UNIVERSITY OF SKÖVDE

School of Humanities and Informatics
Final Year Project in Computer Science 20 p
D-level
Spring 2006

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Daniel Oskarsson

HS-IKI-MD-06-003

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Submitted by Daniel Oskarsson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

2006-06-15

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature: ____

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Daniel Oskarsson

Abstract

Software systems are becoming increasingly larger and more complex. Object-orientation help software developers to manage the complexity by enabling software design with a direct mapping to real-world entities. But modern code may still be hard to review and maintain due to scattered and tangled boilerplate code, i.e. code snippets that cut through the dominant decomposition, increasing the complexity.

In this thesis a case study is carried out reviewing the design and code of a real-world software system, ordered by and delivered to the Swedish defense. The system is refactored so that boilerplate code is identified and modularized using aspect-orientation. The difference between the original system and the refactored system is assessed qualitatively, through interviews, and quantitatively, using a metric suite for aspect-oriented systems.

Keywords*:* Aspect-orientation, Assessment, Complexity, Refactoring

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Daniel Oskarsson

Acknowledgements

I would like to thank all those who have contributed to this thesis in any way. In particular, I would like to thank my supervisors Henrik Grimm and Gunnar Karlsson who has helped me during these weeks, always supplying valuable feedback and sharing knowledge. I would also like to thank my examiner Mikael Johannesson for providing more feedback than we students normally get, Birgitta Lindström for providing valuable feedback as an opponent, and the developers at the company who was kind enough to provide time to be interviewed.

A special thank goes to Alessandro Garcia for helping me find a metric suite for AO, Eduardo Magno for support on AJATO, and Chris Burnely for validating my AspectJ implementation. Last, but definitely not least, I would like to thank all people in the aosd.net mailing lists, the AspectJ users mailing list, as well as all other people contributing to the aspect-oriented community for taking the art of software engineering one step further.

This work was supported by Ericsson Microwave Systems, Skövde.

Table of contents

1 Introduction

Most programmers and software architects have probably experienced how adding additional requirements “clutters” the design and code and increases software complexity. Examples of common additional requirements are logging, database connections, communication, and error handling. These additional requirements, or concerns, are never the main part of a system but are often implicitly requested, e.g. when a database connection has to be established before data can be fetched. Just to close the connection as soon as the data has been fetched. Adding those implicit requirements *always* increases the complexity. The implementation of implicit requirements of that kind is often scattered throughout the system, and may also be tangled with the “main code”. In such times one might wish that there were more than one way of decomposing a design and structure source code. With multi-dimensional separations of concerns, concerns that crosscut the dominant decomposition can be separated from the main code so that the complexity can be kept at a manageable level. When Tarr et al. (1999) discusses separation of concerns (SoC) they describe it as “a new paradigm for modeling and implementing software artifacts” (Tarr et al., 1999, p. 107).

A technique that supports multidimensional separation of concerns is aspect-oriented software development (AOSD). With AOSD it is possible to separate crosscutting concerns from the main code and put those concerns in modules of their own. According to Colyer et al. (2006) aspect-orientation (AO) does reduce complexity, but despite that AO has still not begun to be used in the software industry (Beuche & Beust, 2006). This thesis present a case study where a large real-world system, ordered by and delivered to the Swedish defense, is refactored using AO. The aim is to both qualitatively and quantitatively assess how the introduction of AO affects software complexity.

This remainder of this thesis is structured as follows. In Section 2, enough background need to comprehend the thesis is given. Section 3 is the problem chapter. In the problem chapter the problem area is introduced and the aim and objectives for this thesis are presented together with arguments for why the problem is worth investigating. Section 4 consists of the methodology used to address each of the four objectives presented in Section 3. Section 5 includes a brief introduction to the Java server which is subject to the case study carried out in this thesis. The realization of the methods is discussed in Section 6, i.e. how to apply the methods to the objectives. Section 7 presents the results from applying the methods on the objectives, the results are further analyzed in Section 8, where the results are also discussed. Last in this thesis is Section 9 which is the conclusion of this thesis. Section 9 also position this work with other related work, and provide a few pointers for future work.

2 Background

In this section aspect-oriented software development (AOSD) is introduced. The section consists of an overview of aspect-orientation, a discussion of aspect-oriented concepts and acronyms, and an example. The section also contains a short description of different programming paradigms, some criticism against aspect-orientation and a few definitions of concepts used within this thesis.

The reader does not need to know anything about aspect-orientation as it is being presented and exemplified. However, the reader is assumed to be familiar with object-orientation (OO) (Rentsch, 1982; Booch, 1993), and object-oriented concepts. Readers with knowledge about aspect-orientation may skip this Section.

2.1 Dealing with crosscutting concerns

Currently, the dominating paradigm when building software systems is object-orientation (Elrad et al., 2001). Although object-orientation is regarded as a good idea, and makes software design easier, according to Elrad et al. (2001) object-orientation has certain limitations. The biggest limitation is perhaps that there is no possibility to decompose a system in multiple hierarchies due to the problems with multiple inheritances (Elrad et al., 2001). The class hierarchy in an object-oriented system is usually referred to as the dominant decomposition (Tarr et al., 1999; Baniassad et al., 2006).

One technique to increase the expressiveness of the object-oriented paradigm is aspect-orientation (AO) (Elrad et al., 2001). AO enables the implementation of a crosscutting concern as a separate component.

2.1.1 Crosscutting concerns

A crosscutting concern is a concern that can be found throughout the implementation (and possibly also the design) but can’t be modularized as a part of the dominant decomposition. The typical example of a crosscutting concern is logging, where the same set of instructions can be found throughout the system, other examples of crosscutting concerns are debugging and tracing (Murphy et al., 2001), synchronization, error handling, and persistence (Kiczales & Mezini, 2005).

2.1.2 Boilerplate code

The source code for a crosscutting concern is often referred to as scattered or boilerplate code (Kiczales et al., 1997). The same set of instructions or slightly modified variants are scattered throughout the system. It cannot be modularized using object-orientation since the code cuts orthogonally through the dominant decomposition. An example of such a situation is when a number of operations fetch different things from a database. In the beginning of the operations a connection to the database must be created, just to close the connection at the end of operation when the data has been fetched.

2.1.3 Aspect-orientation

With aspect-orientation and the introduction of aspects it becomes possible to catch crosscutting concerns and modularize boilerplate code. Elrad et al. (2001, p. 30) defines aspects as “mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern” and according to Kiczales et al. (1997, p. 240) "aspects are properties for which the implementation cannot be cleanly encapsulated in a generalized procedure", i.e. properties which implementation cannot be modularized using object-orientation. Further Kiczales et al. (2001, p. 328) argues that aspect-orientation "does for crosscutting concerns what OOP has done for object encapsulation and inheritance".

2.1.4 Implicit invocation

AO are popular among researchers because of the ability to write and weave aspects in to a system, without breaking the dominant decomposition (Elrad et al., 2001). Using aspect-orientation it's possible to get a *less complex* implementation since aspects are based on implicit invocation, "a virtue in this age of increased software complexity" (Elrad et al., 2001, p. 30). Implicit invocations are based on the fact that while there is a correspondence from an aspect to a set of join points there is no explicit invocation from within the implementation of the dominant decomposition to the modularized crosscutting concern.

2.2 Aspect-oriented concepts

In this section the four most important aspect-oriented concepts: aspect, pointcut, join point, and advice are introduced. The concepts are not presented in detail; instead the interested reader is referred to Kiczales et al. (1997; 2001).

Listing 2.1 is an example of an aspect written in AspectJ, including an around advice, a pointcut, and the concept of a join point. The four concepts in Listing 2.1 are clarified using comments.

```
import java.util.Hashtable;

public aspect Memoization {  // <-- An aspect (similar to a class).
        private Hashtable valueCache = new Hashtable();

        pointcut calculate(int i) :  // <-- A pointcut (matches join points).
                args(i) &&
        (execution(int Application.calculateSquare(int)));  // <-- A join point.

        int around(int i) : calculate(i) {  // <-- An advice.
                if(valueCache.containsKey(new Integer(i))) {
                        return ((Integer) valueCache.get(new Integer(i))).intValue();

                } else {
                        int square = proceed(i);
                        valueCache.put(new Integer(i), new Integer(square));
                        return square;
                }
        }
}
```
Listing 2.1: An example featuring the most important aspect-oriented concepts.

In an aspect-oriented system there is also something called a weaver. The purpose of the weaver is to weave aspects together with the rest of the system. Weaving can be done either at compilation time, loading time or even at runtime (Kiczales et al., 1997; Popovici et al., 2002).

2.2.1 Aspect

An aspect is similar to a class and shares many properties with a class, e.g. variables and operations can be defined within an aspect; instead of defining the aspect with the keyword *class* the keyword *aspect* is used. Aspects are not a part of a systems functional decomposition (Kiczales et al., 2001); instead they are modularized solutions to crosscutting concerns, appointed by join points, pointcuts and advices.

2.2.2 Join point

A join point is a certain point in the execution where something happens. It can be reading the value of a variable, setting the value of a variable, invoking an operation or receiving an exception (Kiczales et al., 2001). A join point has no code correspondence. However, with proper tool support a join point can be made visible to the developer. To match a join point one or more designators are used (Colyer et al., 2004). The designators in Listing 2.1 are *execute* which matches the execution of the operation *int calculateSquare(int)* in the class *Application*, and *args(i)* which matches join points where the argument are instances of the type of *i*. The variable *i* must be bound in the pointcut definition. The number of designators depends on which language the aspect is written in.

2.2.3 Pointcut

A pointcut allows programmers to refer to a set of events which crosscut the system (Ségura-Devillechaise et al., 2006). A pointcut is a set of designators which matches a set of join points. Pointcuts are used to tie an advice to a certain state in the execution In Listing 2.1 the pointcut matches the execution of the operation *calculateSquare(int)* where *i* contain the value of the argument sent to that operation.

2.2.4 Advice

An advice is similar to an operation although they are never explicitly invoked and there are normally only three types of advices: before, after and around. Advices are constructs used to define a behavior at join points (Kiczales et al., 2001) i.e. the advice is tied to a set of join point either by referring to those join points or by using a pointcut. The different types of advices refer to when the advice is executed. Statements written in *before* advice are executed before the join point, statements written in *after* advice are executed after the join point, and statements written in *around* advice are executed *instead* of the join point, that is the join point is normally *not* allowed to proceed, however using the keyword *proceed* the join point can be executed inside the advice as if it was the original invocation and its return value can be assigned to a variable inside the advice. In Listing 2.1 *around* advice is executed instead of the original call. However, if certain conditions are met the original call is also executed. The advice will be further explained in the next section where the full example is presented.

2.3 Aspect-oriented acronyms

Within the aspect-oriented community there are a number of acronyms. This sub-section contains a short presentation of these acronyms and the meaning of these in this thesis.

Two of the most used acronyms are AOP (aspect-oriented programming) and AOSD (aspect-oriented software development). AOP are traditionally used when talking about aspect-orientation in any phase of the System Development Life Cycle (SDLC) (The department of Justice, 2006) or a similar software development method. In this thesis this is referred to as aspect-orientation (AO) and AOP is used only when referring to AO within the implementation phase. AOSD is usually used when referring to AO within all phases of a software development method. This use is adopted in this thesis. However, traditionally AOSD has also been used as an abbreviation for aspect-oriented software design, this is unfortunate, and it is my opinion that a more suiting concept when explicitly talking about AO within the design phase is aspect-oriented design (AOD).

2.4 Memoization

The example in this section is based on the memoization technique (McNamee & Hall, 1998) where results of expensive calculations are stored for reuse. The example is adapted from Chapman and Hawkins (2004) and is implemented in AspectJ[1] (Kiczales et al., 2001). It consists of a single class, provided as Listing 2.2, which includes two operations: one that calculates the square of any integer provided and another which prints the result to the console. The example also consists of two aspects (Listing 2.4 and Listing 2.6).

```
public class Application {
	public static void main(String[] args) {
		int number[] = {45,64,12,45,64,102,33};
		for (int i = 0; i < number.length; i++) {
			square(number[i]);
		}
		System.out.println("Application terminated!");
	}

	public static void square(int i) {
		System.out.print("The square of " + i + " is ");
		System.out.print(calculateSquare(i));
	}

	public static int calculateSquare(int number) {
		try {
			Thread.sleep(5000);
		} catch (Exception e) {
			System.out.println("Ex occured in calculateSquare:" + e);
		}
		return number * number;
	}
}
```

Listing 2.2: The Application class.

The *calculateSquare* operation is considered as time and memory consuming since it takes five (simulated) seconds to calculate the square. Listing 2.3 shows a typical output from running the example.

```
The square of 45 is 2025 (calculation took 5007 ms)
The square of 64 is 4096 (calculation took 5007 ms)
The square of 12 is 144 (calculation took 4997 ms)
The square of 45 is 2025 (calculation took 5008 ms)
The square of 64 is 4096 (calculation took 5017 ms)
The square of 102 is 10404 (calculation took 5007 ms)
The square of 33 is 1089 (calculation took 5007 ms)
Application terminated!
```

Listing 2.3: Output from running the example.

The attentive reader notices that there are things in the output in Listing 2.3 which is not included in the source code in Listing 2.2. How can that be? The answer is aspect-orientation. With aspect-orientation the implementation of an aspect can be woven into the main code at certain join points. How an aspect is written differs between different languages/frameworks. Listing 2.4 shows the Timer aspect written in the AspectJ language. The advice around is executed instead of the square operation, as defined in the pointcut. However, since the keyword *proceed* is used the square operation is also executed but proceeded by the start of a timer and followed by a print line clause which prints the elapsed time.

[1] http://www.eclipse.org/aspectj/doc/released/progguide/starting.html

```
public aspect Timer {
	pointcut timer(int i) : args(i)
		&& (execution(void Application.square(int)));

	void around(int i) : timer(i) {
		long start, elapsed;
		start = System.currentTimeMillis();
		proceed(i);
		elapsed = System.currentTimeMillis()-start;
		System.out.println(" (calculation took " + elapsed + " ms)");
	}
}
```

Listing 2.4: The Timer aspect.

If another aspect is included in the build and the example is run again a typical listing is Listing 2.5. Notice that, according to the *Timer* aspect, two of the values have a calculation time of 0 ms.

```
The square of 45 is 2025 (calculation took 5008 ms)
The square of 64 is 4096 (calculation took 5027 ms)
The square of 12 is 144 (calculation took 4997 ms)
The square of 45 is 2025 (calculation took 0 ms)
The square of 64 is 4096 (calculation took 0 ms)
The square of 102 is 10404 (calculation took 5007 ms)
The square of 33 is 1089 (calculation took 5017 ms)
Application terminated!
```

Listing 2.5: Another execution.

```
import java.util.Hashtable;

public aspect Memoization {
	private Hashtable valueCache = new Hashtable();

	pointcut calculate(int i) :
		args(i) && (execution(int Application.calculateSquare(int)));

	int around(int i) : calculate(i) {
		if(valueCache.containsKey(new Integer(i))) {
			return ((Integer) valueCache.get(new Integer(i))).intValue();

		} else {
			int square = proceed(i);
			valueCache.put(new Integer(i), new Integer(square));
			return square;
		}
	}
}
```

Listing 2.6: The Memoization aspect.

The reason for this is of course the inclusion of the Memoization aspect listed in Listing 2.6. Before executing the expensive operation of calculating the square a check is done against a Hashtable so see if the value has not been calculated before. If the value exists in the Hashtable it is returned instead of proceeding to execute the *calculateSquare* operation at all.

2.5 Programming paradigms

In this section, aspect-orientation is compared to object-orientation. The section will answer questions such as: why is not AO a replacement for object-orientation and how can object-oriented languages be extended to support AO?

Aspect-orientation is based on object-orientation in the same way that object-orientation is based on modular programming. This means that classes, methods, and other object-oriented concepts are used when practicing aspect-oriented software development (AOSD). Table 2.1 is a compilation of software engineering technologies and their key concepts.

Technology	Key concepts	Constructs
Structured programming	Explicit control constructs	Do while and other loops, blocks and so forth
Modular programming	Information hiding	Modules with well-defined enforced interfaces
Data abstraction	Hide the representation of data	Types
Object-oriented programming	Objects with classification and specialization	Classes, objects, polymorphism
Aspect-oriented programming	Crosscutting concerns, Aspects	Join points, pointcuts and advices

Table 2.1: Key concepts compared, adapted from Elrad et al. (2001).

The table is adapted from Elrad et al. (2001) but is enhanced to include aspect-orientation. It shows how new techniques is built on older techniques, utilizing the strength and developing new constructs to handle the weakness of an older technique. As Table 2.1 shows, crosscutting concerns and aspects are key concepts within aspect-orientation. An aspect is a construct (similar to a class) consisting of join points, pointcuts and advices. Aspects extend object-orientation by enabling support for multidimensional decomposion of a system (Tarr et al., 1999).

By extending a programming language with support for the new constructs, an object-oriented language can be made into an aspect-oriented language. Kiczales et al. (2001) developed the language AspectJ by extending Java to support the new constructs. Similar extensions can be found for a number of languages including (but not limited to) C++ (Spinczyk & Schröder-Preikschat, 2002) and COBOL (Lämmel & De Schutter, 2005).

2.6 Criticisms against aspect-orientation

While aspect-orientation has the advantage of being able to modularize boilerplate code as aspects and thereby increase information hiding (and perhaps decrease software complexity), it also has disadvantages. Some of the criticisms against aspect-orientation are (Beuche & Beust, 2006):

- AOSD seems to suffer from the chicken and egg problem; is not used by any other than early adopters and researchers, probably since it’s not well known, and it is not directly supported by any major software company such as Sun, Microsoft or IBM.
- According to Beuche and Beust (2006, p. 73) “the conceptual shift to aspect-oriented software design and architecture is radical”.
- It can be hard to know when and how to use AO.
- Aspects are hidden from programmers.
- Debugging gets harder because of implicit invocation.

This criticism suggests that *aspect-orientation may never be used by a major part of software developers* since it’s not publicly available, not supported by any major actor, and might be a to radical shift from object-orientation. It also seems like it’s hard to use aspect-orientation because the aspects are hidden from the programmer and debugging gets harder because of the implicit invocation.

2.7 Definitions

This sub-section contains a few definitions for concepts used in this thesis.

2.7.1 Integrated Development Environment

An Integrated Development Environment (IDE) is an application used by developers to implement other applications or systems. An IDE can also support other phases of the software development method, such as the design phase or the test phase. Examples of popular IDE:s for Java development is Eclipse, JBuilder[2], NetBeans[3] and Java Studio Creator[4].

2.7.2 Programming Language

A programming language (PL) is defined by Oxford Reference Online (2006) as "A notation for the precise description of computer programs or algorithms. Programming languages are *artificial* languages, in which the syntax and semantics are strictly defined", i.e. a syntax which is interpreted by and compiled with a compiler.

2.7.3 Aspect-oriented language

It may be difficult to separate the concepts aspect-oriented (programming) languages and aspect-oriented frameworks. In this thesis the definition of an aspect-oriented programming language include the definition for a PL but is extended to support syntax for aspect-oriented concepts such as advices, join points and advices.

2.7.4 Aspect-oriented framework

According to the Oxford Reference Online (2006) a framework is "an essential supporting or underlying structure". Within this thesis an aspect-oriented framework is defined as an essential set of tools supporting the AOSD process. One such tool is a compiler for the aspect-oriented language. Other tools may be tools that aid the developer when developing aspect-oriented systems. An aspect-oriented framework may or may not, but is preferably, integrated with an IDE.

2.7.5 Complexity

This thesis aim to assess the introduction of aspect-orientation in a real-world system regarding software complexity, therefore this paragraph is spent defining complexity.

Already in 1967 Edsger Dijkstra pointed out that paying attention to software partitioning and structure is worth the work stating that:

> *"It seems the designer's responsibility to construct his mechanism in such a way -i.e. so effectively structured- that at each stage if the testing procedure the number of relevant test cases is so small that he can try them all and that is what is being tested is perspicuous that it is clear that he has not overlooked a situation."* (Dijkstra, 1967, p. 455)

[2] http://www.borland.com/jbuilder
[3] Available as part of Sun's JDK or at http://www.netbeans.org
[4] http://developers.sun.com/prodtech/javatools/jscreator

What Dijkstra meant is what is usually referred to as software architecture. Software architecture is normally seen as the internal structure of the components which a system consists of. Today an architecture is often built using different software patterns (Helm, 1995; Gamma et al., 1994), after an idea presented by Christopher Alexander in the late 1970:s (Alexander, 1977; 1979), which has been adapted to the software community by Gamma et al. as design patterns (Gamma et al., 2003).

AlSharif et al. (2004) uses software architecture as a context to define software complexity and describe software complexity as a general idea which can be difficult to grasp. They proceed with defining software complexity as "the complexity of the structures that make up the whole architecture and their relationship with each other" (AlSharif et al., 2004, p. 99), a definition used within this thesis to quantitatively define software complexity. Kearney et al. (1985, p. 340) says that "The term 'software complexity' is often applied to the interaction between a program and a programmer working on some programming task"; this view is used to qualitatively define software complexity. In this thesis software complexity is often referred to as just complexity.

3 Problem description

In this section the problem area is introduced. Further the project aim is presented as well as a set of objectives which must be fulfilled to be able to reach the aim.

3.1 The introduction of aspect-orientation

Aspect-orientation is an emerging technique to implement crosscutting concerns as separate modules. The separate implementation of crosscutting concerns seems to have many advantages such decreasing code complexity and providing the developers with a consistent way of solving recurrent problems. In contemporary complex software projects (Xia & Lee, 2004), one might think that AOSD would be welcomed with open arms, as it has a big potential to decrease software complexity (Colyer et al., 2006). However, despite recent years growing industry awareness and an increasing number of institutions teaching AOP (Colyer et al., 2006), AO has not had its major breakthrough yet. Beuche and Beust embodies this saying that "We have yet to see a major grassroots movement of 'regular' developers" (2006, p. 75). They also state that "The AOP community still has a lot of work to do communicating the benefits of its ideas to the general software development community and beyond to organizational management" (Beuche & Beust, 2006, p. 74).

Kiczales et al. (1997) argues that in *real-world* programs, *complexity* due to tangling quickly becomes a major obstacle to the ease of development and maintenance. Beuche and Beust, which are software developers, discusses the general feeling among developers stating that "the conceptual shift to aspect-oriented software design and architecture is radical" (2006, p. 73), and question if AO is worth the radical change.

3.2 Aim

According to Kiczales et al. (1997, p. 18) "One important goal is quantitative assessment of the utility of AOP. How much does it help in the development of real-world applications?" The aim of the project is thus to assess how the introduction of aspect-orientation affects software complexity and discuss the introduction of AO in a real-world system. There are not many studies where an object-oriented system is refactored using aspect-orientation, so this thesis aim to extend the small amount of work that has been done investigating and assessing the introduction of aspect-orientation. By comparing the software complexity of the original system with the refactored system.

Murphy et al. suggests that "To thoroughly evaluate the usefulness of AOP, multiple software development organizations would need to build their products both with and without AOP and compare the results" (2001, p. 75), which leads us to the project aim:

The aim is to assess the introduction of aspect-orientation in a real-world system regarding complexity.

Although a number of case studies have been performed within the aspect-oriented area, most of them serve as illustration to a concept or proof to a hypothesis. Also, most of the case studies are based on systems built using AOSD. Few case studies have been carried through with the purpose of assessing *the introduction* of AO into an object-oriented system. By comparing a system designed and implemented using

OO and the same system refactored using AO, the differences between the two techniques can be highlighted and discussed regarding complexity.

3.3 Objectives

To reach the aim, a few objectives have been developed. The specific objectives for the study are:

1. Find a suitable way to measure complexity.
2. Investigate contemporary tools and frameworks available for AOSD in Java.
3. Refactor a system, i.e. find crosscutting concerns and modularize those using aspect-orientation.
4. Compare the original and refactored system regarding complexity.

The objectives have been developed and refined so that they support each other and so that the realizations of all four objectives also involve the realization and fulfillment of the project aim.

4 Methodology

This section presents and describes the methods used to address each of the four objectives presented in the previous section. Each of the sub-section describes one of the objectives and their corresponding methods.

As previously stated the project aim is to assess the introduction of aspect-orientation in a real-world system regarding complexity. According to Murphy et al. (2001) there are two techniques for assessing a programming technology.

> *"There are two basic techniques for assessing a programming technology: experiments and case studies. Experiments provide an opportunity for direct comparison, allowing researchers to investigate how the technology performs in detailed ways. Case studies take many forms, from small-scale comparisons to longitudinal studies of the technology in action. Case studies tend to provide broader knowledge about the use of a technology."* (Murphy et al., 2001, p. 76)

The overall method in this thesis is as a case study which includes refactoring of a large system (implementation) followed by qualitative assessment (open interviews) and quantitative assessment (measurement) of the system. The reason that the case study includes both qualitative and quantitative assessment is to present a more complete result of the case study by providing a broader knowledge than if the refactoring was assessed only qualitatively or only quantitatively.

The system subject to this case study is a large system built between 2003 and 2005 by a large Swedish software company. The system is refactored so that crosscutting concerns are located and modularized as aspects, resulting in two systems: one built by the company using pure OO and one system refactored using AO. The original system is introduced in Section 5, and the results are presented in Section 7.

4.1 Find a suitable way to measure complexity

To compare the original and refactored systems, there has to be a way to measure complexity. Traditionally metrics are used to quantitatively measure software. To reach this objective a small and informal literature study is carried out with the goal of finding a way to measure both the object-oriented system and the aspect-oriented system using the same metrics. By applying the metrics both on the original system and on the refactored system it is possible to calculate a quantitative difference between the two systems and draw conclusions from the results.

4.2 Tools and frameworks for AOSD in Java

To refactor the system using aspect-orientation there has to be some kind of aspect-oriented framework which enables AOSD in Java. To reach this objective a small and informal literature study is carried out to assess which AO frameworks are most referenced for Java, and popular (not necessarily non-peer-reviewed) material is utilized to study the different frameworks and assess which one that seems most suited to use when refactoring.

4.3 Find and modularize crosscutting concerns

To assess the difference between the original and a refactored system, there is work done refactoring the object-oriented system in to an aspect-oriented system by modularizing as many crosscutting concerns as possible. To reach this objective the object-oriented system is first reviewed and found crosscutting concerns are noted. Then the crosscutting concerns are implemented as aspects using the chosen framework.

4.4 Compare the systems regarding complexity

By assessing the difference between the original and the refactored system new knowledge is gained about how the introduction of aspect-orientation affects software complexity. Such knowledge can be used for a number of things, e.g. by a software developer to see whether aspect-orientation seems interesting or by a project leader to evaluate a possible introduction of aspect-orientation in industry AOSD projects. To gain as much knowledge as possible the affection on software complexity when introducing aspect-orientation should be assessed both qualitatively through open interviews and quantitatively through measurement.

4.4.1 Interviews

The purpose of the interviews is to gather qualitative information about what developers think of how the introduction of aspect-orientation affect the complexity, using examples that originate from a system that has been developed themselves or a colleague within their company.

4.4.2 Measurement

The quantitative measurement is based on the metrics found reaching the first objective. By applying the metrics both on the original system and on the refactored system it is possible to calculate a quantitative difference between the two systems and compare and discuss the deviations.

5 The Java server

This section describes the Java server which is refactored in this thesis. The Java server subject to this case study is part of a communication system ordered by the Swedish defense. It was developed by a large Swedish software company between autumn 2003 and spring 2005. Since details around the system are considered a company secret, a full system presentation cannot be given. However, in this section enough details are provided so that the role of the Java server can be understood and put in context.

5.1 Architecture

The system shown in Figure 5.1 is part of an even larger system used by the Swedish defense to communicate both voice and data throughout all of Sweden. The Java server, works as an interface to the outer parts of the system as shown in Figure 5.1.

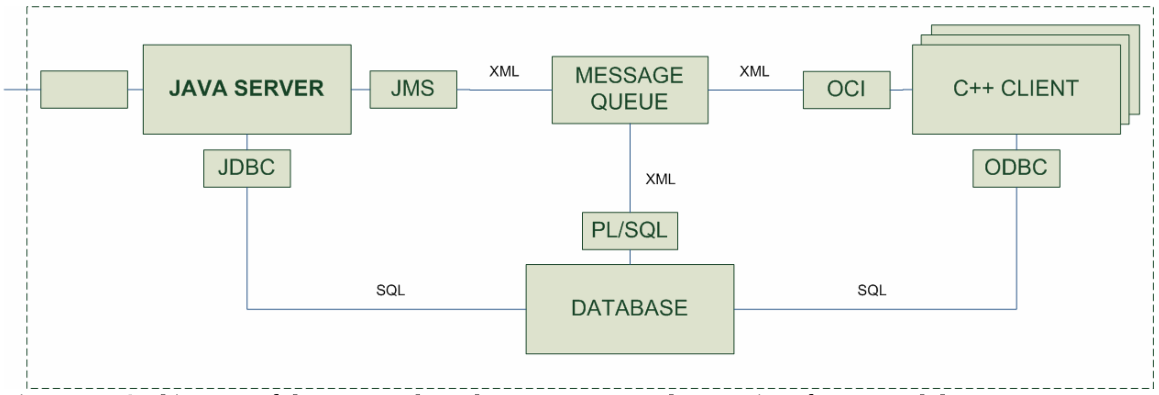


Figure 5.1: Architecture of the system where the Java server works as an interface toward the outer system.

Internally the Java server communicates with a message queue using Java Message Server (JMS) and a database, using Java Database Connectivity (JDBC). Also communicating with the database is the message queue and a number of C++ clients. The clients use Oracle Call Interface (OCI) to communicate with the message queue and Open Database Connectivity (ODBC) to communicate with the database. The information sent to and from the message queue is wrapped in predefined XML-messages. Figure 5.1 illustrates the relationships between these components.

5.2 Details

The Java server consists of 241 regular classes (inner classes included) and 61 test classes divided into 34 packages. The server is implemented using well known techniques as patterns and localization classes. The system must be compiled using Java 2 Platform Standard Edition 5.0 (J2SE 5.0) since the source code contains some J2SE 5.0 constructs, e.g. generics. Most of the time there was only one developer working on the Java server, but for a couple of months there were up to three developers implementing the Java server. The case study is based on the implementation of the Java server built 2005-09-30.

6 Realization

The section describes the realization of the project, i.e. how the methods are carried out in order to reach all the objectives and thus fulfill the aim. The realization of a small and informal literature study is considered self-explanatory so the focus in this section is to describe the realization when refactoring the system, carrying out the interviews and the measurement. Section 6.2 describes the refactoring, Section 6.3 describes the interviews, and Section 6.4 describes the measurement.

6.1 Refactoring the system

After gaining access to the Java server described in the previous section, all source code is thoroughly reviewed and a working document containing the separation of concerns is created. Because of the lack of experience in finding crosscutting concerns some could have be overseen. In an attempt to minimize the bias, the crosscutting concerns are checked against the code and the code is reviewed two times. When the review is completed the document consists of all found crosscutting concerns in the system.

The crosscutting concerns that occurred most times and is possible to refactor is together with a typical implementation put into another working document called the *crosscutting concerns working document*. With the crosscutting concerns working document as a base the system is refactored by implementing the chosen crosscutting concerns as aspects and refactor the original code at all location it exists. The code for the aspects and the refactored code are added to the crosscutting concerns working document, and the document is used as a base for the interviews.

Unfortunately the implementation of the singleton aspect has after the interviews been shown not to work and is therefore fixed before releasing the document (with permission) to the *aspectj-users mailing list*[5]. That way, the community, which has the best knowledge of aspect-oriented software, can help validate the implementation. The crosscutting concerns working document has been refined into and are attached to this thesis as Appendix A.

6.2 Interviewing developers

The implementation is followed by qualitatively assessing the introduction of aspect-orientation into the system through open interviews with the key developer of the Java server as well as with two other developers. Common for the developers is that all three of them work for the software company as a software developer and has done so for more than five years.

6.2.1 Choosing interviewees

The interviewees are all developers and is chosen based on their experience of developing software using object-orientation, and knowledge of Java programming. None of the interviewees has had any previously experience of AO. Interviewee 1 has been employed at the company for more than ten years and has been the key developer of the Java server which is subject to this case study. Interviewee 2 has also been a team member of the project which the Java server was developed in, working mostly with the database and message queue shown in Figure 5.1 and are well

[5] http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg06103.html

familiar with the systems interface. Interviewee 3 has not worked with the project at all and therefore has not as much knowledge of the original code as the other two interviewees. However, Interviewee 3 has as previously stated worked for the company for more than five years and is well familiar with OO. The interviewees are all participating in the interviews voluntarily.

6.2.2 Open interviews

The interview style used is an *open interview* where the control of the interview very much lies in the hand of the interviewee (Berndtsson et al., 2004). The reason for the choice of an open interview is that it is well suited for qualitative research and, if mastered, the topics that are brought up are those that are important to *both* the interviewee and the interviewer (Berndtsson et al., 2004). With proper preparations the open interview seems to be the best alternative, even though the interviewer is not experienced in performing open interviews and has only done so a few times before.

6.2.3 Validity

All the interviews are executed entirely in Swedish to address validity since both the interviewer and interviewees are from Sweden. Before the interview the interviewees receive the introduction and background of this thesis so that they have the chance of studying aspect-orientation basics needed to discuss the refactoring. The thesis is written with the possible bias in mind that the text can influence the interviewees' answers. Because of that, and even though it is supposed to contain only facts, before sending it to the interviewees, the document is reviewed to make sure that it doesn't contain any opinions. The document is sent roughly three days before the interview and the interviewees are encouraged not to discuss AO with anyone else before all interviews is carried out.

6.2.4 Reliability

The interviews rely on some fundamental questions so that the interview agenda is clear and both the interviewer and the interviewee feel that they are a vital part of the interview. These questions are tested in advance so that they are clear, valid and can't be misinterpreted. The transcripts from the interviews are reviewed by each of the interviewee to avoid errors.

6.2.5 Data collection

The interviews are recorded using an electronic recorder. This is agreed in advance and the interviewees are reminded about this just before the interview. The recordings are only handled by the interviewer and are destroyed on the completion of this thesis. Information about this is communicated to the interviewees.

6.2.6 Carrying out the interviews

During the interviews the crosscutting concerns document is used as a basis for the discussion together with a few prepared questions. When discussing the contents of the crosscutting concerns document some of the prepared questions are asked and answered for each example. One such question is:

How do you think that the addition of this aspect affects code complexity?

Important issues are followed up with unprepared questions. All prepared questions and the interview transcripts are attached to the thesis as Appendix B.

6.3 Measurement

To quantitatively assess the introduction of aspect-orientation in the system, a number of metrics is used to retrieve values from each of the two systems and compare the values. The metrics is selected from a number of metrics suggested by different individuals from the aspect-oriented community as a result of asking for good metrics on the *aosd-discuss mailing list*[6]. The metrics selected is based on the fact that they have been used before by several authors in recent work. That way any potential biases when quantitatively assessing the refactoring using the metrics can be identified by reading previous work.

Another reason for the selection of the metrics, although not the biggest, is the tool support that is provided for the metrics. The measuring was supposed to be carried out using two tools developed for assessing aspect-oriented systems; AOP Metrics version 0.3 (Stochmialek, 2006) and AJATO version 0.1 (Magno et al., 2006). But despite a huge effort, none of two the tools, is able to execute properly using the refactored system as input. The reason for this is explained for each of the tool in the following two paragraphs.

6.3.1 AJATO

When trying to assess the Java server using AJATO it seems like the tool is able to measure some files, but for some of the files the tool throws an exception and is therefore not able to calculate any of the metrics for the system. The tool can, however, assess the memoization example in Section 2.4. The problem has been identified as due to Java 5.0 constructs. AJATO cannot assess systems built using Java 5.0 constructs, which the Java server is. This has been verified through personal communication with Eduardo Magno who is the author of AJATO.

6.3.2 AOP Metrics

According AOP Metrics website the tool support Java 5.0 and most metric supported by AJATO is also supported by AOP Metric. However, AOP Metrics does not start in the environment that the project is set up in[7] due to what seems to be a bug; the tool works fine on a personal computer[8] and can assess the memoization example, unfortunately there is no possibility to assess the system outside the project environment at the company. Despite attempts to get in touch with the developers of AOP Metrics using the *aop metrics users' mailing list*[9] the issue is not resolved.

Since both tools are considered open source, there has been a huge effort (more than 20 hours) trying to get these AOP Metrics to work on Windows by manually changing the source code and recompile the tool. Despite this effort the error could not be fixed, so that AOP Metric could run under Windows.

6.3.3 Manual measuring

The results for the metrics are based on a manual measurement. Because of the manual measuring there is a slight possibility that the results are not the exact same result as if a tool would have been used. However, the deviation is not large because the faults that might have been done measuring manually should be considered similar

[6] http://www.aosd.net/pipermail/discuss_aosd.net/2006-March/001875.html
[7] The environment consisted of a PC running Windows 2000 SP4 and Eclipse/AJDT.
[8] The personal computer runs Mac OS X 10.4, a UNIX-like operating system.
[9] http://aopmetrics.tigris.org/servlets/ReadMsg?list=users&msgNo=1

to a graceful degradation within a computer program. Also the faults, if any, would be done similar measuring both the original system and the refactored system. Some metrics are unfortunately also to complex to be measured manually on large systems and could therefore not be included in the assessment. With those two considerations in mind it is apparent that the results can still be used as a way of comparing the two different implementations of the system, which is the reason for measuring the both systems using these metrics. The results from the measurement is presented in Section 7.5, discussed in Section 8.3 and all charts are attached this thesis as Appendix C.

7 Results

In this section the results from investigating the four objectives is presented as Section 7.1-7.5. Section 7.1 describes the metric suite that is used during the measurement of the system and a short presentation of the metrics that is to assess the refactoring. Section 7.2 contains a discussion of aspect-oriented frameworks for Java development. It contains information about which set of frameworks and tools that has been selected for this project. Section 7.3 contains a presentation of the source code from implementing the crosscutting concerns. Section 7.4 contains the results from the open interviews which have been carried out, and Section 7.5 contains the results from the measurements with a comparison between the original and the refactored system.

7.1 Measuring complexity

In this thesis quantitatively assessing the difference between the original and the refactored system is done using a metric suite developed for aspect-oriented systems defined in Sant`Anna et al. (2003). The metrics can also be used to assess object-oriented systems, as many of the metrics are refined version of well known software metrics. The metrics suite has previously been used to measure both object-oriented and aspect-oriented systems, e.g. by Filho et al. (2005) and Garcia et al. (2005).

7.1.1 The metric suite

The metric suite in Table 7.1 consists of four different kinds of metrics: size metrics, coupling metrics, cohesion metrics and separation of concern metrics, of which only size and cohesion metrics will be measured for reasons discussed. Table 7.1 contains the attributes and the corresponding metrics that constitute the metrics suite.

Attributes	Metrics	Definitions
Separation of concerns	Concern diffusion over components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern diffusion over operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern diffusion over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling between components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
	Depth inheritance tree (DIT)	Counts how far down in the inheritance hierarchy a class or an aspect is coupled.
Cohesion	Lack of cohesion in operations (LCOO)	Measure the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.
Size	Lines of code (LOC)	Counts the lines of code.
	Number of attributes (NOA)	Counts the number of attributes of each class or aspect.
	Weighted operations per component (WOC)	Counts the number of methods and advices of each class or aspect and the number of its parameters.
	Vocabulary Size (VS)	Counts the number of system components, i.e. the number of classes and aspects in the system.

Table 7.1: The metrics suite. Figure adopted from Filho et al. (2005).

The metric suite is adapted from Sant`Anna et al. (2003) and the metrics introduced in this section are further described in Sant`Anna et al. (2003) and Garcia (2004). However, the figure is adapted from Filho et al. (2005) since it's not included in Sant`Anna et al. (2003).

7.1.2 The metrics

The metrics that are used to assess the refactoring is only shortly presented in this sub-section and interested reader is referred to Sant`Anna et al. (2003) and Garcia (2004). The results of the assessment can be found in Section 7.4.

7.1.2.1 Size metrics

There are four different size metrics (Garcia, 2004): Vocabulary size, Lines of code, number of attributes, and weighted operations per component.

Vocabulary size (VS)

VS is a simple metric which consists of the sum of all components in the system. A component is defined as a class or an aspect. The higher VS the more complex is the system.

Lines of code (LOC)

One of the simplest and most common metric is LOC. Traditionally LOC is *the* metric used to describe the size of a system from one colleague to another. In this thesis, where complexity is discussed and compared at an operation level, LOC is counted not for the whole components but for the operations that has been refactored. (The reason for that is that the refactoring would most definitely be different if a developer with more experience of aspect-orientation would have carried out the refactoring. In other words, there could have been more done.) This is sometimes referred to as ELOC (Effective LOC).

Number of Attributes (NOA)

The NOA metric counts the internal number of attributes of a component (inherited attributes are not counted). The handling of an exception is counted as one attribute, regardless of how many times that exception is being handled within the component.

Weighted Operations per Component (WOC)

This metric measures the complexity of a component in terms of its operations, since the refactoring did not add or remove any operations this metric will not be used in the assessment.

7.1.2.2 Coupling metrics

There are two metrics which measure the coupling in a system: Coupling between Components and Depth of Inheritance Tree.

Coupling between Components (CBC)

CBS is for a component a tally of the number of components to which it is coupled by counting the number of classes that are used in attribute declarations. The higher the component coupling, the more difficult it is to understand the system. In this thesis the tally is measured only for those components affected i.e. components not affected by aspects are not counted.

Depth of Inheritance Tree (DIT)

The DIT metric calculates how far down in the hierarchy the component is located. The further down the more can be inherited and the more complex the component becomes. Since only eight aspects are added, in their own *aspects* package but without any inheritance (except for inheriting object), the DIT metric is not used to assess the system.

7.1.2.3 Separation of Concerns and Cohesion metrics

The separation of concern metrics (SoC): Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over LOC (CDLOC) are new metrics to measure separation of concerns (Filho et al., 2005;

Figueiredo, 2005). However, neither of these or the cohesion metric Concern Diffusion over LOC (LCOO) will be used assessing the introduction of aspect-orientation. The reason for this is simply that the metrics are to complex to be measured manually on such a large system, and the number of components affected by the refactoring the object-oriented code (finding true crosscutting concerns were hard) is to small in comparison to the number of components in the system, the SoC metrics is not expected to show very much of an improvement.

7.2 Aspect-oriented frameworks for Java development

This section is spent discussing different aspect-oriented frameworks and their constituting components. The purpose is to reach the second objective and find an appropriate framework that can be used when refactoring the Java server.

An appropriate aspect-oriented framework must include tools to weave an aspect into the language or an aspect-oriented language itself (that includes a compiler). It should preferably also include other tools such as the possibility to generate documentation that describes the aspects and how they are woven into the rest of the system. Furthermore an aspect-oriented framework should be integrated with an IDE to be able to assist the developer in the implementation of aspects and reviewing of aspect-oriented source code.

According to Mik Kersten in a 1½-2 year old AOP Tools comparison (Kersten, 2005) the four largest *projects* for using aspect-orientation with Java are AspectJ AspectWerkz, JBoss AOP and Spring AOP. According to the literature that has been read in this project there is no reason at all to believe that anything has changed, except that the two biggest projects, AspectJ and AspectWerkz, has merged[10] into a single project developing AspectJ. However, these four were *projects* and not *frameworks*. According to the definitions of a framework and a language AspectJ is in fact a language and JBoss AOP[11] and Spring AOP[12] are both frameworks since they are part of thee JBoss application server framework and the Spring framework, respectively. However, there is a framework which includes AspectJ as well as additional tools and is integrated into the Eclipse IDE. That framework is called AspectJ Development Tools (AJDT)[13] and is widely used. The combination of Eclipse and AJDT will in the rest of this thesis be referred to as Eclipse/AJDT.

There are mainly two differences between different frameworks. The first difference is which technique the framework uses to enforce aspects to the original code. JBoss AOP for example, actually extends the runtime environment while AspectJ uses static weaving. The second difference is concerned with how the syntax for aspect looks like. Most frameworks only support a single syntax; e.g. with JBoss AOP and AspectWerkz the pointcut has to be defined in an xml-file, while in AspectJ pointcuts is written with an extended Java syntax. Actually AspectJ supports two fundamentally different ways to implement an aspect. The reason for this is the merger between AspectJ and AspectWerkz, who had two fundamentally different ways to write aspects.

[10] http://dev.eclipse.org/mhonarc/lists/aspectj-announce/msg00038.html
[11] http://labs.jboss.com/portal/jbossaop/index.html
[12] http://www.springframework.org/docs/reference/aop.html
[13] http://eclipse.org/ajdt

AspectJ is the oldest implementation of aspect-orientation (Kiczales et al., 2001) and originate from the Palo Alto Research Center (PARC)[14]. It is without doubt the most referenced language in literature and the communities and the one chosen for the refactoring of the Java server with AJDT as the surrounding framework and Eclipse as IDE. Another argument for using Eclipse/AJDT is that AJDT is well integrated into Eclipse and thus provides an easy to grasp graphical development environment. For example, when using AspectJ in Eclipse, the IDE shows the join points so that the developer knows which methods that are affected by aspects. It also provides hyperlinks so that the developer can quickly go from a join point to the aspect and the other way around (Chapman, 2005).

7.3 Refactoring

The system described in the previous section constitutes of a large number of classes divided into different packages. An AspectJ project was created in Eclipse/AJDT. The version of Eclipse used was 3.1.2 and for AJDT it was 1.3.0 (build 20051220093604) which includes support for the language AspectJ 1.5.0. To be able to see AspectJ specific warnings during development, a number of warnings had to be suppressed since there were well over eight hundred warnings apparent in the original source code.

The process of refactoring the system has been described in Section 5 so this chapter will instead be based describing the eight crosscutting concerns and the aspects implemented to modularize these, which is discussed during the interviews which results are presented in Section 7.3. It should also be mentioned that the aspects were placed in a separate package in the same level as the other packages; already available in the original system i.e. the refactored system has one additional package named *aspects*.

7.3.1 Modularization of try-catch clauses

Within the original code there are a number of third-party classes such as *TextMessage* used from different locations in the code. Some of these classes force a behavior which is not wanted or poorly handled. Invoking an operation in the TextMessage object requires that the call is surrounded with a try-catch clause since the operation might throw an exception resulting in code like in Listing 7.1.

```
try {
        category = Category.parse(reqTxtMsg.getStringProperty(QueueDefs.CATEGORY));
} catch(Exception e) {
        // Do nothing, at this pont it's just not available
}
```

Listing 7.1: Invoking operations that might throw.

Since the class is a third-party class there is no possibility of changing this behavior with object-orientation and there are a number of those calls within the system. With aspect-orientation it’s possible to capture the handling of always have to surround the calls with a try-catch clause into a single aspect which affects the whole system such as in Listing 7.2.

[14] http://www.parc.xerox.com/research/projects/aspectj/default.html

```
import javax.jms.JMSException;

public aspect TextMessageExceptions {

	pointcut textMessageExceptions():
		call(String get*(..))
		&& within(com.command.CommandMessage);

	declare soft: JMSException: textMessageExceptions();

	String around(): textMessageExceptions() {
		String result = null;
		try {
			result = proceed();
		} catch(JMSException jmse) {
			// Do nothing, at this pont it's just not available
		}
		return result;
	}
}
```
Listing 7.2: An aspect that handles a thrown JMSException.

The aspect contains a pointcut which matches all join points where an operation is invoked which has a name that starts with *get*, return a string, take any number of parameters of any type and is done from within the package *com.command.CommandMessage*. The advice around takes over and executes instead of the invoked operation until, and if, there is a *proceed* statement. So in this advice the return value is provided to the advice instead of to the code where the invocation was done because the keyword *proceed* act as if the invocation was done at that point. However the advice does return the values of the variable *result* which is of the type *String* and should contain the return value of proceed. If there is an exception thrown that exception is taken care of in the advice (in the same way as in the original code) and the advice returns the value which was used to initialize the variable result, i.e. null. The *JMSException* must be declared soft; otherwise the compiler doesn't allow that the invocation is not surrounded with a try-catch-clause.

Modularizing the exception handling this way gives us much cleaner code since the only thing that has to be kept from the original code is the invocation itself (see Appendix A, Listing 1c).

7.3.2 Implicit handling of non-existing attributes

Another crosscutting concern found was the return of *Attributes*. An operation named *getAttributes* is supposed to return the last element of a vector which contains an object of the type *Attributes*. However, if the vector is empty for some reason the operation returns a new object of the class *AttributesImpl*.

This was a crosscutting concern but the amount of source code within the operation was small and for that reason refactoring this code might give more complex code than leaving it as it was. However, the code was still refactored so that the developers interviewed could state how they thought that the refactoring affected the code complexity.

The aspect, which can be found in Appendix A as Listing 2b, is similar to that in the previous example where the catch clause is used to catch an *ArrayIndexOutOfBoundsException* if the vector was empty and instead return an *AttributesImpl* object.

7.3.3 Softened exception handling

This example is similar to the first example where the exception handling is modularized into an aspect. The creations of a new *XMLHandler* are apparent in several locations in the code and *often* look like Listing 7.1.

```
try {
       xmlHandler = XMLHandler.newHandler();
} catch(ParserConfigurationException pce) {
       throw new OPException(pce, new ExceptionParam(Level.WARNING, CLASS_NAME,
       "Kunde inte skapa XML parser"));
} catch(SAXException saxe) {
       throw new OPException(saxe, new ExceptionParam(Level.WARNING, CLASS_NAME,
       "Kunde inte skapa XML parser"));
}
```

Listing 7.3: Creation of a new *XMLHandler* object.

Not only does the exception handling clutter the code, there are also variants of Listing 7.3, e.g. where only a single exception is caught (see Appendix A, Listing 3d). By modularizing this concern into an aspect the code potentially gets less complex and the way of handling the exceptions become uniform throughout the code.

```
public aspect NewXMLHandler {

       pointcut newHandler():
              call(XMLHandler XMLHandler.newHandler())
              && within(com..*)
              && !within(com.aspects..*);

       declare soft: ParserConfigurationException: newHandler();
       declare soft: SAXException: newHandler();

       after() throwing(ParserConfigurationException pce)
              throws OPException: newHandler() {
              final String CLASS_NAME =
                     thisJoinPoint.getStaticPart().getSignature().getName();
       throw new OPException(pce, new ExceptionParam(Level.WARNING, CLASS_NAME,
              "Kunde inte skapa XML parser (ParserConfigurationException)"));
       }
       after() throwing(SAXException saxe) throws OPException: newHandler() {
              final String CLASS_NAME =
                     thisJoinPoint.getStaticPart().getSignature().getName();
       throw new OPException(saxe, new ExceptionParam(Level.WARNING,CLASS_NAME,
              "Kunde inte skapa XML parser (SAXException)"));
       }
}
```

Listing 7.4: Uniform handling of invocations to create a new XMLHandler.

The difference from how exceptions are handled in the two previous examples is that in this example there is no around advice. Instead the exceptions that can be thrown are declared soft and (as in the first example) an *after throwing* advice is used. An *after throwing* advice is only executed if an exception is thrown. So there are two *after throwing* advices declared, one for each exception that can be thrown.

7.3.4 Implicit handling of non existing record elements

This concern has a pretty low amount of code and the implementation only exists within a single class and might thus not be seen as a *crosscutting* concern. It is refactored as an aspect to support future operations with the same concern. In the original code (Appendix A, Listing 4a) the two operations *get*Records()* return *null* if the array *records* is empty.

This behavior is modularized into an aspect (Appendix A, Listing 4b) and the *after throwing* advice catches any *ArrayIndexOutOfBoundsException* and does nothing. This way the operation ought to return null. That way the original code can be refactored into what is considered the intent of the operation; to extract an array and return the first and last element from that array, respectively.

7.3.5 Aspect-oriented singleton-pattern

The singleton pattern (Gamma et al., 1994) is a design pattern known to most software developers. Shortly described any class implemented according to the pattern can only be instantiated once. Any other attempts to instantiate a second object will result in that a reference to the existing object is returned. Because of this the singleton is normally implemented using a private constructor and instead an operation, usually named *getInstance*, is used to create the object. Listing 7.5 is an example of a typical *getInstance* operation.

```
private Config config = null;

public static Config getInstance() throws OPException{
        if (config == null){
                config = new Config();
        }
        return Config;
}
```

Listing 7.5: A *getInstance* operation.

Hannemann and Kiczales (2002) have taken the Gang of Fours (GoF[15]) 23 design patterns (Gamma et al., 1994) and created aspect-oriented versions of those. They found that with AO it is possible to take another approach. It's possible to allow singletons to be created in the same way as any other classes, i.e. using the keyword *new*, because a around advice can be used to override the constructor unless a new instance should be created which can be done using *proceed* (Gamma et al., 1994).

```
import java.util.Hashtable;

public aspect GetInstance {
        private Hashtable instances = new Hashtable();

        pointcut getInstance():
                execution(static * getInstance())
                && within(com..*)
                && !within(com.database..*);

        Object around(): getInstance() {
                Class instance = thisJoinPoint.getSignature().getDeclaringType();
                synchronized(instances) {
                        if(instances.get(instance) == null) {
                            // Proceed only if we haven't already created an instance
                                instances.put(instance, proceed());
                        }
                }
                return (Object) instances.get(instance);
        }
}
```

Listing 7.6: Aspect-oriented singleton.

In Listing 7.6 a *Hashtable* is used to keep track of object references. If the hashtable contains a reference to an object of the declaring type that reference is returned, otherwise the operation is executed saving the return value into the hashtable as well as returning it from the around advice.

The Java server has no less then 36 singletons so this is truly a crosscutting concern. By implementing this aspect there is no longer any need for a variable per class keeping track of if the class has been instantiated or not, the aspect handles that.

Notice that the aspect doesn't override the constructor but the operation getInstance. The operation has been left in the code because of an inheritance of an old solution where an after advice was used to check the value of what the operation returned and if that value was null, the advice replaced that value with a new object reference. However it was not until after the interviews that the fault was found that the new reference could never be stored in the instance variable. The implementation was then rewritten to be more like Hannemann and Kiczales implementation as implemented in

[15] The four authors of Gamma et al. (1993; 1994) are often referred to as the Gang of Four or GoF.

(Miles, 2004), with the difference that instead of overriding the constructor it is the getInstance operation which is overridden.

7.3.6 Disallowing arguments with a null value

In the Util package a class was found which consist of five operations that all take arrays as input a return the hex value of that input. Even though this concern does not crosscut more than one package and is actually only implemented in a single class *Hex* aspect-orientation can be used to get a uniform behavior of testing the input for errors. In the original implementation all operations started with a condition to test if the input was null (Appendix A, Listing 6a). That condition has been modularized into the simple aspect shown in Listing 7.7.

```
public aspect ToHex {

        pointcut toHex(Object obj):
                execution(String Hex+.toHex(*[],..))
                && args(obj, ..);

        String around(Object obj): toHex(obj) {
                if(obj == null)
                        return "null";
                else
                        return proceed(obj);
        }
}
```
Listing 7.7: A simple aspect.

The around advice returns the string "*null*" if the input is indeed null and proceeds in any other case. Notice how the input is included in the pointcut and advice declaration so that it can be evaluated in the condition. Private variables are normally not visible to aspects so they must be "sent" into the advice like in Listing 7.7 or the problem can be solved in some other way, like in the second example (Appendix A, Listing 2b) where a try-catch-clause is used instead of an if-statement.

It may be so that this concern also is made more complex modularizing this into an aspect, but since the aspect affects five operations the readability of five operations is improved. Also adding a similar operation, and this is likely to happen, that operation will also automatically include the condition, so there is no need to copy and paste boilerplate code.

7.3.7 Implicit creation of directories

The implicit creation of directories is based on the discovery that after every creation of a new *File* object, the operation *mkdirs* is executed. By capturing all calls to the constructor of the *File* class in a pointcut and tie that pointcut to an *after* advice

7.3.8 Implicit formatting of a new DBHandler object

In the *LogSupervisor* class a lot of different kind of handlers are instantiated and formatted like in Listing 7.8.

```
DBHandler serverHandler = new DBHandler();
serverHandler.setFormatter(new DBFormatter());
serverHandler.setLevel(dbLevel);
opLogger.addHandler(serverHandler, SERVER_LOGGER);

DBHandler clientHandler = new DBHandler();
clientHandler.setFormatter(new DBFormatter());
clientHandler.setLevel(dbLevel);
opLogger.addHandler(clientHandler, CLIENT_LOGGER);

DBHandler databaseHandler = new DBHandler();
databaseHandler.setFormatter(new DBFormatter());
databaseHandler.setLevel(dbLevel);
opLogger.addHandler(databaseHandler, DATABASE_LOGGER);
```
Listing 7.8: Instantiation and formatting of a *DBHandler* object.

This behavior has been modularized into an aspect (Appendix A, Listing 8b), so when a new *DBHandler* object is to be created an around advice is executed, actually creating the object, capturing the reference and formatting it using the operations *setFormatter* and *setLevel*, the reference is then returned from the advice. Notice that the arguments for *setFormatter* and *setLevel* are created within the aspect, i.e. to get code where it's apparent what the values of the arguments are.

```
DBHandler serverHandler = new DBHandler();
DBHandler clientHandler = new DBHandler();
DBHandler databaseHandler = new DBHandler();

opLogger.addHandler(serverHandler, SERVER_LOGGER);
opLogger.addHandler(clientHandler, CLIENT_LOGGER);
opLogger.addHandler(databaseHandler, DATABASE_LOGGER);
```
Listing 7.9: Refactored code.

Creating such an aspect renders the possibility to refactor the original code in Listing 7.8 into the code in Listing 7.9.

7.4 Qualitative assessment

In this sub-section the result from the qualitative assessment is presented. The results are based on interviews with three interviewees described in Section 6.2.1.

The interviews include quite much data. This section tries to present as much of the results as possible in a clear and easy to comprehend way.

7.4.1 Advantages

The results of the interviews show that all interviewees think that AO seems reasonable and that it has a very neat and elegant syntax[16]. There seems to be an agreement that AO may, if used correctly, decrease the complexity by refining the code to its actual purpose, modularizing the boilerplate code into an aspect. The result is often said to be an easy to read, compact and focused code where the meaning of the code is all that is left, i.e. high cohesion.

All interviewees agreed that there is a large possibility to simplify parts of the software development process and that AO should be included within the whole software development process if AO is to be used in a sensible way. That way the advantages of AO can be taken into account when creating the system design.

Error handling is an area where AO can really help according to the interviewees. Error handling is hard and easy to forget so if that could be generalized so that it becomes implicit and it could be put somewhere where it's not shown, that would be great, according to Interviewee 1. There might also be opportunities to use AO which

[16] The example which they base their opinion of is written in AspectJ (version 5).

hasn’t been found yet, like using it to format a third-party class which source code is not available according to Interviewee 2.

All of interviewees agree that the best usage for AO seems to be to *extend* OO by moving things like error handling to aspects to decrease complexity. According to Interviewee 1 AO is really interesting and the interviewee is certain that the technique is here to stay in some level.

7.4.2 Disadvantages

The main problem with AO seems to be that there are two places to read the code, which can be hard, since one needs to go into the aspect so see what gets invoked, particularly if someone needs to review the code which has not developed it. Such a task is practically impossible without the use of a tool. Another problem is that AO might do things which the developer is not aware of and that increases complexity. If a developer doesn’t know about AO, he or she will probably be a bit concerned that there e.g. is no exception handling, unaware that there is an aspect that is weaved into the system.

Another disadvantage is if the original code is used only in a few places in the source code or if the original code isn’t very complex. In such cases, where there is only a small amount of code that is being modularized the complexity can, according to the interviewees, be increased instead of decreased. If there is a small amount of code there is no point in writing an aspect for it states Interviewee 3. It might be easier to see what the original code does than what the refactored code with the additional aspect does according to Interviewee 2.

According to the interviewees aspects should be kept short, so that it’s known what they do. There is a danger in putting logic into the aspects; because the code gets more complicated and harder to read. It must not be hard to grasp what the code plus aspect do. Since the aspects themselves are a part of the code they have to be reviewed as every other part of the code. The aspects must not increase the complexity but decrease it, so it’s important that aspects are written to modularize issues that are truly crosscutting, there is a value of showing what’s happening. In some places it might be a meaning behind the code being structured in a certain way. Sometimes a different handling of things is wanted for some reason (deviations from the aspect). There is a possibility to miss something like that when refactoring if care is not taken so that hidden semantic is removed.

To be able to fully take advantage of AO the developer must evolve a certain way of thinking. All interviewees recognize that it probably takes experience to practice AO in a good way and Interviewee 1 believes that it probably requires *a lot* of experience to use AO wisely. The thing that is most difficult and requires experience is to know when to use, and when not to use AO according to the assessment.

7.4.3 Name conventions

Name conventions were according to the interviewees something that plays a key role in the usage of AO. In the interviews all interviewees discussed how the naming of things has to be done more carefully and without a naming convention it can be very hard to create good pointcuts. According to Interviewee 3 it’s not uncommon that operations and attributes are misspelled.

7.4.4 Tool support

It’s apparent that tool support is important for aspect-oriented software development (AOSD). This is particularly pushed by Interviewee 2. If the tool used to develop the system doesn’t include support for aspect-orientation at all, then it is almost impossible to work with aspects because one can’t by just reading the code see that it’s affected by an aspect. All interviewees argue that the complexity of AO requires a tool that, at least, show that there is an aspect affecting the code and provides a hyperlink to that aspect.

Interviewee 2 also discussed the role that documentation plays in a software development process stating that information about aspects must be included in the system documentation, and expressed a need for a tool which can generate such documentation which includes aspects, support which is supported in AJDT.

7.4.5 Consistent way of working

If AO isn’t used a problem might be solved in a particular way, then due to time, guidelines establishments/changes or some other change the problem is solved in a slightly altered way in another way in another location in the same system, i.e. the solutions deviates. This result in boilerplate code i.e. code which is copied and pasted in several locations within a system, something which is common according to Interviewee 3.

According to especially Interviewee 2; with AO there is consistent way of handling things. With AO in the previous example, on a changed behavior then it becomes easy to change the implementation, since such code is only implemented once as an aspect, and it automatically applies to all locations where the problem exists. According to Interviewee 3 it’s good that it’s possible to get such help because often things are changed in nine places out of ten, leaving the system in an erroneous state. So even if there is little to gain in terms of lines of code (LOC), AO can aid developers implement a consistent solution to a particularly problem.

7.4.6 The singleton case

One of the found and refactored crosscutting concerns was the usage of the design pattern Singleton (Gamma et al., 1994). The interviewees all recognized that there is a certain advantage if one can tell if an object is a singleton or not without having to look in the pointcut. The operation *getInstance* is what makes this explicit in a normal singleton case. The operation is kept also in the refactored code because it would be very hard for anyone not familiar with the (AO) pattern to understand the code, and impossible for someone without the knowledge of AO in general it was deleted. In the aspect-oriented singleton as implemented by Hannemann and Kiczales (2002) the Pointcut is what determines if an object is a singleton or not, there are no *getInstance* operation, and according to the Interviewee 1 this decreased the readability although it might be useful.

7.5 Quantitative assessment

This sub-section presents the results from the quantitative assessment. The metrics described in Section 7.1.2 is applied to both the original system and the refactored system and their values are compared.

7.5.1 Vocabulary size

There were 241 classes (not counting test classes) in the original system and since no classes has been removed or added the difference is those eight aspects added. If calculated as a percentage increase, the increase is 3.3% which is to be considered as a small complexity increase.

7.5.2 Lines of code

The lines of code is a simple metric but gives a good rule of thumb of how complex a system is, or how productive one have been during a certain period of time. LOC has been measured for all concerns but the *implicit handling of non-existing record elements* which is not included since there is no difference between the two systems, with the assumption that a clause with a single statement is counted as a single line, i.e. the if-clause.

Concern	**Original**	**Aspect**	**Refactored**	**A+R**[17]	**±C%**
Modularizing try-catch-clauses	20	14	9	23	+15%
Implicit handling of non-existing attributes	40	15	15	30	-25%
Softened exception handling	56	20	8	28	-50%
Implicit handling of non existing elements	14	8	8	16	+14.3%
Aspect-oriented Singleton pattern	374	15	272	287	-23.3%
Disallowing arguments with a null value	15	11	0	11	-26.7%
Ensuring that directories are created	6	9	3	12	+50%

Table 7.2: Lines of code (LOC) on the refactored concerns.

Table 7.2 is a collection of the charts available in Appendix C. The lines of code for the original system should be compared to the sum of the aspect (A) plus the refactored code (R). The values for the original system are calculated by multiplying the lines of code for the concern with the number of classes the concern is scattered into.

So if there is an operation named *foo* which consists of 6 lines and *foo* exists in 6 classes the value for the original code is 36. If 4 of those lines were modularized into an aspect which came to consist of 14 rows then that value is 14 (the aspect only occurs once). The value for the refactored code becomes 12, i.e. 6 minus 4 equals 2 which is multiplied with 6 (the number of components). To compare the refactored solution the original value of 36 should be compared to the sum of 14 and 12 which is 26. According to the LOC metric the refactored solution is, in this case, considered less complex. The column to the right states if the complexity is increased or decreased.

According to Table 7.2 the complexity increases in three cases and that is when modularizing try-catch clauses, when there is implicit handling of non existing elements, and when ensuring that directories are created. Worth to note is that the results relies on the original values which could drastically change if the number of

[17] The sum of the aspect plus the refactored code. To be compared with the original value.

classes the concern is scattered into would change. Also, the values are quite small which implies that conclusions drawn from them might be spurious. The results should be used a pointer and complement to the quantitative assessment, not as an absolute truth.

7.5.3 Number of Attributes

An attribute is defined as an instance or a class variable for a component. The number of attributes is calculated in the same way as the LOC; only instead of counting the lines the attributes are counted. As above the concerns that definitely would not show difference has not been included.

Concern	Original	Aspect	Refactored	A+R	±C%
Modularization of try-catch clauses	8	3	6	9	+12,5%
Implicit handling of non-existing attributes	2	1	1	2	-
Softened exception handling	40	4	8	12	-70%
Aspect-oriented Singleton pattern	204	2	170	172	-15.7%
Disallowing arguments with a null value	15	1	15	16	+6.7%

Table 7.3: Number of attributes (NOA) on some of the concerns.

Table 7.3 shows that for the second concern the modularization has no affect on the complexity. In two of the cases the complexity is quite reduced according to NOA, and that included high values of the original and aspect + refactoring code values which otherwise would keep the increase down. In the two other cases an increase is shown, but in both cases there is only a difference on a single attribute.

7.5.4 Coupling between Components

Only two concerns are refactored in such a way that the coupling between components is changed: implicit handling of non-existing attributes and softened exception handling.

Concern	Original	Aspect	Refactored	A+R	±C%
Implicit handling of non-existing attributes	10	2	5	7	-30%
Softened exception handling	40	5	8	13	-67.5%

Table 7.4: Coupling between Components (CBC) on two of the concerns.

Table 7.4 shows that the refactoring decreased the complexity, in one case with 30% but the softened exception handling refactoring decreased the complexity with more than 67% according to the CBC metric.

8 Analysis

In this section the results from the previous section is analyzed. The analysis is divided into four parts: refactoring, interviews, measurement, and a cross-comparison where the results from the interviews and measurement are compared and discussed.

8.1 Refactoring

When refactoring an object-oriented system in to an aspect-oriented system and measuring the difference, an important thing to keep in mind is that if the assessment is supposed to show the difference between OO and AO, then the system should not be refactored using OO. No class should be rewritten or moved instead focus should lie in finding crosscutting concerns and implement aspects which address such crosscutting concerns. A task which in a well implemented object-oriented system isn't trivial.

A possibility which has been considered but not used widely in this thesis is to use more of the aspect-oriented design patterns[18] to refactor the system. In the refactored system the only aspect-oriented pattern used is the Singleton pattern. The reason for not using more aspect-oriented design patterns is that, it would probably put the project in a state where common object-orientation based refactoring would have been done also, and that would not produce correct results since the aim is to assess the introduction of aspect-orientation *not* to assess the refactoring of a particular system. Another thing that can be refactored is the actual way that some things i.e. exceptions are handled. But since the refactored system must function in the exact same way as the original does that idea has not been realized either. Carrying out such a study of is instead suggested as future work.

8.1.1 Object-oriented and aspect-oriented refactoring

The difference between a "normal" (object-oriented) refactoring and a aspect-oriented refactoring is that with a OO refactoring components are often added, renamed, moved, or even removed so that the dominant decomposition becomes less complex, It seems like the aim with OO refactoring is to reduce coupling. In an AO refactoring focus is to find crosscutting concerns and modularize those to get an easier to read, more compact and focused code, i.e. increase the cohesion of the components. This suggests that the two types of refactoring complement each other.

If the definition that a system is as complex as its most complex part is used, then the results show that there is certainly a gain in refactoring an object-oriented system into an aspect-oriented system, particularly if the developer refactoring the system has good knowledge about aspect-orientation and focuses on finding crosscutting concerns which is known to get good results when modularized using aspect-orientation. However, this thesis also shows that an AO refactoring is very difficult and does not always decrease complexity, in the same way that OO refactoring almost always do. A conclusion that could be drawn from the results is although AO refactoring may decrease complexity in a system, it's better to build a system using AOSD then to refactor it afterwards when the design has already been created. This conclusion is supported by the interviews and the work that has been put into the refactoring done within this project.

[18] http://www.cs.ubc.ca/~jan/AODPs/

8.1.2 The danger of modularizing too much

Implementing an aspect always require an increase LOC value since there is some overhead writing the aspect definition and pointcut before actually writing the advice. It is possible that the LOC value is decreased when simplifying original code, but it is also possible that the value is not increase so that an aspect-oriented implementation of a crosscutting concern has a higher LOC value than the original implementation, this has been shown in the measurements in this thesis. Normally LOC is not seen as a very important metric, but besides that there is also another reason for not care about the increase of LOC, i.e. because an aspect makes a crosscutting concern explicit which decreases complexity. By having crosscutting concerns explicitly modularized it's an easy task to grasp the concern and to update it's implementation. For this reason one might get tempted to use AO more or less everywhere to get "less complex" code, since all concerns would be modularized and hidden from the rest of the code, a property which is both the strength and weakness of AO. However, if not properly done, or if the developer is unaware of the aspects for various reasons such as lack of AO knowledge, or proper tools, a spurious situation might arise. This shows that even though it's good to make crosscutting concerns there is a danger modularizing too much, and that it takes experience to practice AOSD.

8.1.3 Addressing reliability

To address reliability the refactored system would preferably have been executed and tested so that the behavior of the system has not been changed. Unfortunately the refactored aspect-oriented version could only be compiled, and do compile without any warnings, but never executed. This is because the data needed for the system to run is classified and there was no possibility to create all random data needed. However, the refactoring has been reviewed by three developers from the company (in relation to the interviews) as well as by a volunteer from the *aspectj-users mailing list*, so the refactored system is most probably equivalent with the original system.

8.2 Qualitative assessment

In this sub-section I extend the results of the qualitative assessment with my thoughts on certain issues. I would also like to add, for the sake of reliability, that the interviewees seemed to understand all the questions and the refactoring made.

8.2.1 Complexity

Even if not discussed explicitly by the interviewees it is my belief that by modularizing boilerplate code into aspects, not only does the main code become less complex, but also implementation of the crosscutting concern becomes more "visible" then without usage of aspect-orientation. By separating tangled code the result may perhaps be a bit more code[19] but the code is still easier to read because the purpose of each module becomes more visible as the cohesion is increased, as opposed to if the two where tangled into a single module. A conclusion is that less code might not always mean less complex code.

[19] As the result of the measuring using the LOC metric shows.

8.2.2 Tool support

The implicit invocation makes it harder to work aspect-orientation then with object-orientation, according to all the interviewees. This criticism is also expressed by several other developers according to Beuche and Beust (2006). The problem can be reduced with (CASE) tool support which according to Low and Leenanuraksa also increases the quality of the software ”the quality of software developed using back-end CASE tools is better than conventionally developed systems” (Low & Leenanuraksa, 1999, p. 149). However, a problem with new research areas is that the number of tools available to support research within that area is usually not that big. According to Interviewee 2 the tools must not just markup aspects but also produce documentation about how aspects are weaved into the system. I agree and state that proper tool support is vital for any aspect-oriented framework. So to boost the accessibility for other languages than Java, tools to weave, markup, debug and document aspects must be developed. The tools must be small, intuitive and easy to install to gain popularity.

8.2.3 Consistent way of working

Interviewee 2 spoke of a larger use of AO that AO could not only be seen as an extension to OO, but also be used in a wider context to address separation of concerns (Tarr et al., 1999). Colyer et al. (2006) has stated that AO means a major change in how developers think and solve problems. The interviewee primarily saw AO as that extension to OO taking care of error handling, logging and database calls e.g., but also recognized that AO do include a new way of thinking. All interviewees agree that AO should be included in the design phase of a software development process, and Interviewee 1 also noted that it probably was hard to refactor a system afterwards because AO is not, actually, founded on the same way of thinking[20].

So is AO *an* OO extension *or* a new way of thinking? My conclusion is that AO is both! One can treat AO as an extension to OO and refactor systems to take care of error handling, file handling, and such. One of the interviewees said that AO felt like OO with an additional level, and according to another of the interviewee all colleagues he had talked to viewed AO as an extension to OO. Treating AO as an extension to OO doesn’t *require* the inclusion of AO in the design phase, since such aspects can be written afterwards with a little effort.

Or one can use AO as a whole new way of developing software and build multiple hierarchies which are woven into each other where each hierarchy addresses a separate separation of concern. Such a system would not have a dominant decomposion (Tarr et al., 1999) and therefore require much (AO) experience from the developers to grasp the system. There is a big threshold to climb here! However, it is my fully conviction that gradually software development will work this way, and that will also require more from all kind of software developers than what is required today. It is my conviction that much of the software complexity that exists today will moved from the design and implementation of the system to the art of creating the design and implementation of the system.

The first applications we are going to see will not be written this way. If aspect-orientation will be started to be used widely the software will contain aspects that help the developers with small things such as already discussed error handling and logging. Gradually the aspects will be more complex and as an interim state software designs

[20] i.e. something that I can agree on, after done my best refactoring the Java server described in Section 5.

are created using best practices from the AO community, design will have taken a step further and matured into Aspect-oriented design (AOD). It is my belief that if AO is started to be used, that step will not take very long time. The step from the interim state to the final state, as so to speak, where there is no apparent dominant decomposition are much further away.

8.2.4 Object-orientation vs. Aspect-orientation

Most problems which can be solved using AO can also be solved using just OO (otherwise software could not be built). One of those things is modularizing concerns, and in some situations pure OO is better suited to solve a problem than AO. An util package consists of modularized code for common tasks, often implemented as static operations. The difference between solving this using OO and AO is the implicit invocation of the aspect at defined join points. If AO is used correctly it has an advantage over regular operations i.e. that when a new situation occurs in which an explicit invocation to a operation must be added when using pure OO, nothing has to be done when using AO since that join point is included in the pointcut. In that sense AO is superior to normal refactoring.

There are also some crosscutting concerns which cannot be modularized with pure OO such as *log enter* and *log exit* in a system. Interviewee 3 stated to use log enter and log exit very often. It's when there exists crosscutting concerns which can't be eliminated with OO that one have a lot of gain using AO. Since an aspect is much like a class there is a possibility to build powerful aspect hierarchies on the side. Most interviewees saw this as a negative thing, Interviewee 3 stated that one usually gain in keeping things simple and flexible. Although I agree, I believe that what software engineering will come to is that this dominant decomposition which exists today will in a distant future have to step aside for multiple hierarchies. Because if it would be a simple task to define an interface between those, if software engineers can do that, there is much to gain in having multiple teams working on more specialized things, and there is also a possibility to reuse whole hierarchies.

8.2.5 The singleton pattern

The AO version of the Gang of Four's singleton pattern as done by Hannemann and Kiczales took an approach where the *getInstance* operation was omitted (Gamma et al., 1994; Hannemann & Kiczales, 2002). By keeping the *getInstance* operation in the refactored system the usage of AO becomes transparent for those who do not know AO and easier to grasp by an AO beginner.

The other option is to not have a *getInstance* operation and instead override the constructor with an aspect letting it proceed if there isn't an object already instantiated or otherwise return the already existing object. The created objects are kept in a hashtable within the aspect[21] so if there is an object with the correct type that one is returned instead of letting the constructor proceeds.

During the interviews a certain implementation were used where the *getInstance* operation returned the value of the uniformed variable instance. After the interviews an error was found[22] which was corrected by using the Hannemann and Kiczales (2002) variant of the aspect-oriented singleton as implemented in (Miles, 2004).

[21] Aspects are by default singletons (Miles, 2004), i.e. there is only one instance of the hashtable.
[22] The instance variable could not be set since the advice is executed after the operation

According to Interviewee 2 there are often classes which are treated like a singleton, i.e. only instantiated once and must not be instantiated a second time. Despite this they classes were not always implemented as singletons, but as normal classes. All the interviewees thought that using the singleton solution where the constructor was overridden by an aspect seemed appealing if there is a possibility somehow, perhaps through naming, to see which classes that is a singleton and which was not. So, if the AO singleton pattern with a good naming convention, e.g. that all classes which names ends with the string “Singleton” are treated like singletons, then implementing singletons would be as easy as changing the name of the class.

8.2.6 Public availability

One thing discussed during the interviews was the public availability of aspect-orientation. I share Beuche and Beust (2006) view of the state of AO as a chicken and egg problem. It’s a chicken and egg problem because the public availability of information about aspect-orientation is very low, and if AO would be used in the software industry it has to be publicly known.

One way to grow out of the state is if a big actor in the software industry actively started support AO by, e.g. integrating this into their compilers. Today IBM support AO in different ways e.g., having a special (peer-reviewed[23]) AOP@Work section[24], supporting Eclipse which has strong support for AO in AJDT[25], and have staff that does research on AO.

It’s my firm belief that if Sun or Microsoft would bundle Java or .NET with tools for aspect-orientation or even integrate the techniques of wowing aspects into code into their compilers, then AO would gain a lot of publicity and developers would start using it quite fast. This belief is also expressed from all interviewees during the interviews – a big actor must embrace aspect-orientation for it to really take off.

8.3 Quantitative assessment

This sub-section discusses two biases linked to the quantitative assessment.

As previously discussed the quantitative assessment is calculated based *only* on the refactored components (classes and aspects) and operations (including advices). The Java server consists of more than 240 regular classes and is thus considered quite large. Comparing the complete original system with the complete refactored system will therefore not show any difference worth analyzing. To get a more meaningful result the measuring is instead only concentrated on the operations and components which are affected by the refactoring. The measuring was done this way to be able to measure the implications that each aspect has on the original system. Quantitatively comparing the full systems would show values with insignificant differences.

The second bias is that percent column as a consequence sometimes shows very high numbers when the difference might actually be quite low. An example is when measuring NOA for the ‘Modularization of try-catch clauses’ aspect. According to the assessment the complexity increased +12,5%, but the difference is only a single attribute. The percent value serves more as a measurement measuring if the complexity was increased or decreased than an actual value itself.

[23] See box at http://www-128.ibm.com/developerworks/library/j-aopwork3/

[24] http://www-128.ibm.com/developerworks/views/java/libraryview.jsp?search_by=AOP@work

[25] AJDT is developed by a number of people, some of them employed by IBM

8.4 Cross comparison

This sub-section contains a cross comparison of the open interviews (qualitative data) and the measurements (quantitative data), Table 8.1 contains data from both assessments and is used as a reference in the discussion.

Concern	**I1**	**I2**	**I3**	**I1+I2+I3**	**LOC**	**NOA**	**CBC**
Modularizing try-catch-clauses	Yes	Yes	Yes	Yes	No	No	–
Implicit handling of non-existing attributes	No	No	No	No	Yes	–	Yes
Softened exception handling	Yes	Yes	No	Yes	Yes	Yes	Yes
Implicit handling of non-existing elements	No	No	?	No	No	–	–
Aspect-oriented Singleton pattern	?	Yes	?	?	Yes	Yes	–
Disallowing arguments with a null value	Yes	No	Yes	Yes	Yes	No	–
Ensuring that directories are created	Yes	?	?	?	No	–	–
Implicit formatting of a new DBHandler-object	Yes	?	Yes	Yes	–	–	–

Table 8.1: Quantitative and qualitative cross comparison.

I1 represents Interviewee 1 and the column is marked with *Yes* if the Interviewee would have implemented the aspect and *No* otherwise, the same description also applies to I2 and I3. If the interviewee was not sure that is marked with a questions mark. The next column is a sum of the three previous columns. If at least two of three of the Interviewees stated that they would have implemented the aspect it is considered that the sum of the interviewees would have implemented the aspect.

The next three columns consist of LOC, NOA and CBC, and describe whether the measurement showed that the complexity decreased or not. A hyphen is used to describe that the metric could not be applied to the aspect.

Table 8.1 show no correlation between whether the interviewees would have implemented the aspect or not, and the results from the measurements. However, based on the always returning standard examples for aspect-orientation, such as logging and database connections, there is a strong possibility that there are some crosscutting concerns which always give good results when refactoring and should therefore be the crosscutting concerns which are looked for when refactoring an OO-system.

9 Conclusions

In this last section I discuss the results, things derived from or that could have affected the results, and draw some conclusions. The section also contains paragraphs were this work is positioned among others, a short discussion of the contributions that this thesis might have given and is concluded with a few suggestions for future work.

9.1 Discussion

Refactoring an object-oriented system using aspect-orientation, i.e. finding crosscutting concerns afterwards, was hard due to a number of reasons:

- The lack of experience in finding crosscutting concerns. To find crosscutting concerns and build a system with aspect-orientation requires experience, searching for crosscutting concerns afterwards is even harder since the system is already built and the design can not be changed so that crosscutting concerns can be modularized.
- If the system is well written using object-orientation it's probably so that crosscutting concerns has already been modularized in some form. There seems to be an implicit knowledge of crosscutting concerns and such are often already separated, e.g. into utility classes using object-orientation, although such utility classes are often not reused.

It can be therefore being considered not a best practice to use AO to refactor a system.

However the results do show great results on some of the aspects which suggest that refactoring an object-oriented system into an aspect-oriented system is an option if there are certain crosscutting concerns that can be refactored which are known to generate good results. It is therefore suggested as future work to create guidelines for what to look for when refactoring an OO system into an AO system.

The systems complexity did get high perceptual differences but that was often because of small values. The refactored system was not quantitatively much different from the original system, this can have many reasons some more than apparent others:

- Even though I have good knowledge about OO, I have no previous experience in working with AO (that includes the language: AspectJ) which could have some influence on how well the language was used.
- The system has been developed for years and is thus large, which requires a quite dramatically refactoring if the refactoring should be considered a major change.

The Java server subject to this case study was suggested by my supervisor at the company. The suggestion came after that I asked for a representative real-world system developed in Java by the company. The reason to delimit the systems to Java systems was twofold. First, I knew before starting the project that the strongest support of aspect-orientation are Java implementations, and second, I have better experience developing systems in Java versus other languages such as C++. By delimit the systems to Java implementations I address validity and reliability by refactoring a system in a language I am very familiar with the strongest support in form of the best frameworks which should result in a better thesis.

9.2 Summary

The conclusions from this thesis are that aspect-orientation has the possibility to decrease complexity probably independent of whether a system is refactored or built from scratch, even if it seems that the gain is higher if AO is thought of already in the design phase according to the interviewees.

The quantitative assessment of the introduction of aspect-orientation shows that the complexity decreases heavily for a number of crosscutting concerns and increases for others. If the view that a system is a complex as it’s most complex part then the refactoring showed god results since there are at least three crosscutting concerns (Implicit handling of non-existing attributes, Softened exception handling and Aspect-oriented Singleton pattern) which show decreased complexity no matter which metric that was used. No correlation can be found between the crosscutting concerns that the developers would implement and those that decreased complexity according to the metrics.

The results from the qualitative assessment show that aspect-orientation can be used to refactor a system and modularize crosscutting concerns, and that the complexity is often decreased if AO is used in a sensible way. When using AO it’s important that name conventions are held and no logic is modularized. According to the interviews the logic should remain in the main code, otherwise the complexity is increased. Moreover, the AOSD process has to be supported by knowledge of aspect-orientation, tools and proper use of aspect-orientation.

9.3 Related work

This section presents some related work and tries to argue for how this thesis complements related work already published.

In the beginning of AO (late 1990, beginning 2000) there was a lot of technical and modular research going on (Kiczales et al., 1997; Tarr et al., 1999; Hannemann & Kiczales, 2002; Popovici, 2002) and aspect-orientation was implemented into a lot of different syntaxes (Kiczales et al., 2001; Spinczyk & Schröder-Preikschat, 2002; Lämmel & De Schutter, 2005). Only a small amount of the research made at that time was based on investigating the usability of AO. Some of the researches that carried out such studies were Murphy et al. which carried out experiments such as “to investigate whether AOP, as embodied in AspectJ made it easier to develop and maintain certain kinds of application code” and “to investigate the usefulness of AO” (Murphy et al., 2001, p. 76).

Today there are a number of researchers focusing on quantitatively assessing aspect-orientation (Sant`Anna et al., 2003; Cacho et al., 2006; Figueiredo, et al., 2005; Garcia et al., 2005) however, most of the research is very fresh and only a few, e.g. Filho et al. (2005) seems to assess real-world refactored systems.

This thesis complements the related work published for two reasons. The first reason is that it extends the category on work done assessing a real-world refactored system delivered by a Swedish software company. The second reason is that this thesis assesses the systems *both* qualitatively by carry out open interviews with the original developer as well as two other developers from the company, and quantitatively by using the metrics suite from Sant`Anna et al. (2003).

9.4 Contributions

With this thesis I hope to boost the interest for aspect-orientation and other software engineering techniques that have the possibility to decrease the complexity in the software development method. Hopefully this work can also serve as a starting point for companies interested in aspect-orientation but doesn't have any knowledge of AO. The quantitative results in this thesis show that aspect-orientation can be used to decrease complexity also when refactoring a system although to do so there are some requirements that must be fulfilled. Some of those requirements are discussed in the qualitative results and should be representative for other developers as well.

9.5 Future work

This thesis shows that aspect-orientation can be used to refactor a pure object-oriented system, and according to the results the gain is probably even higher if aspect-orientation is included within the software development method. In the interviews all of the interviewees expressed their opinion stating that aspect-orientation should probably be thought of already in the design phase. However, to be able to do that, the knowledge about available tools and frameworks needs to be further investigated by carrying out a more thorough investigation of available tools and frameworks for different programming languages. There is a list of research projects[26] hosted at aosd.net[27] the home of the annual aspect-oriented software development conference.

Moreover, in this thesis, the introduction of aspect-orientation is assessed regarding complexity, but there are also a number of other software quality attributes which is affected by the introduction of aspect-orientation, a suggestion for future work is to assess the introduction of aspect-orientation regarding some other software quality attribute.

The results show that the modularization of some crosscutting concerns gave very good results, especially if the view that the system is only as complex as its most complex part. When talking about AO there are often a number of crosscutting concerns which are more or less always mentioned. A final suggestion for future work is to investigate whether there are any crosscutting concerns which are more grateful to refactoring using AO then others and create guidelines for refactoring an OO system into an AO system.

[26] http://www.aosd.net/wiki/index.php?title=Research_Projects
[27] http://www.aosd.net

References

Alexander, C. 1979. *The timeless way of building*. Oxford University Press, New York.

Alexander, C., Iskikawa, S., Silverstein, M., Jacobson, M., Fliksdahl-King, I., and Angel, S. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

AlShariff, M., Bond, P. W., and Al-Otaiby, T., 2004. Assessing the Complexity of Software Architecture. In *Proceedings of the 42nd annual Southeast regional conference*, Huntsville, Alabama, USA, April 2nd-3rd, pp. 98-103.

Baniassad, E., Clements, C. P., Araújo, J., Moreira, A., Rashid, A., and Tekinerdogan, B., 2006. Discovering Early Aspects. *IEEE Software*, 23(1), January, pp. 61-70.

Berndtsson, M., Hansson, J., Olsson, B., and Lundell, B. 2004. *Planning and Implementing your Final Year Project with Success!* 2nd edition (reprint from 2002). Springer, London, England.

Beuche, D., and Beust, C., 2006. AOP Has Yet to Prove Its Value. *IEEE Software*, 23(1), January, pp. 73-75.

Booch, G. 1993. *Object-Oriented Analysis and Design with Applications*. 2nd edition. Addison-Wesley, London, England.

Cacho, N., Sant' Anna, C., Figueiredo, E., Garcia, A., Batista, T., and Lucena, C., 2006. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *AOSD'05: Proceedings of the 5th international conference on Aspect-oriented software development*, Bonn, Germany, March 20th-24th, pp. 109-121.

Chapman, M., 2005. *New AJDT releases ease AOP development*. [online]. Available from: http://www-128.ibm.com/developerworks/java/library/j-aopwork9/ [Accessed 2006-06-06]

Chapman, M., and Hawkins, H., 2004. *Develop aspect-oriented Java applications with Eclipse and AJDT*. [online]. Available from: http://www-128.ibm.com/developerworks/library/j-ajdt/ [Accessed 2006-05-20].

Colyer, A., Clement, A., Harley, G., and Webster, M. 2004. *Eclipse AspectJ*. Addison-Wesley, London, England.

Colyer, A., Harrop, R., Johnsson, R., and Vasseur, A., 2006. AOP Will See Widespread Adoption. *IEEE Software*, 23(1), January, pp. 72-75.

Dijkstra, W. E., 1967. The structure of "T.H.E" Multiprogramming Systems. *ACM Symposium on Operating Systems Principles CAMC*, 18(8), pp. 10.1-10.6.

Elrad, T., Filman, E. R., and Bader, A., 2001. Aspect-oriented programming. *Communications of the ACM*, 44(10), October, pp. 29-32.

Figueiredo, E., Garcia, A., Sant' Anna, C., Kulesza, U., and Lucena, C., 2005. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In *QAOOSE'05: Proceedings of the 9th ECOOP Workshop on Quantitative Approaches in OO Software Engineering*, Glasgow, Scotland, July 25th, pp. 58-69.

Filho, F., Rubira, F. C M., and Garcia, A., 2005. Quantitative Study on the Aspectization of Exception Handling. In *ECOOP'05: Workshop on Exception Handling in OO Systems,* Glasgow, Scotland, July 25th-29th.

Gamma, E., Helm, R., Johnson, E. R., and Vlissides, M. J., 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, Kaiserslautern, Germany, July 26th-30th, pp. 406-431.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, MA, USA.

Garcia, A., 2004, From Objects to Agents: An aspect-oriented approach. Doctoral Thesis, Computer Science Department, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil, April.

Garcia, A., Sant' Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A., 2005. Modularizing Design Patterns with Aspects: A Quantitative Study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, Illinois, USA, March 14th-18th, pp. 36-74.

Hannemann, J., and Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. In *OOPSLA'02: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov, pp. 161-173.

Helm, R., 1995. Patterns in Practice. In *Proceedings of the 10th annual conference on Object-oriented programming systems, languages and applications*, Austin TX, USA, October 15th-19th, pp. 337-341.

Kearney, J., Sedlmeyer, L. R., and Thompson, B. W., 1985. Problems with software complexity measurement. In *CSC'85: Proceedings of the 1985 ACM Computer Science Conference*, New Orleans, Louisiana, United States, March 12th-14th, pp. 340-347.

Kersten, M., 2005. *AOP Tools Comparison Part 1*. [online]. Available from: http://www-128.ibm.com/developerworks/java/library/j-aopwork1/ [Accessed 2006-05-20].

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M., and Irwin, J., 1997. Aspect-Oriented Programming. In *Proceedings of the 11th Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 9th-13th, pp. 220-242.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W., 2001. An overview of AspectJ. In *Proceedings of the 15th Conference on Object-Oriented Programming*, Budapest, Hungary, June 18th-22nd, pp. 327-353.

Kiczales, G. and Mezini, M., 2005. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 15th-21st, pp. 49-58.

Low, G., and Leenanuraksa, V., 1999. Software Quality and CASE Tools. In *STEP'99: Proceedings of the Ninth International Workshop Software Technology and Engineering Practice*, Los Alamitos, Pittsburgh, USA, Aug 30th-Sep 2nd, pp. 142-150.

Lämmel, R. and De Schutter, K., 2005. What does aspect-oriented programming mean to Cobol? In *Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, Illinois, USA, March 14th-18th, pp. 99-110.

McNamee, P., and Hall, M., 1998. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Notices*, 8(33), August, pp. 17-22.

Magno, E., Garcia, A., and de Lucena, C., 2006. *AJATO*. [online]. Available from: http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/ [Accessed 2006-05-20].

Miles, R. 2004. *AspectJ Cookbook*. O'Reilly, Cambridge, MA, USA.

Murphy, G., Walker, J. R., Baniassad, L.A. E., Robillard, P. M., Lai, A., and Kersten, A. M., 2001. Does aspect-oriented programming work? *Communications of the ACM*, 44(10), October, pp. 75-77.

Oxford University Press, 2006. *Oxford reference online*. [online] Available from: http://www.oxfordreference.com [Accessed 2006-05-20].

Popovici, A., Gross, T., and Alonso, G., 2002. Dynamic Weaving for Aspect-Oriented Programming. In *AOSD'02: Proceedings of the 1st international conference on Aspect-oriented software development*, Enschede, The Netherlands, April 22nd-26th, pp. 141-147.

Rentsch, T., 1982. Object Oriented Programming. *ACM SIGPLAN Notices*, 17(9), September, pp. 51-57.

Sant' Anna, N. C., Garcia, F. A., von Flach G., Chavez, C., de Lucena, C. J, and von Staa, A., 2003. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *SBES'03: Proceedings of Brazilian Symposium on Software Engineering*, Brasilia, Brazil, Oct, pp 1-16.

Ségura-Devillechaise, M., Menaud, J-M., Loriant, N., Douence, R., Sudholt, M., Fritz, T., and Wuchner, E., 2006. Dynamic Adaption of the Squid Web Cache with Arachne. *IEEE Software*, 23(1), January, pp. 34-41.

Spinczyk, O., Gal, A., Schröder-Preikschat, W., 2002. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, Sydney, Australia, February 18th-21st, pp. 53-60.

Stochmialek, M., 2006. *AOP Metrics*. [online]. Available from: http://aopmetrics.tigris.org [Accessed 2006-05-20].

Tarr, P., Ossher, H., Harrison, W., and Jr. Sutton, M. S., 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software Engineering*, Los Angeles, California, USA, May 16th-22nd, pp. 107-119.

The Department of Justice. *Systems Development Life Cycle Guidance Document*. [online]. Avalible from: http://usdoj.gov/jmd/irm/lifecycle/table.htm [Accessed 2006-06-04].

Xia, W., and Lee, G., 2004. Grasping the Complexity of IS Development Projects. *Communications of the ACM*, 47(5), May, pp. 69-74.

UNIVERSITY OF SKÖVDE

School of Humanities and Informatics
Final Year Project in Computer Science 20 p
D-level
Spring 2006

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Appendix A: Refactored source code

Daniel Oskarsson

HS-IKI-MD-06-003

Table of contents

This appendix contains; a) source code from the original system, b) the aspect, including the pointcut and advice c) refactored source code, possible due to the aspect and in some cases, and in some cases also d) alternative original implementation.

1 Modularization of try-catch-clauses

```
try {
   category = Category.parse(reqTxtMsg.getStringProperty(QueueDefs.CATEGORY));
} catch(Exception e) {
   // Do nothing, at this pont it's just not available
}
try {
   reqDef = reqTxtMsg.getJMSCorrelationID();
} catch(Exception e) {
   // Do nothing, at this pont it's just not available
}
try {
   reqXML = reqTxtMsg.getText();
} catch(Exception e) {
   // Do nothing, at this pont it's just not available
}
try {
   reqFromAddress = reqTxtMsg.getStringProperty(QueueDefs.FROM_ADDRESS);
   respToAddress = new String(reqFromAddress);
} catch(Exception e) {
   // Do nothing, at this pont it's just not available
}
```

Listing 1.a: Original code.

```
package com.aspects;
import javax.jms.JMSException;

public aspect TextMessageExceptions {

	pointcut textMessageExceptions():
		call(String get*(..))
		&& within(com.command.CommandMessage);

	declare soft: JMSException: textMessageExceptions();

	String around(): textMessageExceptions() {
		String result = null;
		try {
			result = proceed();
		} catch(JMSException jmse) {
			// Do nothing, at this pont it's just not available
		}
		return result;
	}
}
```

Listing 1.b: Aspect.

```
category = Category.parse(reqTxtMsg.getStringProperty(QueueDefs.CATEGORY));
reqDef = reqTxtMsg.getJMSCorrelationID();
reqXML = reqTxtMsg.getText();
reqFromAddress = reqTxtMsg.getStringProperty(QueueDefs.FROM_ADDRESS);
if(reqFromAddress != null) respToAddress = new String(reqFromAddress);
```
Listing 1.c: Refactored code.

2 Implicit handling of non-existing attributes

```
public Attributes getAttributes() {
        if ( !vec.isEmpty() ){
                return vec.lastElement();
        }
        else{
                return new AttributesImpl();
        }
}
```
Listing 2.a: Original code.

```
package com.aspects;

import org.xml.sax.*;
import org.xml.sax.helpers.*;

public aspect GetAttributes {

        pointcut getAttributes():
                execution(Attributes getAttributes());

        Attributes around(): getAttributes() {
                Attributes result;
                try {
                        result = proceed();
                } catch (Exception e) {
                        result = new AttributesImpl();
                }
                return result;
        }
}
```
Listing 2.b: Aspect.

```
public Attributes getAttributes() {
        return vec.lastElement();
}
```
Listing 2.c: Refactored code.

3 Softened exception handling

```
try {
        xmlHandler = XMLHandler.newHandler();
} catch(ParserConfigurationException pce) {
        throw new OPException(pce, new ExceptionParam(Level.WARNING, CLASS_NAME,
        "Kunde inte skapa XML parser"));
} catch(SAXException saxe) {
        throw new OPException(saxe, new ExceptionParam(Level.WARNING, CLASS_NAME,
        "Kunde inte skapa XML parser"));
}
```

Listing 3.a: Original code.

```
package com.aspects;
import com.util.xml.XMLHandler;
import com.common.ExceptionParam;
import com.common.OPException;
import java.util.logging.Level;
import javax.xml.parsers.*;
import org.xml.sax.*;

public aspect NewXMLHandler {

        pointcut newHandler():
                call(XMLHandler XMLHandler.newHandler())
                && within(com..*)
                && !within(com.aspects..*);

        declare soft: ParserConfigurationException: newHandler();
        declare soft: SAXException: newHandler();

        after() throwing(ParserConfigurationException pce)
                throws OPException: newHandler() {
                final String CLASS_NAME = thisJoinPoint.getStaticPart()
                        .getSignature().getName();
                throw new OPException(pce, new ExceptionParam(Level.WARNING, CLASS_NAME
                        ,"Kunde inte skapa XML parser (ParserConfigurationException)"));
        }
        after() throwing(SAXException saxe) throws OPException: newHandler() {
                final String CLASS_NAME =
                        thisJoinPoint.getStaticPart().getSignature().getName();
                throw new OPException(saxe, new ExceptionParam(Level.WARNING,CLASS_NAME
                        ,"Kunde inte skapa XML parser (SAXException)"));
        }
}
```

Listing 3.b: Aspect.

```
xmlHandler = XMLHandler.newHandler();
```

Listing 3.c: Refactored code.

```
try {
        xmlHandler = XMLHandler.newHandler();
} catch(Exception e) {
        throw new OPException(e);
}
```
Listing 3.d: Variant of the original code.

4 Implicit handling of non existing record elements

```
public synchronized TraceRecord getLastRecord() {
	TraceRecord[] records = _list.toArray(new TraceRecord[]{});
	if ( 0  < records.length ) {
		return records[records.length-1];
	}
	return null;
}

public synchronized TraceRecord getFirstRecord() {
	TraceRecord[] records = _list.toArray(new TraceRecord[]{});
	if ( 0  < records.length ) {
		return records[0];
	}
	return null;
}
```

Listing 4.a: Original code.

```
package com.aspects;
import com.log.gui.*;

public aspect GetRecord {
	pointcut getRecordData():
		execution(* get*Record()) &&
		this(TraceRecords);

	after() throwing(ArrayIndexOutOfBoundsException ex): getRecordData() {
		//return null
	}
}
```

Listing 4.b: Aspect.

```
public synchronized TraceRecord getLastRecord() {
	TraceRecord[] records = _list.toArray(new TraceRecord[]{});
	return records[records.length-1];
}

public synchronized TraceRecord getFirstRecord() {
	TraceRecord[] records = _list.toArray(new TraceRecord[]{});
	return records[0];
}
```

Listing 4.c: Refactored code.

5 Aspect-oriented Singleton-pattern

```
private Config config = null;

public static Config getInstance() throws OPException{
        if (config == null){
                config = new Config();
        }
        return Config;
}
```

Listing 5.a: Original code.

```
package com.aspects;
import java.util.Hashtable;

public aspect GetInstance {
        private Hashtable instances = new Hashtable();

        pointcut getInstance():
                execution(static * getInstance())
                && within(com..*)
                && !within(com.database..*);

        Object around(): getInstance() {
                Class instance = thisJoinPoint.getSignature().getDeclaringType();
                synchronized(instances) {
                        if(instances.get(instance) == null) {
                                //  Proceed  only  if  we  haven't  already  created  an
instance
                                instances.put(instance, proceed());
                        }
                }
                return (Object) instances.get(instance);
        }
}
```

Listing 5.b: Aspect.

```
public static Config getInstance() throws OPException{
        return Config;
}
```

Listing 5.c: Refactored code.

6 Disallowing arguments with a null value

```
public static synchronized String toHex(byte[] b, int len) {
       if(b == null) {
              return "null";
       }
       …

public static synchronized String toHex(short[] s) {
       if(s == null) {
              return "null";
       }
       …

public static synchronized String toHex(char[] c) {
       if(c == null) {
              return "null";
       }
       …


…
```

Listing 6.a: Original code.

```
package com.aspects;
import com.util.Hex;

public aspect ToHex {
       pointcut toHex(Object obj):
              execution(String Hex+.toHex(*[],..))
              && args(obj, ..);

       String around(Object obj): toHex(obj) {
              if(obj == null)
                     return "null";
              else
                     return proceed(obj);
       }
}
```

Listing 6.b: Aspect.

```
public static synchronized String toHex(byte[] b, int len) {
       …
public static synchronized String toHex(short[] s) {
       …
public static synchronized String toHex(char[] c) {
       …
…
```

Listing 6.c: Refactored code.

7 Ensuring that directories are created

```
File file = new java.io.File(logPath + SERVER_FILE);
file.mkdirs();
file = new java.io.File(logPath + CLIENT_FILE);
file.mkdirs();
file = new java.io.File(logPath + DATABASE_FILE);
file.mkdirs();
```

Listing 7.a: Original code.

```
package com.aspects;
import java.io.File;

public aspect NewFile {

        pointcut newFile(String path):
                call(File.new(String))
                && !within(com.aspects..*)
                && args(path);

        after(String path) returning: newFile(path) {
                File file = new File(path);
                file.mkdirs();
        }
}
```

Listing 7.b: Aspect.

```
File file = new java.io.File(logPath + SERVER_FILE);
file = new java.io.File(logPath + CLIENT_FILE);
file = new java.io.File(logPath + DATABASE_FILE);
```

Listing 7.c: Refactored code.

8 Implicit formatting of a new DBHandler object

```
DBHandler serverHandler = new DBHandler();
serverHandler.setFormatter(new DBFormatter());
serverHandler.setLevel(dbLevel);
opLogger.addHandler(serverHandler, SERVER_LOGGER);

DBHandler clientHandler = new DBHandler();
clientHandler.setFormatter(new DBFormatter());
clientHandler.setLevel(dbLevel);
opLogger.addHandler(clientHandler, CLIENT_LOGGER);

DBHandler databaseHandler = new DBHandler();
databaseHandler.setFormatter(new DBFormatter());
databaseHandler.setLevel(dbLevel);
opLogger.addHandler(databaseHandler, DATABASE_LOGGER);
```

Listing 8.a: Original code.

```
package com.aspects;

import java.util.logging.Level;
import com.common.Config;
import com.log.DBHandler;
import com.log.DBFormatter;

public aspect FormatDBHandler {

        pointcut DBHandlerCall():
                call(DBHandler.new(..))
                && within(com..*)
                && !within(FormatDBHandler);

        DBHandler around(): DBHandlerCall() {
                Level dbLevel = null;
                try{
                        Config config = Config.getInstance();
                        dbLevel = Level.parse(config.getProperty(Config.DB_LOG));
                }
                catch (Exception opex){
                        //This should never happen.
                }

                DBHandler dbHandler = proceed();
                dbHandler.setFormatter(new DBFormatter());
                dbHandler.setLevel(dbLevel);
                return dbHandler;
        }
}
```

Listing 8.b: Aspect.

```
DBHandler serverHandler = new DBHandler();
DBHandler clientHandler = new DBHandler();
DBHandler databaseHandler = new DBHandler();

opLogger.addHandler(serverHandler, SERVER_LOGGER);
opLogger.addHandler(clientHandler, CLIENT_LOGGER);
opLogger.addHandler(databaseHandler, DATABASE_LOGGER);
```

Listing 8.c: Refactored code.

UNIVERSITY OF SKÖVDE

School of Humanities and Informatics
Final Year Project in Computer Science 20 p
D-level
Spring 2006

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Appendix B: Prepared questions and interview transcripts

Daniel Oskarsson

HS-IKI-MD-06-003

Table of contents

Disclaimer 1: Some details have intentionally been left out. Such details are names of individuals and systems. The interview was carried out in Swedish which is both the interviewers and interviewees natural language and has been translated afterwards. The intension of having the interview in both natural languages was to make sure that the interview was used at discussing the topic instead of understanding each other. When the interview was translated it was also rewritten a tiny bit to improve readability. However no details have changed! The interviewee has been able to verify that nothing has been missed in the translation and has had the opportunity to give feedback on the content. Questions marked with an asterix are prepared questions.

Disclaimer 2: The implementation of the aspect-oriented Singleton pattern has been changed since this interview since as it did not in fact work as it should. The final refactoring of the system contains an operation getInstance which always return a new object. The aspect also consists of an around construction which overrides the getInstance operation and where proceed is used to create a new object only if there is not already such an object within a hashtable that is defined within the aspect. The return value of proceed is both stored in the hashtable and returned as the result of the getInstance invocation (see Appendix A, listing 5b).

Prepared questions

These questions was prepared and tested before the interviews so that the open interview could be somewhat controlled. Note that perhaps note the exact questions were used but similar.

First some questions were asked to establish background information about the interviewees:

- ➢ Why don't you tell us a little about yourself, your system engineer, object-orientation, and java knowledge's, and how long you've worked at the company and which work assignments you've had?

Questions were then asked regarding aspect-orientation in general:

- ➢ Have you had the chance to read some information about aspect-orientation and aspect-oriented software development?
- ➢ What are you general thoughts about aspect-orientation?
- ➢ In which way to do think that aspect-orientation affects code complexity?

The interviewee was presented with material from the system along with changed made using aspect-orientation (Appendix B) and the following questions were asked related to that:

- ➢ For how long and in which ways have you been involved in developing the java server subject to this case study?
- ➢ If you take a few minutes and have a look at this code (the interview gets a paper transcript with original code and a refactored code including an aspect). I will introduce the code, the aspect and the refactored code, when I would like you to comment on this based on the following questions (I will help you remember these):
  - What are your general thoughts about these aspects?
  - In which way (in your opinion) has the refactoring affected code complexity?
  - Can you find both positive and negative things with the refactored code? What are those?
  - Would you have used this aspect? Please motivate your answer.
  - Do you think that the code would look different if the system was built using aspect-orientation from day one? Please motivate your answer.

Finally some questions about their view on aspect-orientations role in large and in their company in the future:

- ➢ There are quite a lot of research going on right know related to aspect-orientation, do you think that aspect-orientation have a future?
- ➢ Could aspect-orientation be a technique used within this company in, say five years? Please motivate your answer.
- ➢ Aspect-orientation is based on an implicit connection from the code to the aspect. What part does an IDE (Integrated Development Environment) play in your daily tasks and how does that relate to this implicit connection between the main code and the aspects?
- ➢ How would you feel if you got the assignment to build your next system using aspect-orientation?

Interviewee 1

Interview date: 2006-04-26. Length: approximately 1:30 hours.

*** Why don't you tell us a little about yourself, your system engineer, object-orientation, and Java knowledge's? How long you've worked at the company and which work assignments you've had?**

I've worked for the company in 11 years and done many different things such as geographical presentation of network elements, maps and symbol manipulation. It was in that project I first got in touch with Java it must have been around 1997, we used JDK 1.1 and it was the first project within the company where Java was used in an important project. Since then, I've worked with Java and C++, mostly Java. […]

The server is written in Java and the Clients in C++ because of an old GUI-inheritance. […] At on time, we canned the system and rewrote it because of new requirements. […]

I was thinking about what you said that you drove the first important project where Java was used, i.e. is something we have discussed if there are any directions from above which technology you should use and so, or was that something that you where interested in using?

We were interested in using Java, we thought it had many advantages for this project and also found a library for GIS (map handling) which was written in C but could easily be interfaced with Java. Most GIS products we evaluated didn't meet all requirements in that project. We were allowed to develop the project using both C++ and Java for a couple of months. Management was also interested in using Java but didn't trust it enough to use it exclusively so we did parallel development for a couple of months or two. When we saw that Java worked, the C++ line was canned. Java is good when targeting several platforms such as in this case both Sun OS, HP-UX and on Windows, that's one of the reasons that we wanted to introduce java.

This is interesting from this thesis point of view. How does usage of a new technique or PL (programming language) get started at a company such as this? How could the usage of say AO, which isn't widely used, get started?

AO sounds interesting; much depends on tools and a company's view of freeware. It also has to be known to the public and there must be a project where usage of AO fits and were the members has the will to drive through usage AO or the freedom to choose themselves. If the project falls well out, it will spread. There are both positive and negative things. I am personally a bit suspicious.

*** Have you had any chance to read some information about aspect-orientation and aspect-oriented software development?**

I've read most parts of you introduction, background and problem description.

*** What are your general thoughts about aspect-orientation?**

What I see as the big problem is that you can have things done which you aren't aware of. You have to have sensible name conventions on operations and variables.

You have to be a bit clever on an early phase. I can imagine that introducing AO in a later phase may be hard because of this.

It wasn’t that easy.

I can understand that.

Which positive things can you see with AO?

The refinement is that you get less code, but at the same time I, personally, think that it’s less obvious, it becomes a problem reading code. To introduce an inexperienced developer in a new project becomes hard since he or she don’t know the system fully and might wonder when things happen. If you have things that should be changed, say that you have some platform dependent things then you can easily change it at one position in the code and get a whole new behavior which affects the whole system. I guess that’s the large gain.

In which way do you think that aspect-orientation affects code complexity?

I guess I don’t see any advantage, but it depends on what you are used to. I am an old assembler programmer so I am used to read registers and such. That’s why I had trouble getting used to new IDEs, even if I nowadays run Eclipse, because they do things I haven’t asked for. It may also be hard to see minor differences, I see a lot of occasions when operations does *almost* the same things and at such points it may be tricky to see which operation are referenced by an aspect. If you have a tool which shows this, it’s alright. But if you are on the field or in a lab without your toolbox (IDE) you may have trouble in understanding the code. In many projects types you do, perhaps not as much today as before but still, run a bit simpler tools. Today you can put any IDE in a laptop which you can carry with you.

But the basic idea with AO is really interesting. It’s only a question of when to use it and when not to use it. If you start use it you may start uses it on everything. Then you discover that that wasn’t good and almost stop using it entirely before you figure out just the proper usage amount. I think it takes a lot of experience to do this wisely.

I think so to. I’ve noticed that I miss that when refactoring the Java server.

*** For how long and in which ways have you been involved in developing the Java server subject to this case study?**

The server I’ve been with from the beginning. I did the first sketches when we ought to introduce database support, you loose perspective but it must have been a couple of three years ago. I and another gentleman (which later dropped) did a basic architecture for the server. I worked full time on the Java server for more than a year. There also were 2-3 other (different) people working on the server during the project.

How many people have been involved developing the whole project?

G. and A. worked full time during the same period with the database. Five to six persons worked with the clients, which took the longest and had the largest variation of developers. Developing GUI always take very much time. The entire project ended for about a year ago. There was a maintenance release in September 2005 were mostly the clients was changed, but there was almost exactly a year ago we stopped the main development and the project went into maintenance.

*** If you take a few minutes and have a look at this code (the interview gets a paper transcript with original code and a refactored code including an aspect). I will introduce the code, the aspect and the refactored code, when I would like you to comment on this based on the following questions (I will help you remember these):**

- **What are your general thoughts about these aspects?**
- **In which way (in your opinion) has the refactoring affected code complexity?**
- **Can you find both positive and negative things with the refactored code? What are those?**
- **Would you have used this aspect? Please motivate your answer.**
- **Do you think that the code would look different if the system was built using aspect-orientation from day one? Please motivate your answer.**

This ought to be interesting.

1. Modularization of try-catch-clauses

In this advice there is a pointcut matching all invocation to operations which names start with the word get, may return anything, have any parameter list and is a part of the class *CommandMessage* (listing 1a). I have declared an exception soft; otherwise the compiler doesn't allow the handling of an exception being done in an aspect. I have written an around advice, which is executed instead of the join point until and if the keyword *proceed* is used. In this advice there is a try-catch-block, and we have variable which retrieves the return value from proceed and if that statement fails then an exception will be thrown (listing 1b) which we, as done in the original code, ignore and just return result which is initialized to null. This means that we can write the original code using less try-catch-blocks (listing 1c).

This is clearly an improvement of the readability of the code in this case. This seems to be an ideal usage of AO.

I have found a lot of locations in the code where the statements look almost the same but differ on some point. There is a possibility to include and exclude all kind of things but that means that the pointcuts become larger.

Is it possible to transform a parameter into the aspect if you whish to handle the fault with, say different error codes?

Sure. […]

I must say that in this implementation it was clearly an improvement of the readability.

2. Implicit handling of non-existing attributes

I have made some more refactoring, some good other probably less good.

[...] This one is a bit trickier if you look at the use of it.

It depends on how often this control occurs.

One of the tricky parts is that if I had problems with this code, I would start to look for a null pointer here unaware of the fact that an aspect affects the code.

If you use an IDE like Eclipse and AJDT will help you annotate code that is affected by an aspect. I was thinking that we could discuss the role of IDEs later in this interview.

[…]

For this type of problems you have traditionally have worked with macros. So this replaces usage of macros, in good and bad. You must have some kind of annotation showing that something affects this code otherwise we have a Bill Gates-syndrome where a lot of things happen which you are unaware of.

[…]

3. Softened exception handling

Here we have the invocation of an operation named *newHandler* that may throw two different types of exceptions.

That piece of code is somewhere in the response handler.

Actually this piece of code is represented at a lot of locations in the code. There are also some invocations to the *newHandler* operation that instead of catching two exceptions only catch one exception of the type *Exception*. Is that deliberately?

In most cases no, it should not be that way, and in this particular case there is a fault. We catch an exception of a certain type but don't have two different error messages, which it probably should have.

In these cases AO can help, there is research about how exception handling can be done with AO, and I think that error handling is one of the areas where AO can really help.

I can imagine that one of the strengths of AO is to handle error handling.

In the advice you don't have access to private variables and such, but if you use imports you can use those classes and create new objects that you in this case use as argument in the *OPException*. In this example I also use a keyword *thisJoinPoint* to get context information about the join point, in this case the name of the class. The refactored main code becomes one single row. What do you thing about this aspect?

This one is nice. There are a lot of classes like this, where you have to surround the invocation with a try-catch-block, and in the server we take care of exceptions that we know what they are and re-throw them as *OPExceptions*. Other exceptions are possibly taken care of in the bottom because if they are thrown the system is really instable.

For classes created in a lot of places this way is much delicate. Many classes probably have the same exceptions that they may throw so you could collect those into a single aspect, besides that they are implemented as a factory so they can easily be centralized.

The good use of object-orientation has made it hard to find problems that are truly crosscutting. There are of course some crosscutting problems, but not all of the examples that I will show you are truly crosscutting.

Sure, by using factory classes you can eliminate these kinds of problems. This one is not bad, and then if you would redo it from the beginning you will of course name *newHandler* so that you understand that it's managed in someway, like *newCheckedHandler* or something.

You have actually found some uses. I though it would be hard to find when you hadn't the architecture clear.

It wasn't easy but the hard thing was really, not knowing AspectJ.

4. Implicit handling of non-existing record elements

These are operations that return the first and last element from a list. The lists are checked before return so that they contain elements, if not null is returned.

This was a new construction. Right the pointcut matches all operations that return elements from the list.

Yes, there could also have been a get-record-by-index.

Then the condition would have been a bit more advanced and the aspect would not have been enough.

The original code returns null. I'm not sure that this is the correct way to solve this but the thought is to take care of any *ArrayIndexOutOfBoundsException* **that the operation might throw and do nothing, which should make the operation return null.**

That's something we can do in AO, write advices that is executed when something has executed well, when an exception is received and you can also write advices that don't care if it executed well or threw an exception. Like in this case where we imitate the original code by catching an exception and then don't handle it. Then it's the readability.

You can certainly discuss that in this case. Well, it's not worse. It has not become better, or worse. […] It's not crystal clear but it's not nasty either. I really have no opinion about this one.

They are not all supposed to be crystal clear.

The construction in itself is good, but the usability is doubtful.

Perhaps you could say that you should be careful on how you apply AO and look for the crystal clear cases, preferably already in the design phase.

These operations are from the beginning a bit hard to understand if you don't know what's outside them.

Yes, I must confess that I didn't understand the first statement in these operations entirely.

I must also look at the rest of the code to be able to know exactly what they do. Those statements are a bit tricky. I don't remember how the *TraceRecord* class is defined, I haven't written that part.

[…]

5. Aspect-oriented Singleton pattern

When it comes to the GoF-patterns there are AO-versions of all patterns. In the AO-version of singleton the creation of a singleton object is done in the same way as the creation of all other objects using the keyword *new*. In this refactoring I have kept the *getInstance* operation.

In this case the aspect creates a new object if the value off the instance variable is null. A drawback is that it requires a uniform naming convention so that the aspect is able to read the value. What do you think about this construction where the operation just return instance and the aspect create the object if it doesn't exist?

If you have a tool that helps you see that there is something that happens then this is ok.

This is a crosscutting concern; there are a lot of singletons and invocations of *getInstance* in different locations in the code.

How does it work if you have another class which makes a getInstance, the object that is returned must be of the correct type?

Then we use the *thisJoinPoint* keyword in AspectJ to retrieve information about the declaring type and create an object dynamically.

Ok, that was clever. That means that it works on all getInstance operations if you have the same variable name? How does this differ from the other version?

They have created an around advice that takes over the constructor and skipped the usage of a separate operation to create singletons.

Ok, that's not pretty; it breaks how old programmers think about what is a singleton and what's not. Of course that is what you are used to. Might be hard to see when you read the code. […] You have to watch out so you don't get yourself in a situation where the names get to complicated. I don't see any risk of that happening here, but I know people who could make it a risk.

Actually it sounds a bit clever; yes I can see the point of overloading the constructor. You could solve this really neat if you could have multiple inheritance and have *Singleton* as a super class. You can do a lot of fun things with this!

There is potential. There are those who say that this is a whole new way to think. When you look at it at first you don't see that.

You definitely have to get the thinking in already in the design phase if you are to use this in a sensible way, so that it gives as much as possible.

AO (aspect-orientation) was from the beginning (and still are by many) called AOP (aspect-oriented programming), I favor AOSD (aspect-oriented software development) as a general term and to talk about AOD (aspect-oriented design) a term which I have not seen anywhere, which is a bit strange.

Yes, OOD is accepted, if a bit more special. I mean you can do OO even without an object-oriented language. I for many years stubbornly maintained that OOD was just a name of good programming practices; I have looked at my old assembler code and found that it is in some sense object-oriented. If you developed good looking programs they became more or less object-oriented.

You can se similarities here where you actually have aspect-oriented program; the difference is that the connection is implicit instead of explicit.

There is research about AOD; unfortunately they have chosen to refer to that also as AOSD (aspect-oriented software design).

These languages are they commercial? Like AspectJ.

Do you mean propriety? No they are free.

So you can download it and play with it at home in Eclipse?

Yes. […]

6. Disallowing arguments with a null value

We get into the *Util* package.

A typical situation where you want to convert input in this case to a hex value.

We have a situation (listing 6a) where the argument may be null. If that would happen the original code returns a string with the value null. I have written a pointcut which matches operations named *toHex* in the *Hex* class and any sub classes which takes an *Array* as argument and return a *String* (listing 6b). The around advice does the checking, and if the argument is not null the operation is allowed to proceed.

This is pretty crystal clear.

Yes, we got rid of all error handling and the content of the refactored operations are the purpose of the operation (listing 6c).

I think it's pretty crystal clear.

This is an example of a non *crosscutting* concern, but it may still be worth applying AO? There are multiple operations.

Yes. It may be large inheritance hierarchies as well. Besides this is pretty far down in the *Util* class so they are not much read after they have been tested.

Is this *Util* package perhaps something that are reused?

It should be. Unfortunately there is no instance here that collects and maintains reusable code. There are some reuse projects but most times there are a few projects who have noticed that they have a similar code base and started to work together. It's not very well organized, unfortunately.

Perhaps you could exchange aspects? If they are well written then perhaps you could apply one aspect from a project on to another project.

But then you really have to make sure that the naming conventions and such match.

Although you can match pointcuts on other things than names, e.g. types.

When you read and write a variable.

Really any part of the execution of a java program can be caught we AO.

Yes, I can imagine that if you use it in the design phase AO can be pretty strong, that as soon as something changes different variables or classes, certain controls are executed or a message is sent.

If there are exceptions thrown they are logged into a database. You can catch without affecting.

Error handling is one of the areas where I see that AO could really help. Error handling is hard and it's easy to forget, so if you can generalize that so that it's implicit, that would be great.

I think that error handling as it is today strongly contribute to the code complexity.

Yes, there are operations where you actually have to look for the executing statement. The rest is just conditions and error handling. It would be enormously pleasant to be able to get rid of such things. It would be much easier to read.

7. Ensuring that directories are created

Here is the second last, very short. What I have done here is to match the creation of a new *File* object (listing 7b). Each time a new object is created the operation *mkdirs* is executed to make sure that all directories exist (listing 7a).

That one (listing 7c) is also pretty crystal clear.

Yup, since the system is built using god OO there is only file handling in one location in the code, although you could imagine that file handling was a crosscutting concern where you always would like to create the directories.

Yes, it works its fine.

8. Implicit formatting of a new DBHandler object

The last example is a long sequence of statements there are some things that happen but I think that there are some things to improve here (listing 8a). All objects of the type *DBHandler* are formatted in a certain way. Can't you solve this with OO?

Yes, the question is why the *DBHandlers* looks like they do… There is a possibility to change formatter and level, that's why they are available as separate operations and not in the constructor. And then we always put such a handler on the logger.

It was one of those places where I thought that you could use OO to ease the complexity, as you say in the constructor format the object. I refactored it using AO to show how AO can solve this kind of problem (listing 8b).

You could perhaps think that the object also puts itself into the OP-logger (listing 8c)?

Sure, it has with the readability to do.

Couldn't you also think that the *addHandler* actually started the creation of the *DBHandler*? If so it would be more elegant if you think of it in terms of readability, although it wouldn't decrease the number of statements, so the gain is doubtful.

The aspect is larger the most of those we have seen before, and a bit more complex. The reason for that is that you don't have access to the arguments that are used to format the *DBHandler*. That's why there are a lot of imports in the aspect.

Sure, there is the whole code on such a block. How do you get the parameters into the advice? This one was a bit harder to grasp.

Yes, as with the others, it might also contain some minor fault. We don't get any parameters into the advice. Instead we create a new *Config* object and other data we need. [...]

Well, the aspect is a bit more complex to grasp, but it leaves a neat refactored main code, I got to agree on that. Then of course it has to occur a few times to be worth writing the aspect.

Maybe it's not worth it on just these three?

Perhaps you need a few more.

If you wish to do the implementation complicated you could think about inheritance. An aspect is just like a class in many ways and it fully supports things like inheritance among aspects. So below the code in the example there is a file handler which also is formatted in a certain way. An aspect can be built that has several pointcut and advices but which utilizes a common operation. You can build very powerful things on the side, where you take advantage of reuse internally within aspect.

Yes, it can be worth studying a bit to of curiosity. I can understand why you think this is bit fun.

What is your general opinion about AO? Does it seem interesting?

Yes it is interesting; if it is introduced in the design phase I think that there is a lot of strength in this. But it probably requires quite a lot of experience to do this right. Of course that the case with all design. You have to start carefully, but it has potential.

*** How has code complexity been affected by introducing AO?**

It has its advantages and disadvantages. You get two places to read the code which can be hard. I could also suspect that it would be large profit to have some kind of an aspect-library which you could work with internally so that you had built some aspects which could be used in several projects and that people were familiar with those aspects and knew why they where there and how to use them.

To construct aspects it is required that there are some concerns which crosscut the system so that the aspect is worth writing, unless you expect changes and whishes to centralize things for that particularly reason. Centralization can also be done in OO although this is a more neat way. There are some things to dissolve. It is what you are used to I'm sure, it's probably difficult to work in this kind of code the first time before you have gotten used to that there are aspects that affect the main code.

I was thinking about reuse. If you put your aspects in their own packages then you can easily built aspect libraries which can be included into a project.

You said that there wasn't a whole lot of AO usage yet, just a few companies?

Yes. I haven't done a lot of research on this but you don't hear of it anywhere. All material I see is research material.

When did all this start?

The first paper was release by a Xerox research group in 1997.

Ok. Xerox has been first with many things. Ten years is about what it usually takes for things like this to reach a certain maturity. The first three or four years, it's definitively only discussed within research.

A simple thing you can do to measure the public knowledge about such things is to search for books about the subject. There are very few about AO.

If you get to choose a positive and a negative thing with the refactored code, regarding code complexity, what would those be?

What I spontaneously see is the readability. You have the possibility to gather the error handling, put it somewhere where you don't have to look at it and instead you get a larger focus on what the operations do. That's the most positive of these uses.

The most negative is that I don't know how to use the language. If I knew AspectJ as well as I know Java these aspects would probably not be a problem. Besides that the most negative effect is that things happen which you are not aware of (referred to as the Bill Gates-syndrome) and that might be hard to realize.

*** Aspect-orientation is based on an implicit connection from the code to the aspect. What part does and IDE (Integrated Development Environment) play in your daily tasks and how does that relate to this implicit connection between the main code and the aspects?**

Then we get into the usage of IDEs which I thought that we could discuss for a while. The fact that code is evident in two places, available in two files which cooperate to solve the same problem.

If you use a tool such as *Ultraedit* it might be a bit strenuous to jump between two code files which *belong* to the same piece of code. You need some tool which supports the jumping between those, shows the aspects or in some other way solves this.

There are IDEs which support it. AspectJ is available for Eclipse and other IDEs as well. In Eclipse it is very well implemented get hyperlinks and different types of icons for different advices and such things. You get feedback from the IDE which join points this pointcut matches and so on, so if you use a modern IDE such as Eclipse with AJDT (AspectJ for Eclipse) then you get quite a lot of support. But of course there are occasions where you don't have that possibility.

Yes it becomes perhaps a bit more troublesome. Although, when you encounter those occasions your probably already have a good knowledge about the system. When you are in the beginning of the development phase you often have a strong tool support, and if AO becomes more available to the public, tools will spread fast.

If we have a good IDE do you see any disadvantages with AO?

Not really. It's probably just an adjustment period. It should be like that anyhow.

One of the core issues discussed is the implicit connection, if tool support is enough to overcome that?

I think so. Of course it requires that you have a good environment, which also makes it hard to introduce it into existing projects where there already exists a working environment. […] If you get a working environment which includes AO you will probably get use to it relatively quickly.

*** There are quite a lot of research going on right know related to aspect-orientation. Do you think that aspect-orientation have a future?**

Does AO have a future? How does it look in say five years? Please motivate your answer.

Either it will become quite large or it will die out. There are also a possibility that standard compilers starts to include similar functionality and that AO stops to exist as a phenomenon and terminology of its own and that the technique is integrated into the languages in future revisions. The technique, I think, will remain in different levels. [...]

*** What do you think it would take for AO to be used at this company?**

I'm not sure. The basic requirement is that people starts to get knowledge about AO. Today developers don't know what it is. You have to get some enthusiast to start telling his or her colleagues how clever it is and that someone responsible for the design in a project where it fits knows about AO. Then it might start to move. It's probably much of a coincident how fast it might integrate within a company such as this, since there is no part of the company which evaluates techniques and tools and gives recommendations. There really should be. […]

If AO would come as a standard option within Java, ADA, C++ or C#?

If it comes in C# it will be probably be started to be used quite fast. […] If it comes in dotnet it will soon be used, if it comes in a Java release as a little extra package it will probably soon be used as well. […] If it starts to be built in into the Java compiler, and it would not surprise me if small functions from AO was included, then it would spread fast I guess.

*** How would you feel if you go the assignment to use aspect-orientation in your next project?**

That would be fun, although, I would like to have a course to learn the language first. It's always fun to use new techniques, as long as they don't seem stupid, and that's not the case with AO. I would be curious to start using it and see how good it really is.

Thank you for your participation!

Interviewee 2

Interview date: 2006-05-05. Length: approximately 1:30 hours.

*** Why don’t you tell us a little about yourself, your system engineer, object-orientation, and Java knowledge’s? How long you’ve worked at the company and which work assignments you’ve had?**

I am mainly a software developer, which is my main occupation. I have been doing some UML-modeling and such. Some Java but mainly ADA and Oracle stuff such as PL-SQL. I’ve only used Java in one project.

But you have good knowledge of object orientation and such?

Yes, I think so.

Do you have any software developer education?

No not really. I am an electric engineer, of course there were some software development courses but it was not my major.

How long have you worked at the company?

For five years.

*** Have you had any chance to read some information about aspect-orientation and aspect-oriented software development?**

Yes I’ve done that, I have some knowledge of join points, pointcuts and such.

*** What are your general thoughts about aspect-orientation?**

It looks very elegant, the syntax. It probably has big opportunities to simplify some parts of the development and to do make code less complex and cleaner i.e. you refine the code to what the main purpose is. That the positive side.

The negative side is that it can be hard to follow what really happens since it is implicit. Because of that I think that it is important to have tool support, otherwise you can’t handle that, perhaps in when developing but absolutely not in maintenance. It is practically impossible for another person to follow the code if you don’t have a tool.

It reminds me of the database world, a trigger on a table which is executed because of a data manipulation you do. That also, is something which isn’t visible in code, what you invoke you have to keep track of that in some other way.

But a trigger doesn’t affect the original code in the same way. It only affects the data you want to extract?

Yes, you can say that. But something gets invoked at certain conditions. Although in that case it’s still easier because you can see that condition. When it comes to an aspect you have to go into the aspect to see when it gets invoked and that makes it harder to follow.

*** For how long and in which ways have you been involved in developing the Java server subject to this case study?**

I haven't been involved in the code development of the server itself. I was involved in the development of applications that worked with the server. I also was responsible for creating some models and documents which describes the whole system. I also was involved in the design at a bit higher level, including for the server, but not on a detailed level.

So you have worked on the database that the server communicates with?

Yes, I worked with the database and on that part; I worked with details and implementation.

So you have seen code from the server before?

Oh yes.

*** If you take a few minutes and have a look at this code (the interview gets a paper transcript with original code and a refactored code including an aspect). I will introduce the code, the aspect and the refactored code, when I would like you to comment on this based on the following questions (I will help you remember these):**

- **What are your general thoughts about these aspects?**
- **In which way (in your opinion) has the refactoring affected code complexity?**
- **Can you find both positive and negative things with the refactored code? What are those?**
- **Would you have used this aspect? Please motivate your answer.**
- **Do you think that the code would look different if the system was built using aspect-orientation from day one? Please motivate your answer.**

I've brought some code parts which I've refactored to use AO. I thought that we could have a look at them together and discuss each aspect regarding complexity to discuss whether AO makes the code more or less easy to read, and other issues that come up.

1. Modularization of try-catch-clauses

Let's start with this example. This code is directly from the original code for the Java server (listing 1a). We have an object of the type TextMessage. All invocations to this object must be surrounded with a try-catch-block because they may throw. I have made a pointcut that matches all invocations to operations that return a String and an around-advice that is executed instead of the invocation (listing 1b). The advice contains a try-catch-block in which we use the keyword *proceed* within the try-catch-block to get original invocation into the advice. If an exception is thrown we do nothing just like in the original code. This gives a refactored code (listing 1c).

It's absolutely easier to read since what you see is that is the functionality of the operation and not the other things that really disturb the reading. So absolutely, it's more compact and focused on what's the meaning of the function and the important is what's left while you don't have to see all these try-catch-blocks.

My thought here is that in all these catch-blocks you do nothing, they're just catching the exception and move on. Another thought that I have is about the last try-catch-block where you do two invocations in the same block, perhaps there is a meaning with that. If the first invocation throws then the second one should not be done. You miss that by solving the problem this way.

I’ve refactored the code so that is handles that. If you check this if-statement, if the return-value is null then the second statement will never execute. So the semantic is there. But you definitely have to look out for such hidden semantic when refactoring into aspects.

Yes. There might be a certain function in some code being written in a certain kind of way and then you have to catch that semantic somehow. This means that it isn’t as neat, at the end. Then there is always the problem, if you wish to do something in one of the catch-blocks. Then the aspect must be changed not to catch that particularly invocation, or is there any other possibility to disable the aspect from the main code? You must get around that, or you have to have a smart name convention or you exclude the operation in the pointcut. If get too many of these things then you have, as I see it, lost the point of AO. If there are too many special cases then you lift the complexity away from the main code and into the aspect, which gives you no gain, instead the code just gets more complicated and hard to grasp for the user.

So you have to be careful with on what and how you apply AO?

Yes I believe so.

If the system was implemented with and aspect oriented language such as AspectJ, would you create this aspect?

I would probably rewrite the TextMessage class to capture these things there instead. But there can be so that there is another location where there is a point in having these exceptions and in that case you have to write new operations in the TextMessage class, one with and one without exception handling.

But on the other hand, perhaps I don’t have control of TextMessage which might be a self defined class or it might come from another package like a third-party package. It may very well be so that I can’t affect that class and if so this is a good way of solving this.

I think that you are right, TextMessage are from a third-party package.

I can guess that it is from import *javax.jms.*;*. If so we don’t have control of that class and then I must say that this is a smart way of solving it, an appealing way. It depends on which possibilities you have to affect. This might be a usage of aspects which I have not thought about before. If you can’t affect a class, then you can do this and have a less complex code. I think it’s a good usage.

2. Implicit handling of non-existing attributes

Here we have a simple operation which returns some attributes by returning the last element from a specific vector. If there is not element in the vector the operation returns a new object of the type *AttributesImpl* (listing 2a). So what I have done is that I’ve written an aspect which does this control (listing 2b), which gives us a refactored code (listing 2c) where the only statement that exists tries to return the last element from the vector. The control and new object creation is taken care of by the aspect.

Ok, we have another *around* which takes over.

Here we use a try-catch-block also in this example. That is because advices don’t have access to private variables and such. So if you want access to those you have to send them to the advice through the pointcut. So the alternative to sending the vector into the

advice is to try to execute the statement and catch any exception that might be thrown if the vector would be empty.

We could have send the vector and do the same control as in the original code, but that would have created a more complex pointcut since you have to define which values you would like to send to you advice. You are welcome to comment on both solutions.

I don’t think that this provide anything particularly on the contrary things get more complicated. I also think that in this case, besides that the aspect may get a slower execution time then in the original code we also get a try-catch-block around the statement which gives a slower execution compared to using an if-statement.

So if you would have created an aspect you would have sent the vector to the advice and used an if-statement, though you would not have used an aspect at all?

Right, I would probably not have created any aspect. I think that would be unnecessary. This is such a trivial operation. There is a certain value in displaying what’s happening and besides you get the fastest available execution time with the original code. So no, I would not have created an aspect at all.

3. Softened exception handling

In this original code (listing 3a) we create a new object of the type *XMLHandler* by invoking the operation *newHandler* on the *XMLHandler* object. This invocation may also throw. It may throw several exceptions so we have two catch-blocks to each try-block and even if that is not the case in this code, we could imagine that they create two different exceptions.

This is truly a crosscutting concern, since the creation of an *xmlHandler* exists in ten different places in different packages and then *often* looks like that (listing 3a). So what I have done is to write an aspect with two advices (listing 3b) that handle exceptions. What’s left is a single line (listing 3c) where we try to create the object and we once again are let off from the fault handling.

Looks good, very appealing. Since it is spread in the code it’s also more justified to write an aspect. It also makes the code easier to maintain, if you would get a change in the behavior of the *XMLHandler* like a new exception that could be thrown you can easily handle that by changing at only one place in the code. That is very appealing. It doesn’t add any new element into the code which could slow it down.

Once again we have chosen not to send in any arguments in to the advice but instead use the keyword *thisJoinPoint* in AspectJ where we can get, in this case, the name of the class which is an argument used when creating the new exception that is created when receiving an exception from the operation.

One thing that I though about am if you would like to have the actual line from the source file into the logging, I’m not sure if it is possible to solve, perhaps it can, there is perhaps functions for that to, or you have to send it as an argument somehow. Sometimes you want a error logging which not only tells you about the fault that happened but also where the fault where, so that you can go in and look at that particularly statement in your code.

In Visual C++ for example there are macros which tell you which line and file the fault is might be at. If you would like to do that in this case, maybe that is hard, I don’t know?

It depends on how the aspect is woven into the original code, there are different weaving techniques. One way is in lining, which I would guess is used here, that the advice is

inlined by the compiler at all join points that is matched in the pointcut. Then you would get the exact same code as in the original code.

But then the line numbers doesn't match since you got additional code.

Yes, you are right about that. […]

Ok, so that was a bit off track, a part from that I think that this looks like a good way to use aspect-orientation.

Ok. I have one more thing I would like to discuss. I said that the creation of an *xmlHandler often* looks in a particular way. In some parts of the code we only have one single catch-block which catches and exception of the type *Exception* instead of the two subtypes *ParserConfigurationException* and *SAXException*. When I interviewed the developer of the Java server we found no reason to why there would look any different on any of the places. So what we have is places where the code should look the same but don't. With AO you get this uniform handling of code, whether you want it or not.

Yes and that is often an advantage. Sometimes you want another handling fore some reason, but in most cases you want the same handling of certain events each time that is get a system which reacts in a certain way at certain events. A likely cause to the different implementation of creating a new *XMLHandler* is that one way of implementing is was written before you decided how you should do it and then you forgot to change it. That is a very likely cause and then you have a big advantage when using an aspect and only have to change the implementation in one place. So in most cases it's probably an advantage to have it this way. I thought this was a good example.

4. Implicit handling of non-existing record elements

This looks a bit like a previous example. We have two operations, one that gets the first element and one that gets the last element from a list (listing 4a). In the original code there is an if-statement checking the length of the list, and if the list doesn't contain any elements the operation returns null.

I have written an aspect with a pointcut that matches the execution of both operations to an after throwing advice (listing 4b) which is executed if the join point throws. This gives us a behavior where we only need to try to return something from the list (listing 4c) and if an *ArrayIndexOutOfBoundsException* is thrown we do nothing and the operation should return null.

The question is why the original implementation is not done using a try-catch-block. You could have created a catch for the *ArrayIndexOutOfBoundsException* and made a return null.

Yes but here we have access to the list and can use an if-statement which means that we don't need to throw an exception.

Right, but you could have done the try-catch-block in the original code also.

Yes. Here is two operations but we could have more operations which refers to different elements, e.g. one that where you can specify index as an argument. Then the aspect would affect also that operation, even if the additional operation is created after the aspect.

Well, what can I say? I am a bit split. It's a neat way of catching errors and as you say, if you add another operation which is named get*Record that operation is also automatically affected by the aspect. That is pretty clever. My thoughts are not really negative but I wonder why the *_list.toArray(..)* is not modularized as well? […]

I though about modularizing that statement as well, but choose not to, with the motivation that the refactored code would get very hard to understand.

Yes, you are probably right, that would have been very odd. If there are only a few *get* operations in a class, sure it's neat and so, but it isn't very complex code to start with. If on the other hand, similar problems is something that exist in several classes then it can be very good to solve that with an aspect. But as long as it's only a single class it probably isn't worth it. In this case it gets more complex with an aspect than without.

We could have written an aspect which affects the entire system, which follows a certain way which you in the design-phase had established, e.g. that operations which tries to access an index which doesn't exist always return null.

Then you would have gained much more.

The problem is when you refactor it's hard to take those decisions afterwards.

Then it is so that, if you would like to really take advantage of AO you should think it through and follow some rules which you have created. But if you really do that, which AO really supports, AO really gives you something back, if you do that. Then there is a lot to gain. On the other hand that is not unique for AO, the better design, the better implementation, even if you in the AO case might have even more to earn. AO seems to support working consistent, and that's a good thing.

A consistent design makes AO give very much positive and in the same time AO also support, what we talked about before, that something is handled in a new way which doesn't apply on all other locations that probably depends on that you have missed something. Consistent development is supported and the consistent work is simplified because of AO. I believe that you have more to gain if you apply AO from the beginning, then to have to work in a system where the design is already done not using AO at all.

It wasn't that easy finding crosscutting concerns in the system afterwards.

No that was another thought, if you have good object-oriented design then there might be hard to make the system less complex. Of course some things can be refined, some things you can't do with object-orientation like in the previous example, which I thought that was a good use of aspect-orientation.

Yeah, that's a typical CC which you despite god OO can't get rid of.

Exactly, and you can fix that afterwards, but some things you could have done better if you had AO from the beginning and I think that the previous example shows that.

5. Aspect-oriented Singleton pattern

Here we have the singleton pattern (listing 5a). When it comes to singleton and all other GoF patterns, AO-versions has been created in earlier research. In that research they chose to use the constructor to create singleton, instead of using the differential operation *getInstance*. This means that you have to name your singleton classes in a special way or at some other way with help of comments or something, keep track of which classes is a singleton or not. You can choose whether you wish to be informed about which classes are singletons or not. Since I refactored code instead of writing new code, I chose not to use the constructor, but to keep the method *getInstance*. It consists of a simple check if a variable is null and if it is it returns a new instance of the class.

What you have to do if you want to create an AO version, then you have to change the class dependant variable, like *config* which was unique without have a reason for that

other than refereeing to the class which it was defined in, to a uniform name like e.g. *instance* instead. I've done a pointcut which matches all *getInstance* operations except those who are in the database package since those works a bit different and you should keep the original functionality. Then we send the value off the variable instance into the advice and make this control which is originally done in each *getInstance* operation. We use an AspectJ keyword to get which type it should be and can dynamically create a new instance of the class if the variable is null. So what we actually do is that we return the variable from *getInstance* and then the advice after comes in and controls if it's null and then replaces it with a new object reference. What's left in the *getInstance* operation is the statement *return instance*. You are welcomed to comment on both this way of writing the aspect and the way where you can use the constructor instead.

There is no code for the constructor version?

No but that's just like a regular new. Instead of invoking *getInstance* you do a new.

Which means that you always create a new object?

No you don't have to. That control can be done in the constructor.

Ok, and that is solved via an aspect in that case?

Yes.

So you use the pointcut to decide which classes that should be a singleton and which should not. That isn't visible in the class itself?

Exactly, you can make it visible by using e.g. a suffix on the class name like *DatabaseSingleton* or using comments. But it is the pointcut which decides which classes that are singletons, that's correct. There is support in AspectJ to directly in the class write the same thing that what's in the pointcut. There are different ways of writing a pointcut in AspectJ.

You have to use it with care. Using the constructor is pretty appealing, where you have a condition in the pointcut that if the class name contains the word singleton, then you also easily can on the name of the class see that it is a singleton. Although that requires that you have a structured way of working and that there is a name convention and such, which is always good, then you have gained a lot. If you have one singleton class in the entire system perhaps you gain less, but if there is many singletons you gain more. Actually, in most systems I have been involved developing, there has been quite a lot of singletons. Even if they are often not implemented as a singleton, most often there is only one object created, and often there must not be more than one object.

So they should be implemented as a singleton?

Yes, many times it is so.

If you had an aspect with a pointcut based on name conventions all you had to do would be to name the class correct and that class would in fact become a singleton.

You could gain quite a lot on that. But then you have to think and via name conventions or some other way annotate the classes, it also good for those who use the class to know that it is a singleton. That isn't done in this original code either. You have to know that the class is a singleton to invoke the operation getInstance instead of writing a new statement. But there is nothing in the name or so who leads you to know that it is a singleton. Many times it is so that the user should not have to know that the class is a singleton, but that is something that the class itself should keep track of. Yes I think that this is quite good use, although I would like to see the constructor solution so that an object is always created in the same way. And you should use a name convention and not only rely on the tool. What the tool does is to give a hint that there is an aspect, but what happens in the aspect, to know you have to go into the aspect to see what it does. So I believe that together with good naming convention it saves a lot of complexity.

6. Disallowing arguments with a null value

Here we have a class named *Hex*, which consists of a number of operations which all take an input and return the hex value for that input (listing 6a). We have a pointcut (listing 6b) which matches all operations named *toHex* which take an array of any type as the first argument. We have done that because in the beginning of all operations there is a condition that if the first argument is null, because if it is the operation cannot proceed and instead returns the string "null". Since I have matched all these operations it's possible to place that control in the aspect instead. We have an around advice which return a string with the value null or calls proceed to execute the actual operation. What's left in the refactored code (listing 6c) is only the actual purpose of the operation. This might not be a crosscutting concern in that sense that it affects several operations in several classes; instead it affects several operations in a single class.

Yes, this looks quite good if the control is consistent. The disadvantage is that you in the original code clearly can see that if you send in a first argument which is null you will get a string with the value null back. In the refactored code you don't see that without going into the aspect. If you are interested in how it works, in an ideal world you shouldn't have to read the code to understand what you can get out of an operation. In the real world, it's often so that you have do read the code anyway, since the code is not documented properly. You shouldn't have to do that; it should be evident in the documentation that exists, e.g. *Javadoc*. Then the aspect is ok to apply. The disadvantage, I guess is, if you haven't done so. Then you have to go in and read the code, and then the code gets hard to read since you have to look at another place to figure out what's happening.

I can add that there in AJDT is a variant of *Javadoc* which look almost the same but also includes aspects. You could get the same documentation but with extra information about which aspects that affects, how they work and hyperlinks to them.

That's good. As we said in the beginning of this interview, you need good tool support to be able to handle AO. It takes some support since it's implicit and you need to be able to follow things in a good way. But if you have good tool support I think that you can get over some of those problems. And as you say, this is typical example of that.

I have found that AspectJ is the only framework with support for other things than the aspects, and it also, definitely, has the largest user base.

But that means that you are limited to Java, which implies that today it's not even usable for other than research in any other language then Java? That's pity and a clearly limiting factor.

Of course that is based on how much support you need, but if you want support in documentation and IDEs and such, then the answer is no, it's not.

Yes, and you need that. You can't sit and look and guess what affects what. That would take to much time.

Would you have used that aspect on these five operations?

No I don't think so, with the motivation that you don't gain that much. […]

7. Ensuring that directories are created

Here we have a *File* object where a path is send to the constructor (listing 7a). After each *File* object which is created in the system, the operation *mkdirs* is invoked to ensure that all directories exist, before doing IO handling. In this system, because of good OO, there is only three statements in the entire code where a file object is created.

I have written an aspect (listing 7b) in which the pointcut matches the creation of a new file object. The path is made visible to the advice which creates a new file object and invokes the *mkdirs* operation, which means that we get creation of directories implicitly at the creation of each new file object (listing 7c). The creation of the file within the advice is excluded in the pointcut. It was considered easier to solve this that way than sending the newly created file objects into the advice.

Since it's all gathered at one single location in the code, there is not very much gained. But if the file handling exists in a lot of places within the system, I absolutely think that this could be a useful way of working, to use an aspect this way to ensure that you don't miss this, and that it's always done the same way. Although in this case since it's gathered I wouldn't replaced the original code with an aspect in this case. But if there would have been a lot of file handling in a lot of different locations in the code, I definitely think that I would have used it.

Sure, the aspect would match new file handling that you add anywhere in the code.

That's a big plus, what you just said that if you add a *new File* statement you would get the implicit *mkdirs* statement afterwards. That's a plus that I don't think we have touched before. We have discussed getting more consistent code. But the thing where you get things consistent even though you add things afterwards, which often is harder than when doing it from the first place, because then it's another person which hasn't the complete view on the rules which applied at that time. Then that person gets support doing it consistent. That's an argument for using aspects even if they match statements which at the time are only evident in a few locations in the code, to get a consistent handling when adding something somewhere else in the future. At a first glance I would not have written the aspect since the file creation is done in only a few places. But there are many standpoints to consider here. On one hand it's easier to maintain in the future since it forces you to automatically do the same things. On the other hand, you should know that those things happen. Of course a tool can help you with that. What the tools can't help you with is to do something in the same way as they where done previous times. But tools can tell you that there is an aspect. Totally it should be so that it helps making it more consistent.

8. Implicit formatting of a new DBHandler object

The next example is similar (listing 8a). Here we have the creation of a new *DBHandlers* which are formatted in the same way. In this example we have the creation of three *DBHandlers* and we have an aspect (listing 8b) which catches that and formats the *DBHandler* directly after it's created. That way we get an implicit formatting here in the same way that we got an implicit invocation of *mkdirs* in a previous example. That gives us this refactored code (listing 8c).

The reason for why the aspect is quite complex is that we don't send any arguments, instead we create what we need to be able to format the DBHandler. We create a *DBLevel* here from a *Config*, which is a singleton.

[…]

What makes this aspect complicated is that you first create a *Config* object and then a *DBLevel* object; I would probably have sent those variables to the advice. The rest is good; it makes the original good much better. It reduces the code and you see what's essential. Ok, so you create these handles and then add them to the *OPLogger*. That you actually do a lot of formatting on the handlers is hidden, and if you add another handler in the future the same formatting will apply to that. That's very appealing.

We are not talking about particularly many invocations so in some way you don't gain that much, but on the other hand you get a consistent handling even in the future.

I have another dimension to this that I would like to discuss which is not evident in the examples. Just below the original code we have a similar block where other handlers are formatted in the exact same way. The difference is which arguments that are sent to the operations *setFormatter* and *setLevel*. Since AO is all about multiple hierarchies it's fully possible in the aspects to use inheritance and such, but also to share a common operation which could have parameter list where you can send in what should be formatted and how it should be formatted. So all kind of handlers could have different pointcuts and different advices but share a common operation which does the actual formatting.

So there is more than what's shown in the example?

There could be more then what's shown in the example. I haven't refactored more than what's in the examples.

This is all about logging. We log things to different places, which are why there are several handlers. We log things *from* different sources like the server, client and database, which is evident here. That's the input, but there are also several outputs. Things are logged *to* a table, the database, a file and even to the event handler in the server. There are a number of these.

What do you think about the possibility to build a hierarchy side by side with the original implementation? Things become a bit more advanced.

Yes, definitely. It support, even more, working in a consistent way, the more you reuse, the more consistent it gets, so from that point of view it seems good. What you can imagine is that if the aspects become to complicated it might become hard to understand what the system does. I think that you should keep the aspects quite short so that they don't get too complicated. You shouldn't put logic into the aspects, if you do that you have gone too far and the complexity increases instead of decreases. AO should do small things on the side, which doesn't involve the main functionality. The main functionality should be in the main hierarchy. Otherwise, I my world, you have used AO in a wrong way if you have put logic in it. That's not the purpose of AO; it should take care of things on the side which doesn't add something to the system, at least as I see it. It mustn't be the main function the aspect affects, but small things that you want to get rid of in the main hierarchy.

I think that most argue the same way, but I also think that there are some which argues to build as large hierarchies on the side, as the main hierarchy. They argue that there shouldn't be this dominant hierarchy that you speak of. I think that there are researchers which research around systems without a dominant decomposition.

It depends on that I am used to work with OO and see AO as something extra which can help me get rid of things that have messed up the code and make it cleaner. That's my view. Most of those I've talked to that have worked with OO sees this as a small addition to OO, but of course as a researcher you get another view and may be able to see the other possibilities with it. I don't say it's wrong, only that that is not my view at the time.

I think that a system which lacks a dominant decomposition is far away and that we are first going to see applications which support yours (and mine) view.

*** There are quite a lot of research going on right know related to aspect-orientation. Do you think that aspect-orientation have a future?**

Will it take the step from research to industry?

It's always hard to know what will be a hit and what will not. I think that it must be supported by a big actor. That is what's required. There are enough positive effects to supply something. But there might not be enough support in tools and so today to make AO public enough. You perhaps you need to able to use AO in more than one language for it to be a hit. It is required that a big actor like Microsoft supports it for it to become large. As it looks today, most developers I have talked to had never heard about AO before.

That's true. At my first day here we walked around and talked to everybody (~15 developers) and no one had ever heard of it.

Yes. I've also spoken with a few others who had never heard about it before. There is a chance but the research must be more public available so that there is an attraction to look at AO before the next step comes. There has to be an interest and at that point a lager actor might start supporting it in their tools, and then the industry might dare start using it. If it gets more public available it has a future, but as it looks now it is way too unknown. […]

*** Could aspect-orientation be a technique used within this company in, say five years? Please motivate your answer.**

It takes about the same as for AO in general. We are not particular conservative in what we do. We develop products and often do new things. We aren't stuck in an existing infrastructure as many other are. We can more easily change how we work with tools and such. I don't think that we will be after the rest of the industries in adopting AO, rather before.

When I discussed this with another employee he mentioned that someone with the possibility have to dare try AO in a project where it fits. Perhaps it's a bit of a chance if it gets used?

Does he mean internally?

Yes.

At the moment I don't think that anyone dare try it, since you first have to explain what AO is and then have to argue about why anyone should use it. It's hard to argue for the usage of something that the receiver has not heard about before.

What we discussed was the possibility that an enthusiast started using it at home and brought it to work. That's the possibility he saw.

Yes, but I feel that it's not public enough so that you would get that possibility to use AO at least officially. Then there are of course those that introduce things unofficially and if that works well, then it might become official.

*** Aspect-orientation is based on an implicit connection from the code to the aspect. What part does and IDE (Integrated Development Environment) play in your daily tasks and how does that relate to this implicit connection between the main code and the aspects?**

We have already discussed the role of IDEs, could you summarize our discussion? As I understand it, it's quite important that AO is supported by some kind of IDE, that you have that possibility.

Yes, I think so. Since it is so implicit and you can't just by reading the code see that there is an aspect which are executed there has to be good support. Support that both tells the user that there is an aspect and support for documentation that includes this side behavior which comes from the aspect, so that it doesn't get to heavy to grasp what's happening. The basics are that you know that there is an aspect. You should also be able to link yourself to that aspect very easy. The third step is to include the aspect in the documentation for the operation and the documentation that exists for the aspect. The tool support is completely decisive for the future of AO. If there isn't any tool support you can't use it. It is to complex for the brain to grasp.

Would you say that this a big thing or is this just an extension to OO?

It's an extension and no big thing but it's also a new way of thinking and it could affect a design quite much if you start using AO to the fullest. So I don't find it that small. When I think of it, it's quite a revolution, how you think. You can think that it's an extension to OO but it's still a new way of thinking, there might be opportunities to use AO which you at first glance don't even think about. I also think that it's important that you stick to name conventions so that it doesn't become a hard task to create pointcuts.

Yes. Even if names are not the only thing you can match on. You can also match on reading or writing of a variable with a certain type.

Yes, there are other opportunities. But a name convention is a way of making it easy. I think that there are large possibilities if it only could get a bit higher up in the consciousness. It has to be more written about AO in larger forums than so far. At the moment it's too unknown to be tried.

*** How would you feel if you go the assignment to use aspect-orientation in your next project?**

That would be fun!

[…]

It would be really fun and interesting to test it now when you have heard of it. I think that you should bring a consultant into your first project which helps you to think correct, a kind of mentor which works with the project members in the project. I mean you can learn the syntax in a few days, but to think right when designing to fully take advantage of AO, you probably need some help at first. So that it doesn't become more complex than it would be without AO and to fully take advantage of the possibilities that exists.

Thank you for your participation!

Interviewee 3

Interview date: 2006-04-28. Length: approximately 1:00 hours.

* Why don’t you tell us a little about yourself, your system engineer, object-orientation, and Java knowledge’s? How long you’ve worked at the company and which work assignments you’ve had?

I graduated from the program for system programming at the University of Skövde in 1998. After that I worked at in Trollhättan for two years with different things, among those Java at the end. I started working at this company in 2000. I read about Java in Trollhättan and last year I did a few longer projects which involved Java at this company. […]

* Have you had any chance to read some information about aspect-orientation and aspect-oriented software development?

Not really. I’ve quickly glazed through your introduction, background and problem description, but I didn’t get a good picture of what it is.

Then perhaps I might run a quick introduction to AO, so that we have the same view of what it is.

AO is based on the assumption that you want to capture crosscutting concerns (CC). Crosscutting concerns is code which exists at many places and looks more or less identical. In an object-oriented system there is always a dominant decomposion, a main hierarchy. But there are parts which crosscut the dominant decomposion and are needed at several places. With aspect-orientation (AO) you can catch these crosscutting concerns in what’s called an aspect, and link these aspects to places where they are needed, both to be able to do changes quicker and better, and to get a better readability in the main code which becomes more focused on the purpose of its existence. Crosscutting concerns are often not a part of the codes real purpose but exists for another reason.

An example: We have a number of operations which starts with connecting to a database, fetches a value, and then disconnects the connection. If we have ten such operations, we get ten places in the code where a connection against the database is established and ten places where the connection is disconnected. With AO we can capture those crosscutting concerns and put those in a module of their own which automatically is executed when the method is executed. We are in this interview going to see some examples of such crosscutting concerns and aspects.

* What are your general thoughts about aspect-orientation?

Does it seem sensible?

It seems sensible even if I feel that I don’t see a lot of these situations which I believe are because I don’t work with those kinds of applications. It’s probably more usable when you are working with databases and such. MMI can I image is something where AO could be very much used. Also logging, which I often do, I often add log enter and log exit in my application.

Logging is often the main example for AO but there are also many other areas where it could be applied.

*** For how long and in which ways have you been involved in developing the Java server subject to this case study?**

I have not been involved in the project at all. I hardly know what it's called.

I have rewritten some of the code from this Java server using an AO-language called AspectJ. AspectJ looks like Java but has some extra keywords which make it possible to modularize crosscutting concerns. A few examples of such keywords are: within, target, call and execution. They are used to create a pointcut which matches an execution point in a Java program, a join point.

So you tag which join points you would like to match? It feels like object-orientation which an additional level.

Yes, you could see it like that. I usually refer to AO as OO++ when I try to describe what AO is all about in a simple way. Important to notice is that AO does not exclude OO in any way.

It feels a bit like in Visual Studio where they have attributes and macros for tagging methods and classes.

Yes, and there are languages where you can explicitly write such annotations as well. But with AO you don't need to have an explicit tag, instead you have an implicit connection which is both the strength and weakness of AO. The aspects are often put in a file of its own, even if you can put it in the same file.

*** If you take a few minutes and have a look at this code (the interview gets a paper transcript with original code and a refactored code including an aspect). I will introduce the code, the aspect and the refactored code, when I would like you to comment on this based on the following questions (I will help you remember these):**

- **What are your general thoughts about these aspects?**
- **In which way (in your opinion) has the refactoring affected code complexity?**
- **Can you find both positive and negative things with the refactored code? What are those?**
- **Would you have used this aspect? Please motivate your answer.**
- **Do you think that the code would look different if the system was built using aspect-orientation from day one? Please motivate your answer.**

[…]

The code is mainly written by one of your colleges in Gothenburg, and could, and this is my personal opinion, at some places be easier to read. What I investigate is how the introduction of AO in a real system affects code complexity. How you grasp code, how maintainable it is and so on.

1. Modularization of try-catch-clauses

Here we have repeated use of try-catch-blocks since all operations in the *TextMessage* object may throw (listing 1a). I have written an aspect (listing 1b) which consists of a pointcut and an advice, the two most common parts of an aspect. An aspect looks and in many ways works as a class, instead of the keyword class you use the keyword aspect when defining an aspect. Pointcuts are constructions that match what you usually refer to as join points. A join point is really, if we freeze the execution of a program, then we

have join point. For example: an operation invocation, the return of an operation, an operation that throws an exception, when reading a variable and more. Most parts of the execution of a program can be matched by a pointcut. […]

So here is a pointcut with a name (listing b), that matches all operations which has a name that starts with *get* returns a *String*, takes any parameters and exist in the class *CommandMessage*. Then I have written a *declare soft* statement so that the compiler approves that the exception handling is done in the aspect instead of within the original code. Moreover I have written an around advice which has a try-catch-block. There is a variable named result of the type String. When we use the keyword *proceed* the content of the operation is executed and the return value is returned to the aspect. That's the purpose of an around advice. There is also a before advice and an after advice. The before advice is executed before the join point and the after advice after the join point. An around advice is executed *instead* of a join point and we can use the keyword proceed to execute from the join point.

In this case since *proceed* is within a try-catch-block, so if we get an exception we handle that by doing nothing, just like in the original code. The difference is that the advice is used for all invocations, and that gives us a refactored code that looks like this (listing 1c).

I don't think about the try-catch-block, I'm pretty used to see those. But sure, it's nice that way, particular if there is more code than one row. […]

What do you think about this construction, where you can use and advice and handle things like this, things that exists in several places?

You have to make sure that you get used to it, and how does, say, debugging work?

It probably varies from framework to framework but debugging of AspectJ code in say Eclipse/AJDT works in the same way as for all other Java code.

Can I use AspectJ to run unit-tests? That would be optimal; I wouldn't have to affect my code.

Yeah, sure.

I guess it looks good, only that if you don't know about it, you might be a bit concerned that there are no exception handling in the code.

That the advantage and disadvantage, the implicit connection. The connection only exists in the aspect you have to know it are there, unless you have an IDE, I was thinking that would could discuss that a bit later.

I became cleaner this way. Then it's common to spell wrong. If you wrote *gat* instead of *get* in the pointcut you would still get an exception.

Yes, you have to be more accurate in you way of naming things.

Get is probably rather unusual to misspell but receive is quite common to spell wrong.

You can use ? and * to match misspelled operations and variables.

Yes. The spelling is a risk that i see if you break out exception handling and such. And also that you have missed to execute this, and only run the Java code. I guess you have to attach this AspectJ somehow. […]

I have a few more examples. They are of different use so I would like you to comment on that. Would you write the different aspect under the assumption that only you were involved in the code?

I would have used this one, to avoid repetitive code and be able to get rid of copy-paste errors, as long as you shouldn't do something with the exception. But in many case you only log.

Which you can do in the aspect.

2. Implicit handling of non-existing attributes

Here we have an operation *getAttributes* which returns the last element from a vector (listing 2a). If there are no such elements the operation returns an object of the type *AttributesImpl*. I have written an aspect (listing 2b) which has a pointcut that matches such operations. Since the advice don't have access to the operations private variables and instance variables, you have to either send the variables to the advice through the pointcut or solve it in some other way. Again an around advice is used and instead of using the operation *isEmpty* on the vector object, I use a try-catch-block and return a new *attributesImpl* object if there where no elements. Together with the aspect the refactored code only consists of an attempt to return the last element from the vector (listing 2c) and the checking is done in the aspect. There is quite much code to do quite little, but there might be a few of these kind of operations.

It feels a bit like you move the problem, but sure if it's used in many places you gain on using the aspect.

How is the complexity?

Like the previous. Pretty standard code.

Would you have used the original code, or written this aspect?

I would have used the original code, since there really isn't much code and I don't feel that the aspect gives very much in this case. It's easier to see what the original code do then what the refactored code with the additional aspect do.

3. Softened exception handling

In the next example we once again have something that may throw, and that's the creation of a new *XMLHandler* (listing 3a) which occurs within different classes and different packages in the system. Each invocation of the operation *newHandler* must therefore be surrounded with a try-catch-block. This is a statement which occurs at many different locations throughout the systems. Two different exceptions are in most places handled and even if that's not the case, you could imagine that two different type of error messages where logged.

I have written an aspect (listing 3b) which matches the invocation of the *newHandler* operation and uses and after throwing advice to catch any exception that is thrown. The result is one single row instead of the original nine (listing 3c).

[…]

If it's done in many different locations in the code.

Yes it's done in around than ten locations throughout the code.

Then it's very nice to have it that way. If I would have done it without AspectJ then I would have done a utility class or something where you can do refactoring.

That is something that seems quite common. I have discovered that if the system is built using good OO (which this system is) it's quite common to have modules at the side and have an explicit connection, and that is how you would have solved this?

Yes.

I was thinking that we could discuss that in some locations there is only one catch-block which catches an exception of the type *Exception*. The idea is that it should look the same at both places, but since it's written at different times and perhaps by different persons it looks different. With AspectJ you get a uniform handling of such things, it will never differ since the code is only written at one location.

That's good, otherwise you think that you do the same but you end up changing in nine locations of ten. That is troublesome at least on logging. Common copy-paste is hard to find, it messes things up. To avoid repetitive code this is excellent actually. For this type of stuff, when you do things that you want to have good control of e.g. creation of external objects, files and IO this is good.

4. Implicit handling of non-existing record elements

Here we have two operations which do the same thing as in a previous example, checking the length of a vector and return the last and first element in a list (listing 4a). We have one aspect (listing 4b) which handles both operations with an after throwing advice. When an exception is thrown the advice catches that and does nothing which should result in the operation returning null just like in the original code. So instead of trying to inspect if the list consists of more than zero elements you only tries to catch the element we are interested in (listing 4c) and if it throws we catch the exception and do nothing about it.

This seems more dangerous. I am not particularly fond in how the original code is written either because I don't like return in such blocks because it is not uncommon that you add some code after the return statement and that will ever be executed. I would probably have used it since it does the same checking on both operations and possibly a third. On the other hand, I'm not sure; it takes some thinking when you can use these so you don't get a fault in the aspects.

You have to debug the aspects in the same way as if they where a part of the main code. So if you do a boundary analysis and send in incorrect values you have to debug the aspects as well, make sure it handles such values.

I think I would have used this… perhaps.

Perhaps? What do you think about?

I thought that it fells more complex but at the same time, it fells like its more flexible to handle the same situation in the same way.

But the gain is larger to handle things uniformly than the complexity in the aspect?

I think so. I would not have introduced AspectJ just for that, but if you use these kinds of lists in several locations. Perhaps not use the same aspect on all lists but similar aspects on different lists.

Aspects are just like classes, you can put them in a hierarchy and inherit and such things, so it's possible to build complicated structures on the side.

You must not fall into the object orientation-trap and build huge class hierarchies; you usually gain in keeping things simple and flexible.

The purpose of AO is actually to build multiple hierarchies. Normally the other hierarchy is small, but it is actually possible to build *N* hierarchies of different sizes in an AO program.

5. Aspect-oriented Singleton pattern

The next example is a common design pattern (listing 5a), singleton. There is an operation named *getInstance,* and there is a variable which in this case is named *config*. If *config* is null it gets a new object reference and it is returned. If there is an object reference that object reference is returned instead and no new object is created. I have written an aspect which catches the invocations to operations named *getInstance* although one package has been excluded where the operations has a bit different logic, so it's possible to include and exclude things. The after advice checks the value after the operation has returned and if it's null a new object is created dynamically using a keyword in AspectJ. The possibility to dynamically create is nothing new to AspectJ.

A bit complicated.

Yes, but it applies to all instances. The operation *getInstance* itself only contains *return instance,* the variable name must be uniform for all singleton classes so that the aspect can read it, we can no longer have variables which is named the same as the class e.g. *config* and *dbconnection*. What are your thoughts about this?

Singleton is a pattern which is quite used. That's why many developers know how it works. I think that the object creation seems a bit complicated. If you would look at this I would not know that a new object was created, and I would therefore avoid using that. If it is troublesome to understand what it does or to read the code I would avoid it. Therefore I would have let it be as it was from the beginning.

There are also other variants of AO singleton where you instead of using a special operation to create a singleton just create a new object and use the constructor to check if the class is a singleton or not, and if it is a singleton just return the object if there is such an object created. So you don't have to differentiate a normal object from a singleton if you want to. You can be unaware of that you just created a singleton.

That seems pretty nice, but it can be an advantage to know if it is a singleton or not.

You can annotate the class with a special naming convention or a comment, but it's perhaps easier to have a specific operation as in this case?

Templates in C++ were not so fun, because you did not know what things was. […] So to know if the class is a singleton or not I have to read the aspect? Doubtful, I might have used it. It's neat but it is probably more difficult to read if someone must review or debug the system say a year later. What can I say, yes perhaps I would have used it if there was an easy way to know if the class was a singleton or not, via name convention or something like that.

6. Disallowing arguments with a null value

This is the systems *Util* class. There are three operations which all first checks if the first argument is null (listing 6a). If the argument is null the operations return a string with the value null. The purpose of the operations is to return the hex value for the input, but that is impossible if the input is null, so all operations have an if-block in the beginning of the operation that does this check.

I have written a pointcut which matches these operations which is named *toHex* and takes an argument which is an array, returns a String and the operation is located in the class Hex or some subclass to Hex (listing 6b). I have written an around advice which does this checking and returns the null string if the argument is null, otherwise the rest of the operation is returned using proceed. What's left in the refactored operations (listing 6c) is the actual purpose of the operations.

It increases readability, so this I would probably have tried, perhaps. I feel that I am afraid of everything that parses, this is a parser. […]

You where a bit skeptic?

No, not really, this makes sense. I was just afraid that you might accidentally would name the operation to something else than *toHex* and get unexpected side effects. But in this case it feels crystal clear and suiting.

7. Ensuring that directories are created

In this example there is a *File* object which first is declared and initialized with a path. After the creation of the *File* object the operation *mkdirs* is executed to create all folders which exist in the path before doing IO (listing 7a). The aspect does the invocations for us. Every time that we create a new File object the after advice invokes the *mkdirs* operation (listing 7b). We get this code (listing 7c) instead of the original code.

The code does get cleaner, but not a whole lot. Doubtful if I would implement such an aspect for only three invocations. I also feel that the code has to be a bit mature, sometimes you test things to see what works and at that point you probably don't do aspects. If you knew exactly how it should look like from the beginning it would be crystal clear, you would think – I do that with and aspect, and then you write that aspect.

So you don't gain enough to use this aspect in this case?

I am not sure how you should use AO, if you should use it as much as possible or if you should use it on specific chosen locations where it has maximum effect, like logging. I see no problem using it but I am not sure if I would have done it for such small gain.

8. Implicit formatting of a new DBHandler object

Here we have the creation of a DBHandler (listing 8a). After creation the objects are always formatted in the same way. If we apply the following aspect, which match all creations of DBHandler objects, and have an around advice where we do this formatting (listing 8b), all we have to do is to create an object and we get the formatting implicit (listing 8c).

I would probably have used this. I like this example; in particular how you can get rid of the copy-past-errors. I would probably use AO to more specialized things like this, things that are used often. **[…]**

* In which way do you think that aspect-orientation affects code complexity?

What are your general thoughts about these eight aspects?

If used correctly they can probably help get rid of the copy-paste-errors that exist and ease reviewing since you see what the code actually does and don't have to focus on the exception handling. But you also have to review the aspects themselves. So it may increase readability a bit, and in some cases. Negative is that you depart from Java-standard. You can't assign a person, who knows Java to look at the code, the person must also know a bit of AO and also it might be harder to look for errors even if it becomes Java code at the end. It seems a bit plucky.

*** If the system was built using AO from the start, do you think that the system would have been any different? Would you have thought different in the design-phase. Perhaps look for AO modules.**

I think that the first system you do would look more or less the same. But the second one, when you have a bit more experience would differ a bit. If the system would be designed with aspects in mind it would probably look different.

There are quite a lot of research going on right know related to aspect-orientation. Do you think that aspect-orientation have a future?

If aspect-orientation were to get bundled with standard-Java it probably has a future. If it lives on as an extern product it would be though. Some big actors must embrace AO.

Let’s not forget that AO frameworks are available for a number of languages.

I believe that it has to be standard or supported as an official add-on so that the developers get the communities with them. Small companies are probably quicker to start using AO. If I daily worked with database invocations, SQL questions or file handling with the same exception handling so you gain time and resources, sure it sound sensible. It looks a bit like templates in C++, so it is not that new, even if this is more sensible implementation of it.

No that’s true. The idea has been around for a while and there have been different implementations in how to achieve this idea of multiple hierarchies.

*** Aspect-orientation is based on an implicit connection from the code to the aspect. What part does and IDE (Integrated Development Environment) play in your daily tasks and how does that relate to this implicit connection between the main code and the aspects?**

We previously discussed that you don't see whether something is affected by an aspect since the connection is implicit from the main code to the aspect. One of the solutions is to have an IDE. There are support for AO in for example Eclipse where the handling of AO is annotated as small icons in the left margin which says that this line is affected by an aspect. You can click them to the aspect, se which join points a pointcut matches and so on. Although everyone might not be able to, or even want to, work in an IDE.

It would help. But it also becomes such as with common code in general, you must have some control of it. If you have common class/aspect libraries, then you don’t want them to change things for others, if you change an aspect you can easily affect others code, you must be able to protect yourself against such things.

I am very much for IDEs and such, but there are some emacs-users also. But perhaps there’s support also in emacs? If not perhaps it will come in the future.

*** How would you feel if you go the assignment to use aspect-orientation in your next project?**

Well, I would do that. I would probably start a bit moderate, use it for logging, file handling, string handling and such, common stuff.

Thank you for your participation!

UNIVERSITY OF SKÖVDE

School of Humanities and Informatics
Final Year Project in Computer Science 20 p
D-level
Spring 2006

Assessing the introduction of aspect-orientation in a real-world system regarding complexity

Appendix C: Measurement charts

Daniel Oskarsson

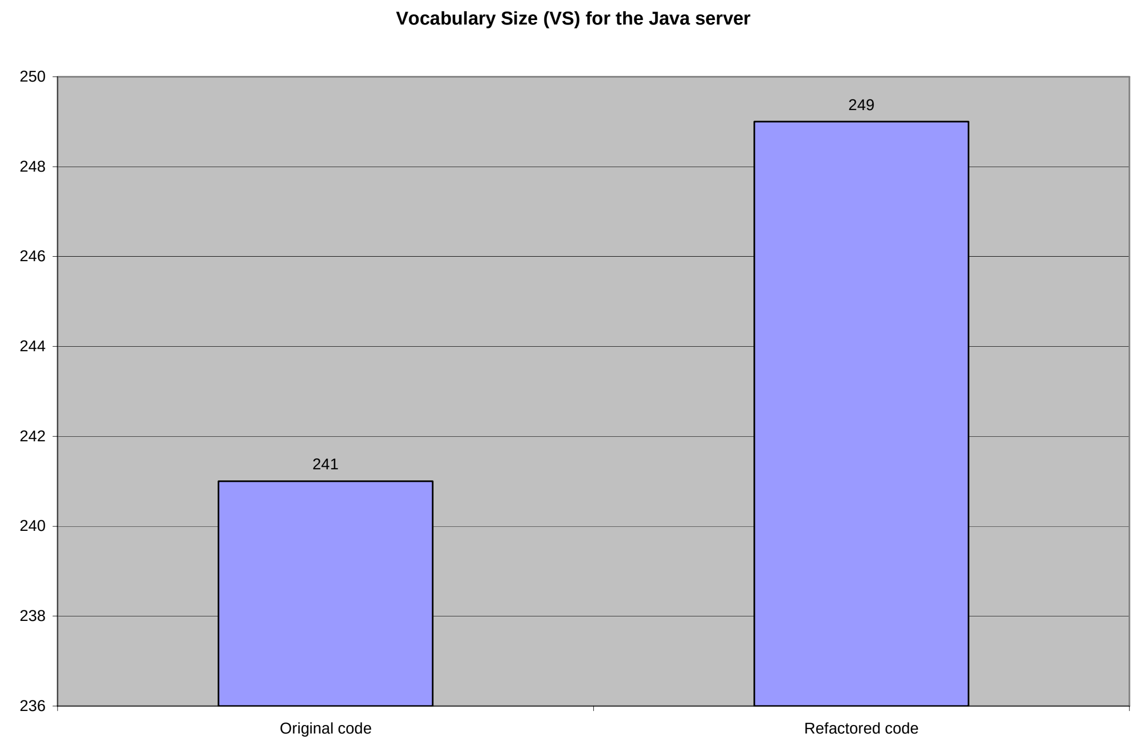
HS-IKI-MD-06-003

Table of contents

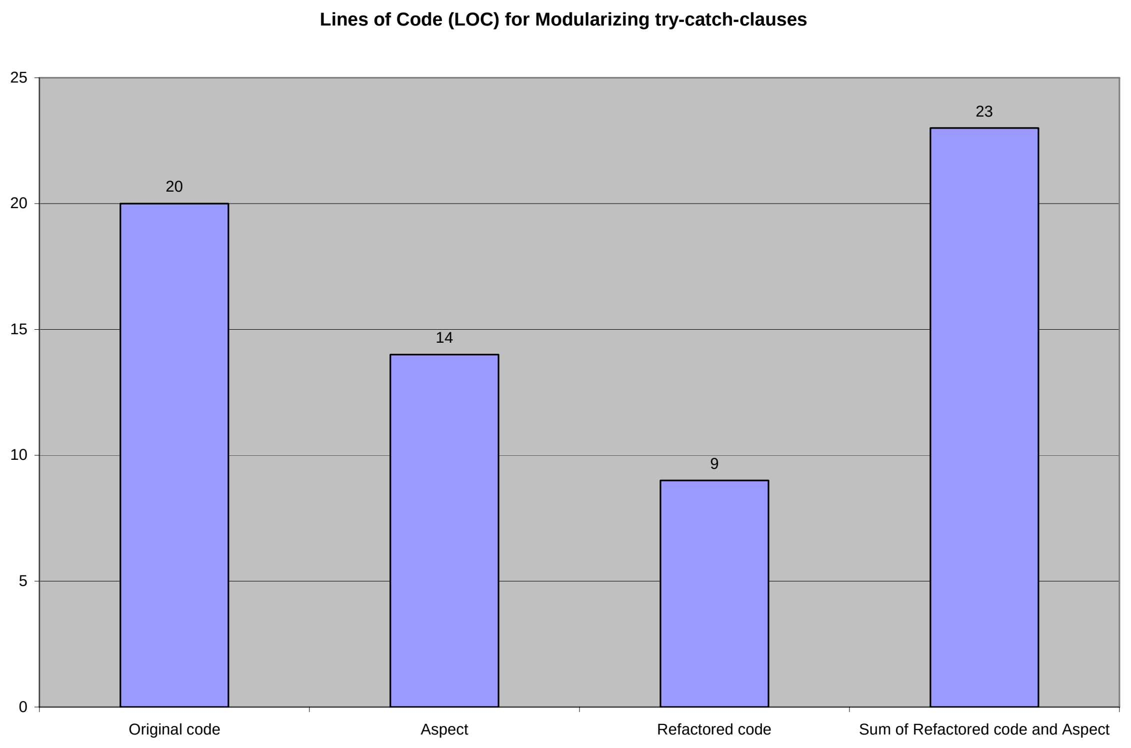
1 Size metrics

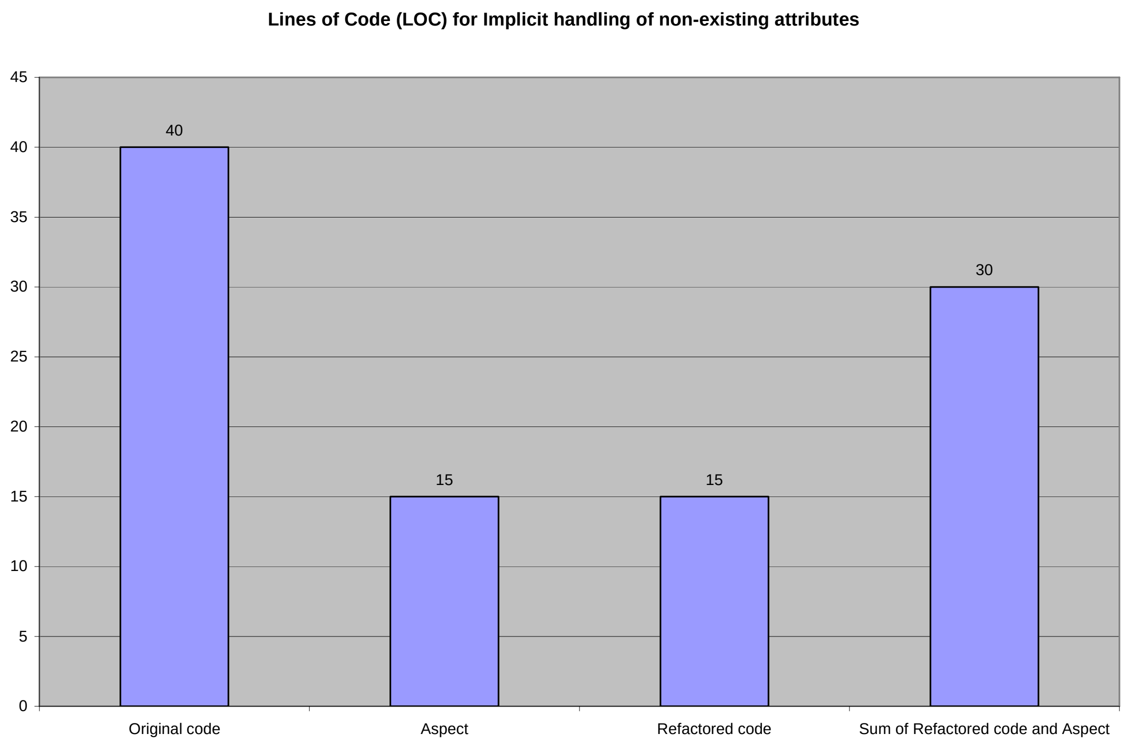
The size metrics measured are VS, LOC, and NOA metrics.

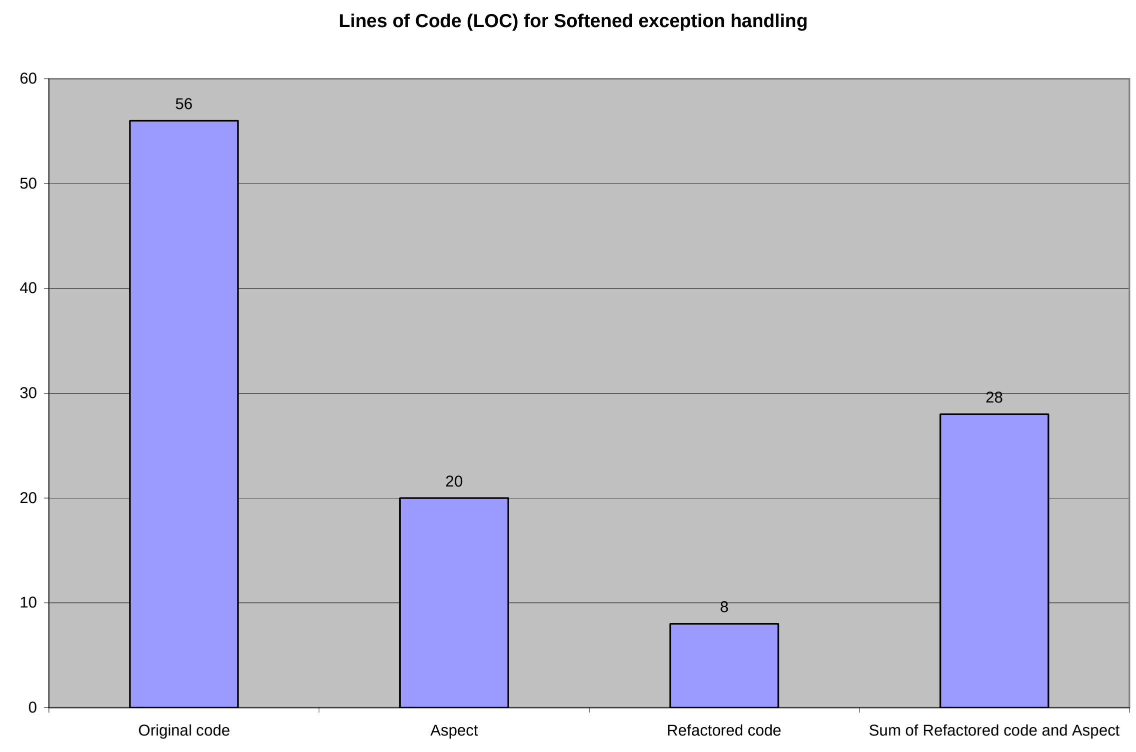
1.1 Vocabulary size (VS)

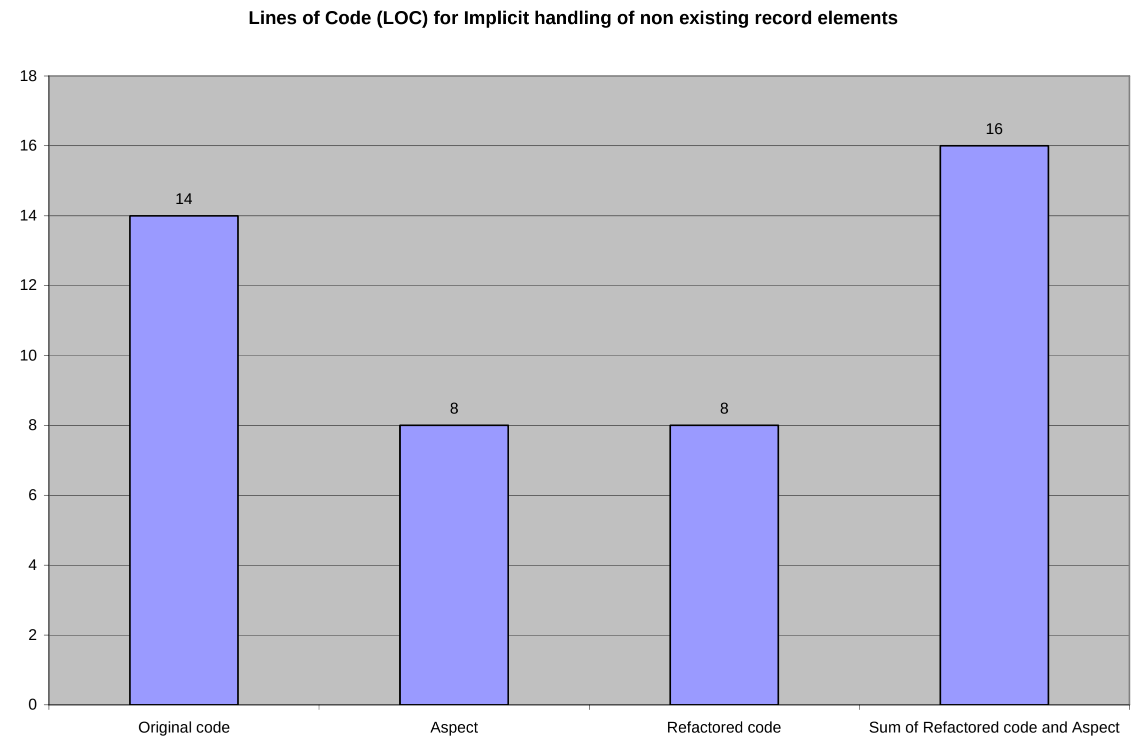


1.2 Lines of Code (LOC)

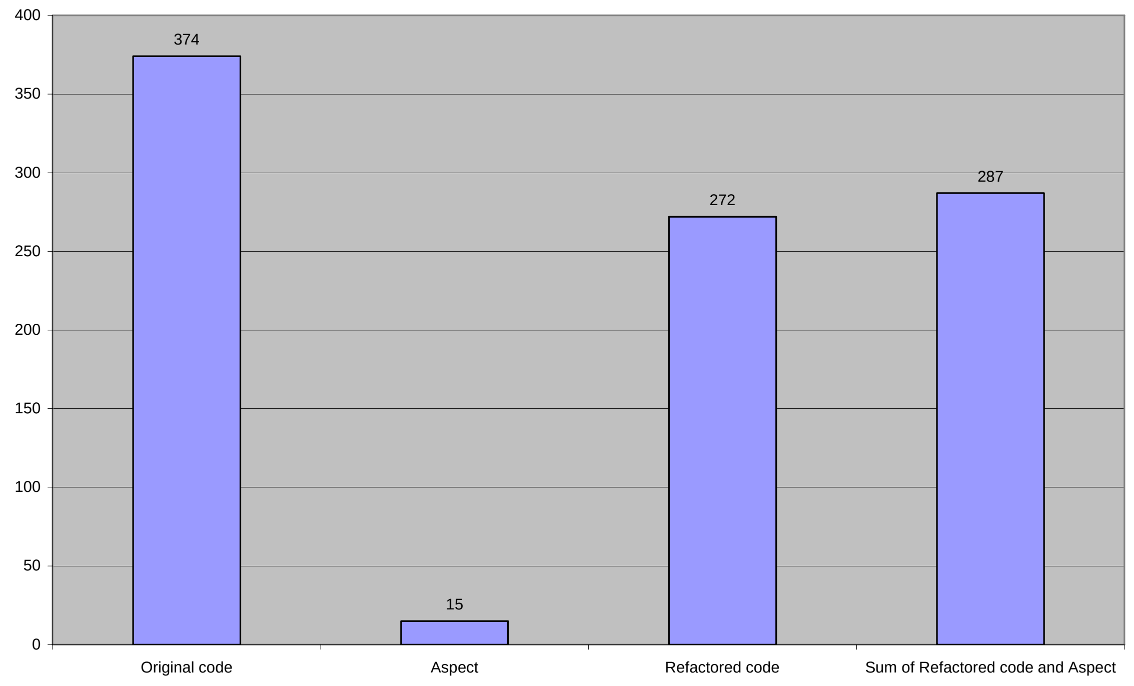




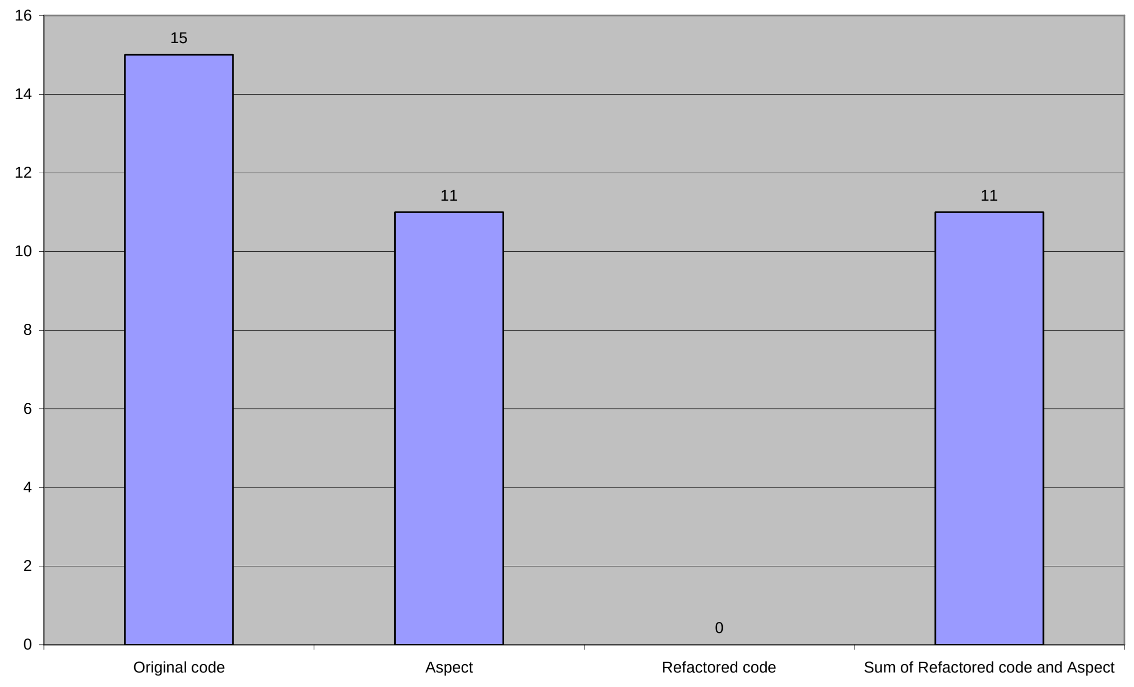

Lines of Code (LOC) for Softened exception handling
60
50
40
30
20
10
0
56
20
8
28
Original code
Aspect
Refactored code
Sum of Refactored code and Aspect


Lines of Code (LOC) for Implicit handling of non existing record elements
18
16
14
12
10
8
6
4
2
0
14
8
8
16
Original code
Aspect
Refactored code
Sum of Refactored code and Aspect

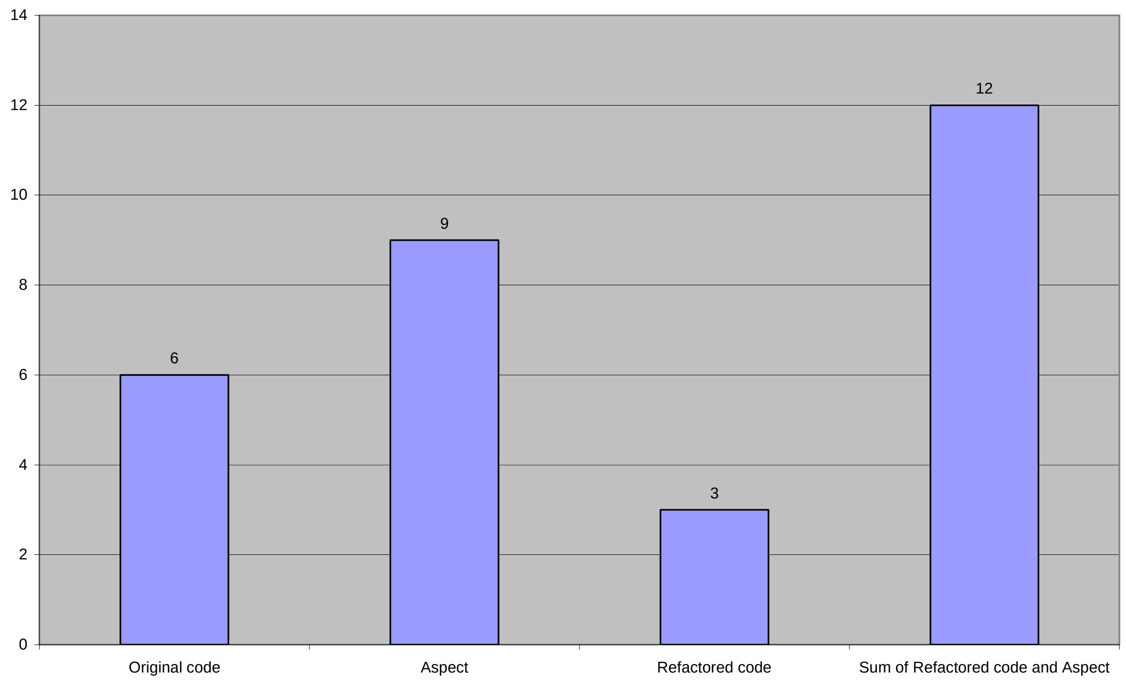
Lines of Code (LOC) for Aspect-oriented Singleton pattern



Lines of Code (LOC) for Disallowing arguments with a null value

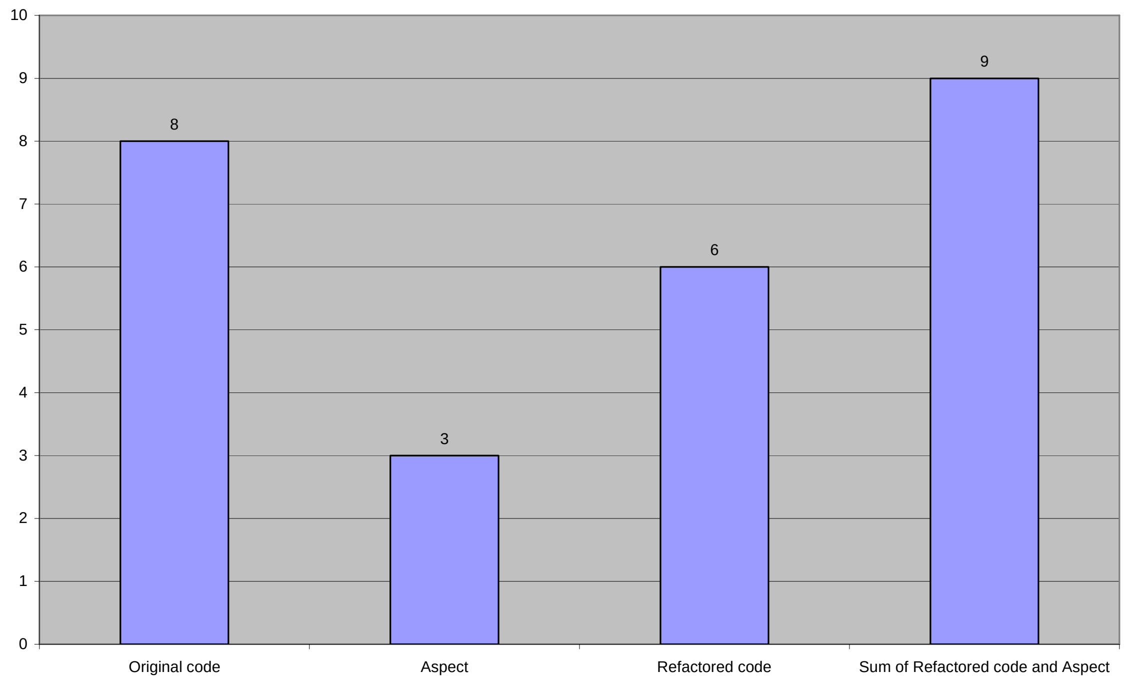


Lines of Code (LOC) for Ensuring that directories are created

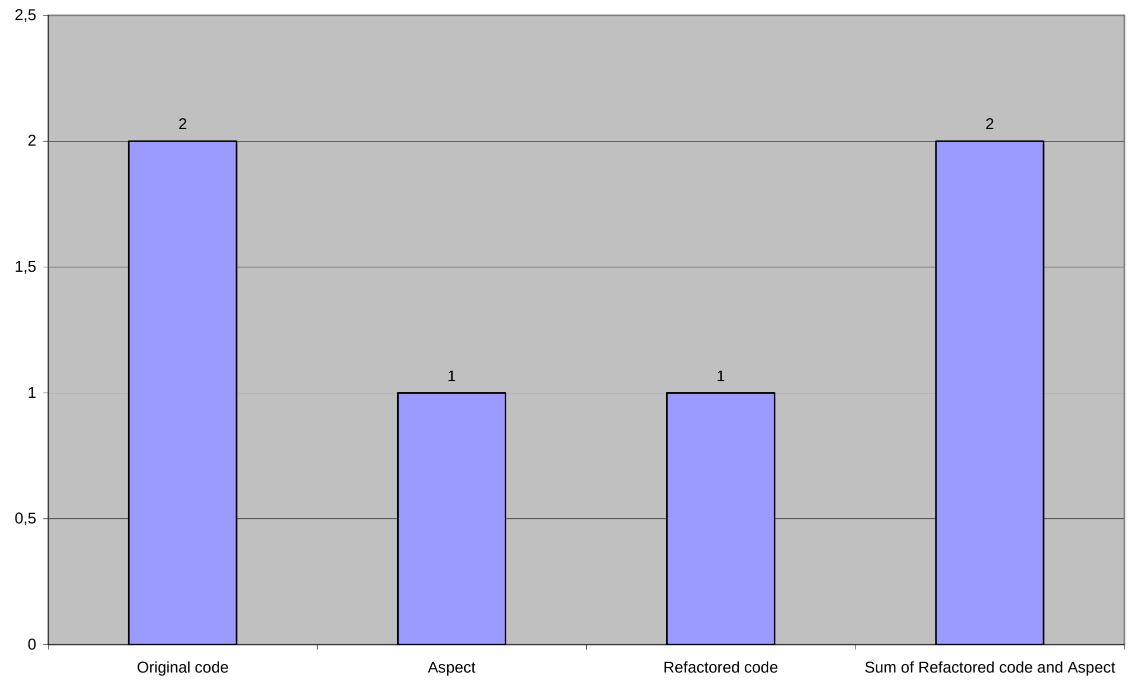


1.3 Number of Attributes (NOA)

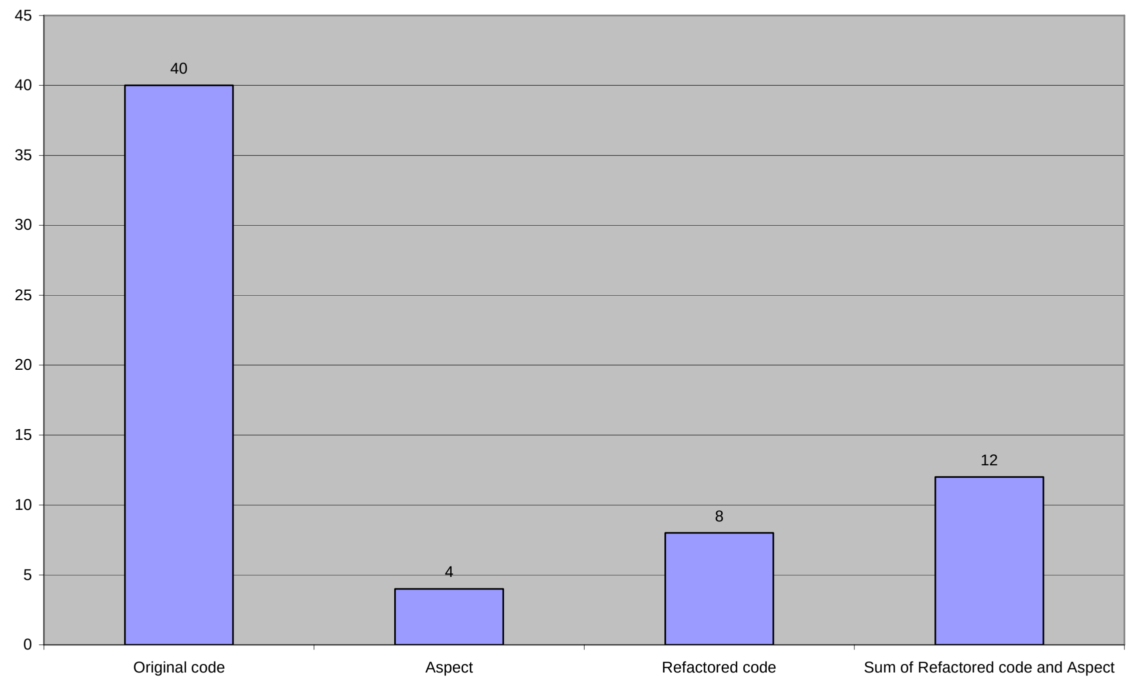
Number of Attributes (NOA) for Modularization of try-catch-clauses



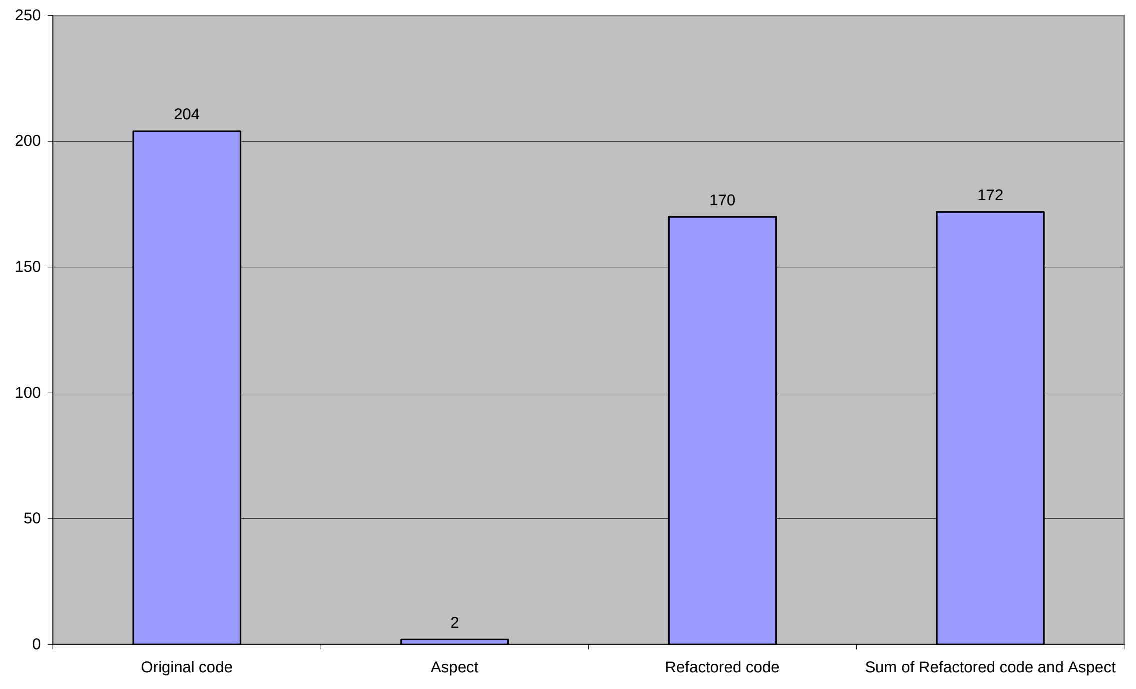
Number of Attributes (NOA) for Implicit handling of non-existing attributes



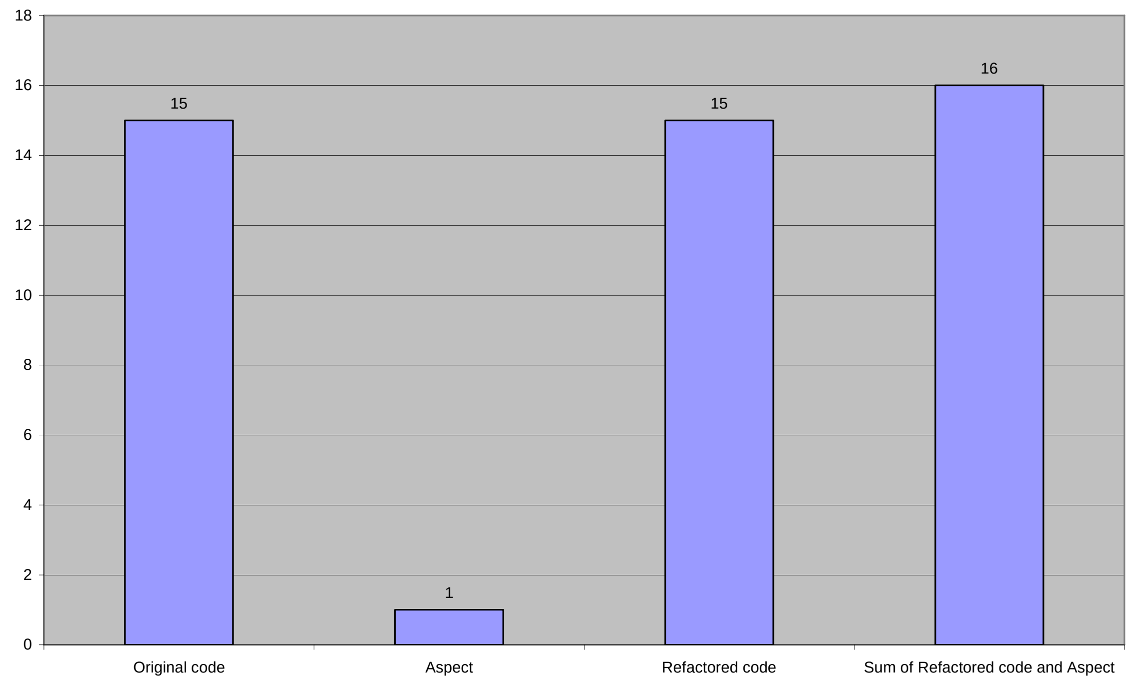
Number of Attributes (NOA) for Softened exception handling



Number of Attributes (NOA) for Aspect-oriented Singleton pattern



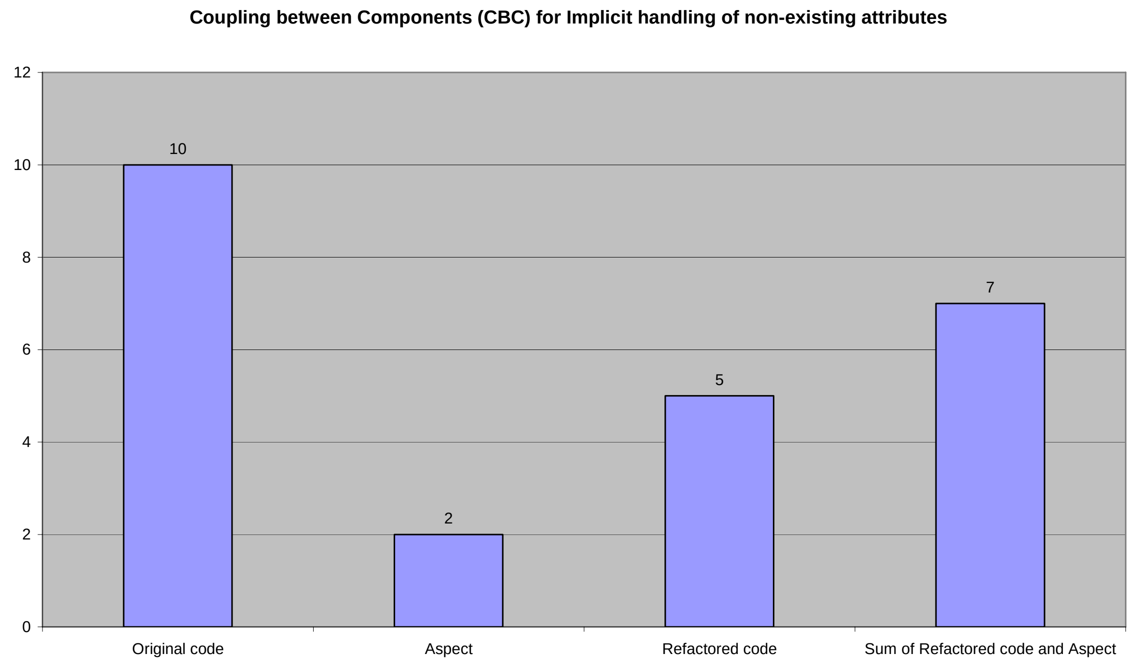
Number of Attributes (NOA) for Disallowing arguments with a null value

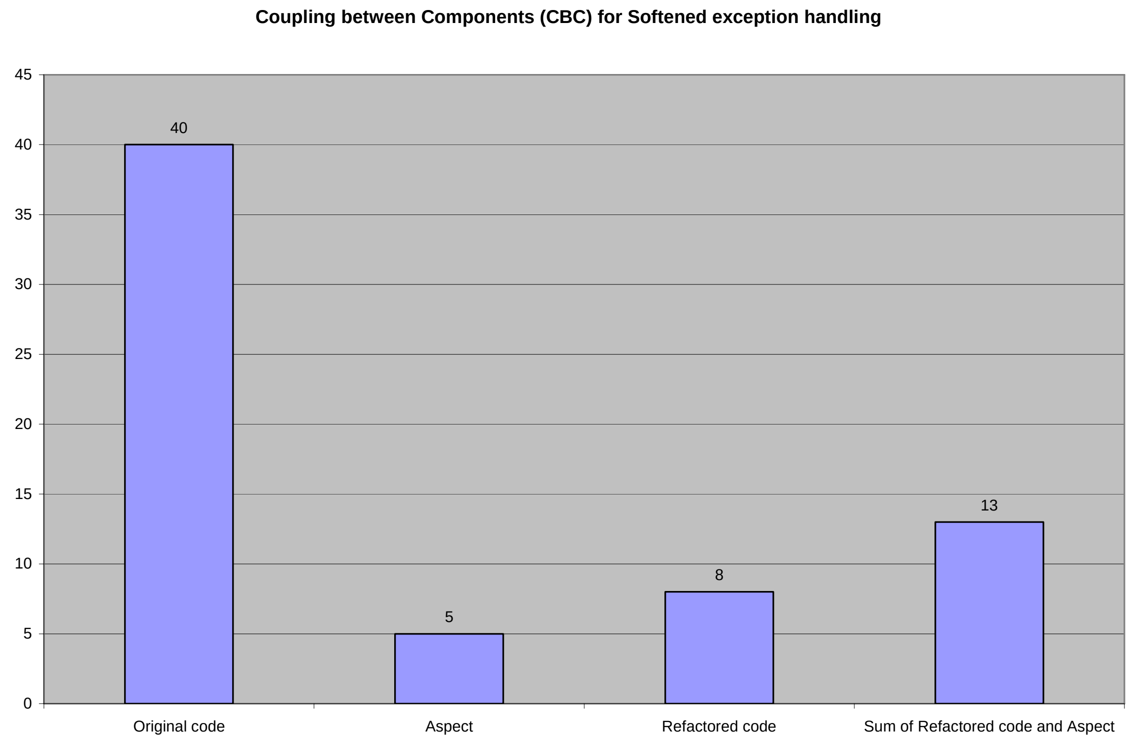


2 Coupling metrics

The coupling metric measured is CBC.

2.1 Coupling between Components (CBC)


Coupling between Components (CBC) for Implicit handling of non-existing attributes
Original code
Aspect
Refactored code
Sum of Refactored code and Aspect


Coupling between Components (CBC) for Softened exception handling
Original code
Aspect
Refactored code
Sum of Refactored code and Aspect