerns causes both *code scattering* and *code tangling*, and make the maintenance of a software an error-prone and complex task.
In [3] the authors define better those concepts. They call *code scattering* when a single requirement that affects multiple design and code modules, so its implementation code is scattered among several modules. They define *code tangling* when material pertaining to multiple requirements is interleaved within a single module.

The difference in decomposition models leads directly to scattering- - and tanglincmaterial pertaining to multiple requirements is interleaved within a single module. These problems compromise comprehension and evolution, as we will see shortly.

In [1] Kickzales et al. analyze this problem and propose a new modularization technique, called *aspect oriented programming*:

> *We present an analysis of why some code decisions have been so difficult to cleanly capture in actual code. We call the issues this decisions address aspects, and show that the reason they have been hard to capture is that they cross-cut the system's basic functionality. We present the basis for a new programming technique, called Aspect-Oriented Programming, that makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.* [1]

Using this new paradigm it is possible to isolate into different modules concerns that cross-cut the system and make easier to read and maintain a software.

3. ASPECT MINING

Cross-cutting concerns cannot be well modularized in Object Oriented software. The resulting software is composed by a tangled set of object, and some requirements are scattered among different classes. Consequently the code implementing cross-cutting concerns is spread among different parts of an application, and the developer has to cope with code tangling and scattering. As result, the presence of cross-cutting prevents from both understanding and maintaining the program.
Identifying such code (semi-)automatically greatly improves the evolvability of the application. For this purpose there are different approaches that analyze the source code, mine for cross-cutting concerns searching for some *code smells* and return to the developer a *marked code* (an example is reported in Figure 3). Using this code it is possible to link the various concerns to the scattered portions of code that implement it.

Using this information, a developer can identify the portion of the application that implements a cross-cutting concern, so he can easily perform the required changes in all the right places. The net result is that the single change will be less time consuming and less prone to errors.

Moreover the marked code allows a developer to refactor the code using higher abstraction mechanism (like AspectJ [4]),

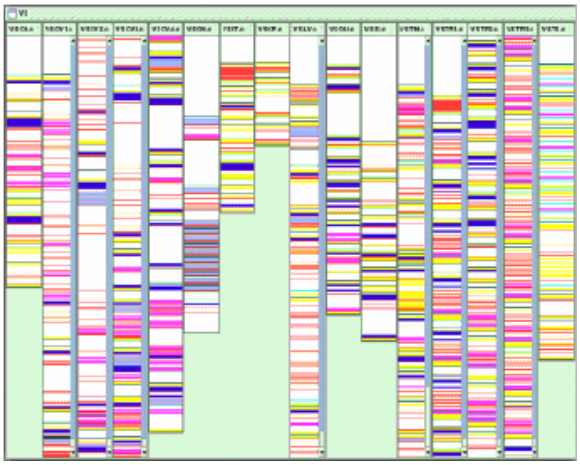

Figure 3: The result of aspect mining.

thereby restoring the modularity of the application.

3.1 Clone Detection Techniques

The *clone detection* techniques allows one to detect copy-cut-paste code, with the intent to refactor it in a single procedure. Generally clones are the result of bad programming techniques, and generate lot of similar code that is very difficult to manage and evolve. For this reason refactoring all clones in a single procedure will greatly improve the quality of the software system.
In some situation the various clones are slightly different, so it is impossible to extract in a common place the functionality. The code that implements this functionality cross-cut the application and cannot be captured cleanly inside a single abstraction, so developers cannot reuse it and they are forced to have (and maintain) multiple copies of similar code scattered among the program.

For this reason in [5] it is presented an approach for aspect mining based on *clone detection* techniques. The authors suggest to use the existing techniques for detecting slightly different software clones also for mining cross-cutting concerns.
Moreover, because of the significant research effort spent on clone detection, the algorithms are both stable and scalable: there are different kind of techniques that are able to detect slightly different software clones:

- **text-based**: attempts to detect identical or similar (sequences of) lines of code.
- **token-based**: applies a lexical analysis to the source code, and use the tokens for clone detection. Comparing tokens instead of raw characters, this techniques are able to ignore useless differences (e.g. white spaces, different comments, new-lines, etc.).
- **AST-based**: converts the code in Abstract Syntax Trees and search for similar subtrees int this AST.
- **PDG-based**: this approach goes one step further working on higher level of abstraction, such as control flow graph of the program.
- **metrics-based**: for each fragment those techniques calculate a value of a metric, which is used subsequently to find similar fragments.

The authors check the validity of their assumption with a case study, that confirms the belief that cross-cutting functionality is often implemented using similar piece of code scattered among the system. In particular they find out that some concerns (e.g. parameter checking) are implemented with very similar piece of code, and in these cases clone detection techniques are able to detect them very well. Anyway there are concerns (e.g. error handling and tracing) that are more problematic and these techniques cannot find them.

3.2 Fan-in analysis

The *fan-in* metric counts the number of location from which control is passed into a module. Formally, the number of distinct methods that can invoke a method M is the *fan-in* value of M.

In [6] it is proposed to use the *fan-in* metric to drive the aspect mining. The authors stated that methods with higher *fan-in* contain functionality that are required in different part of the system, or in other words those methods contain a concern that is scattered among the whole system. Typical cross-cutting concerns that imply methods with high *fan-in* are logging, tracing and contract enforcement. The algorithm for mining cross-cutting concerns with this technique is:

1. Automatically computing the fan-in value for all methods of the program
2. Filtering the result of previous step, eliminating:
   - all methods with low fan-in (e.g. lower than 10)
   - all accessor methods (e.g. getters and setters)
   - all utility methods (e.g. toString)
3. Analyze manually the set of results, checking if there is a cross-cutting concern.

This technique is able to detect cross-cutting concerns that are largely scattered and have significant impact on the modularity of the system. The downside is that concerns with small footprint (i.e. with lower *fan-in*) are omitted.

3.3 Random Walks

In some works researchers study the behavior of a human searching for a cross-cutting concern. They find out that usually the human aspect miner examines the source files, selects a random element and proceeds with two random walks.
During the first round the miner gathers all the program elements that are known by the selected element. The likelihood of visiting the next element depends on how many outgoing relation the current element has. In this way it is possible to calculate the "*popularity*" of each element, simply counting the number of references to that element.
In a similar way, the second round walks through elements

that know about the selected module, allowing one to define the "*significance*" of all elements. Given an element e, its *significance* depends on the number of elements that has a reference to e.

In [7] the authors tried to formalize the concept of *Random Walks*. They model a class or a method with an element, and define that an element A knows an element B if:

- A invokes B
- A extends B
- A has a field of type B

In this way they are able to build the knowledge graph of the system, and using page-ranking algorithms and Markov models they are able to find out both homogeneous and heterogeneous cross-cutting concerns. As defined in [8], homogeneous concerns are used uniformly in the code space – such as in the case of logging – while heterogeneous concerns consists of several different pieces of code scattered throughout the code space.

3.4 Event Traces

Silvia Breu et. al. propose to mine for aspects using event traces. At the beginning they started collecting those traces from a running system; this approach will be explained in 3.4.1. They found out that using dynamic analysis, they have some scalability issues, since they have to run the application in order to collect data. For this reason, they proposed a static analysis approach that is able to identify and collect event traces performing a static analysis. This approach is presented in 3.4.2 and allows them to build a scalable aspect miner, able to mine aspects also in very large software repositories.

3.4.1 Dynamic analysis

Dynamic analysis analyzes the properties of a system according to its run-time behavior. In particular the program is instrumented, so during its execution is able to record its execution traces. Those traces reflect the real run-time behavior of the system and it is possible to investigate for particular execution patterns. The main advantage of dynamic analysis is that works on the actual behavior of the system, instead of potential behavior (as static analysis do). The downside of dynamic techniques is that they strongly rely on the particular execution trace: if a statement is not executed in any recorded trace, it will not be considered during the analysis phase.

In [9] is presented the first approach that mines for aspects using program traces. The algorithm to mine cross-cutting concerns is:

1. Insert probes in the code that trace the beginning of the execution of a method m – creating an entry (m,ent) – and the end of its execution – creating an entry (m,ext).
2. Run the instrumented version of the program and collect the execution traces.
3. Mine for those categories of relations:
   - *outside-before-execution-relation*: the method b is executed after the end of the method a (i.e. in the trace exists $[(a, ext), (b, ent)]$).
   - *outside-after-execution-relation*: the method a is executed after the end of the method b (i.e. in the trace exists $[(a, ext), (b, ent)]$).
   - *inside-first-execution-relation*: the method b is executed at the beginning of the execution of the method a (i.e. in the trace exists $[(b, ent), (a, ent)]$).
   - *inside-last-execution-relation*: the method b is executed at the end of the execution of the method a (i.e. in the trace exists $[(a, ext), (b, ext)]$).
4. Filter out non-*uniform* relations. A relation $a \rightarrow b$ is *uniform* if doesn't exist any relation $c \rightarrow b$ with $c \neq a$.
5. Filter out non-*cross-cutting* relations. A relation $a \rightarrow b$ is *cross-cutting* if exist $a \rightarrow c$ with $c \neq a$, i.e. it occurs in more than a single calling context of the program trace.
6. The resulting set of relations are aspect candidates, as they represent potential cross-cutting concerns of the analyzed program.

3.4.2 Static analysis

The same authors tried to analyze the differences between static and dynamic analysis in aspect mining. In [10] they develop a static analysis variant of their approach. They extract the *execution relations* presented in the previous article from a control flow graph of the analyzed program. Moreover, since they analyze directly the source code, it is possible to extract directly *uniform* relations. Since this new kind of analysis works on potential behavior, the result that it produces are slightly less accurate than the one extracted by their dynamic approach.

The creation of a static aspect mining approach based on the *execution relations* allows the authors to adapt that technique in order to mine software repositories, as presented in [11, 12]. The authors argue that programmers have to cope with cross-cutting concerns while they develop it, so it is possible to search for cross-cutting commit and probably they represent an underlying cross-cutting concern. In particular they report that when it was changed the Eclipse locking mechanism, the developer "Silenio Quarti" has performed a huge commit that involves 2'573 methods with calls to `lock` and `unlock` primitives. The locking is a cross-cutting concern, and it is possible to reuse the work performed by Silenio Quarti to discover the code that implements it in the Eclipse platform.
The approach starts collecting the changelog of the program, collecting for each transaction T this information:

- *developer(T)*: who has performed the change.
- *timestamp(T)*: when the change was committed.
- *locations(T)*: the methods changed in the commit.
- *calls(T)*: what calls were added in the commit.

The algorithm will try to find aspect candidates within the set T of transactions. An aspect candidate represents a cross-cutting concern since it consists of *calls* to methods spread across several *locations*. If the number of methods called is one, the aspect is called "simple"; if it is greater than one, it is called "complex". The authors suggest to filter out aspect candidates with a low number of locations, since those elements do not cross-cut a considerable part of the system. Either if those elements can represent real cross-cutting concerns, the authors suggest to filter them out in order to have more precise results and have less noise.
Subsequently the authors find out that several cross-cutting concerns were introduced within one transaction and later new locations were added to that concern. In order to detect also those new locations, they introduce locality and reinforcement. Two transactions are locally related if they were created by the same developer or were committed around the same time. If there exists locality between transactions, the corresponding aspect candidates will reinforce each other. With the reinforcement process it is possible to merge together two aspect candidates that share the same method calls.
Since many cross-cutting concerns involve more than a single method call, the authors provide the way to merge two simple aspect candidates and create a complex one. If there are two aspect candidates that rely on the same location and with different method invocations, it is possible to merge them and create a new aspect candidate with the same location and the union of calls.
Finally the authors suggest to filter the results and to take care about the first n results. They propose three main rankings:

- **by size**: candidates that cross-cut more locations are more interesting than smaller one. The downside of this ranking technique is the noise, since there are lot of correlated method calls in Java that don't represent a cross-cutting concern. (e.g. `iter()`, `hasNext()` and `next()`).

- **by fragmentation**: when a cross-cutting concern is added to the system, it will not be changed anymore, so it appears in only one transaction. For this reason the authors suggest to sort from the lowest to the highest the aspect candidates using the number on transaction in which they were found.

- **by compactness**: this ranking technique combines the ranking by size with the ranking by fragmentation, keeping the advantages of ranking low common Java sequence of methods.

3.5 Other techniques

The research on aspect mining proposed also a lot of other approaches, some of them are sketched in this paragraph.

In the absence of special language constructs, developers exploit naming convention to associate related but distant program entities. In [13] the authors propose to use *identifier analysis* to identify potential aspects grouping program entities with siminal names. This technique is able to produce a lot of detailed results, but there is too much noise (false positives) and the discovered concepts are often incomplete (a single concern is not completely discovered).

In [14] the authors presented a technique called *Dynamic analysis*. Execution traces for a set of use-case scenarios are obtained by running an instrumented version of the program. Those traces are processed with formal concept analysis that allows to identify both code scattering and code tangling. The main drawbacks of this approach rely on the analysis of the execution traces, that doesn't guarantee the completeness of the result and requires that it is possible to create and run an instrumented version of the program.

4. ASPECTJ

The AspectJ language [4] is an aspect oriented extension of the Java language. An AspectJ system is composed by an object oriented base system and some *aspects*. Aspect is the modularization unit that contains the implementation of a cross-cutting concern, in an aspect oriented program. The system, composed by Java classes and AspectJ aspects, is called *augmented system* and is compiled into regular Java bytecode, using an AspectJ compiler. The compiled bytecode is inserted in those join points, in the base system bytecode, selected by the developer. This process is performed using a *weaver* [15]. The resulting bytecode can be executed by a standard Java Virtual Machine. The weaving process is shown in figure 5.

Using aspects to implement cross-cutting concerns solves

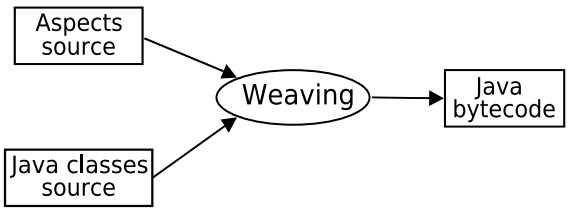

Figure 4: Weaving process schema

both the code tangling and the call scattering problems. The code, implementing a cross-cutting concern, will be totally encapsulated in an aspect, avoiding the tangling problem, moreover this code does not need to be called inside the base system, since it is automatically inserted, by the AspectJ compiler, in some points specified by the developer himself.

AspectJ is composed by a dynamic part and a static part. The dynamic part is represented by *join points*, *pointcuts*, and *advice*. The static part is represented by *inter-type declarations*. Both the dynamic and the static parts of the AspectJ language are implemented within *aspects*.

4.1 Join points and pointcuts

A *join point* is a well-defined point in the execution of a program where a part of the implementation of a cross-cutting concern have to be inserted. Examples of join points are a method call, a field access, an object initialization, a constructor call, and many other points. Join points are not specific to AspectJ programs.
The AspectJ language uses *pointcuts* to pick a join point

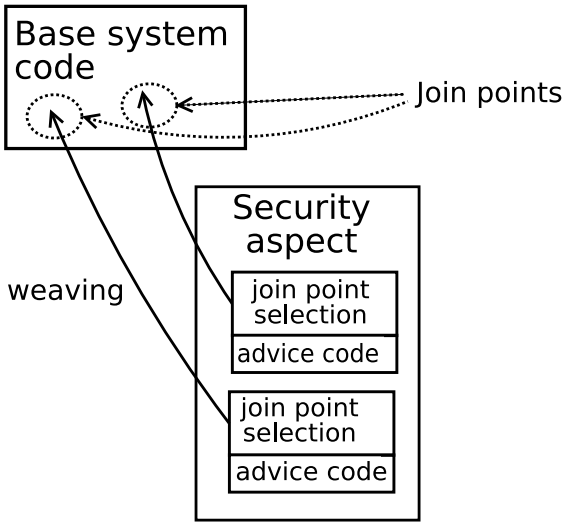

Figure 5: Weaving process schema

or a set of join points, and identify points of interest in the program. AspectJ pointcuts are divided in four main categories:

- methods pointcut
- member or static fields pointcuts
- exception handling pointcuts
- control flow pointcuts.

The first pointcut category captures the join points related to invocation and execution of a given method. Some pointcuts that fall in this category are the `call` and the `execution` pointcut. The `call` pointcut can be used to identify all program points that contain a method call to a specific method, the `execution` pointcut can be used to identify all the join points in a method execution. Line 6 of listing 1 shows a `call` pointcut, that identifies every point in the program where the `equals` method of `Object` class is called.
The member or static fields pointcuts capture read or write access to member or static fields in a class. An example is the `get` pointcut, that captures all read access to a specified class member field. A `get` pointcut, identifying the read access to the field `out` having type `Printstream` of the class `System`, is reported on line 8 in listing 1.
Exception handling pointcuts capture join points associated with the execution of exception handlers for a given exception type. An example of this kind of pointcut, that captures the join point associated with the execution of handlers for `IOException`, is reported on line 11 of listing 1.
Control flow pointcuts capture pointcuts based on other pointcuts' control flow (the flow of program instructions). For example, if in an execution, method `a()` calls method `b()`, then `b()` is said to be in `a()`'s control flow. With control flow based pointcuts is possible to capture all methods, fields access, and exception handlers caused by invoking a method. An example of control flow based pointcut is shown on line 14 of listing 1. This pointcut captures all the join

Listing 1: Some simple AspectJ pointcuts

```
1  package examples;
2  import java.lang.*;
3
4  public aspect PickEquals {
5
6    pointcut equals():
7      call( boolean Object.equals(Object) );
8
9    pointcut exampleAccessPointcut():
10       get(PrintStream System.out);
11
12   pointcut exampleHandlerPointcut():
13       handler(IOException);
14
15   pointcut controlFlowPointcut():
16       cflow(public execution(Myclass.Mymethod(..)));
17
18 }
```

point in the execution of the method `Mymethod` in the class `Myclass`.

Even if pointcuts are classified as static constructs their translation process finds the statical projection of each pointcut and instruments it. The statical projection of a pointcut is called *pointcut shadow* and is a portion of the system code whose join points are captured by the pointcut [16].

4.2 Pieces of advice

Pointcuts are used to specify program join points at which pieces of *advice* are executed. Pieces of advice contain executable code, that can effectively change the behavior of a system. The AspectJ language defines three types of advice:

- *before* advice
- *after* advice
- *around* advice

Before advice is used to execute code immediately before the program point, specified by the associated pointcut, is reached.
After advice is used to execute code immediately after the execution of the program point specified by the pointcut.
Around advice is run *instead* of a method specified by the pointcut. An example aspect, with two pieces of advice, is shown in listing 2. The `Telescope` class is shown in listing 3. Before coordinates of a `Telescope` are set, the *before* advice is executed. After the execution of the `set` method, the *after* advice is executed.

Listing 4 contains around advice. Before the `set` method is executed, the *around* advice checks whether coordinates are valid. If the passed coordinates are valid, the execution of the `set` method is resumed, using a `proceed` statement. Otherwise, the advice resumes the execution of the `set` method using a default coordinate pair. In a piece of *around* advice, the method identified by the pointcut is executed only if an explicit `proceed` instruction is used.

Listing 2: An aspect with two pieces of advice

```
1  package examples;
2
3  public aspect CoordSet {
4    pointcut coordSet(int x_axis, int y_axis):
5      call ( public void Telescope.setCoords(int , int))
6        && args(x_axis,y_axis);
7
8    before (int x, int y): coordSet(x,y) {
9      System.out.println("Changing coordinates to " +
10       x + " , " + y);
11   }
12
13   after (int x, int y): coordSet(x,y) {
14     System.out.println("Coordinates changed to " +
15       x + " , " + y);
16   }
17
18 }
```

Listing 3: The Telescope class

```
1  package examples;
2
3  public class Telescope {
4    public void setCoords(int x, int y) {
5      // set coordinates
6    }
7  }
```

Listing 4: A piece of *around* advice

```
1  package examples;
2
3  public aspect AroundCoordSet {
4    pointcut coordSet(int x_axis, int y_axis):
5      call ( public void Telescope.setCoords(int , int))
6        && args(x_axis,y_axis);
7
8    void around (int x, int y): coordSet(x,y) {
9      if ((x >= 0 ) && (y >=0)) {
10       proceed(x,y);
11     } else {
12       proceed(0,0);
13     }
14   }
15 }
```

Listing 5: The CardHolder interface

```
1  package examples.university;
2
3  public interface CardHolder {
4    public int getCardNumber();
5    public void setCardNumber(int number);
6  }
```

4.3 Inter-type declarations

Aspects can also be used to modify the class hierarchy of a Java system, and to add new fields to existing classes. These features of the AspectJ language are called *inter-type declarations*. Inter-type declarations are mainly useful to modify an existing system. Consider an information system for a university, containing two classes: `Student` and `Professor`. Now, suppose that the university installs magnetic card readers that control access to laboratories. Every student and professor is given a magnetic card with a unique number. The information system has to be extended to allow getting and setting the card number for professors and students.
A new interface can be created, and inter-type declarations can be used to make the original class implement the new interface. The new `CardHolder` interface is shown in listing 5.

An aspect containing the necessary inter-type declarations is shown in listing 6. Lines 4 contains a `declare parents` instruction. This instruction modifies the class hierarchy of the original system, making the `Student` and `Professor` class implement the `CardHolder` interface. However, any class implementing this interface has to implement the appropriate methods.
Inter-type declarations can also be used to declare new fields and methods for existing classes. Line 7 is used to add a private attribute to any class implementing the `CardHolder` interface. The semantics of `private` in this attribute declaration is different from the semantics of `private` in Java class attribute declaration. Declaring the `cardnumber` as private makes the attribute visible only within the declaring aspect. This allows another aspect in the system to declare another `private` attribute for the same classes. Finally, lines 9-15 declare two public methods for every class implementing the `CardHolder` interface.

5. REFACTORING OBJECT ORIENTED CODE INTO ASPECTS

Cross-cutting concerns embedded in object oriented code usually lead to a code difficult to read and maintain, since tangled code is difficult to understand and maintain. Moreover when the code, implementing a cross-cutting concern, needs to be changed it is necessary to change also the points in the system that calls methods in that code. Because of scattered method calls, it is necessary to change many places in the source code. This operation is expensive, since an extensive set of regression tests needs to be generated.
This problem can be overcome if code, implementing cross-cutting concerns, is mined and highlighted in the object oriented code, as it was shown in section 3, and refactored into aspects. Refactoring cross-cutting concern into aspects is an

Listing 6: The CardHolding aspect

```
1  package examples.university;
2
3  public aspect CardHolding {
4    declare parents: (Student || Professor)
5      implements CardHolder;
6
7
8    private int CardHolder.cardnumber;
9
10   public int CardHolder.getCardNumber() {
11     return cardnumber;
12   }
13
14   public void CardHolder.setCardNumber(int number) {
15     cardnumber=number;
16   }
17
18 }
```

operation that needs appropriate tool support. Refactored system must be semantically equivalent to the object oriented code, and, manually moving pieces of code identified as cross cutting concerns, is an error prone operation.
In [17] is presented a tool to semi-automatically extract and refactor cross cutting concerns into aspects. This tool implements an algorithm in five steps:

- Aspect mining
- Discovering refactory actions
- Object oriented to object oriented transformations
- Object oriented to aspect oriented refactoring

A schema of the algorithm is reported in figure 6.
The first step of the algorithm is an aspect mining phase that highlights some portions of code in the object oriented system. This marked code is the one that can implement cross-cutting concerns and is fed to the second phase that discovers the possible transformations from object oriented to aspect oriented code. The discovering takes place comparing the marked code with some pattern rules. These rules can be specified using a rule language that comes with the tool [18]. Each rule is composed by a pattern part, that specifies a pattern to be matched, and a refactoring part, that specifies refactoring actions to be performed when the pattern in the first part is recognized.
If the set of discovered transformations is empty the marked code is fed to the third phase. This phase discovers and performs, on the marked code, object oriented to object oriented refactoring operations. These operations have the purpose of re-engineering object oriented code to enable transformations in the list found at step two of the algorithm. If some transformation is enabled at the end of this step the developer is asked to select a transformation from object oriented to aspect oriented code, within the available ones. The last step of the algorithm actually performs code refactoring. Steps from the second to the last one are repeated iteratively while some marked code exists and is possible to discover some object oriented to aspect oriented refactoring operation.

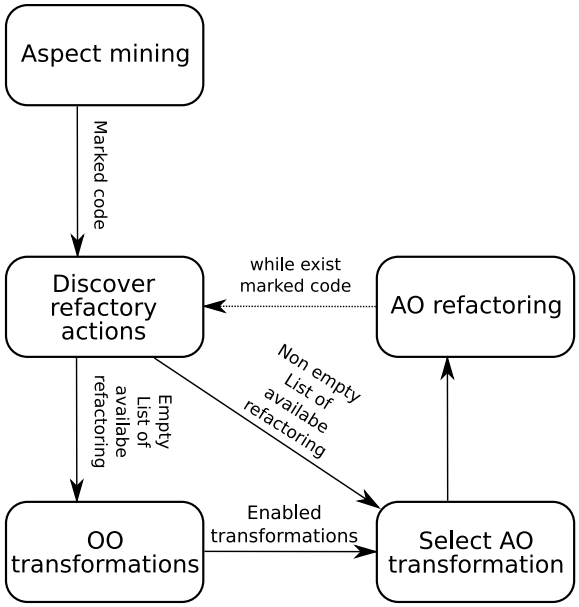

Figure 6: Refactoring Algorithm

The last step of the algorithm can perform three kinds of transformations:

- Marked code is a whole method.
- Marked code is composed by method calls.
- Marked code is composed a sequence of statements.

The first case can be, straightforwardly, refactored moving the marked method into an aspect and using AspectJ intertype declaration to add the moved method as a member of the class it was removed from. The third case can be reduced to the second case, moving the statements in the sequence into a method and substituting them with a method call. The second case can be refactored specifying a pattern and a refactoring action to perform, when that pattern is matched, using the rule language. The tool offers a set of predefined refactoring actions that falls in the second case:

- Extract beginning (or end) of a method or exception handler. The marked code is at the beginning/end of the enclosing method body or of one of the method's exception handling blocks. The refactoring action for this case is moving the call into a before (or after) advice, associated with a pointcut that intercepts calls to the method or execution of the exception handler containing the marked code.
- Extract before or after call. The marked code is always before or after a method call. The refactoring action for this case is moving the marked code into an around advice associated with a pointcut, that intercepts all join points, respectively, before or after the given method call.

- Extract Conditional. A conditional statement controls the execution of the marked code. We can safely assume that the conditional statement is an if statement, since other conditional statements can be reconducted to this one. The refactoring action for this case is removing the conditional statement and its true branch from the base system, moving the true branch into an around advice. This advice is associated with a pointcut that intercepts the execution of the method, containing the conditional statement. The pointcut has also to contain the same condition of the refactored if statement.

- Pre Return. The marked code is just before the return statement. The refactoring action for this case is moving the call from the method body to an around advice, associated with a pointcut that intercepts the call to the method containing the marked code. The advice code contains a proceed invocation. The proceed invocation transfers control to the intercepted method, triggering its execution. The return value of the proceed invocation is stored into a temporary variable and returned by the around advice.

A developer can define more refactoring actions by defining new pattern rules.

6. CONSIDERATIONS ABOUT AOP

The key idea of the aspect oriented programming paradigm is to allow separation of concerns, providing a way to isolate cross-cutting concerns from the base system. Cross-cutting concerns can be implemented in an aspect. An aspect contains some pieces of advice that can be associated with some join points in the base system. Each advice is automatically executed when the join points, it is associated to, are reached in the program.

The association of an advice to a join point is performed only sintactically. For instance a pointcut that intercepts the call to a method identifying the method by name. If the method name changes the pointcut will become ineffective. This change requires the possibility to automatically update the advice or to generate regression test cases [19].

One of the main drawbacks of the aspect oriented approach is that system code can be misleading. This happens because by inspecting the source code of the object oriented part of the system, it is not possible to understand what is actual system behavior at run time, since there is no reference to the execution of an advice in the base system code. To predict the system behavior it is necessary to manually perform the weaving operation, which is not suited for humans. Moreover aspects can be designed independently, both from the base system and from other aspects woven in the system. This allows a better separation of cross-cutting concerns, but can also lead to interference between aspects introduced in the system. Two aspects, independently designed, can read and write the same variables in the system, causing an inconsistent system behavior. These problems can be overcome using program analysis techniques, to better understand augmented system semantics [20].

7. REFERENCES

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241*, 1997.

[2] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2003.

[3] P. Tarr, H. Ossher, W. Harrison, and SM Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 107–119, 1999.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[5] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for crosscutting concerns. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 200–209, 2004.

[6] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 132–141, 2004.

[7] Charles Zhang and Hans-Arno Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD*, pages 226–238, 2007.

[8] A. Colyer and A. Clement. Large-scale AOSD for middleware. *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, 2004.

[9] S. Breu and J. Krinke. Aspect mining using event traces. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 310–315, 2004.

[10] J. Krinke and S. Breu. Control-flow-graph-based aspect mining. 2004.

[11] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for cross-cutting concerns. In *Proceedings of the Third International Workshop on Mining Software Repositories*, pages 94–97, May 2006.

[13] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 97–106, 2004.

[14] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Mik Kersten, Matt Chapman, Andy Clement, and Adrian Colyer. Ajdt. http://www.eclipse.org/ajdt/.

eclipse plugin.

[16] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD'04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[17] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

[18] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Guoqing Xu and Atanas Rountev. Regression test selection for aspectj software. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 65–74, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. On bytecode slicing and AspectJ interferences. In *FOAL'07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, ACM International Conference Proceedings Series, pages 35–43, Vancouver, British Columbia, Canada, March 2007. ACM, ACM Press.