

DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY

MASTER OF SCIENCE THESIS 20 P

ASPECT-ORIENTED PROGRAMMING
COMPARED TO OBJECT-
ORIENTED PROGRAMMING WHEN
IMPLEMENTING A DISTRIBUTED, WEB-
BASED APPLICATION

By Magnus Mickelsson

Thesis reviewer: Olle Eriksson, Department of Information Technology, Uppsala University.
Thesis advisors: Patrik Fredriksson and Tobias Hill, Citerus AB.

ABSTRACT

This Master of Science thesis compares the currently dominating programming method, object-oriented programming (OOP), to aspect-oriented programming (AOP). AOP offers extended mechanisms to OOP for decomposing problem domains into cleanly encapsulated entities.

By having implemented the same distributed, web-based application as two application versions, one using pure OOP methods and one using AOP methods, based on the AspectJ AOP enhancement to Java, an interesting opportunity arises to compare and analyse the approaches, both from a theoretical viewpoint and from actual programming results.

The AOP implementation had quite similar results as the OOP implementation regarding code statistics, however the more generic services that are decomposed into aspects, the bigger the advantage of AOP.

The development tools were better for the OOP implementation. The two major differences are in refactoring and UML-modelling, which are currently not of production quality for AOP. Tracing and debugging were however more efficient using AOP, since development aspects can be dynamically applied to the code to focus on possible problem areas.

The AOP code is more modular and reusable than the OOP version. It is also more flexible in terms of altering behaviour without changing design or code, and contained less or no code tangling and code scattering. In spite of the many advantages in writing code, AspectJ code is not necessarily easier to follow. This may call for new thinking both in terms of IDE development and UML modelling, to grant developers a more intuitive, visual view of the application being developed or maintained.

The thesis and its example code can be downloaded at: <http://www.darkwolf.ws/aop/thesis/>

On a side note, MIT technology review listed AOP as one of "ten emerging areas of technology that will soon have a profound impact on the economy and on how we live and work" [URL5].

ACKNOWLEDGEMENTS

Thanks to all people who have helped me in the making of this thesis, in any way! The following deserve some special recognition:

- Sandra Öberg – for love, support and giving me a non-geek perspective on things
- My family; Anna, Agneta and Mats – for bringing me up, and keeping me up – thanks for all the support!
- Tobias Hill and Patrik Fredriksson– thesis supervisors at Citerus, Java-gurus and friends
- Olle Eriksson – thesis reviewer at the Department of Information Technology, Uppsala University
- Rickard Öberg – for providing feedback, interesting discussions and for implementing a very interesting dynamic proxy-based AOP framework
- Petter Lindborg – for feedback and interesting discussions
- Lee Carver – for a good reference thesis and for being an important part of the AOP community
- Wes Isberg, Jim Hugunin, Mic Kersten and the others at AspectJ.org – very helpful and competent people, who are responsible for a lot of the work being done on AOP in general, and AspectJ in particular
- Satish Arkala – for testing the example code and for good discussions
- Rajesh Honnawarkar – for interesting input on the AOP development process and how to create a good environment for developers
- Clemens Lee – for creating JavaNCSS
- Juri Memmert – for useful information about Hyper/J
- Laurent Martelli – dito, but about JAC
- Vincent Massol – for creating the PatternTesting project
- The people at the AOSD and AspectJ mailing lists – thanks for sharing your knowledge and opinions

CONTENTS

ABSTRACT	2
ACKNOWLEDGEMENTS	3
CONTENTS	4
FIGURES AND TABLES	5
1. INTRODUCTION	6
1.1. BACKGROUND	6
1.2. THE TOPIC: AOP COMPARED TO OOP	6
1.3. HYPOTHESIS	7
1.4. ASSUMED READER KNOWLEDGE	8
1.5. THE AUTHOR	8
2. ASPECT-ORIENTED PROGRAMMING	9
2.1. BACKGROUND	9
2.1.1. <i>Object-Oriented programming</i>	9
2.2. WHAT IS ASPECT-ORIENTED PROGRAMMING?	10
2.2.1. <i>Basic concepts</i>	11
2.2.2. <i>Aspect-oriented design</i>	12
2.2.3. <i>AspectJ</i>	13
2.2.4. <i>Examples</i>	13
2.2.5. <i>Discussion</i>	17
2.2.6. <i>Problems with AOP</i>	18
3. THE DESIGN AND IMPLEMENTATION PROCESS	19
3.1. THE COMPARISON APPLICATION	19
3.1.1. <i>Background</i>	19
3.1.2. <i>The assignment</i>	19
3.1.3. <i>Enhancements to the assignment</i>	20
3.2. THE OOP VERSION	24
3.2.1. <i>Design notes</i>	24
3.2.2. <i>Implementation notes</i>	29
3.3. THE AOP VERSION	30
3.3.1. <i>Design notes</i>	30
3.3.2. <i>Implementation notes</i>	35
4. RESULTS	36
4.1. RESULTS FROM DEVELOPMENT	37
4.1.1. <i>Design methods</i>	37
4.1.2. <i>Implementation methods</i>	37
4.1.3. <i>Time spent on development</i>	39
4.1.4. <i>Code statistics</i>	39
4.1.5. <i>Discussion of development results</i>	42
4.2. CONCLUSIONS BASED ON DEVELOPMENT RESULTS	44
4.2.1. <i>Adaptability</i>	44
4.2.2. <i>Maintainability</i>	44
4.2.3. <i>Modularity</i>	45

4.2.4.	<i>Reusability</i>	45
4.2.5.	<i>Comparison to hypothesis</i>	46
4.3.	SUMMARY	47
5.	SOURCES OF INFORMATION	49
6.	APPENDICES	52

FIGURES AND TABLES

The following figures and tables can be found within the document:

Description	Page
Figure 1: Two objects communicating via a method call.	9
Figure 2: An aspect intercepts a call between A and C by using a pointcut on the call site join point.	11
Figure 3: Crosscutting functionality in two classes, which can be extracted and placed into an aspect. The aspect then uses introduction on the classes to define the method.	12
Figure 4: The OJDB package on a higher level.	21
Figure 5: Basic access control when a client calls a server to invoke a service method.	22
Figure 6: The principle of element locking during an element update in the application.	23
Figure 7: UML model of the main data objects.	25
Figure 10: Analysis model of the OJDB.	27
Figure 8: The library client model.	26
Figure 9: The library server analysis model.	26
Table 1: Time spent on development for the OOP version.	39
Table 2: Time spent on development for the AOP-version.	39
Table 3: This table shows the NCSS Code statistics for the OOP-version.	40
Table 4: NCSS code statistics for the AOP-version.	41
Table 5: Comparing the NCSS statistics of both versions.	42

1. INTRODUCTION

1.1. BACKGROUND

Software developers, designers and architects of today have a very ungrateful job in some respects. An object-oriented application, no matter how well designed and implemented, will eventually fall victim for one of the following scenarios, usually as a result of altered demands on the application, or the sheer complexity of the problem domain:

- Code tangling
- Code scattering
- Re-design – due to major changes to the problem domain. It might not always be needed to maintain a clean, modular design, but may be called for if the problem domain did not allow for a clean decomposition process in the first place.

There are of course exceptions to the statement above, as there are to most rules. By running the business objects in an EJB container that intercepts calls to and from the objects, the desired functionality may be added during the interception instead of using redesign and complicated patterns.

Furthermore, many problems can be avoided by using established design patterns, and keeping the code generally clean from spur-of-the-moment ugly solutions that sometimes seem appealing due to lack of time, external demands or other bad ideas. In the end, those kind of temporary solutions usually cause time loss instead of gain.

However, a certain degree of the above scenarios will still be a problem, even in the purest of object-oriented environments. Null pointers need to be looked out for, exceptions need to be caught, and special tracing statements within the components might be needed to trace its state, if tracing and debugging mechanisms are too complex or too weak to extract the information needed.

Also consider; what if the application is not meant to be deployed on an EJB server?

There is a need for a new line of thinking, considering the many problems developers have had to battle for decades. Applications tend to be more complex than they were a decade ago; they are often distributed client-server applications - perhaps even clustered, transactional, and integrated with all kinds of legacy systems.

1.2. THE TOPIC: AOP COMPARED TO OOP

This Master of Science thesis aims to investigate a concept that is starting to get a lot of attention in the software development community, because it introduces a whole new way of looking at software problem domains. The concept is *aspect-oriented programming* (AOP).

In this document, AOP will be described, analysed, and compared to the currently dominating concept, *object-oriented programming* (OOP). Aspect-orientation introduces a new terminology and new methods to design and implement applications, and by doing so it is supposed to, used properly, solve the scenarios with code tangling, code scattering and re-design problems that pure OOP alone cannot solve.

Two real-world applications are created in Java as a base for comparison, one using strictly typical OOP methods, one using AOP methods and the AOP enhancement to Java called *AspectJ*.

AspectJ specifics such as maturity, security, performance and stability are not included in the main analysis, only in occasional comments. The focus is on the effect aspect-oriented analysis, design and implementation has on real-world enterprise software development; especially concerning distributed applications.

1.3. HYPOTHESIS

AOP is useful for certain individual enhancements; to prove this is trivial, which is demonstrated later on in section 2.2. AOP has also been shown to be effective for general software development in [RWA02], where an application was divided into encapsulated aspectual modules that were composed into the finished application using the Hyper/J framework. Read more about Hyper/J in appendix A.

To fulfil the purpose of the thesis, a test application will be implemented in two versions, one using pure OOP methods, one using AOP enhancements to the OOP methods. The two resulting application versions will be analysed to see what advantages and disadvantages each approach may have in this particular case. Furthermore, minor independent experiments will be conducted to gain insights on technical details.

The hypothesis is that the following will hold for the AOP application version compared to the OOP version of the comparison application:

- There will be less code altogether in the AOP version, since code tangling and code scattering can be decreased or in some cases eliminated.
- Functionality will be more modular, and hence more reusable. This means that future development projects will have a greater chance of reusing code from the AOP application than from the OOP application.
- The actual code will be easier to follow and comprehend in the AOP version, as it is more modular and less code tangling and code scattering.
- The development tools and processes are more mature and well-established for OOP, resulting in less spent time for the development of the OOP version.
- The OOP version will have a better overview model of the system, as UML tools can be used to their full extent. UML tools for AOP are currently in their infancy.

1.4. ASSUMED READER KNOWLEDGE

The reader of this document is assumed to know the following:

- Object-oriented programming
- The essentials of the Java programming language
- The basics of the Unified Modelling Language (UML)
- Java Remote Method Invocation (RMI)
- The basic concepts from appendix D, as they are described there

For information on Java, J2SDK, and RMI, please go to [URL24]. To get to know more about UML get a book about it, like [UML99], or go to [URL30] for an introduction.

All references in the thesis are given using the [<source>] notation. Internet addresses are noted as [URL<number>], whereas documents, papers and books are noted as [<abbreviation of title><publishing year with two numbers>]. To find the actual source, please check the list of references.

1.5. THE AUTHOR

Magnus Mickelsson is a software architect and developer, focusing on the J2EE platform and AOP, from Uppsala in Sweden. He currently works as a systems architect and developer for H&M, where he has, among other things, been responsible for creating the system architecture of <http://www.hm.com> and assembling the company's standard J2EE developer environment. Visit the home page of his consulting company, Darkwolf Development, at <http://www.darkwolf.ws/> for more information.

Contact the author using e-mail: thesis-feedback@darkwolf.ws. Your feedback is welcome.

2. ASPECT-ORIENTED PROGRAMMING

2.1. BACKGROUND

Software development is about implementing a solution in a programming language that satisfies a *problem domain*. The solution may be an application, which is a stand-alone program that is executed for a certain purpose, or a system, which can consist of a set of services or applications working together, usually continuously executing and responding to some set of actions or events.

Depending on the problem domain type, different methods and programming mechanisms may be applied to reach the best possible solution. Aspect-oriented programming (AOP) is an interesting way of looking at problem domains, basically adding a new dimension to object-oriented programming (OOP).

Its purpose is to provide means for software developers to properly model a problem domain containing also crosscutting properties¹ in a modular way; allowing for better reusability, flexibility and understandability.

2.1.1. Object-Oriented programming

Object-oriented programming (OOP) deals with implementing an object-oriented design (OOD) in an object-oriented programming language, for instance C++ or Java. OOD/OOP is about decomposing a problem domain into a set of objects that interact by passing messages between each other. These interactions can produce a result that satisfies the problem domain.

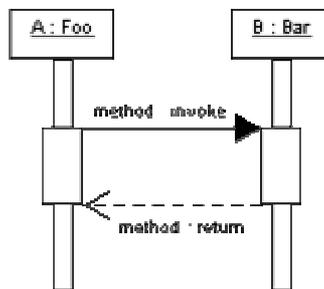


Figure 1: Two objects communicating via a method call.

¹ Such as generic system services like logging, transaction handling, concurrency and security.

The major concepts of OOP are:

- Abstraction – an abstraction deals with the outer perception of an object. It states the characteristics of an object without any focus on its actual implementation.
- Encapsulation – ensures that private object data is only accessible internally. Other objects should only access its information using available methods. The Java language uses the class construct to achieve this, as do many other languages.
- Inheritance – one class' features may be inherited by another class. By using inheritance constructs, like *extends* in Java, the same code does not need to be duplicated in order for the child to inherit the parent's functionality. Inheritance leads to a class hierarchy.
- Polymorphism – the same method may have different behaviour on different type of input objects. This can be achieved using *method overloading*, i.e. defining the same method several times for different situations. The methods may have different implementations.
- Modularity – an application should be built by developing modular, reusable classes.
- Reusability – by making components generic and modular, it is easier to reuse components. This holds for both individual objects as well as larger software libraries, creating a big opportunity to save resources.

Object-oriented thinking has been around for quite some time and has become very popular. This has lead to OOD and OOP having well defined methods, procedures, tools and design patterns. In short, it has come a long way both in theory and practice.

Even so, there are some basic problems with how OOP tries to model and realise a problem domain [OOO92]. When OO was examined more in detail, it was discovered that there are problem domains that cannot easily be expressed in OO terms, due to the fact that some of the essential functionality is crosscutting the class structure.

2.2. WHAT IS ASPECT-ORIENTED PROGRAMMING?

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier and John Irwin defined aspect-oriented programming (AOP) in [AOP97]. It was built on the research on Meta-Object Protocols (MOP) [OIM94], Open Implementation [OID97], and Reflective Programming [TTR93], but also Adaptive Programming (AP) [AOP96].

2.2.1. Basic concepts

AOP enhances OOP², hence inheriting its basic concepts; but also adds a few, very important ones at that. The primary concept, the *aspect*, is a piece of functionality that crosscuts the object hierarchy in a modular fashion. AOP enables developers keeping crosscutting functionality like access control, logging etc. cleanly separated through encapsulation mechanisms.

Aspects are, on the implementation level, constructs used to handle given points of execution in OO code called *join points*. Access to join points can be defined using *pointcuts*, which are mechanisms that allow declarative, reflective access³ to join points. So called *advise* code, which is standard code that has access to AOP implementation capabilities, can be added to the pointcut, and be executed in relation to the join point occurrence.

Aspects are *composed* with other aspects and object-oriented components by an *aspect weaver* to generate an application that solves the problem domain. Doing so in a flexible and modular way, the actual code does not need to be modified to alter the application's behaviour, merely the weaving procedure and composition element set. The image below demonstrates typical call interception by an aspect.

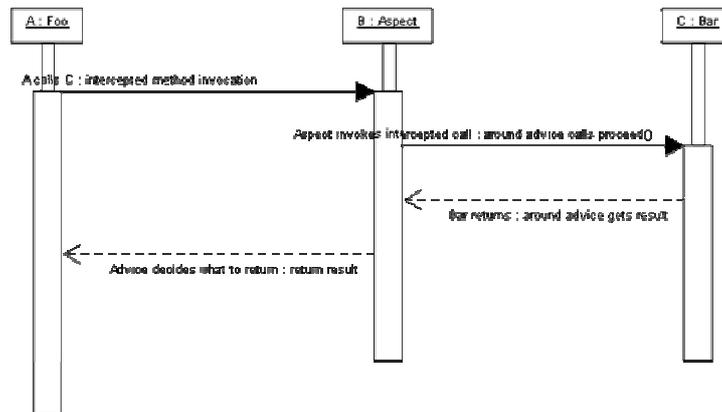


Figure 2: An aspect intercepts a call between A and C by using a pointcut on the call site join point.

Another interesting feature offered by aspects is *introduction*, which is the ability to introduce new properties to a class template. New inheritance behaviour, new methods, new fields etc. can easily be added, and the aspect can also control what kind of privacy settings the new methods and fields have (i.e. private, protected or public). Introduction will be demonstrated more in the example section (2.2.4).

² AOP can however be added to other programming methods as well, such as procedural programming.

³ This means it provides language constructs making it possible to declare a join point interception, by stating “the pointcut with the following label intercepts this join point” and making it possible to refer to the used label in other contexts. The “reflective” part means the pointcut can view and modify the properties of the join point.

2.2.2. Aspect-oriented design

Aspect-oriented design is about modularly decomposing a problem domain, both in terms of ordinary OO components and crosscutting concerns. Attributes and methods; whatever is used in a set of classes but should, from a logical point of view be a separate component, most likely belongs within one or more aspects. If some functionality is not really part of the class' main responsibility, it should be extracted to aspect utility classes, which are standard Java classes used by aspects to provide a specific functionality⁴.

As an example of aspect-oriented design, consider the following situation; objects should be created that each should hold a set of data. The objects should be able to store their current status using some persistence mechanism.

In this situation, each object's main task is to hold the data. Persistence is a typical crosscutting property that can be extracted into an aspect, as in the figure below.

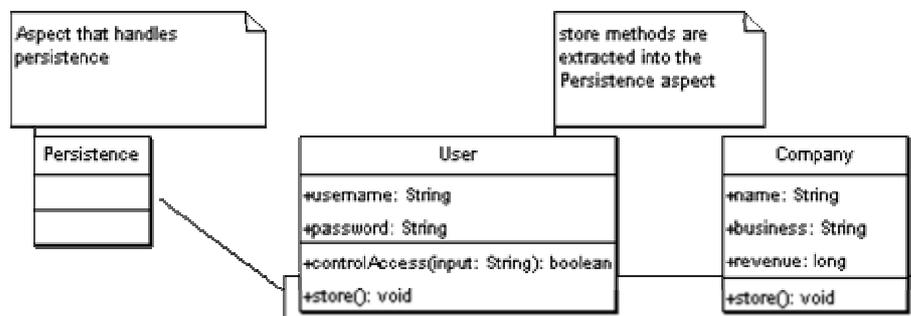


Figure 3: Crosscutting functionality in two classes, which can be extracted and placed into an aspect. The aspect then uses introduction on the classes to define the method.

Now, it is usually good to have some rule of thumb while trying to grasp new concepts, and one was given in the AOP introduction document from 1997 [AOP97], which has been interpreted as the following:

With respect to a system and its implementation using an object-oriented language, a property that must be implemented is:

- A class, if the property can be cleanly encapsulated in an object.
- An aspect, if it cannot be cleanly encapsulated in an object.

This definition, although trivial, says a whole lot about aspects. How often can a software developer **cleanly** encapsulate functionality into an object, or a set of objects? Just consider the problem with application logging once more.

⁴ If the main aspect logic is built in utility classes, it makes moving from one AOP framework to another easier, and also helps making the code easier to follow. This at least holds for AspectJ, where all the functionality potentially can be added within the aspect definition itself, as will be demonstrated later on.

After the crosscutting concerns have been identified, the designers start to analyse how method calls will take place between and in different classes. From such call trees, point cuts are identified where the aspects can inject their advice code functionality.

2.2.3. AspectJ

The framework used for AOP in this thesis is AspectJ [URL1], which is an aspect-oriented extension to Java. The reason why AspectJ was selected is that it has been developed by the same PARC team that coined the expression aspect-oriented programming (lead by Gregor Kiczales), and it is currently the most popular AOP framework, with a considerable amount of active users. Also, since it provides a new kind of sub-language to Java, it is an interesting framework to investigate, considering the advantages and drawbacks of such an addition to the Java language.

AspectJ features new language constructs to Java, such as “aspect” and “pointcut”, as the examples in section 2.2.4 demonstrate. It also adds a set of new tools for AOP development; tools like the ajc compiler, which is an AOP weaver, ajdoc, which is basically Javadoc for AspectJ, AJDE, which is an IDE add-on to offer AOP capabilities in for instance Forte/Netbeans, and ajdb, which is the AspectJ debugger.

If more information on AspectJ and AOP is needed, refer to [AOP97], [URL1] and [URL3].

2.2.4. Examples

Here are two basic AOP examples based on the AspectJ framework that aim to demonstrate aspect interception and introduction in actual code.

The first example implements basic Log4J logging on a class without any logging statements within the target class source code. The second example is a brief demonstration of introduction. Note that import statements and other matters are set aside; as the purpose with the examples is just to demonstrate basic AspectJ syntax and the possibilities of aspects. The examples are given plainly to be analysed in the following section (2.2.5).

Logging aspect

StdOut.java – a simple class that is executed invokes two methods, and then exits.

```
public class StdOut
{
    public String foo()
    {
        return "Hello";
    }

    public String bar()
    {
        return "World!";
    }

    // Main method that is called on execution
    public static void main(String[] args)
    {
        StdOut s = new StdOut();
    }
}
```

```

        System.out.println(s.foo());
        System.out.println(s.bar());
    }
}

```

Logging.java – the Logging aspect. Note the “aspect” keyword in the definition. Also note that pointcut definitions can be made using regular expressions to be able to refer to several join points from the same definition. Read more about pointcuts and pointcut definitions in [AJP02].

```

public aspect Logging
{
    // Pointcut on all method call sites not related to Aspect functionality
    // (Why not Aspects? So we avoid endless looping...)
    pointcut publicMethodCall(): call(* *.*(..) &&
    !within(Logging) && !within(Logger));

    /** This is the advice code on the defined set of pointcuts in
     * publicMethodCall() - definition.
     * Wrap the method call(s) so we can log that it is about to be run,
     * the parameters it will run with and the result we get from it.
     * "around" is instead of the method call, alternatives are
     * "before" or "after".
     */
    Object around(): publicMethodCall()
    {
        // Log information about the currently intercepted join point
        Logger.log("Log alert for " + thisJoinPoint);

        // Print the method parameters by calling the method below
        printParameters(thisJoinPoint);

        // Execute the wrapped method and catch the result
        Object result = proceed();

        // Print result, if it is not null
        if (result != null)
            Logger.log("Result of " + thisJoinPoint + ": " + result.toString());

        return result;
    }

    /** Print the method input parameters
     * @param jp the join point a pointcut has caught an event on
     */
    static private void printParameters(JoinPoint jp)
    {
        StringBuffer buf = new StringBuffer();
        // Reflective access to join point parameters below
        Object[] args = jp.getArgs();
        String[] names = ((CodeSignature)jp.getSignature()).getParameterNames();
        Class[] types = ((CodeSignature)jp.getSignature()).getParameterTypes();

        buf.append("Arguments: \n");

        // If there are any arguments, print them
        if (args.length > 0)
        {
            // Go through the arguments and print them out
            for (int i = 0; i < args.length; i++)
            {
                buf.append(" " + i + ". " + names[i] + ": " +
                types[i].getName() + " = " + args[i] + "\n");
            }
        }
    }
}

```

```

        Logger.log(buf.toString());
    }
}

```

Logger.java – the class that handles the actual logging, by using the Log4J [URL8] framework. It is a special utility class for the aspect.

```

public class Logger
{
    private static Category cat = Category.getInstance(Logger.class.getName());

    /** Log a message on debug level in Log4J
     * @param msg String message to log
     */
    public static void log(String msg)
    {
        cat.debug(msg);
    }
}

```

After constructing the source code above, the aspect weaver is used to join the composition elements, the source code files, into a new set of source code that contains the aspect influenced code, which is then compiled into Java byte code. Thus, the crosscutting concern logging, which was encapsulated in the Logging and Logger source files, have been added to the StdOut class, using interception and the pointcut's reflective properties. The process is transparent to the user, who might believe that just a normal compilation took place. The example above, when assembled together by the aspect weaver and executed, will print something like this:

```

0 [main] DEBUG  Logger  - Log alert for call(String StdOut.foo())
0 [main] DEBUG  Logger  - Result of call(String StdOut.foo()): Hello
0 [main] DEBUG  Logger  - Log alert for call(void
java.io.PrintStream.println(String))
20 [main] DEBUG  Logger  - Arguments:
0.__0: java.lang.String = Hello
Hello
20 [main] DEBUG  Logger  - Log alert for call(String StdOut.bar())
20 [main] DEBUG  Logger  - Result of call(String StdOut.bar()): World!
20 [main] DEBUG  Logger  - Log alert for call(void
java.io.PrintStream.println(String))
20 [main] DEBUG  Logger  - Arguments:
0.__0: java.lang.String = World!
World!

```

If the example had been run as plain OOP, it would have printed simply:

```

Hello
World!

```

Aspect introduction on a class

Below is an example of basic introduction on a class, using the AspectJ syntax:

```
public class TestClass
{
    private int one;
    public String two;
    public void setString(String input)
    {
        two = input;
    }
}

public aspect Introducer
{
    protected static String TestClass.message = "Hello World";

    public String TestClass.getString()
    {
        System.out.println(message);
        return two;
    }
}
```

This Introducer aspect defines an introduction of a static field called message on TestClass. It also defines the introduction of a new public method that returns the “two” variable and prints out the introduced “message” variable. When the files have been woven together by the weaver, the compiled TestClass class will contain the introduced field and method.

For more information on introduction using AspectJ, please refer to the AspectJ programming guide [AJP02].

2.2.5. Discussion

The examples were meant to demonstrate some of the features of aspect-orientation, although they only reveal small parts of what makes it an interesting technology. In the first example given previously, two simple method-calls being printed out caused a lot of activity. Each of the methods was wrapped - the input and results of the method calls were logged using the Logging aspect, eliminating the usual scattered logging statements that would have been called for in ordinary OOP. This can be applied also on a larger scale; one Logging aspect could handle the logging for an entire application. One thing to notice though is that the logging in this case is not specific to certain conditions; it will instead gladly log everything the same way, compared to scattered logging statements, which are more informative of local conditions.

AOP offers several ways to trace also more local conditions; either using local-aware log interceptors for specific pointcuts, targeting for instance a single method, or by analysing generic results (such as in the example) and providing specific error messages based on the analysis. Another method is to make sure that specially crafted exceptions are thrown in some situations that can be caught by an aspect, and dealt with it as seems fit, mapping to reflect upon local conditions.

Another interesting notion AOP offers, is a high degree of flexibility when testing developed applications. Imagine just having to test the actual business code of an application, and then apply the aspects later one at a time, adding more and more functionality. The aspects have all been tested separately to concur with an expected behaviour. The flexible addition of aspects on the applications can provide add-on testing of plug-in functionality when the core has already been approved, and thus need not be changed as a result of test failures. Only the interaction between aspects and aspect target objects need to be analysed.

The concept of introduction is also a very promising one; as it offers aspects the possibility to add encapsulated behaviour to a class, behaviour that is specific for the functionality that the aspect is adding to the system. This is a very important part of being able to encapsulate crosscutting concerns, meaning that a system without aspect behaviour in the AOP design world is not going to be able to do very much. But, as soon as some aspects are added, the behaviour will form a more complete system.

The impact of AOP should be a higher level of reusability since components can be made more generic and be strictly encapsulated, leaving the addition of crosscutting functionality and behaviour composition to also encapsulated and generic aspects.

2.2.6. Problems with AOP

The general problem with AOP is that it adds a new dimension of possibilities, while at the same time adding an extra dimension of complexity. Thus, it is essential that AOP is kept simple yet powerful to focus on its possibilities instead of detailed complexity, so also ordinary developers are able to take advantage of the advantages of AOP tools and methods. More powerful and complex features can be made available, but should only need to be used by power-users.

AOP complexity derives from the fact that there are several new mechanisms to work with, and if not handled correctly, they can wreck more havoc than they do well. There is for instance a risk in the AspectJ framework of ending up in infinite loops, in the case where an aspect catches events on itself, or its related utility objects. The new language constructs in AspectJ are not trivial either. Once they are understood, they are very powerful, but the learning curve is not to be underestimated.

Although unfortunately not being correct in all respects, [CAO02] makes a presentation of a few important challenges to AOP:

- Emergent properties and fault resolution; hunting bugs in AOP code will mean tracing and debugging become an even more complex activity, if not the proper tools or able employees are available to find them.
- Understandability; the complexity referred to above.
- Implicit changes in syntactic structure and semantics; applying aspects on a system should not wreck the system's core functionality and contracts.
- Effects of cognitive burden; aspects woven with aspects might cause unexpected behaviour.

The latter point is above all a problem in AspectJ, where aspect interactions like precedence determination are encapsulated into the actual aspect definition, but is valid for other frameworks as well.

The design of an AOP application is a bit foreign from standard OOD. It involves a new line of thinking; similar to the paradigm shift that occurred when going from procedural programming to OOP. It is important to give designers and developers time to adjust to the new paradigm and get a feel for the philosophy behind it, before throwing people into demanding projects with lack of time to reflect upon the new concepts.

Another important issue is what to do about application modelling. No mature, finished solutions exist today, but a lot of work is underway destined to fill the current void [UNA02, DAC02, and URL6]. Later on, also expect AOP to be added to existing methodologies like Agile and RUP.

3. THE DESIGN AND IMPLEMENTATION PROCESS

This section describes the requirements of the test application, and the design and implementation processes that lead to two executable versions of the thesis comparison application.

3.1. THE COMPARISON APPLICATION

Here, general information can be found about the application's requirements, the origin of it, and the enhancements made to the requirements for the purpose of this thesis.

3.1.1. Background

The comparison application originates from a course at Uppsala University called "Object-oriented programming with Java" (OOPJ), where it was one of the mandatory assignments students had to implement in order to pass the course.

The assignment introduction stated the following:

"The idea about this assignment is that you should be given opportunity to practice analysing, designing and implementing a larger example using object-oriented methods."

It was selected since it is a good candidate for object-oriented design and implementation, obviously since it should not have been selected as an assignment for an OOPJ university course otherwise. Furthermore, it has crosscutting functionality that could test the usability of aspects.

There are also a few seemingly harmless additions to the requirements that have been made for this thesis, in order to make the assignment more like a real-world application.

3.1.2. The assignment

This section sets the basic requirements for the application, based on the assignment text that was handed out to the OOPJ students of the spring semester 2002. It has been translated from Swedish, and a few things have been added or removed to fit the scope of this thesis better.

The assignment instruction

A smaller library is about to convert their business from being paper managed to a computer managed solution. They want a system where all lendable objects; books, CD:s and magazines, are stored in some form of record. Every library object has a unique number, which is used for identification, and is either lent out or available. For lent out objects, information about the user who has lent the object must be available.

Books should have information about the author, title and category, magazines about title and volume (which number and year), CD:s about artist, title and total playtime.

The record should be searchable. Searching for substrings should be possible; if a book with the title “The rise and fall of the Roman Empire” exists, it should be found if for instance the search word is “Roman”.

Of course, there should be methods for lending and returning library objects. The user should be able to pick an object from a result list for lending.

It should be possible to add, remove and edit objects in the library. When an object is added, the unique library ID should be created automatically – this could easily be handled by the base class knowing the currently highest value and simply increasing that number.

When the program starts, the record should be read from the hard drive. When the program finishes, the program should save the record to the hard drive.

The assignment is to write a program to solve the requirements above. The program should have an object-oriented solution. The result should be a well documented and executable program written in Java.

Furthermore, it is specified what should be present in the report the students have to hand in in order to complete the assignment, but that is really of no importance here.

The classes that MUST exist are:

- A base class for library objects.
- The classes Book, CD and Magazine, that inherit from the library object.
- A class for the text-based user interface, Menu.

The program should handle exceptions in an acceptable manner, yielding in an error message of some kind. If so desired, a graphical user interface (GUI) can be developed, but a command prompt UI is sufficient.

3.1.3. Enhancements to the assignment

Graphical User Interface

In order to provide more real-world requirements, it was decided that the application should contain not only a command prompt (text-based) UI, but also a web-based GUI utilizing the Java Server Pages (JSP) technology. In both interfaces, administrators should have more available functions than ordinary users.

General purpose database

The assignment preceding the library assignment was to create a more general purpose, simple text-based database. The students of the OOPJ course were therefore encouraged to reuse code from the previous assignment, to make the library assignment implementation a bit easier. The same kind of reusability thinking is part of the foundation of object-oriented programming, and should be a part of this assignment as well.

The assignment is therefore enhanced with the demand to build the library application on top of a more general purpose, text-based database, that if needed can be reused to implement a different application. The database is called “Open source (simple) Java Database”, OJDB in short.

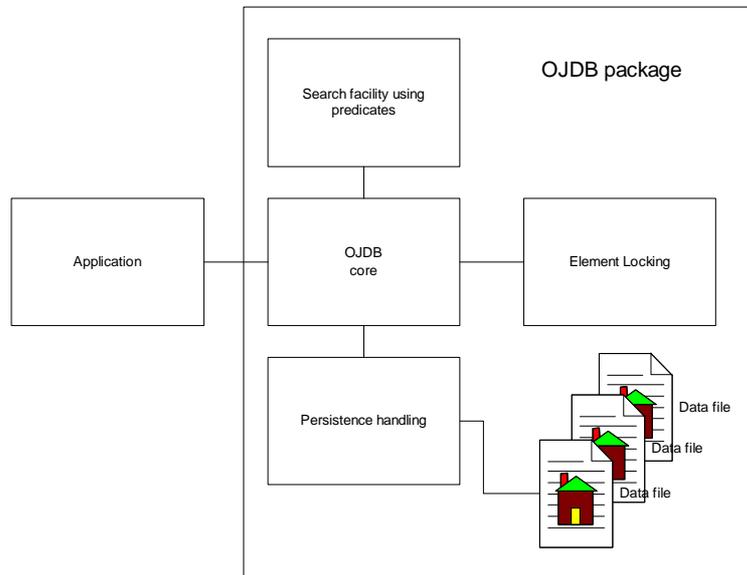


Figure 4: The OJDB package on a higher level.

Access control

The concept of access control, having several users with different sets of access rights, is added to the assignment as it is very common among applications of this category. As a consequence, there must be different user interfaces for administrators and ordinary users, as there are some functions available that ordinary users should not be able to access (influencing the actual contents), such as user data, access rights and the information content of the library.

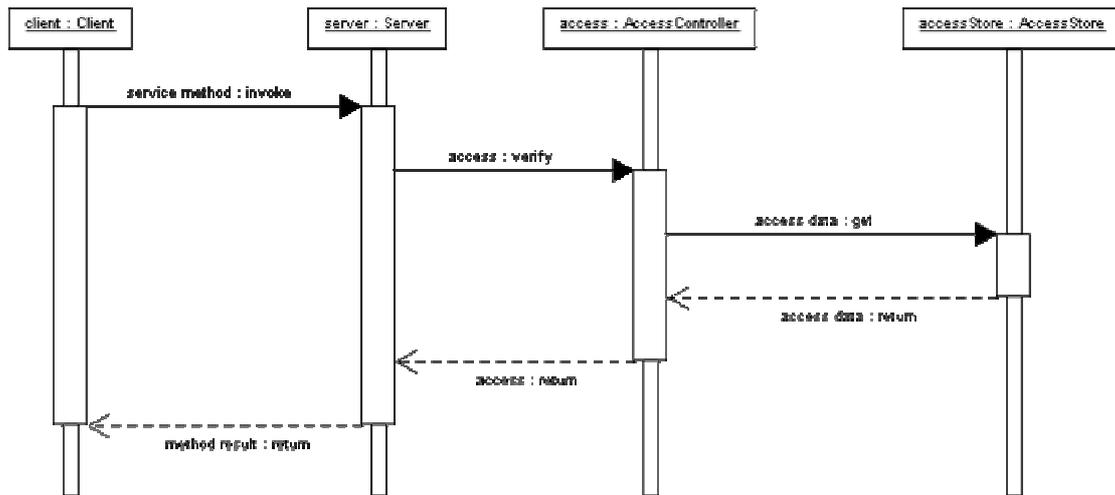


Figure 5: Basic access control when a client calls a server to invoke a service method.

Logging

Logging is added as an enhancement. All important components should be able to log their state and progress. The logging framework Log4J [URL8] was chosen to provide logging functionality, as it allows logging on different levels, debug, information, warnings, error etc, and is the de facto standard component in the Java world for logging. Whatever level is selected (edited in a property file) is the threshold for what is logged. If the level is WARN, then only error and warn levels are logged if those are the only higher levels implemented. Information and debug level statements are ignored in such an event.

Element locking

In a real-world library, one cannot assume that there will be only one user at a time. Therefore, the assignment was enhanced with the concept of *element locking*, which means that when a user calls an update method, the element being updated is locked during the update, so no other user can change it at the same time.

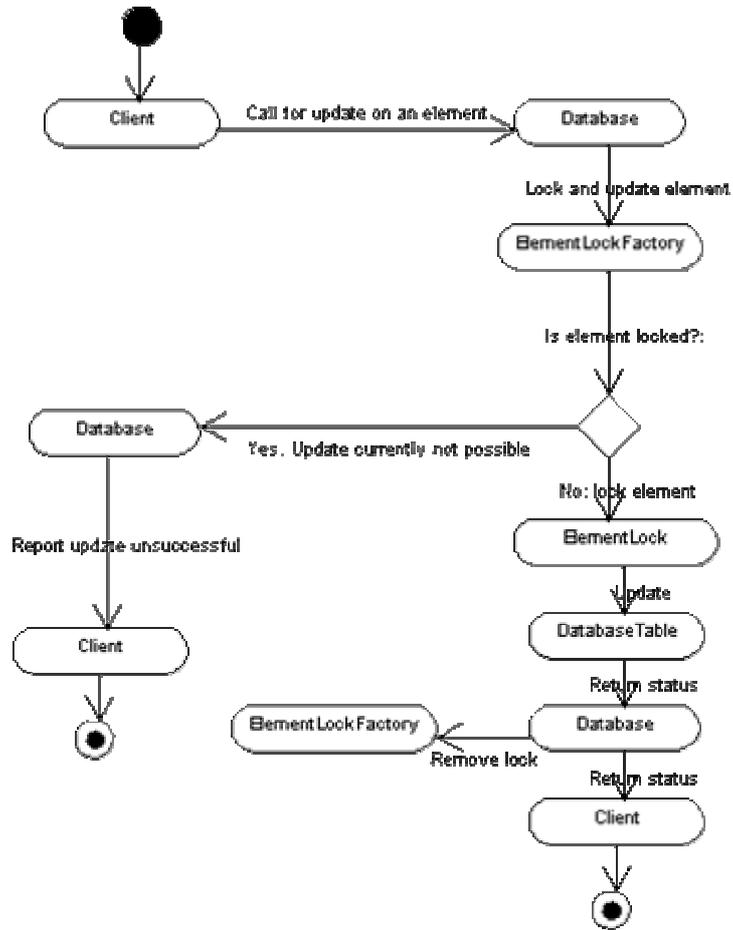


Figure 6: The principle of element locking during an element update in the application.

3.2. THE OOP VERSION

This section describes the actual object-oriented software development process that solved the problem domain of the thesis application requirements.

3.2.1. Design notes

When designing an OOP application, the following steps should be taken:

- A. Analysis: Find the basic business objects needed to get the basic functionality running, in accordance with the problem domain. Detect any sub-objects that are needed by the basic business objects, for instance a database object will probably need database tables, primary keys and some operation objects. One easy way to do it is to write down what the system must do and turn the nouns into entity classes and the verbs into operations on the entity classes. It will not make a perfect model, but a good starting point. Objects are added to class diagrams in UML. Once this has been completed, it's extremely important to review the class diagram to see if the existing objects really are cleanly encapsulated. If not, make the proper changes until there is a modular, clean-cut architecture. For now, focus should not be on object interaction, rather the types of objects needed and what basic services they should offer.
- B. Reuse analysis: Find out if there are any objects that can be reused from other projects.
- C. Design: This is where the idealised model from the OO analysis is made more concrete. The design phase is divided into the following:
 1. System design, which is where decisions are taken about the qualities of the system as a whole, to a certain level of detail (variable according to taste).
 2. Object design, which is about adding details to the objects of the analysis phase.
- D. Interaction: Model the calls that will be passed between the business objects. This will lead to interaction diagrams in UML. Whenever functionality is needed that is not part of the object's core functionality, add new objects that implement the add-on functionality and include calls to them where it is needed. In some cases, non-core functionality may have to be added to an object.

Now it is time to examine the above steps of the design process more in detail, from the view of the comparison application.

A. Analysis

The analysis phase started with identifying the basic objects and functionality of the system, based on the problem domain, which in the design phase were detailed. The basic idea was to make the application an RMI client-server application, that was built on top on a reusable, generic, text-file based, and simple database handler, reused according to the assignment.

The data to be handled within the system was specified in the assignment, and resulted in the following UML model:

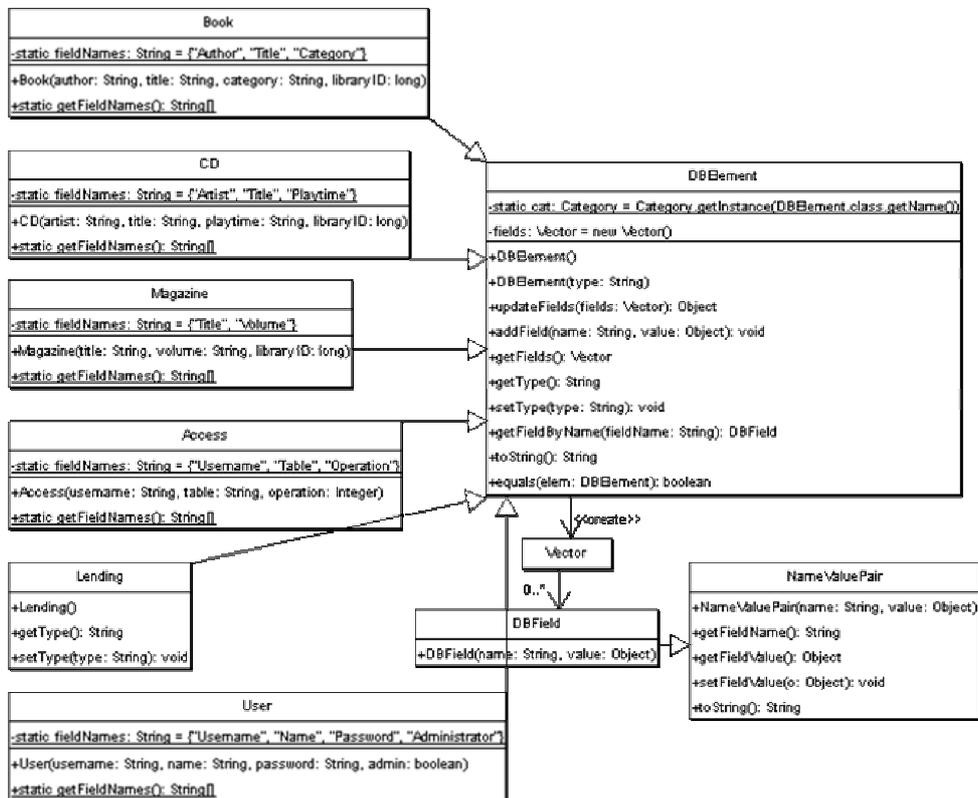


Figure 7: UML model of the main data objects.

The DBElement is a general purpose data carrier that holds a Vector consisting of DBField objects, which are basically name-value pairs. DBElement corresponds to the base class for all library objects, CD:s, Magazines and Books, but also other database objects, like Lending, User and Access, who all inherit DBElement. This construction satisfies the assignment requirements, and also promotes reuse. Note that Lending is the object that holds a lent out library object reference, and the user name of the person that lent the object. Access holds access rights and User holds user information.

The general purpose, simple database foundation was already designed and implemented, and therefore reused. The actual construction of this part was not included in measurements of implementation time, as it was already available according to the assignment specification⁵. Therefore, analysis of the database itself was not needed; but an analysis was required on how to use it.

⁵ Reusing an already constructed, simple database that uses text files for persistence.

The client that connects to the remote library server (IRemoteLibrary is its remote interface) was analysed as below:

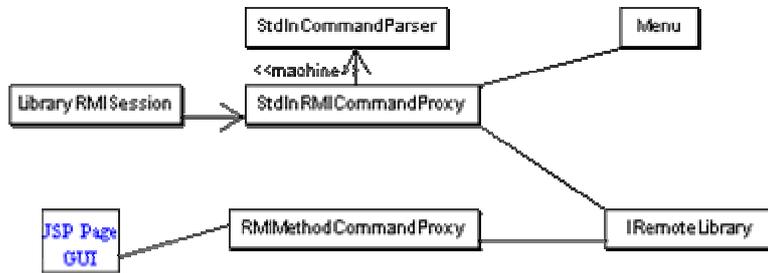


Figure 8: The library client model.

Command proxies are objects that interact with the UI and provide them with what they need, by invoking methods on other objects, like the StdInCommandParser, which enables reading the user's command prompt input, the Menu, which holds the text-based menu and StdOut result printing, or the actual library server remote interface, which defines the server's service methods. As specified in the assignment, there are two UIs.

The recently mentioned library server was analysed as below:

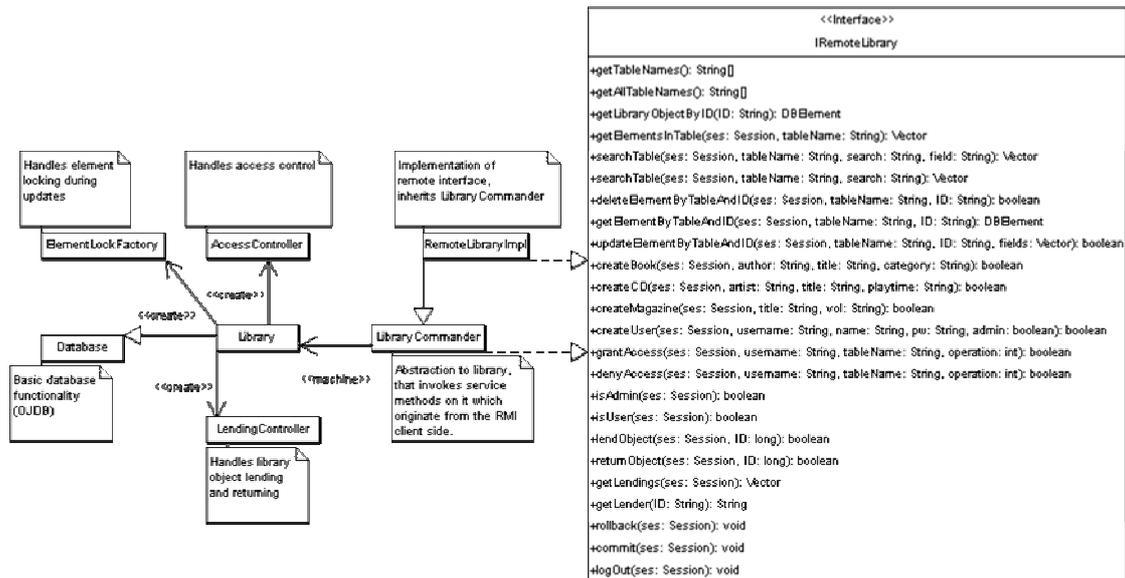


Figure 9: The library server analysis model.

The library server listens to requests and passes them on to the LibraryCommander, which is an abstraction for the actual Library object. Looking back, the LibraryCommander was not all that necessary, since no rigorous status handling was implemented. It might however become useful in such event, or if the problem domain is changed significantly.

The Library uses utility classes to add advanced functionality, such as access control, element locking (during element updates), object lending and returning handling. Furthermore, it extends the functionality of the Database object. Hence, the assignment requirements about access control, persistence, element locking, using a general purpose database and lending are satisfied.

As for logging; Log4J provided the classes to use. The code scattering threat of logging was ignored for comparison's sake, to ensure the application was developed as it might have been by other developers unaware of crosscutting concerns.

B. Reuse analysis

It was assumed that the OJDB classes already existed, so they could be reused, as specified in the assignment. However, other classes that are somewhat generic, like the classes dealing with predicates, predicate filtering, object transformations and the StringToHTML converter, are candidates that could have been easily reused, had they been available.

Apart from the mentioned areas, there is little chance of reuse unless there has been a very similar project before. Note that this is only the reuse analysis for what could have been an input to this project, not what is possible to reuse in future projects. That kind of analysis belongs in the results section.

C. Design

C1. Object design

The OO design models are available separately, in appendix H. However, since the database was mentioned before but not specified, here is a basic view of the OJDB design:

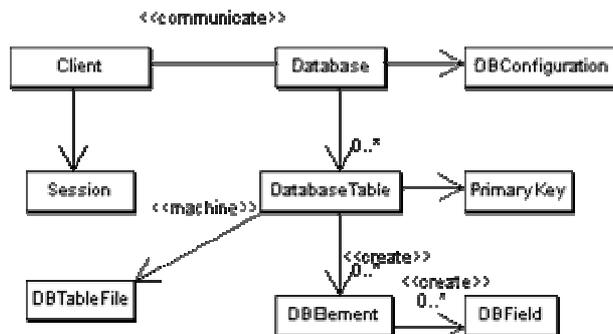


Figure 10: Analysis model of the OJDB.

Some explanations for figure 10:

- The Session is a user session object. Session instances work against a singleton database.
- A Database holds one or more DatabaseTables and one Configuration object.
- A DatabaseTable uses Persistence mechanisms, and holds a PrimaryKey (to give elements ID's) and one or more DBElement objects (data carrying objects).

C2. System design

The application was divided into four sub-systems; the generic database, the library server, the library client (UI) and generic tools. The latter was merely to encourage component reuse by placing completely generic functionality, such as String to HTML-code conversion, in a completely separate Java package, later turning into a separate Java archive file (JAR).

The library client and server were distributed, as that is the area of analysis for this thesis; the server became an RMI-server, registered with the RMI registry (via JNDI), and the client became a RMI-client accessing the RMI-server using its stub.

Log4j [URL13] was chosen as a logging framework for the application, as mentioned. Its property file was placed with the rest of the required JAR-files and included in the class path. Needed policy files and RMI properties were handled the same way.

D. OO interaction

OO interaction diagrams are available in appendix H. However, the focus was not on creating an exemplary UML model considering all aspects of UML, so the model may seem a bit scarce.

3.2.2. Implementation notes

The generic database

The generic OJDB database was reused entirely from an application that had been handed in for the OOPJ course. It is not a good database, just a simple enough foundation for this thesis.

The RMI library server

As always with RMI-applications, the design decisions taken can become critical. It is essential to cleanly separate the client and server, otherwise there is a risk of transferring too many or large objects between the JVM:s, causing terrible performance.

A clean separation was accomplished in this case, with only elements or sets of elements (within a Vector) transmitted. In order to increase performance, a client-side caching/invalidation mechanism could have been created, but that was not within the scope of the assignment.

The remote interface, `IRemoteLibrary`, defines the available service methods. The `LibraryCommander` implements the interface, and the remote implementation object, `RemoteLibraryImpl`, just extends `LibraryCommander`. The `RMIManager` handles the JNDI registry and registers the server object.

The RMI library client

The client comes in two versions, one text-based and one web-based, using JSP pages (the web-based UI is described more in detail further on).

Text client

The text based client that uses the command prompt for input is an instance of the `LibraryRMISession` class. It uses the `StdInRMICommandProxy` to execute commands on the library, which the proxy parses and handles accordingly by delegating calls to the proper objects, but also to retrieve user commands utilising a `CommandParser` instance.

Web client

The web based client is a set of JSP pages, which use the `RMIMethodCommandProxy` to execute commands on the library. The proxy retrieves a reference to the RMI server's remote interface, which it then allows the JSP pages to get a reference to so they can invoke methods on it.

3.3. THE AOP VERSION

3.3.1. Design notes

The application was divided into the same four major packages the OOP-version had, except that a new package was introduced; aspects. Therein, all aspects were located. Aspects that are specialised on the OJDB were placed in a sub-package called “ojdb”, and similarly for the library-specialised aspects.

When designing the AOP version of the application, the following steps were taken:

- A. Analysis: The same as for the OOP version.
- B. Reuse analysis: The same as for the OOP version.
- C. Design: This phase is quite different from the OO design; however, as in the OO case, basic system design and object design are included. Its subphases are:
 1. Basic system design: The basic parts of the system should be identified. It takes place this early in the process, because aspect decomposition is easier with the system as a whole in mind.
 2. OO interaction: Model the calls that will be passed between the business objects from the analysis phase. This will lead to interaction diagrams in UML. If a situation occurs where a destination object will be called by more than one kind of source object, add the destination object to your list of *aspectual object candidates* (objects that might be used by aspects to implement the aspect’s core functionality). Then add the source objects to your list of *aspectual target candidates* (objects that might be in need of aspectual, cross-cutting functionality). The reverse situation, one source object calling several types of destination objects, does not mean the object is an aspectual candidate. Remember to always keep notes on what methods or fields are in focus, i.e. will be needed by aspects or influenced by aspects.
 3. Object responsibility analysis: Consider the problem domain for what is missing between the basic business object interactions and the problem domain to get a solution that satisfies it entirely. Any object that is in need of completing its core functionality is added to the list of aspectual target candidates.
 4. Aspect-object decomposition: Decompose the missing pieces, which are the rest of the aspectual object candidates, into cleanly encapsulated objects, even though they may need to be applied across several objects in the end. It helps to consider these candidates as services that will provide the missing functionality in your system. The decomposed AO objects, like for instance authorisation control, are added to the class diagram, preferably somewhat separate from the pure OO classes.
 5. Aspect design: Take the list of aspectual object candidates and find the crosscutting concerns for the aspectual target candidates. Just pose the question “is this object’s functionality part of the target object’s core functionality?”, and if the answer is no, an aspect has been discovered. From there on, it’s just a matter of trying to keep the aspects cleanly encapsulated, and still be able to provide the missing functionality of

the pure OO system. Aspect inheritance hierarchies need to be found, to promote reuse of generic aspects.

6. Aspect modelling: This step is about completing the draft UML models with the aspect-related material. There are currently several suggestions available on how to handle UML modelling of aspects [UNA02, DAC02, and ADC02]. Since there is no single standard and no available UML tool mature enough, a simple version of the approach of Suzuki and Yamamoto can be used if creating models in a tool that cannot handle aspect-orientation. This basically means that aspects are noted by using the “<< realize >>” stereotype. As an extra addition, in order to be able to tell the difference between a realisation class and an aspect, the name of the object should indicate whether the object is an aspect or not.

This means that there are more steps to the design process of the AOP version; should it then take more time to finish? Since the design is more modular, even regarding crosscutting concerns, it should be easier to express the system behaviour, whereas for the OOP version some functionality will get tangled and complex due to the application’s nature. Thus, the more complex the system, from a crosscutting perspective, the more efficiency an aspect-oriented approach will demonstrate.

The result for the software development process in this thesis was that the AOP version had longer first design iteration, but instead fewer iterations altogether. The first iteration being longer might just depend on the lack of AOD experience of the designer, so the conclusion must be that the AOP version had an easier design process, which was not expected.

As in the OO case, the design phases are now analysed more in depth.

A. OO analysis

The basic classes of the OOP and the AOP system were made very similar; there are no obvious differences on the surface. The general thought was to keep all basic classes modular, to the extreme in fact, not considering where the base class was to fit in. That was thought better handled using introduction, and composition of functionality through the weaving process.

As an example, the Database object’s tight couplings to element locking, access control and persistence were made loose; they were handled as if developing extremely generic services.

On the lowest component level, i.e. the utility classes performing the actual work, there was hardly any difference besides some removed code tangling and code scattering; however the binding of functionality onto higher level classes was quite different.

With AOP, it is quite possible to add the support of generic system services at any join point, making security checking, logging, tracing, and element locking optional add-ons by including or removing their class file reference within the weaver’s source code input file. The definitions of which join points to intercept are defined within the aspects, at least using AspectJ.

B. OO reuse analysis

As in the OO case, it was assumed that only the OJDB database existed and could be reused.

Since reusing OJDB could not be done in its purest OOP form, the OJDB classes were changed a bit. The biggest change was cleaning the classes from logging statements, but statements using generic services not part of an object's core functionality were extracted, as those were good candidates for replacement by an aspect providing the removed generic system service behaviour.

C. OO design

C1. Basic system design

The application was divided into five sub-systems; the generic database, the library server, the library client (UI), generic tools, and aspects and their utility classes, which were on compile woven into the same four major sub-systems as for the OOP-version.

The client-server RMI setup was quite similar to the OOP-version.

Log4J was used for logging purposes in this version as well, although now made completely modular and independent of the rest of the application, not reflecting specifically on local conditions.

C2. Object interaction

This part of the design process was handled by drawing basic interaction diagrams, where only basic objects were considered. Whenever more functionality was needed at a call site or in an object, that functionality was added either through interception, or through introduction. Intercepted join point's advice code may also require call site interception and introduction, in such cases reflecting on the need of "wormhole" functionality; where aspects may need access to other aspects. Since the investigation of wormhole effects was not within the scope of this thesis, it is left for others to investigate. Note that at this stage these changes were only conceptual, for establishing aspect candidate functionality.

C3. Object responsibility analysis

The object interaction diagrams were analysed to determine what aspect candidate functionality was in fact part of an object's main responsibility. If such functionality was found, it was added to the object itself, and hence removed from the aspect candidate list.

C4. Aspect-object decomposition

This part consisted of locating ordinary objects that could serve as utility objects to crosscutting aspects. Such typical objects were ElementLockFactory, ElementLock, and AccessController.

C5. Aspect design

From what was left of the diagram, encapsulated and modular aspects were decomposed, which use the aspect utility objects to achieve their intended functionality. Introductions that related to interception functionality were placed in the same aspects. Also, reusability of the aspects themselves was kept in mind, sometimes calling for adding abstractions to the aspects.

The following aspects were created:

- AspectList – an aspect keeping track on what aspects and aspect utility classes are available, so there are no unnecessary infinite loops. It defined a pointcut set definition, used by other aspects.
- Tracing and LibraryTracing – aspects used to trace calls. Tracing is an abstract, more generic aspect that LibraryTracing inherits and adapts to the library application.
- Logging and LibraryLogging – logging aspects utilising Log4J. Logging is abstract, LibraryLogging the application specific aspect.
- PerformanceTiming and LibraryPerformanceTiming – same as above, but the aspects enable performance timing on any method call.
- ExceptionHunting – an aspect that keeps track of invoked methods, parameters and method results. As soon as there is an exception, it gives information about the problem join point and the preceding call.
- Parsing and ElementParsing – aspects that define how and when to parse stored library data into active objects.
- TablePersistence – defines persistence of library objects.
- ElementLocking – adds element locking during updates.
- Security – makes sure administrator methods are run only by administrators.
- SecurityTableEditing – enables user (administrators only) control of users and access rights.

C6. Aspect modelling

The design could not result in an UML-model in the same way as the OOP-version, as there was no stable and mature UML tool available that could handle aspects. Thus, in order to view the object design, the generated Javadoc or the actual source code needs to be examined.

As an example of how the design of the AOP version was different from the OOP version, here are two code snippets of the same method; the first from the AOP version's Library class:

```
/** This method removes an element from a table.
 * @param table name of table to remove object from
 * @param object object to remove from the table
 */
protected synchronized boolean removeElement(Session ses,
String table, Object o)
{
    return super.removeElement(table, o);
}
```

And the second from the OOP version's Library class (same method):

```
/** This method removes an element from a table.
 * @param table name of table to remove object from
 * @param object object to remove from the table
 */
public synchronized boolean removeElement(Session ses, String
table, Object o)
{
    if (access.hasAccess(ses, table, access.DELETE))
    {
        DatabaseTable tmpTable = super.getTableByName(table);

        if (tmpTable != null)
        {
            tmpTable.removeElement(o);

            return true;
        }
    }

    return false;
}
```

From this example, it is quite clear that the OOP code had to be tangled to perform access control, while the tangled conditions became separated aspect behaviour in the AOP case. The removeElement method in the Database class (which is extended by Library in both versions) looks exactly the same in both versions. In the AOP case, checking if the table to delete an element from is null is also something that can be handled by an aspect.

3.3.2. Implementation notes

The generic database

The result of implementing the generic database was the same classes as before, but the compositional behaviour was changed. The services which had been decomposed into aspects were applied dynamically at compile-time. Note, that not all services were decomposed. The lending and returning mechanisms of the library were judged as a core part of the library's functionality and thus kept as it was. That is a design decision that can be questioned, as it might as well be decomposed into an aspect as well. The chosen way was taken for understandability's sake.

The RMI library server

No special aspect introductions were made on the RMI-components, although the thought of a generic interface that could be altered into a remote interface using introduction was appealing. It was tested and was proved to work, but it requires some considerations on how to handle remote object unavailability and other typical RMI exceptions.

Separate experiments showed that in order to make RMI work; it is a prerequisite that the exact same classes are available to both client and server; including the AspectJ runtime classes. No such classes are transmitted across the network however; just the objects intended to be transferred. The only unexpected thing was the amount of extra byte code that is sent due to AspectJ's cumbersome additions to the code. Measurements of up to 500% more network data compared to the OOP case was not uncommon, if most development aspects were applied. Even if only production aspects were applied, the difference was still apparent, and should be taken into account.

The RMI library client

The client comes in two versions, one text-based and one web-based, using JSP pages (the web-based UI described more in detail below).

Text client

The text-based client works the same way as for the OOP version.

Web client

The JSP pages are compiled into Servlet classes at run-time, but are until that moment not real Java-code; hence they could not be affected by aspects. If so had been desired, the JSP pages could have been made Servlet classes, or JSP pages could have been pre-compiled into Java code, which would have allowed for aspect composition. That was not performed here, as the task specified to use a plain JSP GUI, and other approaches were deemed as not within the scope of the thesis.

The web-based GUI

The JSP pages in the web-based GUI are the same as in the OO-version, except they have minor changes in them to reflect the Library API changes compared to the OO-version. Due to the fact that JSP:s could not be affected by aspects in this case, no special considerations were needed.

4. RESULTS

The applications and their respective development procedure have been analysed in detail, with respect to actual implementation results (4.1):

- Design methods.
- Implementation methods.
- Time spent on development.
- Code statistics.

And more theoretical conclusions (4.2):

- Adaptability, how well the implementation adapts to new demands.
- Maintainability, how easily the implementation is administered.
- Modularity, how well functionally separate code can be separated into own components.
- Reusability, how easily components can be reused in other applications.

Based on the results, some advantages and disadvantages of AOP development compared to OOP development have been noted.

Furthermore, a special section with comments on AOP and AspectJ more in general, supported by the results of this thesis, independent testing, and discussions with AOP community members, was written (section 5).

The hypothesis for the thesis was that the following would hold for the AOP application version compared to the OOP version of the comparison application:

- There will be less code in the AOP version, since code tangling and code scattering can be eliminated.
- Functionality will be more modular, and hence more reusable.
- The code will be easier to follow in the AOP version.
- The development tools and processes are more mature and well-established for OOP, resulting in less spent time for the OOP version, and better possibilities to debug and profile the code.
- The OOP version will have a better overview model of the system, as UML tools can be used.

4.1. RESULTS FROM DEVELOPMENT

This section holds results achieved by performing the design and implementation of each application version, and compares them to the statements of the hypothesis.

4.1.1. Design methods

Advantages of AOP design methods compared to OOP design methods:

- The analysis process was more intuitive and easy, as it was only the question of finding basic encapsulated entities and services. They were later on refined and composed into the final application using the AOP tools.
- Difficult crosscutting design issues could be easily expressed and accounted for.
- The designed objects were modular and more pure in design, as opposed to the OOP version, allowing for more flexible behavioural changes.

Disadvantages of AOP design methods compared to OOP design methods:

- The actual design process is not mature yet as no standard has been agreed upon. Each designer currently needs to invent their own process, instead of for instance relying on RUP.
- There are to this date no mature, good UML tools for modelling introduction, interception and other AOP traits.

It is worth noting that both design processes had the main purpose of creating a clean design with a high possibility of reusing functionality from generic classes (and aspects, in the AOP case).

4.1.2. Implementation methods

Advantages of AOP implementation methods compared to OOP implementation methods:

- Debugging and tracing was easier, as development aspects were used to focus on interesting code and extracting all used calls, input parameters and method results from it. Bugs were found faster using this approach than for the OOP case.
- The class files' code was as much as possible kept clean from crosscutting issues, exception handling and null reference checking. This made coding faster and easier.
- Viewing crosscutting issues was made much easier through the AJDE environment, above all the crosscutting view that was generated upon a compile. The AJDE tool is easy to use, and easy to understand.
- If the design was to be altered somehow, it was usually sufficient to alter a few pointcut definitions instead of remaking entire components.

- Unit testing code became much simpler with AOP. The basic classes were implemented and tested separately from aspects; aspects and their utility classes were tested separately from non-aspect components. Special test classes were used to verify the aspect behaviour. Thus, when the time came for composition, any bugs could be assumed to be an effect of pointcut definition rather than actual programming mistakes, assuming the unit testing was successful.
- The behaviour of the entire application could easily be changed by altering aspect behaviour, either by adding or removing aspects, or by altering pointcut definitions.
- Code completion, which is a very efficient way to make writing code faster, could be used, at least on standard Java classes, methods and attributes.

Disadvantages of AOP implementation methods compared to OOP implementation methods:

- Refactoring could not be used, as RefactorIt did not support aspects within a project. It did not ignore them, as they have the “.java” extension, interpreting them as standard Java classes with inaccurate syntax.
- It is harder to use error messages, reflecting on specific local conditions. It can be accomplished, and if so serve a better purpose in the long run, but still is harder.
- The used AspectJ version does not support incremental compiling, making the build process a bit clumsy. In a project with thousands of classes, it would have become a real problem.
- The AspectJ language and its handling of pointcuts are not entirely intuitive and needs to be properly understood before undertaking a project like this. Sometimes bugs occurred that was the result of a join point not being intercepted properly since the pointcut definition was a bit off key.
- Automated unit testing of aspects using for instance JUnit is not supported for the AspectJ framework integrated with any IDE, out of the box. It can be accomplished with some extra work, but should be provided ready to be used (and easy to use).
- Weaved code that has been packed into JAR files contain the aspects that they were composed with at compile time. Usually, JAR files are kept static. As long as there is no run-time or flexible load-time weaving instead of compile-time weaving, that is not feasible.
- The code completion does not include actual aspect concepts yet.

4.1.3. Time spent on development

OOP version

This section sums the activities needed to complete the OOP-version's implementation. The util package was not included in the analysis, as it is said to be the same for both versions.

Entity	Activity	Time (h)
OJDB	Design, implementation, testing and packaging.	- (reused)
Library	Design	8
Library	Implementation	49
Library	Testing	31
Total	-	88

Table 1: Time spent on development for the OOP version.

AOP version

Entity	Activity	Time (h)
OJDB	Design, implementation, testing and packaging.	- (reused)
OJDB	Removing logging statements	4
OJDB	Purging the package so only the core functionality remains. Persistence, security etc extracted.	6
AOP-Library	Design including aspects	5
Aspects	Implementation, testing and debugging	26
AOP-Library	Implementation	32
AOP-Library	Testing, debugging	9
Total	-	82

Table 2: Time spent on development for the AOP-version.

4.1.4. Code statistics

Note: The statistics below have been assembled mostly using the open source, freeware JavaNCSS tool [URL29]. Currently, aspects cannot be counted the same way, so aspect related statistics have been assembled manually, according to the rules defined by JavaNCSS. NCSS stands for None Commenting Source Statement, which means the number of valid, non-comment (using // or /**) source code statements in a Java class. The Javadoc column stands for counted Javadoc statements, for instance "@param input".

OOP-version

Below are the JavaNCSS-generated results for the OOP-version of the application.

Classes	Functions	NCSS	Javadocs	Package
4	57	521	61	com.darkwolf.library
2	11	132	12	com.darkwolf.library.client
6	13	62	19	com.darkwolf.library.elements
1	24	31	25	com.darkwolf.library.interfaces
4	19	376	23	com.darkwolf.library.rmi
4	47	332	47	com.darkwolf.ojdbc
5	24	150	29	com.darkwolf.ojdbc.elements
2	4	64	6	com.darkwolf.ojdbc.parsers
2	6	53	8	com.darkwolf.ojdbc.persistence
3	6	78	9	com.darkwolf.ojdbc.predicates
2	13	153	15	com.darkwolf.ojdbc.presentation
1	1	12	2	com.darkwolf.util.filtering
1	3	22	4	com.darkwolf.util.html
6	12	26	16	com.darkwolf.util.interfaces
1	7	32	8	com.darkwolf.util.logging
1	6	46	7	com.darkwolf.util.persistence
2	13	55	15	com.darkwolf.util.primitives
1	4	13	5	com.darkwolf.util.rmi
3	15	123	18	com.darkwolf.util.tools
1	1	5	2	com.darkwolf.util.transforming
52	286	2286	331	Total

Packages	Classes	Functions	NCSS	Javadocs	Per
20	52	286	2286	331	Project
	2,6	14,3	114,3	16,55	Package
		5,5	43,96	6,37	Class
			7,99	1,16	Function

Table 3: This table shows the NCSS Code statistics for the OOP-version.

AOP-version

Below are the NCSS results for the AOP-version of the application.

Classes	Functions	NCSS	Javadocs	Package
4	59	348	60	com.darkwolf.library
2	12	115	13	com.darkwolf.library.client
6	13	72	19	com.darkwolf.library.elements
1	24	31	25	com.darkwolf.library.interfaces
4	18	297	22	com.darkwolf.library.rmi
4	43	235	43	com.darkwolf.ojdbc
5	24	123	29	com.darkwolf.ojdbc.elements
2	4	54	6	com.darkwolf.ojdbc.parsers
2	6	41	8	com.darkwolf.ojdbc.persistence
3	6	61	9	com.darkwolf.ojdbc.predicates
2	13	141	15	com.darkwolf.ojdbc.presentation
1	1	12	2	com.darkwolf.util.filtering
1	3	22	4	com.darkwolf.util.html
6	12	26	16	com.darkwolf.util.interfaces
1	7	32	8	com.darkwolf.util.logging
1	6	46	7	com.darkwolf.util.persistence
2	13	55	15	com.darkwolf.util.primitives
1	4	13	5	com.darkwolf.util.rmi
3	15	123	18	com.darkwolf.util.tools
1	1	5	2	com.darkwolf.util.transforming
6	9	140	24	com.darkwolf.aspects
3	2	54	6	com.darkwolf.aspects.ojdbc
5	1	85	11	com.darkwolf.aspects.library
66	296	2131	367	Total

Packages	Classes	Functions	NCSS	Javadocs	Per
23	66	296	2131	367	Project
	2,87	12,87	92,65	15,96	Package
		4,48	32,29	5,56	Class
			7,20	1,24	Function

Table 4: NCSS code statistics for the AOP-version.

Comparison of code statistics

Below is a comparison summary of the two versions and the difference between them on a higher level.

Version	Packages	Classes	Functions	NCSS	Javadocs
OOP	20	52	286	2286	331
AOP	23	66	296	2131	367
Difference	3	14	10	155	35

Table 5: Comparing the NCSS statistics of both versions.

4.1.5. Discussion of development results

This section discusses the results achieved through the actual development process, such as development experiences of the two approaches and code statistics.

The AOP implementation took about as long time as the OOP implementation, and the code statistics values were about the same as well. The immediate reaction was, “well then, what is it good for?” When digging a bit further, the results made more sense.

Design and implementation methods

The development time was 82 hours for the AOP version and 88 hours for the OOP version, a 7.3 % increase for the OOP version. Because these are rather short development times and there is really only one measurement, it cannot safely be stated if one method is notably faster than the other. That is of course depending on the experience of the developer, as well as several other factors. Based on the results here, what can be said is that AOP at least does not mean a notably *longer* development time, even for inexperienced AOP developers. As a hint to the project managers, some minor test project should be undertaken prior to the real project, so developers can be productive with AOP and not just utterly confused, leading to a project melt-down.

Since aspects had to be developed from scratch, a large factor of the total AOP implementation time consists of aspect development time. In a normal case, many aspects should be able to be reused, if the project is not the first AOP-project. Thus, a development effort without aspect reusing have to be prepared for the fact that the project time might be longer than necessary, but it is an effort that, if done correctly, only needs to be performed one time. Also, in the given measurement (26 hours), some aspects are included that were used only during development.

The developer had very little experience in AOP before the implementation, but a lot of experience with OOP. Still the results are, even if only by little, to the advantage of AOP.

Testing and debugging was better for the AOP-implementation, mostly due to more dynamic tracing and logging capabilities, which meant that it was easier to find bugs. The main reason was that the aspects are more flexible, and can dynamically be applied anywhere without altering the ordinary Java classes. They then filter out useful information, so the important things are in focus.

In the end, both approaches did their job quite well, although both had disadvantages compared to the other.

Code statistics

The amount of NCSS in the OOP-version is a 7.3 % increase over the AOP version⁶. This mostly comes from the fact that code tangling is less frequent in the latter, but most importantly, AOP allows the assembling of generic service usage in one place, such as logging.

As shown previously, the overall source code statistics are quite similar for both versions, at least on the surface. However, in-depth analysis of the results and the actual code states the following:

- In bigger systems, where more components might need access to more generic services than in this example application, AOP should excel in code statistics. The tendency is clear, after examining the code of the application versions.
- The aspect version contains several aspects that are only used during development (Tracing, PerformanceTiming and ExceptionHunting, for instance). If they had been removed from the statistics comparison, the gain of AOP would be more obvious. The result when removing development aspects was 11.8 % compared to the 7.3% above.
- Code tangling exists in the AOP version in some cases as well (a few try/catch blocks and null pointer checks), although the OOP version code contains more tangling in general. However, when looking in the code, it is apparent that if optimal stability is to be achieved, the OOP version in fact needs even more code tangling (i.e. better exception handling, more null checking and so forth). If that had been the case, the difference between the two versions would have been bigger also in this case. This holds even if the AOP version should be changed the same way, as exception checking and null checking occurs on the crosscutting level through interception.
- Javadoc measurement showed that AOP had more Javadoc statements, which comes from the fact that the AOP version had more classes and methods; hence also Javadoc statements. When comparing average values, the results were quite similar.

When considering the statements above, the code statistic results rule in favour of the AOP version.

⁶ Neither version is optimised for as little code as possible - this is just the way it turned out.

4.2. CONCLUSIONS BASED ON DEVELOPMENT RESULTS

This section deals with what cannot be bound to specific programming results, but are residing on a more theoretical level, and originate in experiences drawn from this thesis development project and parallel studies that have been conducted independently of the project.

4.2.1. Adaptability

Adaptability is an area where AOP should truly excel, given its flexible nature. However, in the case of this application, the two approaches will not differ much in the amount of work required to implement a change as long as they are not a complete revolution. As long as an application is designed properly from the start by rigorous encapsulation and constructing generic components, with an interface and corresponding implementations, most changes just mean adding a couple of methods to an interface, a corresponding implementation and to the GUI front-end.

When radically switching the behaviour of an application, AOP will be far more superior as it allows for more loosely coupled behaviour definitions and implementations. While it is really not that common during enterprise software development that the problem domain is so profoundly changed so the entire design must be remade, AOP does add other values specific to enterprise development, such as the ability to flexibly apply or remove development aspects like Tracing and ExceptionHunting.

4.2.2. Maintainability

The maintainability when it comes to the amount of code is a clear case; the AOP version will, because of less code tangling, code scattering, and better handling of service access, have less code, thereby reducing cost of maintenance.

However, there is more to maintenance than the amount of code. If a company that has only one developer that knows about the AspectJ syntax, the reduction in cost might just be a mirage. In the event that the developer resigns, is hospitalised, or whatever, the maintainability may not be all that cheap in the end. This is not an AOP-specific problem, but rather applies throughout the entire knowledge reserve of the company.

As AOP currently lacks mature visualisation tools, for instance UML-tools, it is a very important thing to let knowledge spread within the organisation, as not all will instantly comprehend the design of an AOP application. Additionally, independent brain cells working together as a neural net is more efficient than the standard “Newtonian” organisations, which is another reason for why knowledge exchange is essential. Please refer to [TQS97] for more details.

4.2.3. Modularity

The increased modularity when applying AOP correctly is quite obvious by looking at the resulting source code of the two applications. Not only are the components themselves modular, but also crosscutting parts of their behaviour and generic service usage.

In the event of producing more generic type of code that is not reliant on the OJDB, for instance JDBC connection handling, the increased modularity should provide some exciting opportunities.

Imagine a JDBC connection factory that is being used by a set of components. By adding an aspect that wraps itself around the methods to get a connection and return a connection, a connection pool can be added in the blink of an eye. A test case of this statement has been implemented, and can be downloaded from [URL15].

In such an example, the connection pooling is modular. The components accessing the connection factory are modular, and the factory itself is modular. In the OOP case, applying pooling behaviour (not the pooling utility classes themselves, of course) would usually be included in the connection factory's code.

4.2.4. Reusability

Both applications have many reusable components, above all in the “util” package, which is identical for the two. Apart from the util package, some components may be reusable in the OOP-version, in the event that a project will develop a very similar application. The components will in any case most likely need a bit of tweaking to fit into another application, or cannot be reused at all.

In the AOP case, things are a bit different. Even if the components mentioned above still cannot be easily reused, the actual composition sites, the aspects, may be reused. That is very useful, in case the aspects (from section 3.3.1, C5) have been made generic and encapsulated in the same way Java components should be. The generic, abstract aspects made in this thesis project have actually been reused in another context, just for testing purposes, and they all work fine in their new environment. All that takes for reuse is making new aspects inheriting the abstract ones, and in those new aspects define the application specific pointcuts needed.

As was acknowledged in the previous section, AOP code is more modular. If code is more modular and more generic, it is more reusable, and AOP offers the mechanisms to let components become just that.

4.2.5. Comparison to hypothesis

What of the issues raised in the hypothesis? The actual results are now weighed against the hypothesis statements, which are marked with bullets below:

- There will be less code altogether in the AOP version, since code tangling and code scattering can be eliminated.

The OOP implementation required more code than the AOP version, although less than expected. That has to do with the fact that the OOP application is not a full-blown enterprise example. The more generic services, contract enforcements, exception handling etc. there is, the more AOP excels in this area.

- Functionality will be more modular, and hence more reusable. This means that future development projects will have a greater chance of reusing code from the AOP application than from the OOP application.

The AOP-version is more modular than the OOP version, hence making it more reusable, as noted.

- The code will be easier to follow and comprehend in the AOP version, as it is more modular.

The AOP code is not easier to follow, due to the lack of mature visualisation tools for AOP, both regarding UML and Javadoc, but also when developers are unfamiliar with AOP techniques. The OOP-version has a better overview.

- The development tools and processes are more mature and well-established for OOP, resulting in less spent time for the development of the OOP version.

The development tools were better for the OOP implementation, but the differences were fewer than expected, as most tools and processes easily could be used with AOP as well. After all, AOP is basically an enhancement to OOP.

The two major differences regarding tools are in refactoring and UML-modelling, which are almost non-existing for AOP; even if the UML tool used for OOP visualisation, ArgoUML, is not at this date a mature one.

The IDE did not recognise aspect-oriented code, and thereby labelled them as having incorrect syntax (in Forte, a symbol is shown when a class does not follow proper Java syntax). It was a minor detail, although an annoying one. This phenomenon will also affect other programs, as it did with RefactorIt and JavaNCSS, which could not work with AspectJ code. The major difference in the development process concerns design issues, as previously stated.

4.3. SUMMARY

This thesis has compared the benefits of AOP to OOP when implementing a distributed, web-based application.

On the whole, AOP is a very promising technology that intuitively solves some of the problems that are currently giving architects, designers, and developers many headaches, such as code tangling, code scattering, handling unexpected demands, and making sure resources can be reused properly in a natural way. As a technology, it has become mature enough to use in larger development projects, also in projects using more advanced J2EE concepts.

Now, there are some points in general that must be considered before adding AOP to the development toolbox:

- Is the project ready for this technology? If the personnel are already having problems using and understanding OOP core concepts, then jumping directly to AOP is not a good idea.
- What AOP implementation suits the project the best? AspectJ, Hyper/J, and JAC are currently the leading alternatives. Which new solutions may come in the future, and will it be possible to easily reuse invested resources to a new framework? AspectJ is considered as not easily replaced by a similar framework. The aspect code invested in might be lost; hence it is of the utmost importance to keep most functionality in generic standard utility classes, not within the aspects themselves. Currently, there is no standard for defining and exchanging aspect behaviour and definitions as Meta data.
- What value does AOP add? If an EJB server is already being used for all application development and works fine, the technology may not be needed at the moment. Other matters may be more crucial.

Another very important thing to notice is that AOP is not the solution to bad code being written. In case developers cannot see how encapsulation, modularity, interface and implementation decomposition, and extracting generic functionality for reuse can be an asset, general AOP just offers one more level of complexity for things to mess up.

Despite these considerations, AOP should be regarded as a very powerful tool for improving application development productivity that, if used correctly, will pose a considerable business advantage to those who incorporate it into their development process. That is a fact already in its current state when IDE:s and other tools have not yet caught up on the phenomenon, not utilising its full power. For those companies using a Netbeans-compatible IDE or for those using Eclipse⁷, the question is more about how to go ahead with the adaptation process of including AOP as a core component in development, without adding an extra, although temporary, burden on the staff.

⁷Two IDE:s that support the AspectJ AJDE tool as an add-on module. Other IDE:s will have to use AspectJ-tools via Ant or another workaround.

The following features of AOP should be kept in mind when deciding on whether to stick to plain OOP or incorporate AOP into the development process:

- AOP allows better reuse and more modular code, being able to express also crosscutting concerns that plain OOP has a hard time handling.
- AOP may come to replace the tedious EJB work that has to be made to deploy an application.
- AOP supports the injection of generic services at every available join point.
- AOP decreases the amount of code that has to be written, enabling developers to focus on business logic instead of contracts, exception handling etc. Contracts and exception handling are still very important, but may be handled separately from other components and applied flexibly where they seem to fit.
- AOP can provide more efficient debugging and error tracing, thereby saving time and frustration.

Many of the above features are opportunities that have not fully been incorporated into developer tools, but can be used today with some initial setup work. In the beginning of next year, 2003, there are sure to be more support for AOP in all sorts of tools, offering an opportunity to software developers to increase their productivity by making complicated issues, like crosscutting concerns, a lot easier to express.

On a side note, MIT technology review listed AOP as one of "ten emerging areas of technology that will soon have a profound impact on the economy and on how we live and work" [URL5].

After reviewing how AOP affects software development, comparing it to standard OOP, it is not that hard to imagine, considering the importance of software and software development in today's information overflowing society.

For further thoughts and information on AOP from the author, please refer to <http://www.darkwolf.ws/aop/> and <http://www.darkwolf.ws:8080/blog/>.

5. SOURCES OF INFORMATION

Note: If there is a star symbol on an item below (*), it is not freely available. Download other sources at [URL15](#).

Designation	Title	Author(s)	Year
[AGS89]	Assuring good style for object-oriented programs	Karl Lieberherr and Ian Holland	1989
[OOO92]	Obstacles in Object-Oriented Software Development	Mehmet Aksit and Lodewijk Bergmans, faculty of Computer Science, University of Twente	1992
[NMA92]	Towards a New Model of Abstraction in Software Engineering	Gregor Kiczales, Palo Alto Research Center (PARC).	1992
[MOP93]	Metaobject protocols: Why we want them and what else they can do	Gregor Kiczales, J. Michael Ashley, Luiz Rodriguez, Amin Vahdat and Daniel G. Bobrow.	1993
[TTR93]	Towards a Theory of Reflective Programming Languages	Anurag Mendhekar and Dan Friedman, PARC.	1993
[AOI94]	Abstracting object interactions using Composition Filters	Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa.	1994
[OIM94]	Open Implementations and Meta-object Protocols	Gregor Kiczales and Andreas Paepcke.	1994
[DSP94] *	Design Patterns: Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.	1994
[APP95]	Adaptive parameter passing	Christina Vidiera Lopes	1995
[AOO96]	Adaptive Object-Oriented Software – The Demeter Method	Karl Lieberherr	1996
[OID97]	Open implementation design guidelines	Gregor Kiczales, John Lamping, Christina Vidiera Lopes, Chris Maeda, Anurag Mendhekar of PARC and Gail Murphy of the University of B.C., Vancouver	1997
[APPC97]	Adaptive Plug-and-Play Components for Evolutionary Software Development	Mira Mezini and Karl Lieberherr, College of Computer Science, NorthEastern University of Boston	1997
[AOP97]	Aspect-Oriented Programming	Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, PARC.	1997
[TQS97] *	The Quantum Society	Danah Zohar	1997
[UML99] *	The Unified Modeling Language User Guide	Grady Booch, James Rumbaugh, and Ivar Jacobsen.	1999

[ODR00]	On the Need for On-Demand Remodularization	Harold Ossher and Peri Tarr, IBM Thomas J Watson Research Center.	2000
[RWA02]	Building a Real-World Application with Aspect-Oriented Modules and Hyper/J	Lee Carver, M.Sc. thesis at the University of California, San Diego.	2002
[CAO02]	Challenges of Aspect-Oriented Technologies	Roger Alexander and James Bieman, Dpt. of Computer Science, Colorado State University.	2002
[DAC02]	Designing Aspect-Oriented Cross-Cutting in UML	Dominik Stein, Stefan Hanenberg and Rainer Unland.	2002
[UNA02]	A UML Notation for Aspect-Oriented Design	Renauld Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier and Laurent Martelli.	2002
[AJP02]	AspectJ programming guide	The AspectJ team.	2002
[ADC02]	An Analysis of Design Approaches for Crosscutting Concerns	Ruzanna Chitchyan, Ian Sommerville and Awais Rashid.	2002

Designation	Title	URL	Last visited
[URL1]	AspectJ	http://www.aspectj.org	2002-07-10
[URL2]	WikiWeb	http://www.c2.com/cgi/wiki	2002-06-12
[URL3]	Aspect-Oriented Software Development	http://www.aosd.net	2002-07-10
[URL4]	JAC	http://jac.aopsys.com	2002-06-12
[URL5]	MIT technology review	http://www.ccs.neu.edu/research/demeter/aop/publicity/mit-tech-review.html	2002-06-12
[URL6]	UMLAUT	http://www.irisa.fr/UMLAUT/	2002-07-16
[URL7]	The Server Side	http://www.theserverside.com/	2002-07-18
[URL8]	Log4J	http://jakarta.apache.org/log4j/	2002-07-18
[URL9]	Law of Demeter introduction	http://www.enteract.com/~bradapp/docs/demeter-intro.html	2002-07-18
[URL10]	Demeter	http://www.ccs.neu.edu/research/demeter/	2002-07-18
[URL11]	JGuru EJB introduction	http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html	2002-07-19
[URL12]	Dynamic proxies	http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html	2002-07-19
[URL13]	JBoss	http://www.jboss.org/	2002-07-19
[URL14]	Link page for all tools in the developer environment	http://www.darkwolf.ws/java/tools.html	2002-09-10
[URL15]	The author's site about AOP	http://www.darkwolf.ws/aop/	2002-07-19

[URL16]	A notation for Aspect-Oriented Distributed Software Design	http://jac.aopsys.com/papers/uml/uml.html	2002-07-19
[URL17]	Subject-Oriented Programming	http://www.research.ibm.com/sop/	2002-07-22
[URL18]	ConcernJ	http://trese.cs.utwente.nl/prototypes/concernj/index.htm	2002-07-22
[URL19]	Hyper/J	http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm	2002-07-22
[URL20]	PROSE	http://ikplab11.inf.ethz.ch:9000/prose/Wiki.jsp?page=AboutProse	2002-07-22
[URL21]	DJ	http://www.ccs.neu.edu/research/demeter/DJ/	2002-07-22
[URL22]	DemeterJ	http://www.ccs.neu.edu/research/demeter/DemeterJava/	2002-07-22
[URL23]	MixJuice	http://staff.aist.go.jp/y-ichisugi/mj/	2002-07-22
[URL24]	Java home page	http://java.sun.com	2002-07-22
[URL25]	ArgoUML	http://argouml.tigris.org/	2002-08-26
[URL26]	Ant	http://jakarta.apache.org/ant/	2002-08-27
[URL27]	Open Source Initiative	http://www.opensource.org	2002-08-28
[URL28]	Sourceforge	http://www.sourceforge.net	2002-08-28
[URL29]	JavaNCSS	http://www.klee.com/clemens/java/javancss/	2002-09-02
[URL30]	UML introduction	http://www.microgold.com/version2/articles/article1.html	2002-09-16
[URL31]	Patternstesting project	http://patternstesting.sourceforge.net/	2002-09-24

6. APPENDICES

Appendix	Content	Details
A	AOP implementations in Java	This appendix reviews several AOP Java implementations, such as JAC and Hyper/J.
B	AOP related concepts	This appendix describes alternatives to AOP.
C	Development environment	Lists the tools used to make a developer environment for AOP.
D	Basic concepts of the thesis	To understand the thesis, understanding the concepts herein are crucial.
E	Code distributions	Gives important information about the downloadable code.
F	OOP application distribution	The ZIP distributions file with the OOP Library application. It can be downloaded from [URL15].
G	AOP application distribution	Same as above for the AOP version.
H	Object design UML models	Contains an ArgoUML file with the result of the object design phase of the OOP application version development.

Appendices can be downloaded at <http://www.darkwolf/aop/thesis/appendices/>

1. APPENDIX A

1.1. AOP IN JAVA

This section presents some AOP-implementations, some pure AOP, and some delivering AOP-like functionality.

1.1.1. ASPECTJ

AspectJ comes with a compiler (AJC), a debugger, a Javadoc generator and an IDE add-on called AJDE.

The AJC tool is actually a weaving pre-compiler written in Java. It quickly compiles aspects and classes together and produces Java source code that is described as a “weave.” AJC then invokes javac to actually compile the weave into byte code. If you prefer, you can tell AJC to produce only the weave and then invoke javac or the compiler of your choice manually.

An important limitation of AJC is that you must provide it with all of the source code that will be affected by your aspects. Many times, this is either impossible or impractical and limits what you can apply aspects to. Hence it is nearly impossible to divide a more advanced Java application into strict aspectual modules, as was the case in [RWA02]. Why “nearly”? Well, there is of course the possibility to supply the entire Java 2 SDK source code to the AJC, but that was not done.

AspectJ has been around for a while now, and is becoming increasingly popular. It is also the AOP environment of choice for this thesis. There is a lot of material on AOP and AspectJ on the AspectJ home page [URL1].

1.1.2. JAC

JAC wraps method calls so that aspect can do things when a method is called without the method being aware of this. This is done on a JVM byte code level during run-time (Note: actually load-time, which is when classes are loaded), whereas the AspectJ implementation currently only handles compile-time.

JAC also includes a set of already finished aspects so a programmer can experience the world of aspects without actual AOP (actual programming, that is). Aspects are configured using configuration files, and there is a user interface to help configuring an application so it can use existing aspects for some purpose.

JAC was not chosen for this thesis because of previous experiences with AspectJ. Due to the time limit, both versions could not be tested as they have different semantics.

Read more about JAC on their home page [URL4].

1.1.3. HYPER/J

IBM has developed Hyper/J, which supports multi-dimensional separation of concerns (MDSOC), described in appendix B.

Aspect-Oriented Programming and Subject-Oriented Programming, both described further on, are said to be usable within Hyper/J. Hyper/J supports dividing components into independent modules known as hyperslices, which can be dynamically combined to form a ready application. The

hyperslices consist of standard code, but also a set of configuration files that define each slice's behaviour. Hyper/J works in compile-time, just like AspectJ. Future versions will probably use run-time weaving.

Read more at [URL19]. [RWA02] used Hyper/J as the implementation tool.

1.1.4. DYNAMIC PROXY-BASED SOLUTIONS

There are several ways to use Java's dynamic proxies to achieve reflective mechanisms that allow for AOP-like behaviour.

One of the more promising is a framework being developed by Rickard Öberg, who has also been involved in developing the dynamic proxy-based JMX server core in JBoss. The framework exists in alpha version on the Sourceforge project "Meinds", but has evolved a lot since. The evolved version is currently not available for download, as it is a part of a product.

It's built on the concepts of interceptors and aspects, where the interceptors are the dynamic proxy-based mechanisms that intercept calls and, based on their assigned aspect functionality, perform actions accordingly. The aspects, which are standard Java classes, are configured with XML-files (meta-data) to tell the interceptors how their functionality is to be handled. Interceptors in this case, correspond to AOP's pointcuts, and can be ordered using the mentioned configuration files, where AspectJ instead defines advice precedence using, for instance, the "*dominates*" keyword.

However, the framework is not only about interception, but can also handle introduction by configuring interfaces and implementations on a context, and can add the functionality of the introduction elements onto any object handled by the framework. This constitutes a similar behaviour as AspectJ, but a quite different line of thinking, since only standard Java mechanisms are used instead of weaving at compile-time (using a special weaver/compiler) or run-time (using a byte code weaving JVM).

1.1.5. PROSE

PROSE is a JVM extension that can perform call interceptions at run-time. A non-AOP application will not see any difference to a standard JVM, according to [URL20]. The PROSE JVM provides an API for inserting and removing code extensions to already executing applications.

There are two small components in each Java Virtual Machine; one resides in the JVM itself, the other is a simple Java application. Aspects are written as ordinary Java classes, and can be compiled, then instantiated and inserted into PROSE. After an insertion into the PROSE JVM, the application will be extended at all points where it has been specified.

1.1.6. MIXJUICE

MixJuice is an enhancement of the Java language that adopts difference-based modules instead of Java's original class mechanism. One can decompose classes into sets of modules that can be independently tested. This means that crosscutting issues can be decomposed as well, tested separately and joined together as needed. Read more at [URL23].

DemeterJ is a tool for applying Adaptive Programming, which is basically a subset of AOP, to Java. The people that coined the expression Adaptive Programming have developed it. Among them are Karl Lieberherr and Mira Mezini.

While DemeterJ requires you to use a non-standard environment with class dictionary files and behavior files, DJ fits seamlessly into Java development environments. All you need to do is to import the `edu.neu.ccs.demeter.dj` package into your Java programs and you will be able to program in a traversal-visitor style. Thus, DemeterJ and DJ are not the same tools!

DJ is a tool for run-time AOP in a JVM, as are JAC and PROSE, and is thereby tied explicitly to Java, which DemeterJ is not in the same sense.

An interesting feature is the APStudio, which is supposed to allow for creating UML models that can be used by DJ/DemeterJ. Read more at [URL21] and [URL22].

ConcernJ integrates the concepts of aspects, classes and composition filters, which are described in the next section. It is currently only a research prototype, but seems promising. Read more at [URL18].

2. APPENDIX B

2.1. AOP RELATED CONCEPTS

Concepts that are related to AOP techniques are listed in this section. They can on a higher level be divided into object-oriented reflective architectures, particularly techniques of reification by meta-object protocols, and open implementation-based systems, including Aspect-Oriented Programming, Subject-Oriented Programming and Adaptive Programming.

Furthermore, the IBM technology about on-demand re-modularization and multi-dimensional separation of concerns should be mentioned on the side of the above main groups, as it strives to incorporate most of the above mentioned technologies under its umbrella.

2.1.1. META-OBJECT PROTOCOL

This is a technology that encompassed many of the other emerging technologies for abstracting object-oriented systems. The foundation is that developers write two sets of programs; both the normal base-language program (in for instance Java or C++) and also code in a meta-language that describes some additional value to the base-language program, for instance transaction handling, security settings, logging etc.

This technique is the father of aspect-oriented programming, and many other forms of programming methods to abstract functionality from the standard base-language programs and make it easier to maintain and evolve. Mathematically speaking it can be seen as extending the standard object-oriented class graph with another dimension of the graph.

Read more about the meta-object protocol technology and its development in [NMA92].

One note described in [NMA92] that is very interesting is the suggestion that compilers for a programming language should have open implementations, so that the programmer can affect the compiler behaviour directly. This allows for better fitting the high-level Meta code into the compiler, which can solve the problem of getting good enough performance out of high-level languages.

2.1.2. REFLECTIVE PROGRAMMING

An approach of Mendhekar and Friedman in a more mathematical way using sets, subsets and operators, derived that there is a possibility to construct a mechanism for base language abstraction. Using reflection and reification, a base system can be mapped onto a new base, where for instance crosscut concerns can be abstracted away from the base implementation and into some form of meta-expression and reflection. This was proven in [TRR93] and some theorems for these kinds of methods were established.

In plain English it means that provided that there are reflective mechanisms available, like in Java, it is possible to express crosscutting concerns using meta-data and the reflective mechanisms. The mechanism can provide information about given points in the application's execution and also allow actions to be performed based on that information. An actual implementation of this scenario is described in the Appendix E section "Dynamic proxy-based solutions".

2.1.3. ADAPTIVE PROGRAMMING

Adaptive programming (AP) is an extension to ordinary object-oriented programming that allows relationships between functions and data to be loosely coupled through navigation specifications. The adaptive part is from the fact that the adaptive system heuristically changes itself to handle altered requirements related to changing the object structure.

AP is basically a special case of Aspect-Oriented Programming.

Please refer to [AOO96] for more information about adaptive object-oriented programming, foremost regarding using the Demeter method.

2.1.4. ADAPTIVE PLUG-AND-PLAY COMPONENTS (APPC)

APPC's are components that encapsulate collaborations between objects, thus they can be used to capture cross-cutting mechanisms. They were introduced by Mira Mezini and Karl Lieberherr from the Northeastern University in Boston [APPC97]. APPC's were suggested as a result of the work on adaptive programming.

2.1.5. SUBJECT-ORIENTED PROGRAMMING

Subject-Oriented Programming (SOP) is an approach to building OO software systems by composing several sub-systems, which are known as subjects, according to a composition expression which describes the rules for what the subjects correspond to, and how they should be merged to implement required system functionality. This supports the separation of concerns, since each subject encodes the important aspects of the system as perceived from a particular perspective.

The approach attempts to solve the same problems as AOP, but in a slightly different way. AOP and SOP are said to be complementary. Read more at [URL11].

2.1.6. COMPOSITION FILTERS

The term composition filters was introduced in [OOO92]. Composition filters have the following important characteristics:

- **Modular extension:** Each filter enhances a class abstraction in a modular way. A modular extension means that a filter can be attached to a class without necessarily modifying the definition of that class.
- **Orthogonal extension:** Orthogonal extension means that each filter extension to a class is independent from other filter extensions. This allows easy composition of multiple filters.
- **Open-ended:** New filters may be introduced if necessary.
- **Aspect-oriented:** Each filter represents an aspect. Filters are predefined and have a well-defined semantics.
- **Declarative:** A filter specification describes what it means but not how it is implemented.

The people behind the work with composition filters are now co-operating with among others the people at PARC, since their work is much related. In 2001, Patricio Salinas integrated the concepts of aspects, classes and Composition Filters and started working on a tool for this, ConcernJ [URL18].

2.1.7. ABSTRACT COMMUNICATION TYPES (ACT)

This technology is based on the fact that object-oriented programs are mainly based on two things, objects and messages between objects. ACT is a means of intercepting messages between components and performing actions based on the message. Messages can be forwarded, altered, cloned or obstructed, if some condition is not met. The research team at the University of Twente in Belgium, in which the leading persons are Mehmet Aksit and Lodewijk Bergmans, introduced ACT's. Aksit and Bergmans are still very active in AOP-related work.

A lot of the work by the Twente team is founded on their own programming language, Sina, but the ideas and concepts hold for other object-oriented languages as well. Sina has built-in support for solving some of the design problems that other object-oriented languages have to deal with programmatically.

Read more on ACT's in [ACT97].

2.1.8. ON-DEMAND REMODULARIZATION

This technology actually is a more generic form of modular decomposition that can contain mechanisms like SOP or AOP to achieve a good system structure with high reusability, separation of concerns and modularization. Harold Ossher and Peri Tarr of the IBM Thomas J Watson Research Center first described on-demand re-modularization in a paper for the ADC 2000 conference. They state that “support for on-demand re-modularization is critical for achieving the benefits of multi-dimensional separation of concerns”. Multi-dimensional separation of concerns (MDSOC) suggests that this work provides even more dimensions of separation than AOP, since AOP handles one dimension on top of OOP, cross-cutting concern separation.

The question is if there is really a need for MDSOC, perhaps technologies like SOP and AOP are quite enough to solve the problem domains that OOP cannot decompose cleanly.

The IBM team has developed HyperJ as an implementation of their theoretical work.

3. APPENDIX C

3.1. THE DEVELOPMENT ENVIRONMENT

The development environment consisted mostly of free tools from the open source Java world, with a couple of exceptions. The following software was used to create the development environment:

- Windows NT as operating system.
- Forte 4, a.k.a. Sun ONE Studio, as Java IDE. It is based on the open source software Netbeans, and is free in its community edition.
- J2SDK 1.3.1 and 1.4.0 as Java environments.
- Windows Commander 5.0 as multi-tool (ZIP archive handling, file manipulation, FTP transfer etc.).
- Tomcat (3.3 and 4.0.1) and JBoss (3.0 with Jetty) for deploying and testing the application.
- DJ Java Decompiler 2.9.9.61 (based on Jad as decompilation engine) for occasionally decompiling and analysing Java classes.
- Acrobat Reader 5.0.5 for reading PDF documents.
- Ant 1.4 for compiling, packaging, deploying etc.
- AspectJ 1.0.6, with AJDE added to Forte and extra Ant tasks to Ant, for AOP enhancements to Java.
- Log4J 1.0.4 for logging.
- ArgoUML 0.10.1 for UML modelling.
- RefactorIT evaluation version, which is a refactoring plug-in to IDE:s.

4. APPENDIX D

4.1. BASIC CONCEPTS

This section briefly explains some of the concepts in the thesis that might not be completely obvious. If a concept is not listed here, then it is most likely explained in its logical context, or is assumed to be known by the reader.

4.1.1. PROBLEM DOMAIN

The problem domain is the set of requirements on a system that need to be satisfied in order for the system to solve some given assignment. It can be defined through a set of use-cases.

4.1.2. CODE TANGLING

When code becomes unnecessarily complex because it needs to take into account some circumstances, it is tangled. Source code after adding method access control or exception handling is a good example of tangled code, as demonstrated below:

```
try
{
    if (access.hasAccess(Session ses, DatabaseTable dbTable, access.SELECT))
        elem = dbTable.getElementByID(id);
    else
        elem = null;
}
catch (Exception e)
{
    elem = null;
}
```

4.1.3. CODE SCATTERING

Code scattering is when functionality that logically belongs in an encapsulated class of its own is spread throughout several classes. Logging statements are good examples of scattered code lines. Below is a demonstration (logging statements in **bold**) from one single class. However, as mentioned above, the real problem is when the same kind of statements are scattered throughout several different classes.

```
/** Log4J logging Category object */
private static Category cat = Category.getInstance(Database.class.getName());

cat.debug("Retrieving the table with name " + tableName + "...");

DatabaseTable temp = new DatabaseTable(tableName);

cat.debug("Get primary key for table with name " + tableName + "...");

// Get primary key for this table
PrimaryKey tempKey = tableConfig.getPrimaryKey(tableName);
```

A programming language or an operating system is called reflective, if it has mechanisms that provide programs with some special ability to read and write properties of themselves (or related programs).

The reflection API in the Java language enables Java code to discover information about the fields, methods and constructors of loaded classes. It also allows code to use reflected fields, methods, and constructors, as long as the operations are within security restrictions, which can be defined by altering the JVM's security policy.

Please refer to appendix F for some words on reflective programming, which is based on reflection mechanisms.

An Interceptor intercepts messages between two classes, and can take action based on various parameters:

- The source or destination object (their type or publicly accessible information, for instance)
- The input parameter(s)
- The result of the message

A dynamic proxy is a mechanism in the Java programming language that appeared in JDK 1.3. It's an example of an *interceptor* (as defined above).

From [URL12]: “Method invocations on an instance of a dynamic proxy class are dispatched to a single method in the instance's *invocation handler*, and they are encoded with a `java.lang.reflect.Method` object identifying the method that was invoked and an array of type `Object` containing the arguments.”

This basically means that when calls to an object are intercepted, Java code can be included in the object's invocation handler. The code will have access to the method parameters, it can affect the method execution's result, or even dispatch new method calls elsewhere as a reaction to the call.

The EJB definition below originates from [URL11]:

“The Enterprise Javabeans specification defines architecture for the development and deployment of transactional, distributed, server-side software components. Organizations can build their own components or purchase components from third-party vendors. These server-side components, called enterprise beans, are distributed objects that are hosted in Enterprise JavaBean containers and provide remote services for clients distributed throughout the network.”

In order to understand EJB, which is of course not the within the scope of this thesis, one needs to understand how EJB containers work.

An EJB container wraps the enterprise beans and handles a set of services that the components need, such as resource pooling, logging, transactions, security, persistence and concurrency. As there are several open-source EJB containers around, a good thing is to do a little research on how those containers work. As long as an EJB container implements the specification, it can be constructed in any way. Below is an image of how a container might work.

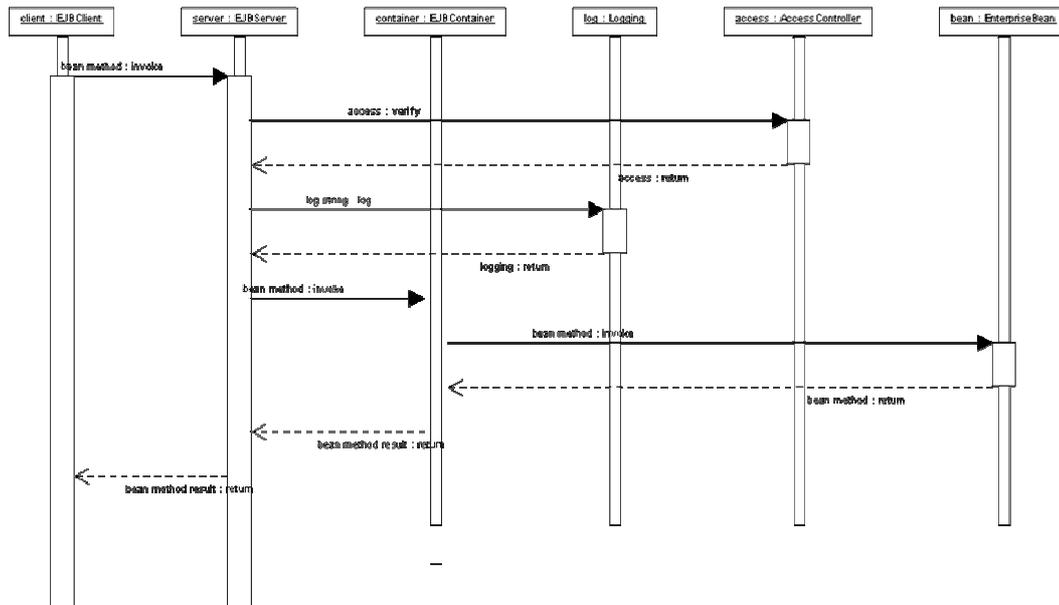


Figure 1: Simple view of an EJB server and an EJB container, and how a incoming call might be handled.

JBoss is one example of an open-source EJB container. It uses dynamic proxies to intercept calls and perform actions based on the call. Read more about JBoss at [URL13].

A join point is a given point in object-oriented code. It can be a method call, object initialization, variable retrieval, to mention a few scenarios. The best way to picture it is to look at the figure below, where a method call is being performed from one object to another.

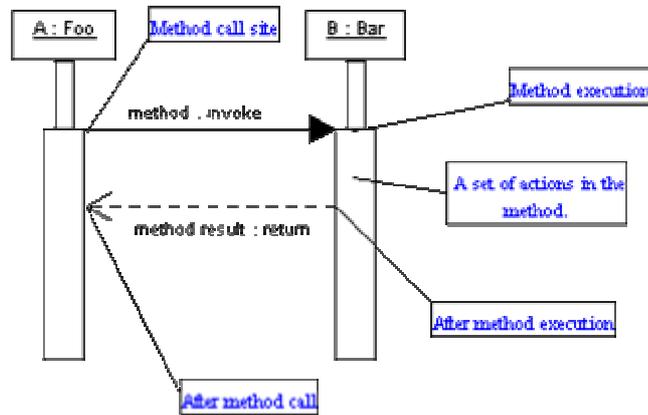


Figure 2: Join points during object interaction via method calls.

The join points in the picture are (assuming Foo and Bar are already initialised objects):

1. The call site of the method call Foo.methodA
2. Foo.methodA's execution starting point
3. Any join points within the method code (calling new methods etc.)
4. Foo.methodA returns or throws an Exception
5. Method call returns result to Bar

In case the objects have had to be initialised, the actual object construction would have contained many more join points.

Not only actual code may be reused, but also design solutions on a higher level. Such design solutions are called *design patterns*. They describe the design solution of a certain situation, and then it is up to the developer to make an implementation of the pattern, if one isn't already available.

Below are a couple of examples of design patterns:

- Seppuku pattern – a distributed object cache, where the idea is that each cached object removes itself from the cache after a time of inactivity.
- Visitor pattern - the purpose is to encapsulate an operation that you want to perform on the elements of a data structure. Using a Visitor pattern allows you to decouple the classes for the data structure and the algorithms used upon them.

Read more at The ServerSide.com, who have got an extensive library of user-contributed design patterns [URL7] or check out [DSP94].

The Law of Demeter [AGS89] defines a design and programming style guideline for objects, which, if applied to them, increases their modularity.

The definition is retrieved from [URL9]:

“A method "M" of an object "O" should invoke ONLY the methods of the following kinds of objects:

- itself
- its parameters
- any objects it creates/instantiates
- The object's direct component objects” (i.e. super component parts)¹

Research work trying to create mechanisms that made it easy to follow the Law of Demeter in OOP lead to the concept of Adaptive Programming (please refer to the appendix), which is a close relative to AOP.

¹ For instance, a Car object consists of an Engine, a Steering Wheel, four Wheel components, etc.

5. APPENDIX E

5.1. SOURCE CODE DISTRIBUTIONS

In order to get the most from the actual implementation and the source code distributions, it is good to know the following:

- Java Naming and Directory Interface (JNDI) [URL24]
- Java Server Pages (JSP) [URL24]
- Jakarta Ant [URL26]
- Log4J [URL8]

The distributions have been made for Windows-flavoured operating systems, but can be used on *NIX-systems with a few changes. It's a matter of directory-structures, and a matter of using, for instance, shell-scripts instead of batch scripts. Since Java is platform independent, getting the Java code itself running is not a problem, assuming there is a Java Virtual Machine (JVM) being able to run JDK 1.3 code or higher available on the system.

Furthermore, the following "Readme" file describes a bit about the structure of the distributions in general.

5.1.1. OOP VERSION

This section describes the package structure of the OOP version distribution.

The developed or reused library application packages, not belonging to external components, were:

`com.darkwolf.library` – the main library classes

`.client` – the client related classes (text-based client)

`.elements` – the basic library element classes, like CD and Book.

`.interfaces` – the library service method interface for the RMI server

`.rmi` – other RMI-related classes

`com.darkwolf.ojdb` – the main OJDB classes

`.elements` – DBElement related classes

`.parsers` – persistence-parsers used to turn file data into elements

`.persistence` – other persistence-related classes

.predicates – predicate classes used for element filtering and searching

.presentation – classes for presenting element data to the JSP GUI

com.darkwolf.util – generic utility classes

.filtering – classes used to filter a set into a new set (used for predicate filtering of element vectors)

.html – HTML presentation classes

.interfaces – global generic interfaces

.logging – logging utility classes

.persistence – generic persistence classes, like a class for accessing a text file

.primitives – generic data types, like the name-value pair class

.rmi – generic RMI utility classes

.tools – generic tools, like a timer

.transforming – classes for applying transformation operation classes on an object

5.1.2. AOP VERSION

This section describes the package structure of the AOP version distribution, which is the same as for the OOP-version, but with the following additions:

com.darkwolf.aspects – where all aspects are assembled

.ojdb – aspects specific to the OJDB sub-system

.library – aspects specific to the Library application

The following JSP-pages are used in the web-based GUI:

admin.jsp – the main page for administrators. From here, commands can be started, which leads to the user being dispatched to the page that handles the actual command execution.

element.jsp – a page only administrators can access. Here, edit and delete commands for an element can be executed, which leads to a call to the element_cmd.jsp page, which checks input, performs the actual command and checks the result of it.

element_add.jsp – this page allows administrators to add elements to tables.

index.jsp – the first page, featuring a login procedure. It then calls login.jsp with the given parameters to verify that the user is valid. After the verification, administrators are redirected to admin.jsp and ordinary users to their corresponding main page, user.jsp.

iteminfo.jsp – is a closer look on a specific element, where also lending status is given. If the library object is available for lending, a link to lend.jsp appears which enables the user to lend the object. If the object is already lent out to the current user, a link to return.jsp appears, that, enables the user to return the object. If the object is lent out to another user, the lender's user name is given.

results.jsp – this page gives a result list of the command that was executed on the page, for instance "List Books". The latest command is saved in the JSP session, so that when the user returns, the last command is pre-selected. Also, other pages can use the command to let the user return to the current result set.

Finally, there is a page called rollback.jsp, which rolls back all changes since the last save. However, since a save occurs every time a user logs out, it might not work exactly as desired. There is currently no such thing as a per-user rollback, such an effort was not within the scope of the thesis.