Automated Refactoring of Object Oriented Code into Aspects

Dave Binkley(1), Mariano Ceccato(2), Mark Harman(3), Filippo Ricca(2), Paolo Tonella(2)

(1) Loyola College, Baltimore, MD, USA
(2) ITC-irst, Trento, Italy
(3) King's College London, UK
binkley@cs.loyola.edu, ceccato@itc.it, mark@dcs.kcl.ac.uk, ricca@itc.it, tonella@itc.it

Abstract

This paper presents a human–guided automated approach to refactoring object oriented programs to the aspect oriented paradigm. The approach is based upon the iterative application of four steps: discovery, enabling, selection, and refactoring. After discovering potentially applicable refactorings, the enabling step transforms the code to improve refactorability. During the selection phase the particular refactorings to apply are chosen. Finally, the refactoring phase transforms the code by moving the selected code to a new aspect. This paper presents the results of an evaluation in which one of the crosscutting concerns of a 40,000 LoC program (JHotDraw) is refactored.

1. Introduction

Aspect Oriented Programming (AOP) provides explicit constructs for the modularization of the *crosscutting concerns* of programs: functionalities that traverse the principal decomposition of an application and thus cannot be assigned to a single modular unit in traditional programming paradigms. Existing software often contains several instances of such crosscutting concerns, such as persistence, logging, caching, etc. The AOP paradigm is expected to be beneficial for software maintenance because it separates the principal decomposition from other crosscutting functionalities.

However, in order to take advantage of the potential benefits of the AOP style of programming, there is a need for migration of existing applications and systems. This paper presents a semi-automated approach to the process of migration from the Object Oriented Programming paradigm (in Java) to the Aspect Oriented Paradigm (in AspectJ). The approach builds on existing work on aspect mining, combining this with novel work on refactoring to produce an end-to-end, human-guided OOP to AOP migration approach. The paper reports the results of a medium sized (40,000 LoC) migration effort using the proposed approach.

Software refactoring consists of the modification of internal program structure without altering the semantics (*i.e.*, external behavior). It aims at improving internal quality factors (*e.g.*, modularity), in order to make the code easier to understand and evolve. As with other migrations and conversions [4, 17, 27], the maintenance–migration effort involved in OOP to AOP must be automated to avoid unsustainable cost.

However, in common with other re-factoring and code migration work [27], human guidance is both necessary and desirable; the process requires value-judgments regarding trade-offs best made by a maintenance engineer. The process of migrating existing software to AOP is highly knowledge-intensive and any refactoring toolkit therefore should include the user in the change-refine loop. However, notwithstanding this inherent human involvement, there is considerable room for automation. Previous work has been concerned with two potential avenues for automation:

- aspect mining – identification of candidate aspects in the given code [2, 19, 23, 25], and
- refactoring – semantic-preserving transformations that migrate the code to AOP [11, 20, 26].

This paper focuses on the second of these two. It considers the challenging, and hitherto unsolved, problem of determining sensible pointcuts to intercept the execution and redirect it to the aspect code. Such a process must be guaranteed to preserve the original behavior, while modularizing the code of the crosscutting functionality.

Our approach to this problem derives from the field of program transformations and amorphous slicing [13, 14]. Its outcome entails one or more aspects containing the crosscutting code, with pointcuts able to redirect the execution whenever necessary. Manual refinement of such an

outcome, for example to generalize the pointcut definitions, remains an advisable step. Overall, the cost associated with the refactoring activity to migrate OOP to AOP is expected to be greatly reduced through automation.

The primary contributions of the paper are as follows:

1. Novel refactorings are introduced for the migration from OOP to AOP.

2. These refactorings are combined with existing transformations in a prototype tool for automating the refactoring process.

3. The feasibility of automating the migration from OOP (in Java) to AOP (in AspectJ) is demonstrated using one, selected, crosscutting concern of the JHotDraw case study (a 40,000 LoC Java program to which the approach and prototype tool is applied). The results demonstrate that the end-to-end migration of a medium size system can be achieved with only 5 relatively simple refactoring rules and two enabling transformations.

The paper is organized as follows: Section 2 briefly presents background material on AspectJ to make the paper self-contained. Sections 3 and 4 introduce the overall refactoring process and the detail of the refactorings respectively. Section 5 presents the results of the case study, while Section 6 sets the approach presented here in the context of related work. Finally, Section 7 concludes with directions for future work.

2. Background on AspectJ

Among the programming languages and tools that have been developed to support AOP, AspectJ [16], an extension of Java, is one of the most popular and best supported. The main new programming constructs provided by AspectJ are *pointcuts*, *advices*, and *introductions*.

The behavior of an aspect is specified inside *advice*, which takes a form similar to a method. The advice is woven into the program by a weaver at *join points*. These well-defined points in the program flow, are identified using *pointcuts*. Pointcuts define where execution is to be intercepted and redirected to the advice. Finally, *introductions* may be used to add members (attributes or methods) or generalization/implementation relationships to a class. Unlike advices, which alter the dynamic behavior, introductions operate statically on the class members and structure. It is these changed classes that are instantiated by the rest of the program.

For pointcut p, advice can be woven in “before” the join points identified by p, “after”, or “in place of” them. The advice associated with these pointcuts are referred to as *before-advice*, *after-advice* and *around-advice*, respectively. For example, if a persistence aspect is defined to serialize all objects of class *Person* as soon as they are created, an appropriate pointcut can be used to intercept calls to any constructor of this class. In AspectJ, it looks like

```
aspect PersistentPerson {
  pointcut personCreation(String data):
    call(Person.new(String)) && args(data);
  after (String data):  personCreation(data)
    { /* save Person data to database */ }
}
```

The advice executed after the pointcut *personCreation* saves information about the *Person* object being created into a database. The AspectJ keyword `args` is used to expose the `String` parameter of the constructor, making it available inside the after-advice.

3. Refactoring Process Overview

This section introduces the iterative process for the migration of existing OOP code to AOP code. It assumes that prior aspect mining has been conducted and that the output of aspect mining consists of a *marked* program, in which the code fragments to be aspectized are surrounded by the markers $\alpha()$ and $\omega()$.

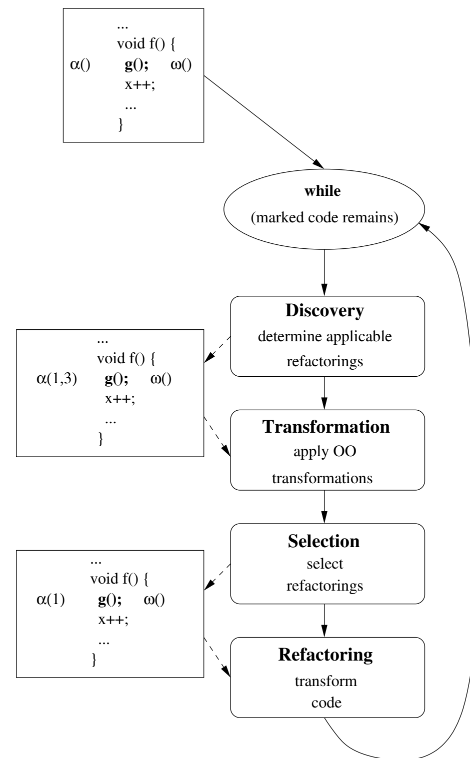

Figure 1. Steps of the refactoring process.

Figure 1 overviews the refactoring approach in which four steps are iterated until no marked code remains (*i.e.*, all the identified concerns have been aspectized).

The first activity aims at determining which refactorings for the aspectization of OOP code are applicable to each marked code fragment. The output of this activity is a (possibly empty) list of applicable refactorings (*e.g.*, $\alpha(1,3)$ indicates that the Refactorings 1 and 3 are applicable). This activity can be fully automated.

The second activity is used when no refactoring applies to a block of marked code and it is believed that this situation will not change in later iterations. In other words, future aspectization of other (usually, neighboring) code fragments will not make any refactoring applicable. In such cases, OO transformations are executed in order to make the refactorings applicable in the future.

OO transformation consists of applying well-known transformations [7] (for example, *extract-method*) in the hope of enabling refactoring. Identification of the places where it is appropriate to apply transformation is driven by the markers – empty refactoring lists suggest that OO transformation might be useful. However, the final decision about which transformation to choose and where and how to apply it, must necessarily rest with the user, since only the user is in a position to evaluate the trade-offs and value–judgments involved. Once such a decision is made, the application of the OO transformation is fully automated.

The third activity is refactoring selection: whenever more than one refactoring labels a marked block of code, a selection must be made to reduce the number of refactorings to one or zero. Here, zero means processing of the fragment is to be deferred to a later iteration, in the hope that a better refactoring will become applicable. A single element is left within a marker if the associated refactoring is the most appropriate for the given code fragment.

Selection is aided by a prioritization scheme among the refactorings; some are considered better choices than others. Priority is based on the quality of the resulting aspect code. Section 4 provides example priority rules used to select among the refactorings considered in this paper.

In some instances, the priority rules alone can be sufficient to select one of the possible refactorings. However, in other cases a human decision is more appropriate. Thus, the overall activity is automated, but allows for human guidance.

The final step refactors the code through transformation. It consists of the removal of the code marked with one refactoring and the generation of the associated aspect code. This step is fully automated. The paper adopts the convention that the word *Transformation* (Step 2) is used to refer code manipulation which takes and returns OOP code. This is to be contrasted with *Refactoring* (Step 4), which takes OOP code but which returns AOP code.

4. Refactoring Process

The refactorings described in this section create the pointcuts and advices necessary to aspectize the code. Each refactoring provides definitions for *pointcuts* and *advices* that replace marked code. The regions of code, marked by $\alpha()$ and $\omega()$, to which the refactorings apply fall into three cases:

1. whole methods,
2. calls to methods, or
3. statement sequences (blocks of code).

The paper focuses on call extraction and statement sequences (Cases 2, 3). Method extraction (Case 1) is straightforward and requires simply moving the whole method from a class to an aspect, where it is turned into an AspectJ introduction. Case 3 is conceptually similar to Case 2, (provided that the local variables referenced in the marked block of statements are not referenced outside it). With Case 2, the marked call is moved into the body of the aspect advice being generated. With Case 3, the marked block of code is moved into the body of the aspect advice being generated.

The four steps that make up each iteration of the algorithm are now detailed. The first step identifies potential refactorings. A refactoring can be applied when the following applicability conditions associated with the marked block of code apply.

Extract Beginning and Extract End. The block of code is at the beginning or end of the body of the enclosing method. (As these two are virtually identical, in the sequel, Extract Beginning is used to represent both these refactoring.)

Extract Before Call and Extract After Call. The block of code is always before or after another call. (Hereafter Extract Before Call is used to represent these refactorings.)

Extract Conditional. A conditional statement controls the execution of the block of code.

Pre Return. The block of code is just before the return statement.

Extract Wrapper. The block of code is part of a wrapper pattern, in which the wrapper code is to be aspectized.

4.1. Discovery and Transformation

The TXL language [6] was used to implement the discovery and refactoring phases of the process. TXL supports the definition of grammar–based rules to perform a source-to-source code transformation. Each rule (*e.g.*, see Figure 2) is divided into two parts, a `replace` part, containing the pattern to be matched, and a `by` part, containing the replacement. A pattern is composed out of pattern variables (conventionally uppercase), followed by their type (a grammar non-terminal, in this case from the Java Grammar, which appears within square brackets), and terminals, optionally preceded by the quote character. When the pattern of a rule is matched, all the pattern variables are bound to the roots of the related sub-trees in the program's syntax tree. The replacement consists of a syntax tree that is constructed by composing terminals, pattern variables (bound to some syntax sub-tree) and possibly invoking library or user defined functions or rules (mixed case, within square brackets, followed by an argument list), which are applied to a syntax sub-tree and return a new sub-tree of the same type.

```
rule ExtractBeginning
  replace $ [ method_declaration ]
    M    [ repeat modifier ]
    TS   [ type_specifier ]
    MD   [ method_declarator ]
    T    [ opt throws_declaration ]
    '{
        α(MARKUP [ markup ])
        STM [ declaration_or_statement ] ω()
        REST [ repeat declaration_or_statement ]
    '}
  by
    M TS MD T
    '{
        α(MARKUP [ AddRef EB ])
        STM ω()
        REST
    '}
end rule
```

Figure 2. TXL rule to check the applicability of the Extract Beginning refactoring.

The TXL code for two of the five discovery rules is given in Figures 2 and Figures 3. Others are similar, but are omitted due to space considerations. The code in Figure 2 discovers marked code to which the Extract Beginning refactoring applies. This rule applies to all syntax tree nodes of type `method_declaration` (the first line within the rule body). The pattern variables `M, TS, MD, T` are matched by the Java code that defines a new method (*e.g.*, modifiers, including `public` or `private`, the return type, then the method name and parameters, and finally optional raised exceptions). After the header, the method body (within braces) is matched. Its first statement, `STM`, of type `declaration_or_statement`, has been marked. `STM` is followed by a sequence of zero or more statements (`REST`).

The output of the rule is identical to its input, except for the AST matched by `MARKUP`, which is modified by an invocation of the user defined function `AddRef`. This function adds `EB` (Extract Beginning) to the list of applicable refactorings.

```
rule BeforeCall
  replace $ [ method_declaration ]
    M    [ repeat modifier ]
    TS   [ type_specifier ]
    MD   [ method_declarator ]
    T    [ opt throws_declaration ]
    '{
      DECLSTATLIST [ repeat declaration_or_statement ]
    '}
  by
    M TS MD T
    '{
      DECLSTATLIST [ AddBeforeMark ]
    '}
end rule

rule AddBeforeMark
  replace $ [ repeat declaration_or_statement ]
    α(MARKUP [ markup ])
    STM [ declaration_or_statement ]    ω()
    C [ id ] '. H [ id ] MA [ method_argument ] ';
    REST [ repeat declaration_or_statement ]
  deconstruct not * [ declaration_or_statement ] REST
    CC [ id ] '. H MMA [ method_argument ] ';
  by
    α(MARKUP [ AddRef BC ])
    STM    ω()
    C '. H MA ';
    REST
end rule
```

Figure 3. TXL rule to check the applicability of Call Before refactoring.

The TXL code to check the applicability of the

OO Transformation 1 (Statement Reordering)

$$\frac{\mathtt{DEF}(\mathtt{st}_1) \cap \mathtt{REF}(\mathtt{st}_2) = \emptyset,\quad \mathtt{REF}(\mathtt{st}_1) \cap \mathtt{DEF}(\mathtt{st}_2) = \emptyset,\quad \mathtt{DEF}(\mathtt{st}_1) \cap \mathtt{DEF}(\mathtt{st}_2) = \emptyset}{[\![st_1;\ st_2]\!] \Rightarrow [\![st_2;\ st_1]\!]}$$

DEF(s) : Defined variables of s.
REF(s) : Referenced variables of s.

Figure 4. Statement Re-ordering Transformation.

second Refactoring, *Call Before*, is shown in Figure 3. The first rule, `BeforeCall`, simply binds the pattern variable `DECLSTATLIST` to the body of the `method_declaration`. In the replacement, the user defined rule `AddBeforeMark` is applied to `DECLSTATLIST`. In this way, all sub-trees of type `[repeat declaration_or_statement]` are evaluated against the pattern of `AddBeforeMark`.

The rule `AddBeforeMark`, shown at the bottom of Figure 3, checks if the first statement (`STM`) in the input statement list is marked. In the rule pattern, the marked statement is followed by a method call, captured as an identifier (`C`), a dot, another identifier (`H`), the arguments of the method call (`MA`), and finally a semicolon. The remainder of the input statement sequence (`REST`) must not contain the same call. In fact, the pointcut that intercepts the execution point preceding the call is not unique if multiple identical calls are present. The TXL instruction `"deconstruct not *"` is used to verify that the subtree `"CC . H MMA';"` is not matched inside `REST`. Similar to Extract Beginning, the TXL code adds BC (Before Call) to the list of applicable refactorings.

After discovery, the second step applies OO transformations where no refactoring applies to a marked block of code. The tool currently employs two OO transformations: Statement Reordering and Extract Method.

Statement Reordering, shown in Figure 4, allows the order of two statements (st_1 and st_2) to be exchanged. Above the line in the rule is the pre-condition. It requires that the defined and referenced variables of the two statements do not overlap. When some overlap does occur between defined and referenced variables, it may be possible to make this transformation applicable by introducing fresh local variables that store a value that must be preserved.

The second OO Transformation, Extract Method, allows a sequence of statements to be turned into a separate method [7]. Method arguments might be required if local variables or parameters of the original method are referenced in the marked statement block. This transformation makes it possible to aspectize any marked block of code. However, it impacts the structure of the base code very deeply, so it is used as sparingly as possible and as a 'last resort'.

4.2. Selection and Refactoring

The algorithm's third step chooses which refactoring to apply in cases where multiple refactorings label a marked block of code. It is easier to explain the motivation for the selection process after the refactorings have been described. Therefore steps three and four are described out of order.

The final step applies the *refactorings*. Each refactoring is described and illustrated by an example of the transformation from Java to AspectJ. The first refactoring, Extract Beginning (and Extract End), deals with the following case

> *The call/block to be moved to the aspect is at the beginning/end of the body of the calling method.*

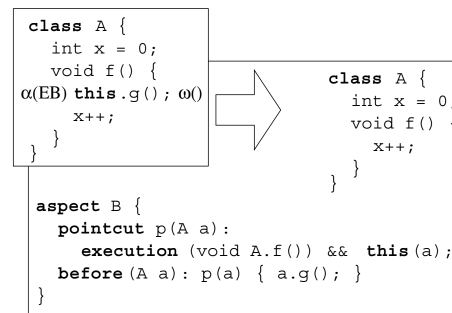

Figure 5. Extract Beginning.

Figure 5 shows the result of applying Extract Beginning to a small code fragment that matches the applicability condition shown in Figure 2. The call to method `g` is removed from the body of `f`. A new aspect, named `B`, is introduced to intercept the execution of `f` and insert a call to `g` at the beginning. The target of the call (`this`) is accessible within the aspect advice thanks to the advice parameter `a`, which is bound to `this` by pointcut `p`.

If the target of the call had been a method parameter, it can be made accessible within the advice thanks to the `args` construct supported by AspectJ. If the target of the call is a class attribute, it is accessible within the advice through the variable bound to `this` (e.g., `a.x` for the attribute `x`). Unfortunately, it is not possible to easily access `x` if it is a local variable. In such a case, it is possible to create a copy of the local variable or move it to the advice, assuming it is not used outside the extracted code. Alternatively, an OO transformation can be first applied.

In general, the marked block of code will be more than a single parameterless method call. For such cases the variables used in the code to be aspectized, including variables used as actual parameters, must be exposed to the aspect (*e.g.*, using the `this` and `args` AspectJ constructs). Of course, when the marked call/block is at the end of the enclosing method, an after-advice is used instead of the before-advice.

The second refactoring deals with the following case

> *The call/block to be moved is always before/after another call.*

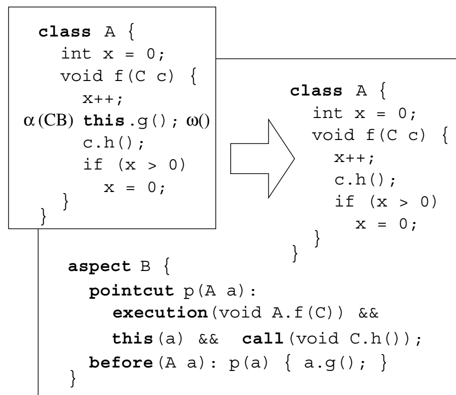

Figure 6. Call Before.

Figure 6 shows the code transformation produced by Call Before. In the aspect `B`, the pointcut `p` intercepts the call to `h` that occurs within the execution of method `f`. A before-advice reintroduces the call to `g` at the proper execution point. If the target of the call to be aspectized is a method parameter or a class attribute, the associated pointcut must be modified as described for Extract Beginning refactoring. Finally, for After Call, when the marked block follows the intercepted call, an after-advice is used.

The third refactoring deals with the following case

> *A conditional statement controls the execution of the call/block to be moved to the aspect.*

Figure 7 shows the mechanics of this refactoring. The conditional statement `if (b)` is considered to be part of the aspect, in that it determines the execution of the call being aspectized (`g()`). Thus, it becomes a dynamically checked condition incorporated into the aspect's pointcut (using the AspectJ syntax `if(a.b)`). For the execution to be intercepted by pointcut `p`, the condition `a.b` must be **true**. In which case, the new body of method `f` is replaced by the call to `g`, as specified in the around-advice. Two variants of Conditional Execution are worth mentioning. First, if the `"x++;"` were not under the control of condition `b`

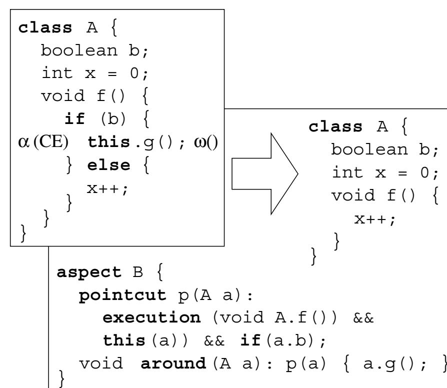

Figure 7. Conditional Execution.

(placing it at the top-level in `f`) it would be sufficient to add `proceed()` at the end of the around-advice to ensure that it is always executed (both when the advice is triggered and when the execution flows normally). Second, if `g()` is in the else-part of the conditional statement, it is sufficient to use `if(!a.b)` instead of `if(a.b)` in the pointcut.

If the block to be aspectized includes references to class attributes, method parameters or local variables, the considerations described above for Extract Beginning apply. This includes variables referenced in the condition `b`.

The fourth refactoring deals with the following case

> *The call/block to be moved is just before the return statement.*

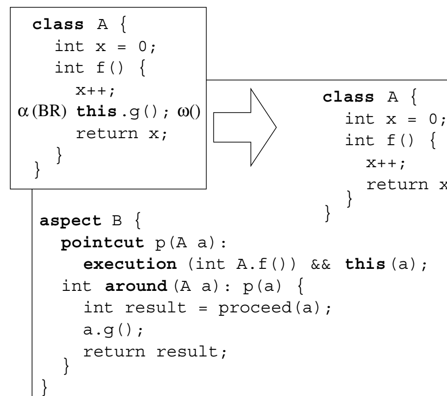

Figure 8. Before Return.

Figure 8 shows the mechanics of Before Return. The call to `g()` is moved from the method body to the around-advice. The advice code contains a `proceed` invocation that triggers the execution of the intercepted method `f()`. Its return value is stored into a temporary variable (`result`) and returned after the invocation to the aspectized statement (*i.e.*, `g()`). The underlying applicability

condition is that there is no dependency between the marked code and the returned value, which can thus be computed before the call/block is executed.

This refactoring is a variant of Extract End that occurs whenever the code to be aspectized is at the end of a method that returns a value. Since the applicability condition and the generated aspect are quite different from those associated with Extract End, this case is considered a separate refactoring.

The fifth refactoring deals with the following case

> *Objects from a given hierarchy are wrapped before being used and the wrapper class is to be aspectized.*

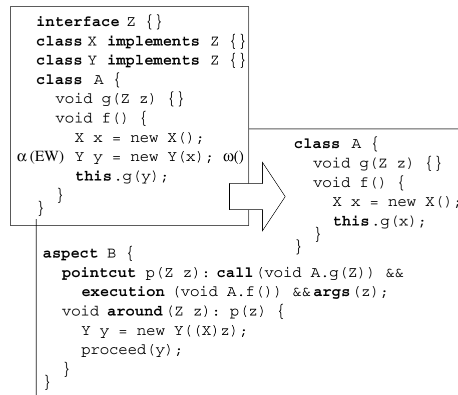

Figure 9. Extract Wrapper.

Figure 9 shows an example of Extract Wrapper. The object `x` is wrapped into `y` before being used as the actual parameter of a call to `g`. In order to move the creation of the wrapper object (second statement inside `f`) to the aspect, the un-wrapped object `x` is used in the refactored code for the method `f` as the actual parameter of the call to `g`. Such a call is intercepted by the pointcut `p`, which exposes its argument. The associated around-advice uses this argument, which is known to belong to class `X`, to create the wrapper object `y`. This object is passed to `g` by restoring the original method invocation (`proceed` construct), with a new argument.

Similar to Call Before, Extract Wrapper is applicable only if the body of `f` contains just one call to `g`. If this is not the case, application of this refactoring in an alternative form can be considered. The pointcut `p` may intercept the creation of the object `x`, instead of the call to `g`, by means of the pointcut designator `"call(X.new())"`. By exposing the target of the call to the constructor (`target` construct in AspectJ), the un-wrapped object `x` can be made available within the around-advice, which will contain exactly the same code as the around-advice shown in Figure 9.

4.3. Priorities

The third step makes use of priorities to help guide an engineer when multiple refactorings apply to the same marked code. The outcome is the selection of a single refactoring or the decision to defer the marked code to a later iteration. A relative scale of priorities among the refactorings, is motivated below. It is based on the impact that each refactoring has on the original code and the complexity and quality of the aspect code that is generated.

First off, Extract Condition and Extract Wrapper are not included in the priority scale, because they are associated with very specific patterns that are not compatible with the other refactorings. When they match, they are always applied, unless refactoring of the given code is deferred to a later iteration by the engineer.

At the top of the scale, Extract Beginning and Extract End, is the preferred refactoring. Here, the base code is not altered except for the removal of the call/block to be aspectized. The aspect code relies only on the fact that the execution of the original method can be intercepted. Thus, this is a very simple and non-invasive refactoring.

Next highest priority is given to Before Return. The motivation here is similar to that of Extract End and is based on the impact on the base code and complexity of the aspect code. It is only in the case of a call/block before a return that a more complex advice is necessary and that the returned value must not depend on the aspectized code.

The final choice is Before Call and After Call. These refactorings are applicable only if the call used to intercept the original execution is unique. Therefore, this assumption must be verified to see if this refactoring is actually applicable, but it must also remain true during the evolution of the source code. In fact, if a second call, similar to the intercepted one, is added later, the aspect ends up intercepting more execution points than necessary and introducing a bug. Thus, the aspect code generated at this priority level is more fragile than at the previous levels, so that its usage should be limited as much as possible.

Two priority modifiers are used. First, although in principle there is no difference between the *before-call* and the *after-call* refactorings, in practice, the user might prefer the former or the latter according to the semantics of the aspect and of the intercepted call. This is one of the reasons why the automation of refactoring selection allows human guidance. The second modifier places a refactoring at the lowest priority if one of the two OO transformations is necessary to enable it. OO transformations introduce minor code changes in order to enable some refactoring. It is preferable to avoid such changes wherever possible in order to preserve the original code structure.

5. Case study

To evaluate the approach, it was applied to the problem of extracting the ubiquitous and important UnDo crosscutting functionality from the program JHotDraw. JHotDraw version 5.4b1 is a Java program consisting of approximately 40,000 lines of code and 249 classes. It is an Object-Oriented framework for the development of applications supporting the interactive creation and manipulation of graphical objects. It was originally conceived as representative of the best practices in the usage of the design patterns [8].

It was selected for the case study because it has been subjected to several different aspect mining techniques, becoming the "de-facto" reference benchmark for the works on aspect identification and refactoring (*e.g.*, it was used in a comparative study based on the results obtained on JHotDraw using three different aspect mining techniques [5]).

The implementation is based on two TXL modules: the *Refactoring detector* and the *Refactoring executor*. The first module takes as input the marked source code and adds the (possibly empty) list of applicable refactorings to the code markers used to markup the code.

The output of the *Refactoring detector* is processed manually by the user, who selects the refactorings to apply (by leaving at most one refactoring in a list). Moreover, in this phase the user can choose to apply OO transformations in order to enable other refactorings.

Then, the TXL module *Refactoring executor* executes the selected refactorings, by removing the aspect code from the original source files. The generation of the related pointcut and advice code, poses no conceptual difficulty. It is not currently implemented, but it is simply a matter of some further TXL development.

5.1. Results

Refactoring of all the JHotDraw code pertaining to the UnDo functionality was achieved in four iterations. Table 1 describes the effects of the OO transformations (Step 2) applied. The last row gives the percentage of refactorings that were enabled by the OO transformations, which provides an indication of the proportion of refactorings that require some transformation in order to become applicable. Taken together 19.8% of the refactorings required an OO transformation; thus, a large majority of the refactorings (80.2%) are applicable without any need for OO transformation.

Table 2 shows the number of each refactoring applied during each iteration. One reason for deferring a refactoring to a later iteration is the hope that another refactoring will enable it without the need for OO transformation. Another reason is that it might be possible to apply a higher priority refactoring (*e.g.*, a *before-call* might become a *method-beginning* if the preceding code fragment is first moved to an aspect).

OO Transformations Instances		
Iteration	*Statement Reordering*	*Method Extraction*
1	5	23
2	0	2
3	0	0
4	0	0
Total	5	25
Refactorings Enabled	3.3%	16.5%

Table 1. OO transformations applied to JHotDraw at each iteration.

Refactoring Instances					
Iteration	*Extract Begin (End)*	*Call Before (After)*	*Extract Cond*	*Before Return*	*Extract Wrapper*
1	32	26	4	3	40
2	14	16	0	1	0
3	5	8	0	0	0
4	1	1	0	0	0
Total	52	51	4	4	40
	34.4%	33.7%	2.7%	2.7%	26.5%

Table 2. Refactorings applied to JHotDraw at each iteration.

Finally, it should be noted that the actual number of iterations executed (4) may vary, depending on the way the source code is marked. For example, it is possible either to mark individual statements or to mark compounds. In the first case, refactoring of a statement might enable the refactoring of another statement during the next iteration, so that more iterations are required to achieve the final result. In contrast, when a whole block is marked, the refactoring can be achieved in a single step. In both cases, the refactorings used and the resulting code are the same.

5.2. Lessons Learnt

The case study was conducted to understand three things: whether the five refactorings are sufficient to migrate an existing application to AOP; how often the OO refactorings are required and what is the quality of the resulting code. The results collected using JHotDraw indicate that the refactorings are sufficient, but are not equally important. The refactorings *Extract Beginning*, *Extract End*, *Call Before*, *Call After*, and *Extract Wrapper* made up 94.6% of the refactoring instances applied. The *Extract Condition* and

Before Return refactorings are less important; they make up only 5.4% of the refactoring instances applied. This is expected to hold for other software systems, except for the identification of *Extract Wrapper*, which was somewhat application-specific.

It is encouraging that over 80% of the refactorings required no OO transformation. However, OO transformations are required to achieve 100% extraction of the UnDo cross cutting concern. The most heavily used OO transformation is Extract Method. Such a refactoring is very powerful because of its general applicability. However, it must be considered the final 'extreme recourse that solves all refactability problems'. This refactoring might reduce the code recognizability and quality in general; it is applied only when absolutely necessary. The proportion of code fragments that required it was fortunately, reassuringly low (around 17%). This percentage forms one possible indicator of the good quality expected from the refactored code. In fact, all the other refactorings produce aspect code that is very close to the one a programmer would write manually. At the same time, the base code remains the same, except for the removal of the aspect code.

6. Related work

In the migration of existing OOP code to AOP, the problem that has received most attention is the detection of candidate aspects (aspect mining) [2, 3, 9, 12, 15, 19, 21, 22, 23, 25], while the problem of refactoring [1, 11, 20, 26] was considered only more recently. The present paper takes the results of aspect mining as its starting point and focuses on the problems associated with automating the refactoring process. Human guidance is important to ensure that inherent value judgments are taken into account.

Some of the various aspect mining approaches rely upon the user definition of likely aspects, usually at the lexical level, through regular expressions, and support the user in the code browsing and navigation activities conducted to locate them [9, 12, 15, 21]. Other approaches try to improve the identification step by adding more automation. They exploit either execution traces [2, 23] or identifiers [25], often in conjunction with formal concept analysis [23, 25]. Clone detection [3, 22] and fan-in analysis [19] represent other alternatives in this category.

The most closely related works are by Marin [18], Hanenberg et al. [11], Monteiro and Fernandes [20], Gybels and Kellens [10] and Tourwe et al. [24]. Marin, manually refactored the UnDo concern from JHotDraw to AOP (AspectJ). The primary difference between this work and that reported in the present paper lies in the degree of automation available. Marin's goal was to understand the degree of tangling between the UnDo concern and the base code. Similarly to the work reported here in, Marin applies preliminary OOP refactorings to reduce tangling; thus, producing easier to migrate code.

Hanenberg's work deals with the re-definition of popular OOP refactorings taken from Fowler [7] in order to make them aspect-aware. This work and the work by Monteiro and Fernandes [20] consider refactorings to migrate from OOP to AOP and refactorings that apply to AOP code. Among them, the *Extract advice* refactoring [11] (or *Extract Fragment into Advice* [20]) is the one we aim to automate in this paper.

Gybels and Kellens, and Tourwe's work uses inductive logic programming to transform an extensional definition of pointcuts (that just enumerate all the join points), into an intensional one, which generalizes the former by introducing variables where facts differ (anti-unification). The underlying assumption is that the pointcut definition language is rule-based (this is not the case, for example, with AspectJ, the target language of the present paper). This work is complementary to that reported in the present paper, because the problem of generalizing and abstracting the automatically produced pointcuts is not our focus, but is definitely a desirable supplementary step in the overall process.

7. Conclusions and future work

This paper introduces a semi-automated approach to support the migration from OOP code to AOP code. In particular, the applicability of semantic-preserving code refactoring transformations to automate the migration task is considered. Given a source program with the aspectual fragments marked, the tool produces a semantically equivalent program with the marked fragments migrated to aspects.

The approach was evaluated using a medium size (40,000 LoC) case study program, JHotDraw. One of its crosscutting concerns, the UnDo functionality, was successfully migrated to AOP using our refactorings.

When combined with existing (and automated) approaches to aspect identification, tool-supported migration from OOP to AOP is achieved. Overall, the tool applied 151 refactorings in order to extract the UnDo crosscutting concern. A large fraction of the code to be aspectized is extracted automatically and in most cases the separation of concerns was achieved with only minor impact on the structure of the base code.

Acknowledgments

The authors would like to thank Jim Cordy and Tom Dean for the interesting discussions and the help given for the TXL implementation. Dave Binkley is supported, in part, by National Science Foundation grant CCR0305330. Mark Harman is supported, in part, by EPSRC Grants GR/R43150, GR/R98938, GR/S93684 and GR/T22872 and by two development grants from DaimlerChrysler.

References

[1] P. Borba and S. Soares. Refactoring and code generation tools for AspectJ. In *Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, Seattle, Washington, USA, November 2002.

[2] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. of Automated Software Engineering (ASE 2004)*, pages 310–315, Linz, Austria, September 2004. IEEE Computer Society.

[3] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 200–209. IEEE Computer Society, September 2004.

[4] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. In 6th *IEEE International Workshop on Program Comprehension*, pages 136–144, Ischia, Italy, June 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[5] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *Proc. 13th IEEE International Workshop on Program Comprehension (IWPC)*, page (to appear). IEEE Computer Society, May 2005.

[6] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[7] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[8] E. Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[9] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of the 2001 International Conference on Software Engineering (ICSE)*, pages 265–274, Toronto, Canada, March 2001. IEEE Computer Society.

[10] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts. In *Proceedings of the First European Interactive Workshop on Aspects in Software*, 2004.

[11] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, September 2003.

[12] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001.

[13] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[14] M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering*, 11(1):27–61, Jan. 2004.

[15] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, Boston, Massachusetts, USA, March 2003. ACM press.

[16] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[17] S. Klusener and C. Verhoef. 9210: The zip code of another IT-soap. *Software Quality Journal*, 12(4):297 – 309, Dec. 2004.

[18] M. Marin. Refactoring jhotdraw's undo concern to aspectj. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE2004)*, November 2004.

[19] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, The Netherlands, November 2004. IEEE Computer Society.

[20] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, March 2005.

[21] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, Orlando, FL, USA, May 2002. ACM press.

[22] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In *MASPLAS*, April 2004.

[23] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, November 2004.

[24] T. Tourwe, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse. Inductively generated pointcuts to support refactoring to aspects. In *Proceedings of the Software Engineering Properties of Languages for Aspect Technology (SPLAT) Workshop at AOSD '04*, 2004.

[25] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 97–106, Chicago, Illinois, USA, September 2004. IEEE Computer Society.

[26] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*, Waterloo, Canada, November 2003.

[27] M. Ward. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)*, Oxford, UK, Aug. 1999. IEEE Computer Society Press, Los Alamitos, California, USA.