An Initial Experiment in Reverse Engineering Aspects from Existing Applications

Magiel Bruntink & Arie van Deursen* & Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
{*Magiel.Bruntink, Arie.van.Deursen. Tom.Tourwe*}*@cwi.nl*

Abstract

In this paper, we evaluate the benefits of applying aspect-oriented software development techniques in the context of a large-scale industrial embedded software system implementing a number of crosscutting concerns. Additionally, we assess the feasibility of automatically extracting these crosscutting concerns from the source code. In order to achieve this, we present an approach for reverse engineering aspects from an ordinary application automatically. This approach incorporates both a concern verification *and an* aspect construction *phase. Our results show that such automated support is feasible, and can lead to significant improvements in source code quality.*

1. Introduction

In recent years, aspect-oriented software development (AOSD) [11] has become increasingly more popular, thanks to its undeniable benefits for software engineering. AOSD techniques offer additional language constructs, which allow software developers to capture crosscutting concerns in cleanly separated modules. This improves important quality attributes such as maintainability, evolvability and understandability, among others [11, 19].

Up till now, the primary focus of AOSD research was on aspect language technology. In particular, the requirements, design and implementation of the aspect languages have been studied in extensive detail. Now that this research has reached a mature level, numerous applications start to appear that employ AOSD techniques, thereby showing its benefits in practice. However, little or no attention is paid to how these applications evolved from ordinary, mostly object-oriented, applications into aspect-oriented applications. Some applications were simply developed from scratch, whereas others were ported to the aspect-oriented paradigm manually. Research concerning the migration of large-scale legacy applications to aspect-oriented applications is practically non-existent, even though this is an extremely important and relevant issue.

We can distinguish between two types of crosscutting concerns. Homogeneous concerns implement the same behaviour repeatedly at different locations in a system, whereas heterogeneous concerns implement different behaviour, related to the same functionality, at such locations [5]. In the absence of specifically-designed language constructs, lower-level means, such as naming and coding conventions, are often used for implementing homogeneous crosscutting concerns. The primary advantage of such techniques is that they are lightweight, i.e. they do not require special-purpose tools and are easy to use. At the same time, they still allow developers to recognise concern code in a straightforward way. The downside however is that these techniques require a lot of discipline, make concern code evolution extremely time consuming and may lead to code size explosion. Naming and coding conventions are but a poor man's means of implementing crosscutting concerns, and they do not scale very well, in particular when they are not enforced. Additionally, concern code, like all other code, is subject to evolution, which turns out to be very difficult if it is spread all over the application. Consequently, when an application has reached a particular size, developers start to realise the need for explicitly handling crosscutting concerns. At that point, automated support for transforming the naming and coding conventions approach into an aspect-oriented approach is indispensable. Such support includes identifying, verifying and extracting these concerns.

Our goal in this paper is two-fold. First, we want to evaluate whether and how AOSD techniques influence the code quality of an application. In other words, we want to verify the often touted benefits of AOSD when compared to simple naming and coding conventions. Second, we want to assess the feasibility of automatically reverse engineering crosscutting concerns from the source code of a large-scale industrial application. Based on an initial experiment with such an ap-

*CWI and Delft University of Technology

plication, the contributions of this paper are the following:

- we show that AOSD techniques can help in reducing the code size and (thus) in improving the understandability and evolvability of an application.
- we present a systematic approach and tool support for reverse engineering characterisations of homogeneous crosscutting concerns from the source code and for verifying whether the implemented concerns adhere to these characterisations.
- we provide insight into how concerns can be extracted automatically from the source code and defined in the appropriate aspects.

It is important to note that, in this paper, we do not focus on automatically *discovering* crosscutting concerns. Rather, we show how such concerns can be extracted from the code, based on a-priori knowledge we gathered about them by talking to developers and reading the documentation. However, this does not mean that we know the precise characterisations of these concerns. As we will show, these can be extracted from the code using an iterative process.

The remainder of the paper is structured as follows. The next section provides a short introduction to AOSD. Section 3 introduces the case study and presents the concerns we considered in our experiments. Section 4 presents our approach to aspect extraction, while Section 5 shows the results of applying this approach on our case study. A discussion of these results, together with opportunities for future research are presented in Section 6. Section 7 discusses related work, and finally Section 8 presents our conclusions and future work.

2. Aspect-Oriented Software Development

2.1. General Introduction

Aspect-oriented programming languages introduce a new language feature, called an *aspect*. Aspects are an extra abstraction mechanism, on top of the already existing mechanisms, that allow us to capture crosscutting concern code in a well-modularised manner. By introducing aspects, AOSD aims at offering several software engineering benefits [11, 19]:

- it improves understandability, since aspects allow all concerns to be cleanly separated, and concern code is thus no longer scattered all over the source code.
- it improves maintainability, since aspects reduce code duplication and bugs are easier to correct when the code is well modularised.
- it improves evolvability, since both the original code and the concern code itself become easier to change or adapt.
- it improves reuseability, since existing decomposition techniques, such as functional decomposition and inheritance, can be used inside the aspect itself, and since aspects themselves can be reused in different contexts.

Clearly, such benefits are particularly important and highly desired in an industrial context, where applications spanning millions of lines of code are the rule rather then the exception. In this paper, one of our goals is to evaluate whether such benefits are indeed feasible.

2.2. Aspect Definition

The definition of an aspect consists of, amongst other things, *pointcut definitions*, that pick out particular *join points*, and *advice definitions*, that associate code to be executed to the places identified by a particular pointcut. We will explain these terms in more detail below, and refer to the AspectJ documentation [1] for an in-depth discussion of all features.

Any aspect-oriented programming language is based on a *join point model*. Join points are basic events in the execution of an application, such as method calls, method executions, object instantiations, and so on (in an object-oriented setting). AspectJ, for example. defines `call`, `execution` and `initialization` join points for these events.

Pointcuts pick out particular join points of interest to the aspect. They do this by specifying a *type pattern*, that can contain wildcards. For example, the pointcut `execution(* m(..))` picks out the join point representing the execution of a method `m` with a variable number of arguments (the `..` pattern) returning a value of any type (the `*` pattern). Different pointcuts can be composed using the *and* (`&&`), *or* (`||`) and *not* (`!`) operators.

Advice definitions are associated to the pointcuts of an aspect, and define the code that should be executed at those places. In AspectJ, such code can be executed *before, after* or *around* the original code, so three different kinds of advice code exist: *before, after* and *around* advice. Moreover, pointcuts can expose to the advice code some information about the context in which they occur, such as the specific method signature or the number of arguments of the method, and so on.

We will illustrate these concepts with concrete examples in subsequent sections.

2.3. Terminology

In the remainder of this paper, we will use some aspect-related terminology, that is explained here.

[1] www.aspectj.org

In an AOSD context, the code implementing the aspects is normally called the *aspect* code, whereas the code implementing the basic functionality of the application is referred to as the *base* code.

With regard to the nature of the concerns, we can distinguish between *tangled* and *non-tangled* concerns. The latter kind of concern stands on its own, and its code does not interfere with the basic functionality of the application. The former kind, however, does interfere, which means the concern code and the basic functionality are interwoven. In this paper, we will show examples of both tangled and non-tangled concerns. Additionally, *homogeneous* and *heterogeneous* concerns exist [5], as already mentioned. For a homogeneous concern, the code fragments are semantically (and syntactically) very similar, whereas a heterogeneous concern is implemented by a number of code fragments which are not semantically similar. We will only consider homogeneous concerns in this paper.

3. The ASML Case Study

3.1. Background

The research reported on in this paper is part of a larger research effort, in which we try to provide automated support for identifying and refactoring crosscutting concerns in industrial applications. The subject system upon which we perform our experiments is an embedded system developed at ASML, the world market leader in lithography systems. The entire system consists of more than 10 million lines of C code. Our experiment, however, is based on a relatively small, but representative, software component (which we will call the *CC* component in this paper

Because the C language lacks explicit support for crosscutting concerns, ASML uses an idiomatic approach for implementing such concerns, based on coding conventions. As a consequence, a large amount of the code of each component is “boiler plate” code. A code template is typically reused and adapted slightly to the context.

3.2. Considered Concerns

The crosscutting concerns we focus on for our experiment are the following:

error handling is responsible for raising, catching and logging errors. Since the C language has no support for exceptions, such behaviour has to be implemented explicitly. Since errors can occur everywhere, this implementation is scattered all over the system.

parameter tracing is responsible for logging the values of input and output parameters of C functions, in order to facilitate debugging.

Concern	LoC	Fraction
Error handling	1716	9%
Parameter tracing	1539	8%
Parameter checking	1441	7%
Total	4696	24%

Table 1. Code percentages devoted to various concerns in the CC component

parameter checking is responsible for implementing pointer checks and raising warnings whenever such a pointer contains a non-expected value.

These concerns appear over the entire ASML source code base, and comprise roughly 24% of the code of the CC component, as shown in Table 1. All three concerns are homogeneous, but the parameter checking and tracing concerns are non-tangled concerns, whereas the error handling concern is a tangled concern.

In the remainder of this paper, we will frequently use the parameter checking concern for illustrating our approach and discussing the results we obtained. Therefore, we explain it here in more detail.

3.3. Parameter Checking Concern

The ASML code distinguishes between three different types of function parameters: *input, output* and the special case of *output pointer* parameters. Input parameters can be pointers or values. Output parameters are represented as pointers to locations that will contain the result value. The actual values returned can be references themselves, giving rise to a double pointer. The latter kind of output parameters are called *output pointer* parameters. Note that the set of output pointer parameters is a subset of the set of output parameters.

Both input and all output parameters should be checked against null pointers. These checks occur at the beginning of a function and always look as follows:

```
if(queue == (CC_queue *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Input parameter error (NULL)",
            func_name));
}
```

where the type cast of course depends on the type of the variable (`queue` in this case). The second line sets the error that should be logged, and the third line reports that error in the global log file. Although it is not strictly specified which string should be passed to the `CC_LOG` function, it is in principle always the same. Checks for output parameters look exactly the same, except for the string that is logged.

Since output pointer parameters are output parameters as well, they should also be checked for null values. Additionally, one extra check is required to prevent memory leaks.

The convention at ASML is that output pointer parameters may not point to a location that already contains a value, because the function will overwrite the pointer to that value. Since the original value is then never freed, a memory leak could occur. In order to avoid such leaks, the following test is added for each output pointer parameter:

```
if(*item_data != (void *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Output parameter may already "
             "contain data (!NULL). This data will "
             "be overwritten, which may lead to memory "
             "leaks.", func_name));
}
```

The only difference with the previous test lies in the condition of the `if`, and in the string that is written to the log file.

4. An Aspect Extraction Approach

In this section, we present the approach we use for reverse engineering aspects from an existing software application. We first provide a general overview and afterwards discuss each phase in more detail in subsequent sections. We will illustrate important concepts by means of representative real-world examples, taken from the actual source code of the CC component.

4.1. General Overview

The reverse engineering of an ordinary application into an aspect-oriented one can be decomposed into two different tasks:

Aspect mining: identify and locate the relevant crosscutting concerns in the source code.

Aspect refactoring: define the appropriate aspects and restructure the base code in an aspect-oriented way.

Aspect mining is an active research field of its own [3, 7, 2, 16]. As already explained previously, we will not consider it here, since in our experiment the concern code was identified manually by a domain expert. Moreover, we will only focus on how aspects should be re-engineered from the source code (a process we call *aspect extraction*), and not on how this source code should be restructured afterwards. This last step should be considered future work.

The approach we propose for aspect construction consists of two distinct phases, as depicted in Figure 1 (where the arrows show the chronological flow of information between them). This partitioning is directly related to the use of simple naming and coding conventions for implementing crosscutting concerns. Because such conventions do not scale and are not actively enforced, slight variations, violations and local deviations in their implementation are inevitable. Consequently, we first want to evaluate to what extent the source code adheres to the required conventions, as documented in developer manuals. The result of this *concern verification* phase is a set of rules to which the code should adhere, and a set of exceptions to these rules. A concern verifier is thus needed in order to characterise the rules to which the concerns should adhere, to ensure that correct aspects are extracted, as well as to facilitate advice code construction, as we will discuss later on. Due to the exceptions detected by the concern verifier, defining the appropriate aspect is not at all a trivial task. It requires advanced reasoning, provided by the *aspect construction* phase, for defining the right pointcuts and constructing the advice code.

In the next two sections, we will discuss each of these two phases in more detail.

4.2. Concern Verification

In the concern verification phase, a *concern verifier* will analyse the source code and verify whether it implements the concerns in a correct and consistent way. The term "correct" refers to the fact that the code implementing the concern should adhere to the coding conventions that hold for that concern. The term "consistent" on the other hand, means that the concern should be implemented wherever necessary, e.g. all locations in the source code where the concern should be implemented, implement it. Clearly, correctness and consistency cannot be guaranteed (easily) for a number of reasons: the code base is large, the concern code is spread over the entire application, there is no way to reuse such code, and coding conventions are not actively enforced most of the time. It is however of prime importance that concerns are implemented correctly and consistently if we want to extract them. If this is not the case, we will end up with incorrect aspects.

From a reverse engineering point of view, the concern verifier can also be used as a means to characterise the existing idioms and coding conventions used to implement the concerns. In order to be able to verify the concern code, the verifier should of course know these idioms and conventions, and needs to verify these in a strict manner. Unfortunately, such idioms and coding conventions are only documented informally in manuals, and the specific way in which they should be implemented is often not strictly specified. Moreover, different developers almost certainly have different coding styles, and may even interpret the documentation in different ways.

In order to overcome this problem, we consider implementing a concern verifier as an iterative process, that is used to recover the particular interpretations of idioms and conventions used by different developers from the source code. In our experiment, we implemented a first version of the concern verifier based on the concern requirements specified in the developer's manual. The resulting deviations that were reported by the verifier where then presented to the main de-

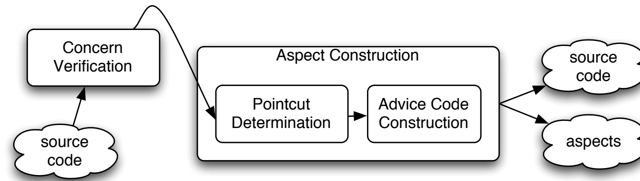

Figure 1. Aspect Extraction Approach

veloper of the CC component. Based on his findings, we were able to refine the verifier further, resulting in less reported deviations, that were also more accurate. This process continued until it was no longer sensible to refine the verifier.

4.3. Aspect Construction

The goal of the aspect construction phase is to generate the appropriate aspects corresponding to the way the concerns are implemented in the source code and are verified by the concern verifier. Such construction involves *advice code construction* and *pointcut determination*. The former activity defines the code that implements the concern. The latter activity determines the locations in the source code where this concern code is located, and makes these explicit.

4.3.1. Pointcut Determination

We should take considerable care when determining pointcuts for a particular concern. If these pointcuts are not defined with caution, the nominal benefits of AOSD technology may not become apparent. Pointcuts therefore have to be determined such that:

- code duplication present in the ordinary source code is reduced to a minimum in the aspect code.
- understandability of the (base and aspect) code is improved, or in other words, the intention behind the pointcuts should be as clear as possible.
- they are as robust as possible with respect to evolution, of the base code as well as of the aspects themselves.

In order to achieve this, pointcut determination is characterised by two different activities. First, we identify the locations in the source code where the concern is implemented. Second, once these locations are known, we analyse them and try to uncover the underlying commonalities between them. The goal of this analysis is to optimise the pointcuts with respect to the three issues mentioned above.

As an example, consider the parameter checking concern. Since input and output parameters require the same check, we need to define only two different aspects: one for input and output parameters, and one for output pointer parameters respectively. Parameter checking always occurs at the beginning of a function, so we will need *before* advice, and should in principle be implemented by every function. A naive way for implementing an output pointer parameter checking aspect, would be to define a pointcut for each function with one or more output pointer parameters, as follows [2]:

```
aspect outputPointerParameterCheckingAspect {
   ...
   pointcut pc324(void ** x) :
      args(x) &&
      execution(* CC_queue_data(..));
   pointcut pc316(void ** x) :
       args(x) &&
       execution(* CC_ReadMsg(..));
   pointcut pc304(void ** x) :
      args(x) &&
      execution(* queue_extract(..));
   pointcut pc383(void ** x) :
      args(x) &&
      execution(* CC_iterator_peek(..));
   ...
   before(void **x) : pc324(x) {
     if(*x != (void *) NULL) {
        r = CC_PARAMETER_ERR;
       CC_LOG(r,0,("%s: Output parameter may already "
            "contain data (!NULL). This data will "
            "be overwritten, which may lead to memory "
            "leaks.", func_name));
     }}
   before(void **x) : pc316(x) {
     if(*x != (void *) NULL) {
        r = CC_PARAMETER_ERR;
       CC_LOG(r,0,("%s: Output parameter may already "
            "contain data (!NULL). This data will "
            "be overwritten, which may lead to memory "
            "leaks.", func_name));
     }}
   ...
}
```

These pointcuts each pick out the execution of a particular function, by means of the `execution` pointcut, and expose an output pointer parameter of that function, by using the `args` join point. Note that wildcards are used, so that the actual number of arguments of a function does not matter, and neither does its return type. This improves the robustness of the aspect w.r.t. evolution. For the CC component, this naive approach results in 384 pointcuts.

Clearly, such pointcuts will not improve the code size of the application, nor its understandability or evolvability.

[2] We use AspectJ and AspectC [4] syntax to define our aspects. Also, we do not show all generated pointcuts and advice code, due to space restrictions.

First of all, for each pointcut, a corresponding advice is defined, that will often contain exactly the same code (as is shown), since the checks are the same for the same type of parameters. Second, since the advices will contain a lot of duplication, the evolvability of the aspect is reduced. Last, the understandability of the aspect can still be improved upon, because it is not explicitly apparent now that all parameters of the same type should perform the same check.

By carefully analysing the locations where the concern should be implemented, and by using knowledge about the particular purpose of the parameter checking concern and the way it can be implemented, we can come up with a more advanced pointcut:

```
aspect outputPointerParameterCheckingAspect {
   ...
   pointcut pc50(void ** x) :
      args(void **) &&
      (execution(* CC_queue_data(..)) ||
      execution(* CC_ReadMsg(..)) ||
      execution(* queue_extract(..)) ||
      execution(* CC_iterator_peek(..)));
   before(void **x) : pc50(x) {
     if(*x != (void *) NULL) {
        r = CC_PARAMETER_ERR;
       CC_LOG(r,0,("%s: Output parameter may already "
            "contain data (!NULL). This data will "
            "be overwritten, which may lead to memory "
            "leaks.", func_name));
     }}
   ...
}
```

This pointcut captures the exact same parameters and the exact same function executions, but is a higher-quality pointcut. It exploits the fact that the four functions each share an output pointer parameter of the same type (`void **` in this example), and that no other functions define such a parameter. Additionally, the aspect extractor analyses the call graph of the functions in order to determine the best location for the parameter checks. In this way, no redundant checks are inserted. The pointcuts also allow us to reuse the advice code, since that code is the same for the same type of parameters. For different types of parameters, however, the advice code can still not be reused, since it differs slightly in the type cast that is being used. Nonetheless, only 39 pointcuts are generated by this approach.

4.3.2. Advice Code Construction

In order to achieve its goals, the aspect extractor collaborates closely with the concern verifier. As explained previously, the concern verifier already checks the conformance of the concern code with respect to some coding conventions, and allows a developer to resolve these. Once this has been dealt with, the idioms and conventions that are followed in the source code are explicitly known. Consequently, this information can be hard-coded in the aspect extractor, as a set of possible advice templates. The extractor then needs to construct the pointcut definitions that match the uses of each template in the code.

The advantage of this approach is that we don't need to extract and reason about the concern code and that the extracted code does not need to be transformed, in order to remove local dependencies. In the generated aspects, the advice code will already be free of references to local variables, for example. The downside of the approach however is that the aspect construction becomes concern-specific. In other words, for every concern, a different extractor needs to be implemented. On the positive side, the extractor only needs to be tuned once for every concern, and can be reused later on. E.g. in our case, we tuned it for the CC component, and we expect to be able to reuse it for all other ASML code, with slight modifications at worst.

5. Case Study Experience

In this section we discuss the results we obtained by running the concern verifier and aspect extractor on the source code of the CC component. Both the verifier and the extractor have been developed as *plugins* in the *CodeSurfer* source code analysis and navigation tool [3]. This tool provides us with programmable access to advanced data structures, such as system- and program-dependence graphs, and defines advanced analysis techniques over these structures, such as control- and data-flow analysis and program slicing.

5.1. Concern Verification

The concern verifier consists of algorithms that verify the error handling, parameter tracing and parameter checking concerns. The algorithms were implemented according to our interpretation of the requirements for the concerns, and were gradually refined until they characterised the concerns as good as possible. In order to achieve such refinements, the results of the algorithms were presented to a domain expert, who pointed out those results that could be improved.

For example, one of the earlier versions of the parameter checking algorithm expected a check at every level in the call chain, and reported 44 deviations from this rule for input parameters. After consultation of the expert, it was pointed out that the checks are often only placed in functions which directly access the parameters, and not in functions that merely pass the parameters on to other functions. In a refined version of the algorithm, which takes into account the call graph of a function, only 25 deviations to the rules remain. These deviations cannot be captured by refined concern rules, but require local (semantic) information about the location of the rules. Some of these deviations are incorrect implementations of the concern (unwanted deviations), but a large group are explicitly created deviations due to the requirements of a function. An example of such a deviation is when a function explicitly allows a parameter to be null on entry and adapts

[3] www.grammatech.com

Concern	Reported Deviations	Unwanted Deviations	% of total
Error handling	18	6	3
Parameter tracing	64	6	1
Parameter checking	75	28	7

Table 2. Results of the concern verification phase

Parameter	Now	Naive	Advanced	% reduced
in || out	1170	2835	536	54
pointer	272	344	58	79
Total	1441		594	59

Table 3. Lines of Code for Parameter Checking concern

its behaviour accordingly. This is why we conclude that at some point it is no longer sensible to refine the rules for a concern.

Table 2 contains the results obtained from running the resulting refined algorithms on the source code of the CC component. The first column of this table represents the concern that is checked, the second contains the number of deviations that were reported, the third contains the number of these deviations that are considered as unwanted, and the last column contains the percentage of the total source code that contains unwanted deviations. This last column may need some clarification:

- for the error handling concern, it states that 3% *of all functions* in the CC component contains one or more unwanted deviations
- for the parameter tracing concern, it states that 1% *of all function parameters* are not traced correctly.
- for the parameter checking concern, it states that as much as 7% *of all function parameters* are not checked correctly. If we consider the different kinds of function parameters separately, we see that only 2% of the input parameters are not checked, whereas 11% of the output parameters and 16% of the output pointer parameters are checked incorrectly.

The results in this table show the advantage of applying AOSD techniques. Although the CC component is quite a small component, it already contains some unwanted deviations. Moreover, we believe these deviations are only due to developer mistakes, because the concerns themselves are quite simple to implement. In turn, such mistakes are due to the fact that, despite their simplicity, implementing the concerns requires effort, concentration and discipline.

Another observation is that a large number of the reported deviations are intended deviations. Approximately 30% of the reported error handling and parameter checking deviations are unwanted deviations, whereas for tracing this number even drops as low as 10%. Unfortunately, there is no way our algorithms can be refined further in order to reduce these false positives. They are simply ”exceptions to the rule” to which the code should adhere. Luckily, the number of reported deviations is quite low, considering the size of the CC component, and they can thus be consulted manually. Moreover, from a developer's point of view, it is important to know these exceptions explicitly and document them appropriately, so it is worthwhile to study them. Such exceptions are typically not explicitly written down in the traditional documentation, as they are very specific to the implementation of a particular function.

5.2. Aspect Construction

As explained in Section 4, aspect construction consists of pointcut determination and advice code construction. Our aspect extractor currently only extracts aspects for the parameter checking and tracing concerns. These are non-tangled, homogeneous concerns, and are therefore easier to extract than tangled concerns, such as error handling. Extraction of tangled and heterogeneous concerns should be considered future work.

In order to evaluate the difference between a naive way of extracting pointcuts and advice code and a more advanced way, we have implemented two construction algorithms for each concern, as explained in Section 4.3.1. The following subsections provide a detailed description of the obtained results.

5.2.1. Parameter checking concern

The aspect extractor for the parameter checking concern will define two different aspects, one that checks input and output parameters, and one that checks output pointer parameters. The resulting aspects were already discussed in Section 4.3.

Table 3 contains a comparison, in terms of lines of code, between the source code as it is now, as it would be with the naive aspects and as it would be with the advanced aspects. As can be seen, an improvement of 59% in terms of lines of concern code is achieved when implementing the concern as an aspect instead of using a simplistic coding convention. The specific reason is that, although there are a lot of different functions, there are only a few different parameter types that need to be checked in each aspect. Because the aspect specifically targets this property, a lot of the code that was previously duplicated can now be factored cleanly. Moreover, the results also show that extracting the aspect in a naive way results in much more lines of code than simply using a coding convention.

Parameter	Now	Naive	Advanced	% reduced
in	1115	1246	548	51
out	424	807	262	38
Total	1539		810	47

Table 4. Lines of Code for Parameter Tracing concern

These results indicate that the total amount of lines of code of the CC component can be reduced by 4%.

5.2.2. Parameter tracing concern

The parameter tracing concern does not differentiate between normal output parameters and output pointer parameters. Therefore, the extractor will generate two different aspects: one responsible for generating the tracing code for input parameters, and one for generating the tracing code for output parameters. Furthermore, the parameter tracing concern differs from the parameter checking concern in that the latter implements a check for each parameter, whereas the former implements one single tracing call for all input parameters of a function and one call for all output parameters. In order to take this into account, the naive pointcut determination algorithm will generate a pointcut for each function that picks out all its input or output parameters. Such pointcuts look as follows:

```
aspect outputParameterTracingAspect {
   ...
   pointcut pc178(CC_queue * x1, void ** x0) :
      args(x1,x0) &&
      execution(* CC_extract_front(..));
   pointcut pc126(CC_queue * x1, void ** x0) :
      args(x1,x0) &&
      execution(* CC_extract_back(..));
   pointcut pc123(CC_queue * x1, void ** x0) :
      args(x1,x0) &&
      execution(* CC_peek_front(..));
   pointcut pc149(CC_queue * x1, void ** x0) :
      args(x1,x0) &&
      execution(* CC_peek_back(..));
   pointcut pc67(complete_type_desc * x0) :
      args(x0) &&
      execution(*  initialise_reply_id_desc(..));
   ...
}
```

The pointcut `pc178` for example, is generated for the `CC_extract_front` function that has two output parameters of type `CC_queue *` and `void **`, respectively.

The advanced algorithm will first reason about the different parameters, however. It realises that pointcuts can only be shared by functions that contain exactly the same set of input or output parameters. It thus generates pointcuts like the following:

```
aspect outputParameterTracingAspect {
   ...
   pointcut pc66(CC_queue * x1, void ** x0) :
      args(x1,x0) &&
      (execution(* CC_extract_front(..)) ||
      execution(* CC_extract_back(..)) ||
      execution(* CC_peek_front(..)) ||
      execution(* CC_peek_back(..)));
   pointcut pc15(complete_type_desc * x0) :
      args(x0) &&
      execution(*  initialise_reply_id_desc(..));

   ...
}
```

which picks out exactly the same join points as the pointcut generated by the naive algorithm, but in a more concise way. These four functions all define two output parameters, and no other functions define exactly these two parameters.

Just like with the parameter checking concern, the difference between the naive and the advanced algorithm algorithm in terms of generated pointcuts is significant: 191 for the naive version versus 61 for the advanced one.

The difference in terms of lines of code between the ordinary source code, the naive aspects and the advanced aspects are summarised in Table 4. The conclusions that can be drawn from these results are similar to the ones drawn for the parameter checking concern: an improvement in terms of lines of code is seen when the concern is implemented in a clever way using AOSD technology, and a naive way of defining the aspects may lead to an even larger number of lines of code then already present. In this particular case, the improvement amounts to 47%, meaning that the amount of lines of code in the CC component can again be reduced by 4%.

6. Discussion and Future Work

As the results in the previous section clearly show, the code devoted to the crosscutting concerns can be reduced significantly by using AOSD technology. Moreover, by using aspects, the concerns become an explicit part of the source code, and can be manipulated as such. As pointed out previously, this may improve important quality attributes such as understandability. maintainability and evolvability.

Although our aspect extractor tries to optimise the reuse of advice code, it can still be observed that a lot of the code looks similar. Especially for parameter checking, only the type cast in the condition of the `if` statement differs between the different advices, but still many different advice code fragments are generated. Unfortunately, AspectJ nor AspectC offers the required meta-programming capabilities to remove such duplication. We plan to study other approaches to aspect definition, such as domain-specific languages, in order to evaluate whether these offer more flexibility and better suit our specific needs.

The lines of code metric we used in order to assess the benefit of applying AOSD techniques may be relatively weak. After all, different developers may have different

coding styles and naming conventions, which influences the number of lines of code. We do believe our results are valid, however. In extracting the aspects, we made sure the advice code was formatted exactly as it was in the original source code. The gain in the amount of lines of source code is thus only due to the fact that the advanced pointcuts allow us to reuse advice code, whereas previously this was not possible.

We have applied the concern verifier and aspect extractor on the CC component only, which is a small, but representative, component in the total ASML code base. In order to confirm the obtained results, we should however apply these tools to other components as well. Besides ensuring statistically more stable results, this may also lead to an even more refined concern verifier. The current verification algorithms are based on the implementation of the concerns by only one developer. Other developers may have slightly different programming styles and may follow idioms in another way.

The results of the aspect extractor presented above are only valid for non-tangled homogeneous concerns, and we can thus say little about what they would look like for tangled concerns, such as error handling, or heterogeneous concerns. However, we have no reason to believe that the amount of code duplication for tangled concerns would be lower than for non-tangled concerns. Furthermore, because tangled concerns interfere with the "ordinary" source code of the functions, separating them inside an aspect will certainly improve important quality attributes. Regarding the extraction of heterogeneous concerns, more study is required before we can draw any conclusions.

Our experiment is only a first, but important, step in a larger effort dealing with the migration of legacy applications to aspect-oriented applications. As such, we have not yet considered important issues such as testing whether the resulting aspect-oriented application preserves the behaviour of the original application, whether automatically transforming millions of lines of C code is feasible, whether adoption of automatic code generators is accepted by developers, and so on. Our long term goal is to investigate all of these issues.

7. Related Work

Our concern verifier resembles tools that verify the quality of the source code. A number of tools for this purpose have been developed over the years, [9, 1, 15]. Most of them are only able to detect basic coding errors, such as using = instead of ==, and are incapable of enforcing company-specific coding styles. More advanced tools exist [10, 18, 13, 17], but these are restricted to detecting higher-level design flaws in object-oriented code.

The work described in this paper has some similarities with work done by Coady et. al. [4]. They consider a single concern (prefetching) within a large C program, the FreeBSD OS kernel. The code implementing the concern is shown to be scattered across the layers that make up the architecture of FreeBSD. Furthermore, the prefetching code is heterogeneous and tangled with other code, which makes it both hard to identify and change. Coady et. al. describe how they (manually) identified the prefetching code, and propose a new (manually obtained) solution in terms of AspectC.

A number of other studies have investigated the applicability of aspect-oriented techniques to various (domain specific) crosscutting concerns. [14] shows the impact of refactoring several small crosscutting concerns on the code structure of two Java applications. [12] targets exception detection and handling code in a large Java framework, and shows how AspectJ can be used to improve its implementation.

An approach to refactoring which specifically deals with tangling is presented by Ettinger and Verbaere [6]. Their work shows how slicing techniques can help automate refactorings of tangled (Java) code. [8] provides a more general discussion of both refactoring in the presence of aspects, and refactoring of object-oriented systems toward aspect-oriented systems.

8. Conclusion

In this paper, we have proposed an approach for extracting aspects from existing applications. This approach consists of two phases: concern verification and aspect construction. We have applied these two phases to an industrial case study, implementing a number of both tangled and non-tangled, homogeneous concerns. Our results showed that automated support for verifying and extracting such concerns is both useful and feasible. Furthermore, the benefits from using AOSD technology, compared to simplistic naming and coding conventions, can be high. We observed that the reduction in amount of lines of code was quite significant: 50% of the parameter checking and tracing concern code can be removed, which leads to an overall reduction of 8% of lines of code in the considered component.

Acknowledgements

We would like to thank Remco van Engelen from ASML, for discussing the results of the case study and for proofreading drafts of this paper, and all members of the Ideals project team, for input about the topic. This work has been carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program. Arie van Deursen is also supported by ITEA (Delft University of Technology, project MOOSE, ITEA 01002).

References

[1] Alfred V. Aho, Brian W. Kernigan, and Peter J. Weinberger. Awk - A Pattern Scanning and Processing Language, 1980.

[2] Silvia Breu and Jens Krinke. Aspect Mining Using Dynamic Analysis. In *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*, volume 23, pages 21–22, Bad Honnef, Germany, May 2003.

[3] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2004.

[4] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98. ACM Press, 2001.

[5] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[6] Ran Ettinger and Mathieu Verbaere. Untangling: A Slice Extraction Refactoring. In *Proceedings of the Aspect-Oriented Software Development Conference (AOSD)*, pages 93–101. ACM Press, 2004.

[7] William G. Griswold, Yoshikiyo Kato, and Jimmy J. Yuan. AspectBrowser: Tool Support for Managing Dispersed Aspects. Technical Report CS1999-0640, University Of California, San Diego, 3, 2000.

[8] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of Aspect-Oriented Software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35. Springer Verlag, 2003.

[9] Stephen Johnson. Lint, a C Program Checker, 1978.

[10] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated Support for Program Refactoring using Invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 736–743. IEEE Computer Society, 2001.

[11] Gregor Kiczales, John Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[12] Martin Lippert and Christina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 418–427. IEEE Computer Society Press, 2000.

[13] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[14] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE Computer Society Press, 2001.

[15] Santanu Paul and Atul Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[16] David Sheperd, Emily Gibson, and Lori Pollock. Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, 2004.

[17] Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 91 – 100. IEEE Computer Society, 2003.

[18] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2002.

[19] Robert J. Walker, Elisa L.A. Baniassad, and Gail C. Murpy. An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.