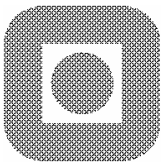
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn:	Erlend Rønningen og Tore Steinmoen
Fag:	Datateknikk, systemutvikling
Oppgavens tittel (norsk):	Metrikker for aspektorientert programmering av middelvare systemer
Oppgavens tittel (engelsk):	Metrics for Aspect-Oriented Programming of middleware systems.

Oppgavens tekst:

The motivation behind the Aspect-oriented programming (AOP) is to improve system modularity. Scattered code due to crosscutting concerns are to be gathered in single locations, called aspects. The AOP concept and the first aspect-oriented programming language, AspectJ, were introduced at Xerox Parc in the 1990s. In this study we aim to study the quality trend of a system implemented in AspectJ and Java, and possibly containing aspects. The analysis is based on Telenor Mobile's middleware system COS. By using the GQM method we have studied the system quality factors maintainability and reusability and their subcharacteristics. As a result, we have identified a set of metrics enabling us to demonstrate changes in these system quality factors. Most metrics are implemented in a metrics tool named AspectMetrics.

Oppgaven gitt:	20. januar 2004
Besvarelsen leveres innen:	15. juni 2004
Besvarelsen levert:	8. juni 2004
Utført ved:	NTNU
Veileder:	Tor Stålhane (NTNU), Kristoffer Kvam (Telenor Mobil)

Trondheim,

Tor Stålhane
Faglærer

Summary

In this diploma thesis we have aimed to identify metrics that accommodate two chosen system quality factors and implementing the selected metrics in a metrics tool. The metrics chosen should measure change in the system quality factors reusability and maintainability for the middleware system COS at Telenor Mobile and similar systems. The metrics tool should support the aspect-oriented programming language AspectJ, and is planned to be a plugin to the open source code analysis framework XRadar. Changes due to introduction of aspects are of particular interest.

We have through a GQM process identified the following subcharacteristics for the chosen system quality factors: modularity, testability, analyzability, changeability and stability. Questions are formulated to analyze these sub factors, and metrics that can answer the questions are chosen.

We have implemented the tool AspectMetrics, which calculate metrics on Java and AspectJ code and generates an XML report containing the measurement results. A transformation from XML to HTML web pages is also provided. The metrics tool can measure size metrics, like the number of statements and the number of classes, coupling, fan-in/fan-out, cohesion and advice-in/advice-out. Advice-in and advice-out are two new metrics which respectively measures how many advice a class (or aspect) is affected by and how many joinpoints an advice hits on. These metrics are inspired by the concept for the fan-in and fan-out metrics.

The tool has been used to analyze two versions of the system DIAS v.2.0, which is a part of a diploma study in 2000. We have in our preparation project in 2003 added aspects to the DIAS system while keeping the system functionally equal to the original version. We have used our metrics tool to calculate the differences between the system with and the system without aspects. The introduction of aspects gave a positive change in coupling, fan-in/fan-out and size measures, while cohesion was negatively affected. The metrics thus, overall, indicated a positive change to the subcharacteristics testability, analyzability, changeability and stability and both the main quality factors. There was no indication of a positive change to modularity.

The analysis of the measurement results indicates that most of the metrics perform as intended. The size metrics, coupling, fan-in/fan-out, and advice-in/advice-out all gave results that corresponded to what we had expected. However, the cohesion measure did not behave in a way that could be correlated to the actual changes performed on the code. A closer analysis showed that moving and merging of functionality could result in either an increase or a decrease in cohesion. Thus we find that cohesion, at least in its current form, is not a suitable metric when using aspect-oriented programming. Further, this gave reason to reinvestigate the disappointing modularity results. With a reworked set of criteria we also found indication of improved modularity.

Preface

This diploma thesis is the finalization of our five years study to become masters of technology, specialization computer science, at the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU).

We have studied Aspect-Oriented Programming (AOP) and metrics in the context of the Java-based middleware system Custom Order Server (COS) at Telenor Mobile AS. The work on this diploma is the continuation of a project carried out last fall, named "Increasing readability with Aspect-Oriented Programming". In this diploma thesis we have studied on the quality factors maintainability and reusability, with consideration to how AOP affect these factors.

We would like to thank our advisor at Telenor, Kristoffer Kvam, for comments and guidance throughout project.

We would also like to thank the developers of AspectJ, which gave us technical help through the mailing list aspectj-dev@eclipse.org.

Last, but definitely not least, we would like to thank our advisor at NTNU, Professor Tor Stålhane. You have given us most valuable guidance, motivation and suggestions for our work and report.

Trondheim, 08 June 2004

---------- ----------

Erlend Rønningen Tore Steinmoen

Contents

Figures and tables

Figures

Tables

Chapter 1. Introduction

The concept of Aspect-oriented programming (AOP) was first introduced to us by Telenor Mobile[1]. In the summer of 2003 they were exploring how AOP could contribute to their system development process. Telenor Mobile was our partner for the readability study [1] and has also partnered us for this diploma thesis. We also found that no one in the academic staff at NTNU had any experience with AOP. Thus, it seemed beneficial for the university to explore this paradigm as well.

This diploma thesis is a continuation of our study on increasing readability with AOP [1] done in the fall of 2003. We studied the effects of introducing aspects in an object-oriented system, to see if it would increase system readability. The system, DIAS2, was implemented in Java, and we used AspectJ to restructure it. We introduced nine new aspects and measured the effects in three areas: code size, complexity[2] and structure. By using GQM we found nine metrics suitable for analysing the system to see if we had achieved improved readability. The use of AOP especially affected the values for lines of code, fan-in and fan-out. The conclusion of our study was that we had increased the readability of the chosen system through reduced code size, reduced complexity and improved structure. We also found several areas of possible further work: developing technique for identifying crosscutting concerns, metrics and metrics tools for AOP, refactoring with AOP, using AOP in resilient systems development, and using aspects for testing purposes. What we concentrate on in this diploma thesis is the task of developing a metrics tool that support AOP-implementations. The aim of the tool is to measure the code quality[3] and impact of introducing aspects into an OO-system.

1.1 Background, Metrics and Code Measurements

The goal of software metrics is, among other things, to improve understanding of a product or process. With better understanding one might better control the development and reach the goals for the product. According to Fenton [5] measurement has been considered as a luxury in software engineering. He claims that:

1. We fail to set measurable targets for our software projects
2. We fail to understand and quantify the development costs of software projects

---

[1] Telenor Mobile is Norway's leading supplier of mobile telephone services, personal paging and mobile data communication. At the turn of the year 2003/2004, Telenor Mobile had approximately 2.3 million mobile subscribers.

[2] Complex - Plaited together, interwoven. A whole comprehending in its compass a number of parts, especially of interconnected parts or involved particulars, a complex or complicated whole. Opposite to simple.
Complexity – the quality of being complex.

[3] Quality – ISO/IEC 8402 define quality as "'the totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs".".

3. We do not quantify or predict the quality of the products we produce and therefore we cannot tell a potential user for instance how reliable a product will be.
4. We allow anecdotal evidence to convince us to try yet another revolutionary new development technology, without doing a carefully controlled study to determine of the technology is efficient and effective.

Further, Fenton [5] states that code measurements are essential to good software engineering because:

1. They help us understand what happens during system development and maintenance.
2. They give us the opportunity to control the evolution of our projects.
3. They encourage us to improve our development process and our products.

1.2 Background, Aspect-oriented programming

Aspect-oriented programming is a programming paradigm invented at Xerox Parc in the 1990s. MIT Technology review wrote in 2001 that AOP is among "ten emerging areas of technology that will soon have a profound impact on the economy and on how we live and work" [2]. Today AOP is rapidly evolving as a concept within software development. There have been a great number of articles published during the latest three years. According to Kiczales et al. [3], AOP research is in many aspects similar to OOP research 20 years ago.

The motivation behind AOP [3] [4] is that existing programming languages cannot represent independent concerns like synchronisation, resource sharing, distribution or debugging in a single module. Rather than being localised within a program unit, like a class, these concerns are orthogonal to the system's basic program units and module structure.

1.3 Background, Project context

Custom Order Server (COS) is Telenor Mobile's middleware platform and is one of Norway's largest Java based systems [6]. COS is designed to give front-end applications such as retail outlets, customer support, large corporate customers and internal functions, a consistent view across multiple backend systems such as databases, network connections and mainframes. COS has during its five year lifetime evolved into a large system, composed of many subsystems.

Telenor Mobile has a slightly different approach to software metrics than that of Fenton. Their focus on measuring serves two purposes:

1. They want to watch trends over time, not the quality of the product but the way it changes
2. They use metrics on code together with design principles to ensure the removal of certain bad code blocks which they call broken windows. The anecdote behind the term "broken window", described by Hunt and Thomas [71], tells us that if we allow broken windows, the whole system will rapidly deteriorate.

XRadar [80] is an open extensible code analysis framework and report tool for Java, designed to support reengineering tasks [6]. XRadar was developed during a major refactoring project at Telenor Mobile, the Pareto Reengineering Project. Its vision is to bring the advantages of different open source analysis frameworks together, through being extended to support new languages, frameworks, measurements and reports. In this context, making a plugin tool that support measurements on AOP-code is seen as a suitable extension of the XRadar framework.

1.4 Problem definition

One of the suggested areas of further work from our readability study was to develop a metric tool that supports AspectJ code [1]. During our study we found that the metrics intended for examining AOP-based code was not well defined and the tools we used did not support the AspectJ programming language. This forced us to count several metrics within the aspects manually.

As a continuation of this problem area we wish to pursue the following problem definition:

- **We will find useful metrics for measuring AOP-based programs and**
- **We will develop a prototype metric tool for use with AspectJ.**

The main goals for the AOP-metrics we find are:

- When combined they should measure high level quality factors for the Telenor Mobile COS system.
- They should be used to support or disprove claims about the abilities and usefulness of AOP; not by achieving a specific level or numerical value, but to show improvement, or opposite, and trends through time and change when using AOP.

1.5 Project work goals

With the problem definition as a starting point, we have defined the following work goals for this diploma thesis:

1. Present a theoretical foundation for AOP and the concerns relevant for studying this concept.
2. Evaluate the system quality factors for the COS system. Derive sensible quality factors for an aspect-oriented system.
3. Find possible transitions from quality factors to metrics
4. Justify our choice of metrics. This will answer our first problem definition.
5. Construct a grammar/tree for AspectJ-programs
6. Create a design for the metric tool
7. Implement the metrics tool prototype
8. Find a good way to present the metric results. This will answer our second problem definition.

In the following section the outline of this work is presented.

1.6 Report outline

This is a presentation of the contents discussed in the remaining chapters of this report.

- **Chapter 2:** Theoretical foundation. A presentation of concepts and principles relating to AOP and metrics.
- **Chapter 3:** Relevant technologies. An introduction to the technologies relevant to the metrics tool developed.
- **Chapter 4:** Quality analysis. A discussion of what criteria must be present to satisfy the system quality factors and which of these we are able to measure to verify that the criteria have been met. Further, we consider the expected effects AOP will have on our chosen measurements.
- **Chapter 5:** The metrics tool. A description of the measurement tool, from functional and non-functional requirements to implementation.
- **Chapter 6:** Systems explored. A presentation of measurements results we have gathered when comparing the DIAS system before and after restructuring it with aspects.
- **Chapter 7:** Discussion and critique. The work and choices done in chapter 3-6 are discussed and criticized.
- **Chapter 8:** Conclusion.
- **Chapter 9:** Further work. A presentation of possible extensions to the tool, verification of results over a longer time period, inclusion of other quality factors and further studies on the effects of AOP.

Chapter 2. Theoretical foundation

In this chapter we introduce theory that is the foundation for our work on Aspect-Oriented Programming and metrics.

2.1 Introduction to Aspect-Oriented Programming

It is important to note that AOP is not a standalone programming paradigm like object-oriented programming and imperative programming. AOP is always used together with other programming paradigms. It can best be described in terms of concerns, joinpoints and aspects.

Concerns are system properties or areas of interest in a system. Separation of concerns is a main principle in software engineering [7]. Concerns crosscut if the methods related to those concerns intersect [8], either inside a class or over several classes. AOP provides a way of encapsulating crosscutting concerns. This can be seen in Figure 1, where all the marked methods have to check user rights.

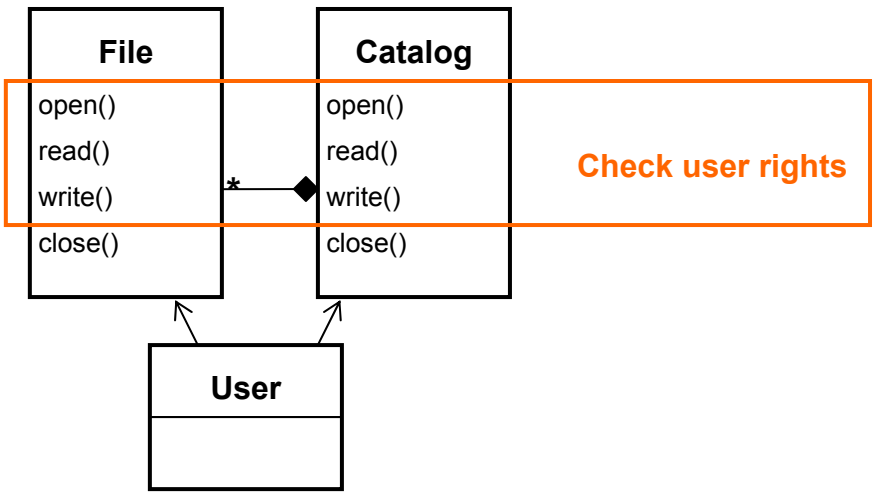


Figure 1 Crosscutting concerns in UML class diagram: all the methods encapsulated have to check if `user` has the right to perform this action.

Such concerns can be based both on functional and non-functional requirements. Examples are logging, security, caching and buffering.

Joinpoints are the locations which are affected by one or more crosscutting concerns. In Figure 2 we can see examples of possible joinpoints in the calling and returning points of a

method. Joinpoints are the locations where we can hook on new actions before or after executing the original code.

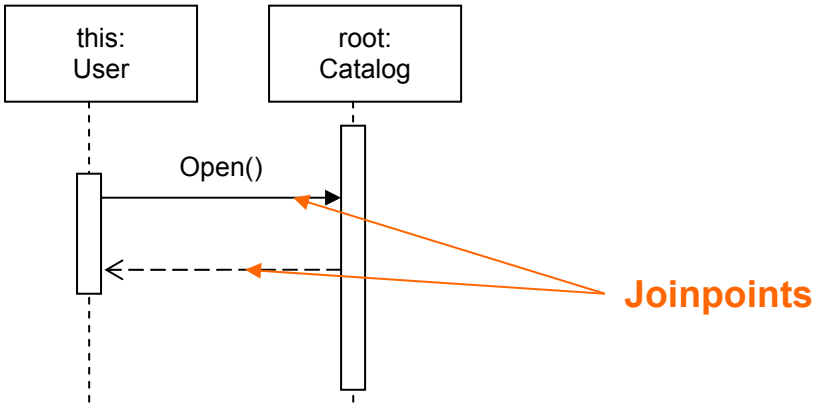


Figure 2 Joinpoints in UML sequence diagram. The calling and returning of methods can be used as joinpoints.

Aspects are fundamental to the definition of AOP. Aspects are design decisions that are difficult to address in regular OO-code because they crosscut the system. With AOP we can separate aspects from the underlying structure of the code; as in the example shown in Figure 1. We can move the updating of the display to a separate subprogram. This subprogram is called an aspect.

2.2 AspectJ

AspectJ is a general purpose aspect-oriented extension of Java. It is a free implementation and language specification developed at Xerox PARC. The language specification defines several constructs and their semantics to support aspect-oriented concepts. The language implementation consists of tools for compiling, debugging, and documenting code. AspectJ's language constructs extend the Java programming language. Every valid Java program is also a valid AspectJ program. The byte code produced by the AspectJ compiler is standard Java byte code, thus it keeps Java's advantages. AspectJ enables both name-based and property based crosscutting. Aspects that use name-based crosscutting tend to affect a small number of other classes. But despite their small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property-based crosscutting can have small or large scale. [74] [78]

The core constructs of AspectJ are discussed in the following sub-sections.

2.2.1 Joinpoints

Joinpoints are well-defined points in a program's execution. In AspectJ the following points in the program execution can be used as joinpoints [74]:

- Method call and execution

- Constructor call and execution
- Read/write access to field
- Exception handler execution
- Object and class initialization execution

2.2.2 Pointcuts

Pointcuts are program constructs used to designate joinpoints; they let you specify a joinpoint collection. They can also expose context at the joinpoint location for use in an advice implementation. If we study the code example from Figure 4 on page 11 we find this pointcut:

```
pointcut publicGet(Object key) : execution(*
          AdressList.get*(..) && args(key);
```

First we have the construct, `pointcut`, which declares that what follows is the declaration for a pointcut. Next, `publicGet`, is the pointcut's name and `Object key` is the context that is collected from the joinpoint location.

What captures joinpoint locations is the statement `execution(* AdressList.get*(..) && args(key)`. `execution` indicates that the poincut captures the execution of a method, as opposed to the call to a method. The `* AdressList.get*(..) && args(key)` is the signature for methods to be captured. The first `*` indicate that there is no discrimination of type of method, as opposed to only public/private, static, with return type String/int/void and so on. `AdressList.get*` dictate that the captured method should belong to the class AdressList and the method name should begin with 'get'. Moving on, `(..)` indicate that methods should be captured regardless of their arguments, as opposed to for instance only those that carry a String argument.

The `&&` is used to combine the named pointcut in front of it with the anonymous pointcut following it. Finally, `args(key)` captures the information needed, called context, about a method and the arguments to it.

In Table 1 we present examples of the different available types of pointcuts [78].

Pointcut	**Description**
Call to methods and constructors pointcuts	
`call(public void MyClass.myMethod(String))`	Call to `myMethod()` in `MyClass` taking a `String` argument, returning `void`, and with `public` access
`call(public * com.mycompany..*.*(..))`	All public methods in all classes in any package with `com.mycompany` the root package

Execution of methods and constructors pointcuts	
`execution(public void MyClass.myMethod(String))`	Execution of `myMethod()` in `MyClass` taking a `String` argument, returning `void`, and with `public` access
`execution(public * com.mycompany..*.*(..))`	All public methods in all classes in any package with `com.mycompany` the root package
Field-access pointcuts	
`get(PrintStream System.out)`	Execution of read-access to field `out` of type `PrintStream` in `System class`
`set(int MyClass.x)`	Execution of write-access to field `x` of type `int` in `MyClass`
Exception-handler pointcuts	
`handler(RemoteException)`	Execution of catch-block handling `RemoteException` type
`handler(IOException+)`	Execution of catch-block handling `IOException` or its subclasses
Class-initialization pointcuts	
`Staticinitialization(MyClass+)`	Execution of static block of `MyClass` or its subclasses
Lexical-structure-based pointcuts	
`within(MyClass)`	Any pointcut inside `MyClass`'s lexical scope
`withincode(* MyClass.myMethod(..))`	Any pointcut inside lexical scope of any `myMethod()` of `MyClass`
Control-flow-based pointcuts	
`cflow(call(* MyClass.myMethod(..))`	All the joinpoints in control flow of call to any `myMethod()` in `MyClass` including call to the specified method itself
`cflowbelow(call(* MyClass.myMethod(..))`	All the joinpoints in control flow of call to any `myMethod()` in `MyClass` excluding call to the specified method itself
Self-, target-, and arguments-type pointcuts	
`this(JComponent+)`	All the joinpoints where this is instanceof `JComponent`
`target(MyClass)`	All the joinpoints where the object on which the

	method is called is of type `MyClass`
`args(String,..,int)`	All the joinpoints where the first argument is of `String` type and the last argument is of `int` type
Conditional-test pointcuts	
`if(EventQueue.isDispatchThread())`	All the joinpoints where `EventQueue.isDispatchThread()` evaluates to true

Table 1 Examples of available pointcuts in AspectJ. Pointcuts are program constructs that let you specify a joinpoint collection.

2.2.3 Advices

Advices specify the executable code when reaching a certain pointcut. There are three types of advices, as seen in Table 2 [78].

Advice-type	**Description**
before()	A before advice runs just before the joinpoint.
after()	An after advice runs just after the joinpoint, and can be specified to run after a normal return, after throwing an exception, or regardless of what kind of return from a jointpoint it is.
around()	An around advice encapsulates a joinpoint and controls if the joinpoint is to be executed or not. It can also execute a different argument set.

Table 2 The advice-types available in AspectJ. Advices specify the executable code when reaching a certain pointcut.

In the example from Figure 4 we find two around advices. One of them is:

```
after(Object key)  returning(value): publicGet(key)
        {
     cacheValue(key, value);
}
```

This advice is executed after the method `publicGet` has executed and returned (but before execution is handed over to the method that called `publicGet`). The advice fetches the

context `Object` key, and executes the method `cacheValue`, with `key` and `value` as arguments. Then execution continues as normal.

2.2.4 Inter-type declarations

Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. [74]

Consider the problem of expressing a capability shared by some existing classes that are already parts of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds a method to each affected class that implements the interface. AspectJ can express the problem in a single place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields to the existing classes.

An example of using inter-type declarations could be if we had an existing class `Point`, and we want `Screen` objects to observe changes to `Point` objects. We can implement this by writing an aspect declaring that the class `Point` has an instance field, `observers`. This field then keeps track of the `Screen` objects that are observing `Point`s, as seen in Figure 3.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    ...
}
```

Figure 3 An example of an inter-type declaration. An inter-type declaration can declare members that cut across multiple classes, or change the inheritance relationship between classes.

The `observers` field is private, so only the aspect `PointObserving` can see it. Observers are added or removed with the static methods `addObserver` and `removeObserver` on the aspect. Then further functionality can be added to declare what we need to do when we observe a change in a `Point`. [74]

2.2.5 Aspects

Aspects are AspectJ's unit of modularization, the same way classes are in Java. An aspect contains pointcuts and advices, and as classes, can have methods and fields, extend other

classes and aspects and implement interfaces. Aspects differ from classes in that you can not create an aspect object using `new`, as aspects can not be accessed directly. By default, each aspect is a singleton, so one aspect instance is created. Only aspects can hold advices, but classes can declare static pointcuts. Both aspects and pointcuts can be declared as abstract and act similarly to a class' abstract methods; they let you defer details to the derived aspects. A concrete aspect extending the abstract aspect can then provide concrete definitions for abstract pointcuts. An aspect can be declared dominant over others to control precedence when multiple aspects affect the same joinpoint. If an aspect is declared `privileged` it is given access to private members of aspected classes. [74] [78]

2.2.6 Example of an aspect

We end our introduction of AspectJ by giving an example of how caching can be implemented as an aspect. In this example, shown in Figure 4, we have an address list which can be accessed using keys like phone number or name, and it returns values like address or name. The real address list data might be stored in a database somewhere else and accessed with SQL statements. Therefore caching might be needed to improve performance. The aspect provides this function by checking if the called key is found in a caching table. If it is, return the value. Every time, when returning a value that is not cached, the value and its key is cached. How the caching in the table is, however, performed is not shown in the example.

```
public class AddressList {
  ...
  public String getAdressByName(String
name) {
    …
  }

  public String getAdressByPhone(Integer
phone) {
    …
  }

  public String getNameByAdress(String
name) {
    ..
  }

  …
}
```

```
public aspect Caching {
  ...

  pointcut publicGet(Object key) :
    execution(* AdressList.get*(..) &&
    args(key);

  around(Object key) : publicGet(key) {
    Object cachedValue =
              getCachedValue(key);
    if (cachedValue == null) {
      proceed(key);
    }
    return cachedValue;
  }

  after(Object key)  returning(value):
            publicGet(key)  {
        cacheValue(key, value);
  }


  public Object getCachedValue(Object
key) {
    …
  }

  public void cacheValue(Object key,
Object value) {
    ..
  }
}
```

Figure 4 An AOP example in AspectJ, caching of values[4].

[4] Remark that this example requires the key to be distinct for all the methods. This is because AspectJ implements an aspect as a singleton.

2.3 Quality frameworks and methods

The quality of software can be seen as multidimensional in the way quality factors are connected to other quality factors. There are many approaches that can be used to define and understand such quality. The ISO 9126 standard [21] provides a framework for evaluating software quality. The Goal-Question-Metrics (GQM) approach aims at giving us a method to define software quality metrics. Alternatives to both ISO 9126 and GQM exist, e.g. design by measurable objectives [43] and GQM++ [44].

According to the ISO 9126 standard [21], software product quality should be evaluated using a defined quality model. However, "it is not practically possible to measure all internal and external subcharacteristics for all parts of a large software product." Resources for evaluation need to be allocated depending on the business objectives.

The ISO 9126 standard consists of six main quality characteristics that are generally important for software. The importance of the characteristics, shown in Figure 5, must be derived in accordance to your software and business objectives.

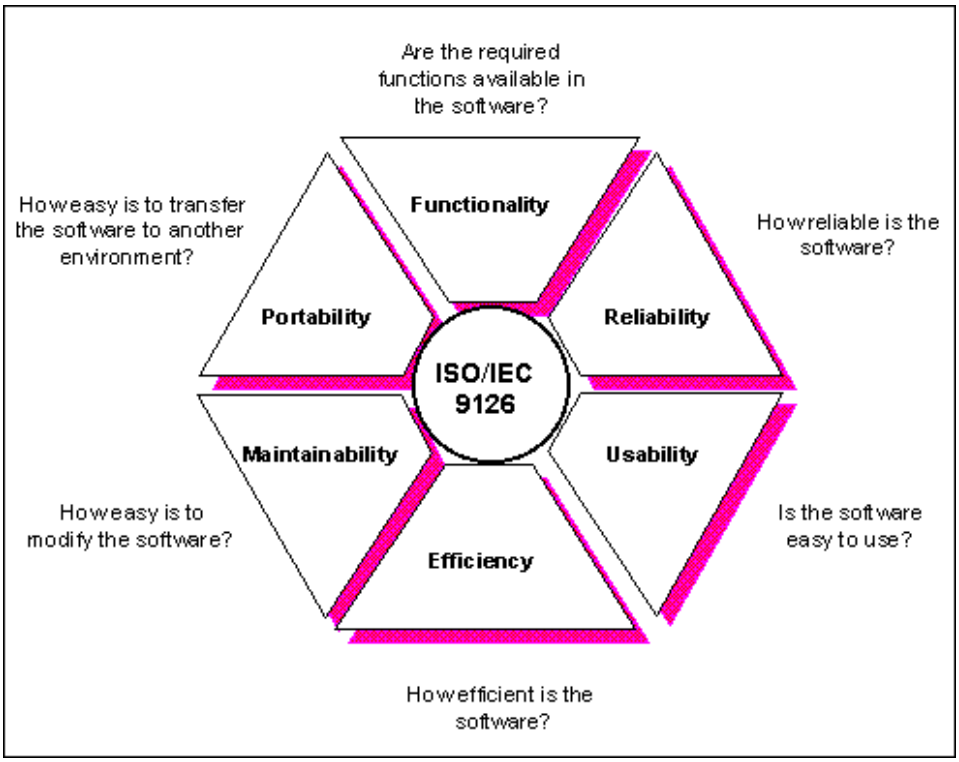


Figure 5 The six quality characteristics of the ISO 9126 standard

The six quality characteristics of the ISO 9126 standard are all divided into subcharacteristics. These characteristics and their definitions are shown in Table 3.

Characteristics	Subcharacteristics	Definitions
Functionality	Suitability	Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks.
	Accurateness	Attributes of software that bear on the provision of right or agreed results or effects.
	Interoperability	Attributes of software that bear on its ability to interact with specified systems.
	Compliance	Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions.
	Security	Attributes of software that bears on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data.
Reliability	Maturity	Attributes of software that bear on the frequency of failure by faults in the software.
	Fault tolerance	Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface.
	Recoverability	Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it.
Usability	Understandability	Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability.
	Learnability	Attributes of software that bear on the users'effort for learning its application.
	Operability	Attributes of software that bear on the users'effort for operation and operation control.
Efficiency	Time behaviour	Attributes of software that bear on response and processing times and on throughput rates in performances its function.
	Resource behaviour	Attributes of software that bear on the amount of resource used and the duration of such use in performing its function.

Maintainability	Analyzability	Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
	Changeability	Attributes of software that bear on the effort needed for modification, fault removal or for environmental change.
	Stability	Attributes of software that bear on the risk of unexpected effect of modifications.
	Testability	Attributes of software that bear on the effort needed for validating the modified software.
	Adaptability	Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered.
Portability	Installability	Attributes of software that bear on the effort needed to install the software in a specified environment.
	Conformance	Attributes of software that make the software adhere to standards or conventions relating to portability.
	Replaceability	Attributes of software that bear on opportunity and effort using it in the place of specified other software in the environment of that software.

Table 3 The subcharacteristics of the ISO 9126 standard

The model proposed in ISO 9126 is not the only model that describes quality using a decompositional approach. Early models have been proposed by McCall in 1977 and Boehm et al. in 1978 [5]. In addition to such fixed models, you might also make your own quality model based on own and prospective users' notions. A fixed model will then just be a guide in the work of defining our own model. According to Fenton [5], the define-your-own-model approach has been pioneered by Gilb and by Kithenham and Walker.

Gilb's method can be thought of as "design by measurable objectives". According to Gilb [43], quality attributes are performance attribute which tell us how well the system performs. Gilb claims that "performance requirements must express quantitatively the stakeholder's requirements." With this in mind, Gilb introduce a three-step process to describing performance requirements:

1. Identify components/objectives of the attribute
2. Describe the essence of each component/objective
3. Specify the requirements in measurable and testable terms.

The Goal-Question-Metrics (GQM) paradigm aims at tying measurement to the overall goals of projects and process [5]. This approach was first suggested by Basili and colleagues (1984), and has turned out to be an effective way of selecting and implementing metrics. To use GQM in an organization, you must first express the organization's overall goals within a chosen scope. This scope could be a project, a department or the whole organization. The second task is to generate questions whose answers you must know in order to determine if your goals are met. Finally, each question must be analyzed in terms of what measurements you need in order to answer this question. The result of a GQM process is shown in Figure 6.

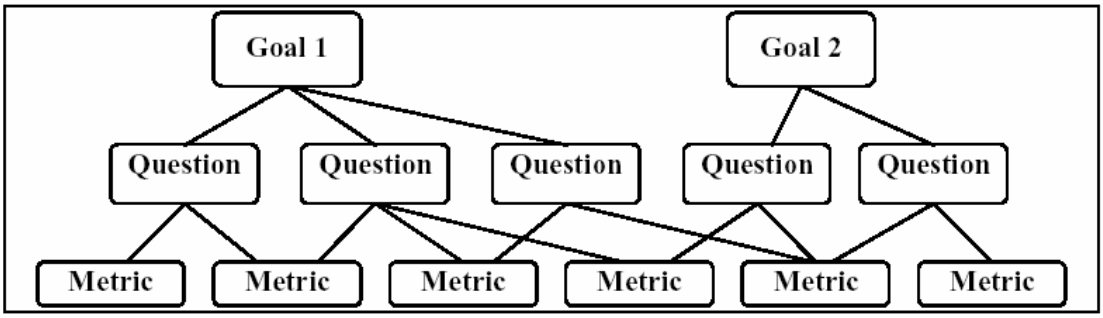


Figure 6 The Goal-Question-Metrics approach (Basili et al., [42])

In Fenton [5] we can find the following input for goal definition in GQM:

- **Purpose:** To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to (understand, assess, manager, engineer, learn, improve, etc.) it.

- **Perspective:** Examine the (cost, effectiveness, correctness, defects, changes, product measures, etc.) from the viewpoint of the (developer, manager, customer, etc.)

- **Environment:** The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc.

The GQM template can be related to attribute frameworks, like the one defined in ISO 9126 shown earlier.

There are a number of alternatives to GQM, some of which take a different approach, like Omnibus Software Quality Metrics, and some that extend GQM, like GQM++. A summary of methodologies is given by Lee and Chang [44].

Instead of the top-down approach given in the above mentioned methodologies and models, we may go in the opposite direction starting by describing a metric, and then showing which quality factors is affected by this metric. This approach is taken in the Chidamber & Kemerer metrics suite [32].

2.4 Construction of metrics

According to Fenton [5], we seek to formalize our intuition about the way the world works by attempting to represent it through a set of measurements. The data we obtain should therefore represent the entities we observe, and manipulation of the data should preserve relationships that we observe among entities.

According to V.R. Basili et al. [42], in order to be effective, measurement must be:

1. Focused on specific goals
2. Applied to all life-cycle products, processes and resources
3. Interpreted based on characterization and understanding of the organizational context, environment and goals.

2.4.1 Formal rules

Measurement is formally defined as a mapping from the empirical world to the formal, relational world. A measure is the number or symbol assigned to an entity by this mapping in order to characterize an attributes. A measure must specify the domain and range and the rule for performing the mapping. The mapping must map entities into numbers and empirical relations into numerical relations; this is called the representation condition. An example of such empirical relation is "taller than". [5]

2.4.2 Models and measurement types

To focus on what we find important, we make a model which is an abstraction of reality. This allows us to strip away detail and view an entity or concept from a particular perspective. When using such models it is important to give careful thought to the empirical system, not just the mathematical. We must focus on how the model maps onto the real world. Once we have a model of the entities and attributes, we can define the measure in terms of them. Sometimes when the relationships are simple, it is possible to define direct mappings from attribute to number. When we have complex relationships among attributes, or when an attribute must be measured by combining several of its features, we need a model of how to combine the related measures. We thus have two ways to do measurement:

- Direct measurement of an attribute which involves no other attribute
- Indirect measurement of an attribute which must be measured in terms of other attributes.

Indirect measurement can also be useful in making visible interactions between direct measurements. In addition to dividing measurements into direct and indirect, we must also differ between subjective and objective measures. An objective measure only depends on the object at hand, not on judgement. Example of such measurement is length. Subjective measures depend on judgement, e.g. personal skills and quality. [5]

2.4.3 Scales and statistics

The mapping of measurement can be done in several ways, driven by measurement scales. The choice of scale can restrict the kind of analysis we can do. According to Fenton [5], we

generally have five main types of scales, and they are restricted in terms of statistical analysis. Wohlin et al [25] argue though that the fifth, the absolute scale, is just a special case of the ratio scale. The four scales he uses are nominal, ordinal, interval and ratio, as can be seen in Table 4.

Scale type	Defining relations	Measure of		
		central tendency	dispersion	dependency
Nominal	Equivalence	Mode	Frequency	
Ordinal	Equivalence Greater than	Median Percentile	Interval of variation	Spearman corr. coeff Kendall corr. Coeff
Interval	Equivalence Greater than Known ratio of any intervals	Mean	Standard deviation Variance Range	Pearson corr. Coeff
Ratio	Equivalence Greater than, Known ratio of any intervals Known ratio of any two scale values	Geometric mean	Coefficient of variation	

Table 4 Scales and relevant statistics [5] [25].

2.4.4 Main types of product metrics

There is more than one way of classifying metrics, and we will not elaborate on all possibilities. Classification might be done with respect to quality factors, time of use, attribute that is measured, indirect or direct use and so on.

As our focus in this project is a metrics tool that can give us measures related to the code, we want to concentrate on main types of internal product metrics. Fenton [5] focuses on the two areas size and structure. He has found some important aspects of size which are:

- Length, the size of the product
- Functionality, what the user gets
- Complexity, in terms of the underlying problem that the software is solving
- Reuse[5], the extent to which the software is genuinely new.

[5] We define reuse as for “parts of a system”, not only the “system as a whole”.

The structure of the product and its quality might be linked. Researchers think, according to Fenton [5], that structure metrics might help us understand the difficulties we have in converting a product to another (e.g. design to code), in testing code or in predicting external software attributes from early internal product measures. Although such metrics vary in what they measure or how they measure, they are often called complexity metrics[6].

Fenton [5] has divided structure (complexity) metrics into three classes:

1. control-flow structure - addresses the sequence in which instructions are executed
2. data-flow structure - follows the trail of a data item as it is created or handled
3. data structure - is the organization of the data itself

2.4.5 Frameworks for cohesion and coupling

Briand et al. [30] [31] have proposed frameworks for cohesion and coupling measurements in object-oriented systems. Their motivation is to facilitate comparison, evaluation and validation of existing metrics and support definition of new metrics. They also want to support the choice of metric according to chosen goals.

The frameworks will, according to Briand et al.:

- ensure that measure definitions are based on explicit decisions and well understood properties
- ensure that all relevant alternatives have been considered for each decision made
- highlight dimensions of cohesion/coupling for which there are few or no measures defined

The coupling framework consists of six criteria which determine basic aspects of the resulting measure, as shown in Table 5.

Criterion	**Description**
Type of connection	The mechanism that constitutes coupling between two classes, e.g. references, is type of and invokes on.
Locus of impact	Import or export coupling
Granularity of the measure	Level of detail at which information is gathered, meaning domains, like method, class and package, and how connections are counted, like every individual connection or number of distinct items at the other end.

[6] Fenton finds the term “complexity metrics” misleading, but also uses the same term in his book [5].

Stability of server	How stable the classes analyzed are, like build or release. (No existing measure addresses this criterion.)
Direct or indirect coupling	If we also count indirect couplings, connections of the form a→b→c.
Inheritance	How inheritance-based coupling and polymorphism is accounted for.

Table 5 Six criteria of the Briand et al. coupling measurement framework

In the cohesion framework, there are five similar criteria, shown in Table 6.

Criterion	**Description**
Type of connection	What makes a class cohesive, which element links are counted, e.g. method to attribute and method to method
Domain of the measure	Level of detail at which information is gathered, meaning domains, like method, class and package.
Direct or indirect connections	If we also count indirect couplings, connections of the form a→b→c.
Inheritance	Two aspects are considered: a) how do we assign methods and attributes to classes b) for method invocation, shall we consider static or polymorphic invocations
How to account for access methods and constructors	Access methods: how to account for get- and set-methods. Treat constructors like ordinary methods or exclude them.

Table 6 Five criteria of the Briand et al. cohesion measurement framework

Briand et al. have also proposed some properties for the theoretical validation of coupling and cohesion metrics. They claim that measurements not fulfilling those properties must be ill-defined, but fulfilling all does not guarantee a valid measure. The properties of the two frameworks are shown in Table 7. We have based our choice of coupling and cohesion metrics on the discussion of alternatives presented in [30] and [31].

Coupling properties	**Cohesion properties**
1. The coupling count is nonnegative for a class. 2. The coupling for a class is null if there are no relations with other	1. Nonnegativity and normalization, meaning the cohesion count can be between zero (null) and some max value. 2. Null value and max value, where null

classes. 3. Monotonicity, adding relations means monotony increasing the coupling count. 4. Merging of classes means less or equal coupling counts in total 5. Merging of unconnected classes means equal coupling counts in total.	means no relations and max means all possible relations within a unit. 3. Monotonicity, adding internal relations means monotony increasing the cohesion count. 4. Merging of unconnected classes result in decreased or unchanged cohesion count.

Table 7 Validation properties of the coupling and cohesion frameworks by Briand et al.

2.5 OO-metrics

As the base language for AspectJ is the object-oriented language Java, we introduce metrics and metrics suites relevant to object-oriented structures.

2.5.1 A selection of well-known OO-metrics

The following is a collection of metrics that have been used in order to measure code quality in object-oriented systems. When implementing our system we have considered which metrics are best suited to capture the system quality factors of Telenor Mobile COS and the properties of an AOP-system.

Lines of code

There are several ways to measure lines of code.

- The straight-forward counting of every line.
- Counting all non-blank lines.
- Counting all non-comment and non-blank lines.
- Counting all non-comment and non-blank lines inside method bodies.
- Counting all code statements.

Fan-in / fan-out

Sommerville [11] define fan-in and fan-out as follows: “Fan-in is a measure of the number of functions that call some other function (say X). A high value for fan-in means that X is tightly coupled to the rest of the design.” Changes to X will have extensive ripple effects. “Fan-out is the number of functions which are called by function X. A high value for fan-out suggests that the overall complexity of X may be high due to the control logic needed to coordinate the called components.”

Complexity measures

A summary of complexity measures is given in Appendix C.

Coupling

According to Schach [19], coupling can be defined as "the degree of interaction between two modules". Coupling can be divided into five levels, from bad to good, as seen in Table 8.

Type of coupling	Description
1. Content coupling (bad)	Two modules are content-coupled if one directly references the content of the other.
2. Common coupling	Two modules are common-coupled if they both have access to the same global data.
3. Control coupling	Two modules are control-coupled if one module explicitly controls the logic of the other.
4. Stamp coupling	Two modules are stamp-coupled if a data structure is passed as an argument, but the called module operates on only some of the individual components of that data structure.
5. Data coupling (good)	Two modules are data-coupled if all arguments are homogeneous data items. Every argument is either a single argument or a data structure in which all elements are used by the called module.

Table 8 Levels of coupling.

Afferent coupling is the number of classes outside a specific module that depend on this module. Efferent coupling is the number of dependencies a specific module has to classes outside the module [10]. A similar notation is used by Briand et al. [63]. They introduce a client-server-relationship between the classes. The client class uses (imports services), the server class is being used (exports services). The distinction between an importing and an exporting class is important. A class which mainly imports services may be difficult to reuse in another context because it depends on many other classes. On the other hand, defects in a class which mainly exports services are particularly critical as they may propagate more easily to other part of the system and are more difficult to isolate. For the remaining parts of the report we will use the terms import and export coupling, as we find them more intuitive than the terms afferent and efferent coupling.

Hitz and Montazeri [53] discuss the concept of coupling in OO systems. Two levels of coupling are defined, as seen in Table 9.

Coupling	Description
Class level coupling (CLC)	CLC can occur if a method of a class invokes a method or references an attribute of another class.
Object level coupling (OLC)	OLC represents the coupling resulting from state dependencies between two objects during the run-time of a system.

Table 9 Hitz and Montazeri's coupling definitions

CLC is given weights according to stability, access type and scope of access. OLC is given weights according to type of access, scope of access and complexity of interface (number of arguments). The values from the couplings are ordinal and cannot be summed; thus the two scales are incomparable.

Cohesion

According to Fenton [5], "the cohesion of a module is the extent to which its individual components are needed to perform the same task". Schach [19] presents seven levels of cohesion, from bad to good, as seen in Table 10.

Type of cohesion	Description
1. Coincidental cohesion (bad)	A module has coincidental cohesion if it performs multiple completely unrelated actions.
2. Logical cohesion	A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module.
3. Temporal cohesion	A module has temporal cohesion when it performs a series of actions related in time.
4. Procedural cohesion	A module has procedural cohesion if it performs a series of actions related by the sequence of steps to be followed by the product.
5. Communicational cohesion	A module has communicational cohesion if it performs a series of actions related by the sequence of steps to be followed by the product and if all the actions are performed on the same data.

6. Informal cohesion (good, object-oriented programming)	A module has informal cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure.
7. Functional cohesion (good, structured programming)	A module that performs exactly one action or achieves a single goal has functional cohesion.

Table 10 Levels of cohesion

Chidamber and Kemerer [32] propose the metric Lack of Cohesion in Methods (LOCM). LCOM is based on the number of disjoint sets of instance variables that are used by a method. The refined version of Chidamber and Kemerer's LCOM is defined as follows:

Consider a class C_1 with n methods $M_1, M_2, \ldots, M_n$.

Let I_j = set of instance variables used by the method M_j. There are n such sets $I_1, \ldots, I_n$. Let $P = \{ (I_i , I_j) \mid I_i \cap I_j = \emptyset \}$ and

$Q = \{ (I_i , I_j) \mid I_i \cap I_j \neq \emptyset \}$

If all n sets $I_1, \ldots, I_n$ are Ø then let P = 0.

$$\begin{aligned} LCOM &= |P| - |Q|, && \text{if } |P| > |Q| \\ &= 0 && \text{otherwise} \end{aligned}$$

The smaller the number of disjoint sets, the more cohesive is the class. LCOM = 0 indicates a cohesive class since all methods share instance variables. For LCOM > 0, the developer should consider splitting the class into two or more classes since instance variables belong to disjoint sets.

There are doubts about the usefulness of LCOM in Java since it penalizes the proper use of getter and setter methods as the only methods that directly access an attribute and the other methods using the getter/setter methods to accessing an attribute [15]. To achieve high cohesion, all methods in a class should use most of the instance variables. Getter and setter methods only access one variable, which evidently leads to poor cohesion values. It is clear that this way of measuring cohesion is not suited for measurements within the object-oriented paradigm.

Hitz and Montazeri [53], present a graph theoretic version of Chidamber and Kemerer's cohesion metric:

Let X denote a class, I_X the set of its instance variables and M_X the set of its methods.

Consider a simple undirected graph $G_X(V,E)$ with $V = M_X$ and

$E = \{ (m , n) \in V \times V \mid \exists i \in I_X : (m \text{ accesses } i) \wedge (n \text{ access } i) \}$

LCOM is then defined as the number of connected components of G_X.

Hitz and Motazeri identified a problem concerning LCOM with regards to access methods. An access method provides read or write access to an attribute of a class. Access methods typically reference only one attribute. If other methods of the class use the access methods, they may no longer need to directly reference any attributes at all. Thus, the presence of access methods artificially decreases the class cohesion.

To overcome the limitations of LCOM they defined connectivity metric to be used in conjunction with the LCOM metric. The connectivity metric would measure the structural differences between the methods of a set of classes with the same value for LCOM.

$$C = \left| \frac{E - (n-1)}{(n-1) * (n-2)} \right|$$

C is a measure of the deviation of any given graph from the minimally cohesive case.

Bieman and Kang [64] have also based their cohesion measure on the proposal by Chidamber and Kemerer. A method can use an attribute directly by referencing it, or indirectly by calling a method referencing the attribute. They define a predicate $cau(m_1, m_2)$ (common attribute usage) which is true, if an accessable method ($m_1, m_2 \in M_I(C)$) directly or indirectly use an attribute of class c in common. The measure TCC (tight class cohesion) is then defined as the percentage of pairs of public methods of the class with common attribute usage. The measure LCC (loose class cohesion) is calculated as the ratio of all directly connected methods and indirectly connected methods in a class to the maximum possible number of connections in a class.

Bieman and Kang identified a problem with constructor methods for the calculation of TCC and LCC. Constructor methods provide the class attributes with initial values and therefore access most or all of the class' attributes. The presence of a constructor method artificially increases cohesion as measured by TCC and LCC. Bieman and Kang recommend excluding constructors (and destructors) from the analysis of cohesion.

Package stability

Import and export coupling can be used to determine how stable[7] a package is. According to the stable dependencies principle [69] packages should depend on packages that are more

[7] Stability - Difficult to move or change. A change in the package might impact or break many other packages that are package dependent on upon it.

stable than themselves. An instable package that is used by many other packages is a potential problem in the system. This leads on to another principle: the stable abstraction principle. The more stable a package is, the more it should consist of abstract classes. The foundation for this principle is that methods are changed more often than the interfaces between modules. Thus interfaces are more stable than executable code.

Then we have the open/closed principle: "Software entities[8] should be open for extension, but closed for modification" [70]. This means that you should never change the code in your modules, only extend them. This can be done by creating concrete implementations of the abstract interfaces. All packages should not be stable, since a stable package is also an inflexible package. There is consequently a balance between a package's stability and its abstractness. Package instability can be calculated as:

$$I = \#\text{Import coupling} / (\#\text{Export coupling} + \#\text{Import coupling})$$

Package abstractness is simply:

$$A = \#\text{Abstract classes} / \#\text{Total classes}$$

Then we can plot the instability and abstractness values in graph as shown in Figure 7.

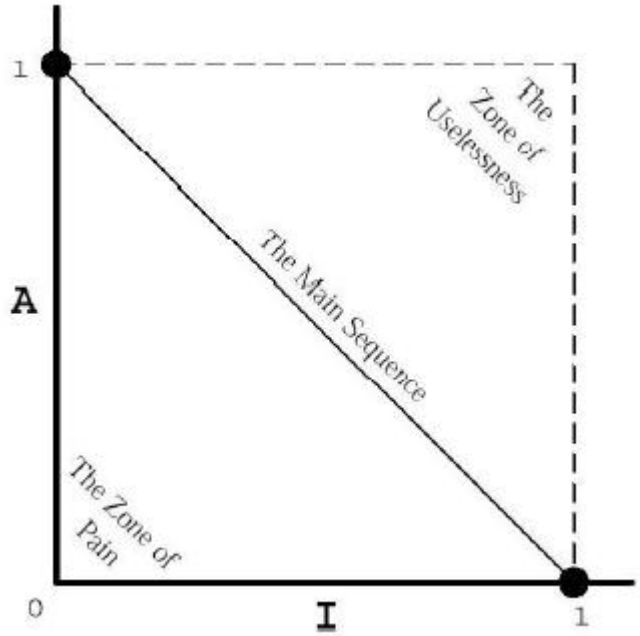


Figure 7 The ideally balanced package is placed along the main sequence

The main sequence represents the line where packages are balanced between how abstract they are and how stable they are. The further away a package is from the main sequence, the harder it is to use and maintain. The zone of pain indicates concrete packages that a lot of other packages depend upon. Changes to a package in this area will have a high possibility of ripple effects throughout the dependent packages. The zone of uselessness are abstract packages that few or none other packages depend upon. A package distance from the main sequence can be calculated as:

$$D = | A + I - 1 |$$

[8] Classes, modules, functions, etc.

A package witch distance from the main sequence that is not close to zero should be examined for possible restructuring.

2.5.2 Object-oriented metrics suites

There have been proposed several collections of metrics that have aimed to show the bigger picture in terms of one or more system properties.

C&K metrics suite

Chidamber and Kemerer [32] suggest a suite of measures for object-oriented systems. Six metrics are defined, as seen in Table 11.

Metric	**Description**
Weighted Methods per Class (WMC)	WMC is the number of methods included in a class weighted by the complexity of each method. The larger the value for this metric, the more complex the object class is [11]. A common way of measuring complexity of the methods is to calculate McCabe's Cyclomatic Complexity [62].
Depth of Inheritance Tree (DIT)	In an object-oriented design, the application domain is modelled as a tree, called the inheritance tree. The DIT metric is the length of the maximum path from the node to the root of the tree. DIT is a measure of how many ancestor classes can potentially affect this class.
Number of Children (NOC)	NOC is the number of immediate successors of the class.
Coupling between objects classes (CBO)	A class is coupled to another class if it uses its member functions and/or instance variables.
Response for class (RFC)	This measure is defined to be the size of the response set of a class. The response set consists of all the methods called by local methods. RFC is the number of local methods plus the number of methods called by local methods.

Lack of cohesion of Methods (LCOM)	Cohesion is characterized by how closely the local methods are related to the local instance variables in the class. LCOM is defined as the number of disjoint (non-intersecting) sets of local methods.

Table 11 The Chidamber-Kemerer metrics suite.

The six metrics shown in Table 11 are designed to meet the three non-implementation steps in Booch's definition of object-oriented design [79], and have been categorized in Table 12. The columns of the table concerns identification of classes, behaviour of classes, and the communication between classes.

Metric	Identification	Semantics	Relationships
WMC	X	X	
DIT	X		
NOC	X		
RFC		X	X
CBO			X
LCOM		X	

Table 12 Chidamber and Kemerer's metrics mapped to Booch's OOD-steps.

Churcher and Shepperd [35] [36] points out that definitions of some of the basic direct counts were imprecise, such as the number of methods in a class. There are several ways to count methods, and Churcher and Shepperd show that the results could vary dramatically, and thus lead to confusion. They conclude that "it is vitally important to precisely specify the mapping from a language-independent set of metrics to specific programming languages".

Hitz and Montazeri [37] further claim that CBO is not a sensitive enough measure for coupling, since it considers all couplings to be of equal strength. They argue that access to instance variables should constitute stronger coupling than pure message passing, and message passing with a wide parameter interface is stronger than one with a narrow interface. They also show that LCOM exhibits an anomaly, since the same value is computed for classes that intuitively appear to have different cohesion levels.

Henderson-Sellers [38] also criticizes the LCOM measure, because while "a large value suggests poor cohesion, a zero value does not necessarily indicate good cohesion".

Basili et al. [39] show that five of the six C & K metrics (WMC, DIT, NOC, CBO and RFC) were useful in predicting class fault-proneness during the high and low level design phases of the life cycle. They conclude that the C& K metrics proved to be better predicators than "the best set of the traditional metrics", which are only available at the latter phases of the software life cycle.

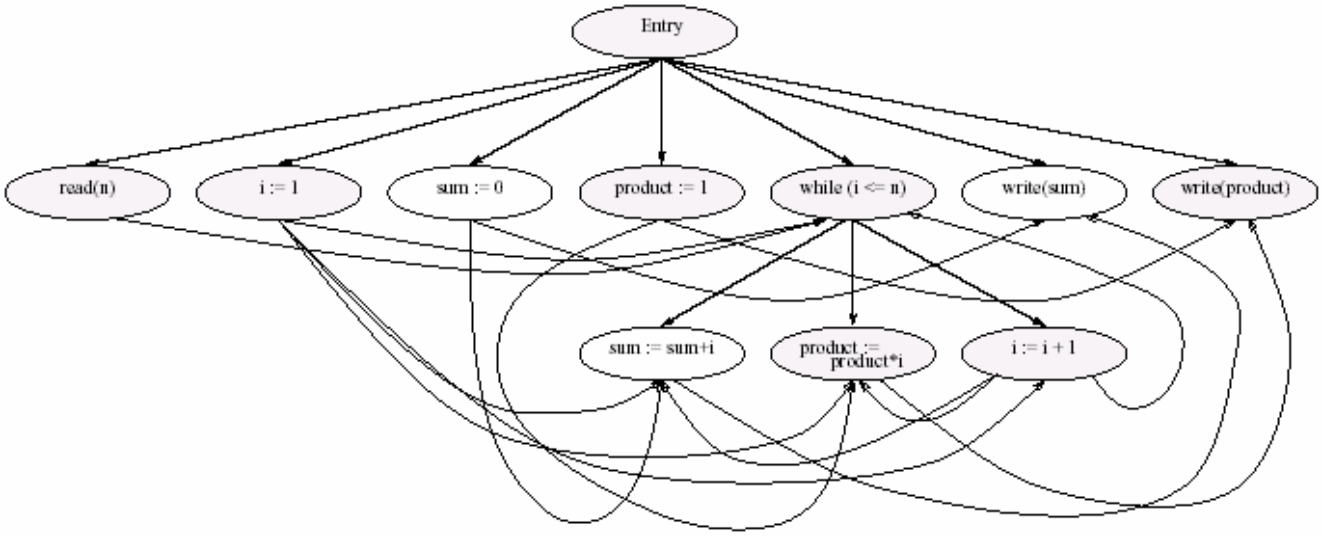
Other metrics suites are elaborated on in Appendix C.

2.6 Dependency trees and program slicing

When measuring structure or complexity metrics, specific nodes from the code must be sliced out of the program to get the information needed.

According to Tip [59], a program slice consists of the parts of the program that affect the values computed at some point of interest, referred to as a slicing criterion. With this technique we can extract the information we need, whether we want to make a debugger, a program understanding tool or a metrics tool. The task of computing program slices is called program slicing. The first definition a program slice was presented by Weiser [83] in 1979; later several slightly different notions have been proposed. An important distinction is between static and dynamic slices; the former notion is computed without making assumptions about a program's input, whereas the latter relies on some specific test case.

In Weiser's approach to static slicing, slices are computed by computing consecutive sets of indirectly relevant statements, according to data flow and control flow dependencies. Ottenstein and Ottenstein [82] have suggested an alternative method for computing static slices. They have restated the problem of static slicing in terms of a reachability problem in a program dependence graph (PDG). A PDG is directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies. An example of a PDG and the corresponding code is shown in Figure 8.




```
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     sum := sum + i;
(7)     product := product * i;
(8)     i := i + 1
      end;
(9)   write(sum);
(10)  write(product)
```

Figure 8 A simple program and its program dependency graph (PDG), (Tip, [59])

2.7 AOP-metrics

Some studies ([1] [26] [27] [28] [29]) have been conducted with the help of metrics, usually with the metrics defined for object-oriented programming. Little work has though been done in the field of defining metrics suitable for Aspect-Oriented Programming.

2.7.1 Metrics used in system studies

In the project we focused on readability when choosing metrics [1]. Using the GQM approach we found nine metrics that could be used to answer questions about the size, structure and complexity of the code when introducing aspects to Java-code. The metrics we chose were lines of code, efferent coupling, afferent coupling, fan-in, fan-out, Henry's and Kafura's complexity metric, lack of cohesion in methods, weighted method per class and hidden

concerns. The introduction of aspects especially affected lines of code, fan-in/fan-out and Henry's and Kafura's complexity[9].

In his study, Mickelson [27] focused on size measures like number of classes, functions and source statements. Coady and Ciczales [28] studied runtime costs and the placement of hidden concerns in operating system code. Zhang and Jacobsen [26] used cyclomatic number, size, weight of class[10], coupling between objects and response time for their evaluation. Tsang, Clarke and Baniassad [29] applied the C&K metrics suite in their evaluation of aspect-oriented techniques. They focused on the quality factors understandability, maintainability, reusability and testability, and the C&K definitions of metrics suited for measuring these factors. These metrics are weighted method calls, depth of inheritance tree, number of children, coupling between objects, response for a class and lack of cohesion in methods.

To sum up, we have found these metrics used in studies of AOP systems:

- size (lines of code/statements)
- number of children
- depth of inheritance tree
- coupling (efferent/afferent/between objects)
- fan-in/fan-out
- Henry's and Kafura's complexity
- cyclomatic complexity/cyclomatic number
- weighted method per class/weight of class[11]
- lack of cohesion in methods
- response time
- response for a class

2.7.2 Metrics reviewed

Zakaria and Hosny [16] have discussed how the metrics in the C&K metrics suite are affected by aspects. They found that all the metrics in the suite might be affected in some way as shown in Table 13.

Metric	**Affectations**
Weighted methods per class	Might be reduced because crosscutting functionality is moved into aspects
Depth of inheritance tree	Subclasses with the purpose of implementing special crosscutting behaviours might be moved into aspects, thus reducing the depth

[9] Henry's and Kafura's complexity is based on fan-in/fan-out and in our study also lines of code.
[10] Weight of a class is the number of methods in the class
[11] Weighted method per class with complexity equal to one is the same as class weight

Number of children	Same argument as for depth of inheritance tree
Lack of cohesion in methods	Aspects filters out crosscutting behaviours, thus increasing cohesion
Coupling between objects	Likely decreased between core classes, yet increasing coupling between core classes and aspects. Zakaria and Hosny argue that since core classes are more likely to be reused, it is more important to reduce coupling between them.
Response for a class	Likely to increase because the entities that a class need to communicate with increases when needing to communicate with aspects.

Table 13 C&K metrics and AOP affectations [16].

Zakaria and Hosny [16] suggest Depth of Inheritance Tree (DIT) and Number of Children (NOC) as measures for improved understandability when using AOP. We need a special kind of code behaviour to observe changes in these two metrics. "Subclasses that might be defined only for the purpose of applying their own implementation of aspectual behaviour will not exist in systems designed using the AO Paradigm, because aspects will be responsible for that." [16]

Response For a Class (RFC) is also recommended by Zakaria and Hosny [16]. RFC measures the number of methods that can be invoked in a class in response to a message and can be used to evaluate understandability, maintainability and testability.

Tsang, Clarke and Baniassad [29] found that the empirical inference based on the metrics in the C&K suite is limited. They argue that since modularity had improved, but understandability, maintainability and testability had not, the empirical inference, that the C&K metrics suite is based on, is limited. Their results suggests that control-flow metrics like response for a class and weighted methods per class should be used to fully assess the effect of aspects.

2.7.3 Proposed new metrics

Based on dependencies within a program, Zhao [23] proposes a number of new metrics in three levels. They are supposed to measure complexity on module-level, aspect-level and system-level counting control, data, call, parameter-in, parameter-out and program dependencies on the appropriate level. The overall goal for these measures is to quantify the information flows of aspect-oriented programs.

Zhao and Xu [24] have proposed three new cohesion metrics for aspects which are named inter-attribute, module-attribute and inter-module cohesion. They measure how tight aspect

instance variables cohere with aspect modules (advice, introduction, pointcut and methods). They have shown that these metrics satisfy the properties given by the framework of Briand et al. [31], shown in Table 7, but they give no empirical evaluation of them.

Influenced by their work with mobile agents, Dospisil and Khemgoen [45] [46] have proposed a metric based on the theory of entropy[12]. The basic idea of the metric is that increasing amount of control flows between units gives increasing complexity. The chosen units are methods. Local code within a unit does not contribute significantly to complexity, thus they can exclude local variables. Measuring can either be done to order units without information or with subjective knowledge of relevance, significance or utility of units. Real empiric evidence for the measurement is not yet provided as it is only used with very small systems.

2.7.4 AOP dependency tree

The system dependence graph for aspect-oriented programs is an extension of the existing graph for the language that AOP extends. For Aspect/J, this means that we need to extend an existing SDG for Java. To study SDGs for Java, see Kovacs et al. [56] or Walkinshaw [57].

Zhao and Rinard [54] [55] are the first that have made an SDG-extension to suit Aspect/J. They have included join points, advices, introduction, aspects and aspect-inheritance in their graph definiton. The extensions they have made are described in Table 14.

Construct	Description
Method dependence graph (MDG)	Represents a method
Introduction dependence graph (IDG)	Represents an introduction
Advice dependence graph (ADG)	Represents an advice
Module dependence graph (MDG)	Because introduction and advice dependence graphs can is similar to the method dependence graph, they all can be denoted module dependence graphs (MDG).
Aspect interprocedural graph (AIDG)	Represents a single aspect and can consist of several module dependence graphs plus a unique start vertex.
Aspect-Oriented system-level graph (ASDG)	Represents a complete aspect-oriented program and is a collection of the above mentioned dependence graphs.

Table 14 Extensions to Java system dependence graph (SDG) for Aspect/J (Zhao, [23])

[12] Entropy, a term from physics, is based on increasing amount fof information, introduced by Davis and LeBlanc (1988) [60]

To make the complete ASDG, Zhao and Rinard propose to first construct a dependence graph for the object-oriented program and then insert weaving vertices. These vertices are connected to ADGs with coordination dependence arcs. In Table 15 a list of possible interactions between objects and aspects is summarized.

Type	Description
Creating objects	Similar to ordinary Java when an object is created in an advice.
Making calls	Calls to public methods in other aspects or classes.
Using introductions	Declaring a public introduction to a class in an aspect.
Using join points	Connection between and advice and a join point somewhere in the code. The joinpoints are found through examination of used pointcuts[13].

Table 15 Possible interactions between objects and aspects (Zhao, [23])

Balzarotta and Mange [58] have found some difficulties in constructing a dependence graph for Aspect/J programs. They argue that the expressive powers of Aspect/J "forces program slicers to take into account big portions of programs". They propose to make stricter boundaries for aspect interactions.

Zhao has used the ASDG as a starting point for change impact analysis [22] and for constructing metrics on AOP-code [23] [24]. The main feature of his change impact analysis program is to asses the effects of changing the code. The metrics he has proposed used with AOP have already been presented in section 2.7.

[13] A join point is declared through pointcuts. Pointcuts are collections of zero to many join points.

Chapter 3. Relevant technologies

We begin by presenting metrics tool that can be compared to what we have implemented. To be compatible with XRadar [80], some external technologies must be evaluated and considered. Another reason for evaluating such technology is convenience in terms of not needing to "re-invent the wheel".

3.1 Metric tools

There are a number of metrics tools for object-oriented languages available, both as open source and commercial tools. The aim of metrics tools is to help the development process by highlighting weak points in the system code. This is done by parsing the source code of the system and calculating a range of metrics to measure system properties. Some tools concentrate on one property, for instance complexity or robustness, while other provides a wider range for its analysis. The results can be presented several ways, from ordinary names and numbers to graphs and diagrams. Results can be presented at both high level (project or package level) down to low level (method or even variable level).

Verifysoft's CMTJava [66] is a complexity measure tool intended to be an aid in testing, quality assurance and enforcing company standards for code complexity. CMTJava uses McCabe's cyclomatic number, Lines-of-code metrics, Number of semicolons and Halstead's metrics in its calculations.

JDepend [10] aims to analyse the design of the system in terms of extensibility, reusability and maintainability, and is an aid to manage and control package dependencies. The tool's intended use is to automatically check that the designs exhibit expected qualities while undergoing continuous refactoring by the developers. JDepend provides metrics for Number of Classes and Interfaces, Afferent and Efferent Couplings, Abstractness, Instability, Distance from the Main Sequence and Package Dependency Cycles.

JMetric [68] is a result of a research project at Swinburne University, and aims to bring current OO-metrics and metrics research to the practioners. JMetric collects information from Java source files and compiles a metrics model. The model is then populated with metrics information such as Lines of Code, Statement Count, LCOM, and Cyclomatic Complexity.

Metrics 1.3.5 [15] is an open source plugin for the Eclipse IDE [67]. It calculates 17 different metrics and provides a package dependencies analyser, and is as such an all-round tool. We used this metric tool in our readability study [1].

We have not found any metrics tool that takes into account the constructs of the AspectJ language.

3.2 XRadar

XRadar is, as mentioned in the introduction, an open-source code analysis framework [80]. We want our metrics tool to be a compatible plugin to the XRadar.

XRadar gets results from more than 8 open source projects and a couple of in house grown projects and presents the results as unified html/svg reports. The architecture is based on java, xml and xsl. Presentation starts at the sub-system level, but drilldowns are available down to class and method level. [80]

XRadar is available in two forms:

- Statics – reports on current build.
- Dynamics – include time dimension[14] as more vesions are analyzed and reported on.

In Figure 9 the number of statements in the different version of Telenor COS is shown as an example a dynamic report.

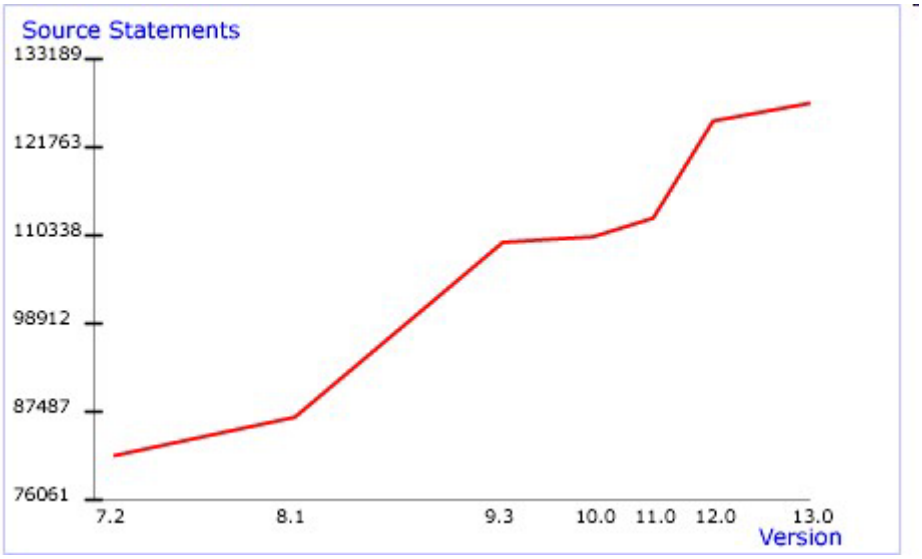


Figure 9 Example of dynamic report from XRadar.

The developers of XRadar see five areas of future directions for the tool [80]. These are:

1. Support other programming languages, as it currently only supports Java
2. Support other build frameworks than Apache ANT
3. Add new measurements and reports

[14] The time dimension is represented by version numbers [80].

4. Add more GUI-models
5. Support cross project analysis

Our metrics tool supports two of the above mentioned areas, area 1 and area 3.

3.3 Candidate technologies for the metrics tool

Some of the tasks in the metrics tool may be done in external libraries or program. In this section we give an introduction to such technologies.

3.3.1 Program startup

Program startup means the actual environment that the application runs in or more simple how it is started.

Apache ANT is a Java-based build tool. ANT differs from other build tools like make, gnumake, nmake and jam that are shell-based. Instead ANT is extended using Java classes. Instead of writing shell commands, the configuration files are XML-based with defined targets that execute defined tasks. You may choose which targets that shall run. Each task is run by an object that implements a particular Task interface. ANT does not give the expressive power we get by using a shell command, but it gives the ability to be platform independent. [18]

ANT is used by Telenor Mobile in their build process and to schedule running of XRadar. To be compatible with the work process in Telenor Mobile, our metric tool will need the ability to be run as an ANT-task. Because of this we will not evaluate alternative technologies for running.

3.3.2 Building code tree

Getting a structure suitable for metrics calculation from the code in text files is a complex task due to the structure and possibilities of the programming language. The structure we need is a simple kind of a program dependence graph (PDG). To get it we need to read the code and fetch the nodes that are interesting, like class declaration, import or method calls. This is the same as what needs to be done when compiling code.

A parser can be used to read and get the structure of languages like Java and C. Example of parsers are Bison [72] and JavaCC [73]. They parse the code by breaking down sentences and statements into tokens that are acceptable in the language. By defining tokens that are interesting elements, we can build a code tree. Another possibly feasible solution is to use regular expressions to search for interesting code blocks.

A problem with AspectJ code is that only parsing the code does not provide enough information. In addition, the aspects must be weaved in. A tool that does this is the AspectJ [74] compiler itself. Figure 10 illustrates how the AspectJ compiler weaves in aspects. A public interface intended for IDE tools like Eclipse and Emacs is included in the compiler. The compiler provides a tree view of the code including packages, files, classes, aspects, interfaces, methods, advices, imports, fields, field set/get and method calls. To being able to use the compiler a build configuration file and a classpath must be provided. In addition, some classes must be implemented to support AspectJ's public interface, including some GUI classes that we will not be using in our tool.

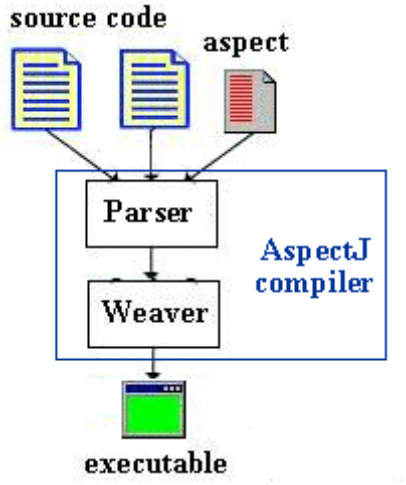


Figure 10 An illustration of the AspectJ weaver

A problem with using the AspectJ compiler is that it is not intended for our use, thereby introducing some new risks:

- Rare bugs that will not be noticed when used as intented may occur with our use.
- New versions of the compiler may change the public interface in a way that is incompatible with our use.

As we have not found any other tool that provides a code tree for the AspectJ code, the only alternative to using the compiler is to make our own parser and weaver. The complexity and time cost of such solution is too high to be considered.

3.3.3 Reporting

XRadar uses XML to present measurement results. All reports from supporting tools, like JDepend [10], generate XML-reports which are then transformed to HTML pages and svg graph figures.

XML is a markup language for documents containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role the content plays. A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents. [17]

XML is not only a language but also a collection of technologies. Two of them that are relative to our project are Cascading Style Sheets (CSS)[15] and Extensible Stylesheet Language (XSL). XSL Transformations (XSLT) can be used to generate HTML pages and CSS is used to give the page some specific formatting, e.g. using a specific font. [75]

An XML document must be well-formed to be valid. By well-formed we mean following the syntax rules. In addition, the XML document should be accompanied with a description how the XML document is structured. Such description is called a Document Type Definition (DTD) which can be embedded in the XML file or described in a separate file. The XML document is valid if it conforms to the DTD. [75]

To be compatible with the work process in Telenor Mobile, our metric tool will to produce results in an XML-format. Because of this we will not evaluate ways of reporting.

There are many technologies that can be used for generating XML reports in Java. Our goals for such technology are a simple API and good performance.

One tool that has a simple API is the XmlWriter [76], an open-source tool that outputs simple XML to a file. A benchmark comparison for XML outputters is included on their site and shown in Table 1.

Records Written	**XmlWriter 1.0**	**J2EE 1.3**	**JDOM 1.0 Beta 8**	**NanoXML/ Lite 2.2.1**	**ECS 1.4.1**
5000	491	1262	1473	1822	81017
10000	821	2103	2434	3345	350805
15000	1051	3715	3144	***	***
20000	1442	3886	4416		
25000	1642	6339	6059		
30000	1853	***	***		

Table 1 A benchmark comparison for XML outputters [76].

The following must be noted:

- few measurements are done, only three for each setup
- Standard deviation is not given (claims that it is small)
- All times are given in milliseconds
- "***" means an out of memory exception is generated

[15] A CSS file is not a well-formed XML file.

We find no reason to believe that the numbers in the benchmark comparison are deliberately false, given the fact that the tool is freeware and open source. For more information about the benchmarks, see [76].

Chapter 4. Quality analysis

To verify if a certain property is well-made you have to express that property in terms of quality. Further, quality has to be understood in terms of a context. For this analysis the context is the Telenor Mobile COS-system. The first part of our problem definition is "to find useful metrics for measuring AOP-based programs". The metrics should help us measure high-level quality factors. As a consequence we need to find metrics that can be related to the quality criteria we want to measure.

We use the development of COS as a basis for our choice of metrics, starting by defining quality goals supported with scientific questions. Thereby we justify our choice of metrics. The ISO 9126 framework and application knowledge of the COS-system influence the choice of factors. Following the measurement philosophy of Telenor Mobile, the point of these factors are not to reach a specific level or numerical value, but to show improvement, or opposite, and trends through time and change when using AOP. We will not be able to identify a "complete AOP-metric suite". That is not by any means achievable with the given project resources and manpower, nor do we have the ability to thoroughly test and prove such a suite.

4.1 GQM

4.1.1 Quality criteria/goals

In Figure 11 we can see system quality factors for the Telenor Mobile COS system.

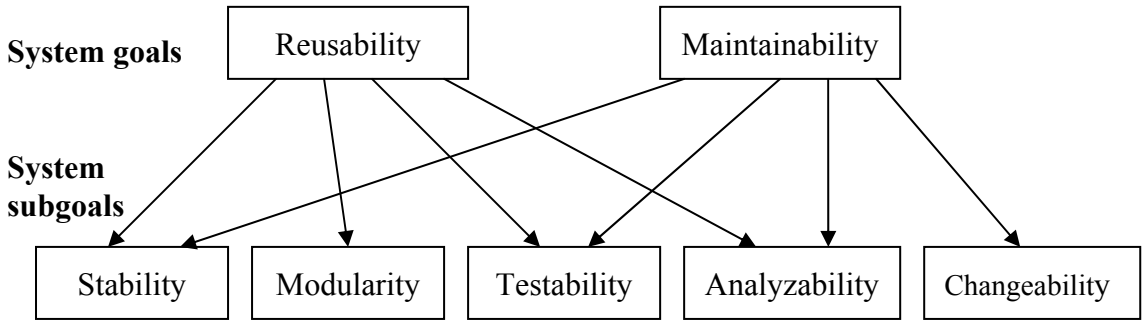


Figure 11 The system quality factors we want to measure with our AOP-metrics tool

Due to time restrictions we have to prioritize and concentrate our work on the two system goals we find most important for the COS application system; reusability and maintainability. Other system goals that were not prioritized were functionality, usability, efficiency, reliability and portability. By using the ISO 9126 framework we have divided the two system goals into subgoals. It is not possible for us to test the adaptability of the system, so this subgoal will be disregarded. Otherwise, adaptability would have been a subgoal of maintainability

4.2 Description of system quality factors

4.2.1 G1: Reusability

Reusability is needed for faster development. Making reusable modules will be more time demanding than making custom-made modules, but this is worth while when reusing the module.

Sub characteristics of reusability are:

- SG1: modularity
- SG2: testability
- SG3: analyzability
- SG5: stability

4.2.2 G2: Maintainability

COS' source code is constantly changed and maintained. Maintainability costs are as such an important part of the COS system.

Sub characteristics of maintainability, according to ISO 9126, are:

- SG2: testability
- SG3: analyzability
- SG4: changeability
- SG5: stability
- adaptability

Adaptability will not be considered because it is not possible for us to carry out such a study.

4.2.3 Scientific questions

In order to check if the system subgoals are fulfilled, GQM tells us that we need to create questions we can find measurable answers to. For each subgoal we present questions that can be used to determine the state of that subgoal. We can see that most of the subgoals can be at least partially answered by use of a metrics tool, but that some question falls outside the scope of such an application. If the question cannot be answered by using a metrics tool to analyze the source code, then the type of measurement has been marked with parentheses.

4.2.4 SG1: Modularity

Improved modularity results in less scattered functionality. This supports reusability, since it reduces the ripple-effect when making changes in the code. The questions for modularity can be seen in Table 16.

Question	Suggested metrics	Effect
Q1.1 Are the functions in modules related?	Coupling inside the package Coupling outside the package Cohesion	+ - +
Q1.2 How many functions do modules provide?	#classes per package #methods per class	(Depends on the coupling results.)

Table 16 GQM-questions to determine the systems modularity.

4.2.5 SG2: Testability

We need to know how suitable the code is for testing. The tester needs to understand the code, and the amount of dependencies to other modules can complicate testing. The questions for testability can be seen in Table 17.

Question	Suggested metrics	Effect
Q2.1 What responsibilities do the modules have?	(Design documents)	N/A
Q2.2 Are the modules strongly connected to other modules?	Coupling outside the package	-
Q2.3 How many dependencies do the modules have towards other modules?	Import-coupling outside the package Fan-out	- -
Q2.4 How many other modules depend on the modules?	Export-coupling outside the package Fan-in	(Depends on how abstract the modules are.)
Q2.5 Are all classes and methods documented?	#commented methods	+

Q2.6 Is the documentation of high quality?	(Manual analysis)	N/A

Table 17 GQM-questions to determine the systems testability.

4.2.6 SG3: Analyzability

By analyzability we mean the effort needed to retrieve information from the code; being able to identify causes of failures, problem areas and parts to be modified. The questions for analyzability can be seen in Table 16.

Question	**Suggested metrics**	**Effect**
Q3.1 How large are the modules?	#classes per package #methods per class #code size of class	(Must be seen in relation to the module's dependencies.)
Q3.2 Do the modules have clear and distinct responsibilities?	Cohesion Import-coupling Fan-out + Advice-out for aspects	+ - -
Q3.3 Do the modules have sensible names?	(Manual analysis)	N/A
Q3.4 Are sub modules (classes and aspects) placed in sensible modules (packages)?	Coupling outside the package	-
Q3.5 Are all classes and methods documented?	#commented methods	+
Q3.6 Is the documentation of high quality?	(Manual analysis)	N/A
Q3.7 Are the connections between modules sensible?	Dependency graph	+ legal dependencies - illegal dependencies

Table 18 GQM-questions to determine the systems analyzability.

4.2.7 SG4: Changeability

Changeability can be defined as the efforts needed to carry out modifications and removal of defects. The questions for changeability can be seen in Table 19.

Question	Suggested metrics	Effect
Q4.1 Do we trust that the modules perform the tasks they are supposed to?	(Subjective opinion)	N/A
Q4.2 Do we have trust in the developer's abilities?	(Subjective opinion)	N/A
Q4.3 Do we understand the code?	#commented methods Coupling Fan-out Advice-in	+ - - -
Q4.4 Do the modules have clear and distinct responsibilities?	Cohesion Coupling Fan-out + advice-out for aspects	+ - -
Q4.5 Are the interfaces of the modules precisely defined?	(Manual analysis)	N/A
Q.4.6 Are the modules heavily used?	Fan-in	-

Table 19 GQM-questions to determine the systems changeability.

4.2.8 SG5: Stability

Unexpected behaviour caused by modifications deteriate stability. The questions for stability can be seen in Table 20.

Question	Suggested metrics	Effect
Q5.1 Do the modules have clear and distinct responsibilities?	Cohesion Import-coupling Export-coupling Fan-out + advice-out for aspects	+ - + -
Q5.2 Do the modules have many dependencies towards other modules?	Import-coupling Fan-out	- -

Q5.3 How many other modules have dependencies towards other modules?	Export-coupling Fan-in	+ +
Q5.4 Have the modules been changed over time, without failures and complications?	(Historical data)	N/A

Table 20 GQM-questions to determine the systems stability.

4.3 Analytical evaluation of metrics

The choice of metrics is highly dependent on what kind of software system you are going to measure, and what system properties you are interested in. We have identified reusability and maintainability as our selected properties and have thus chosen metrics that on their own or combined can provide information about these system properties.

4.3.1 Size

We want our size-measure to give a good indication of the amount of code in the system. As such we have discarded the counting of all lines and all non-blank lines as alternatives for our implementation. The most common way to implement a size measure has been to count non-blank non-commented lines. We have implemented a measure for non-commented non-blank lines inside methods as a comparison to our chosen measure. Our main size-measure is to count all statements inside methods. We have chosen to disregard statements outside method bodies, such as import declarations and class variables. These statements are inherently simple and add minimal complexity to the system. It is inside methods that the more difficult to understand statements are found, both as individual statements and the effect of a group of statements.

Other minor size-measures are the number of classes per package and the number of methods per class. In relation to other metrics, such as coupling and cohesion, these measures can indicate if a package or a class is too big.

The number of commented methods gives an indication of the level of documentation in the system. However, it cannot measure if the documentation is of high quality and comply with a good standard.

4.3.2 Cohesion

Cohesion is a metric designed to establish to which degree parts of a module belong together. Many ways have been proposed to measure cohesion. We have based our choice on the discussion given in [31] and [65]. Most cohesion measures presented in the literature have either not been suited for our intended use, been designed for an earlier phase in the

development cycle and not for measurements on implemented code, or been of only academic interest and very difficult to implement.

Chidamber and Kemerer's Lack of Cohesion in Methods [32] violate the properties for not have a fixed maximum value for cohesion; this will differ from class to class. The measure is also not able to distinguish between the structural cohesiveness of two classes, i.e. in the way which the methods share instance variables. The measure gives conflicting results for cohesion. Adding a method to a class that shares an instance variable with existing methods should decrease the value of LCOM. However, adding a method to the example shown in Figure 12 increased LCOM in Figure 13 and decreased LCOM in Figure 14 [53].

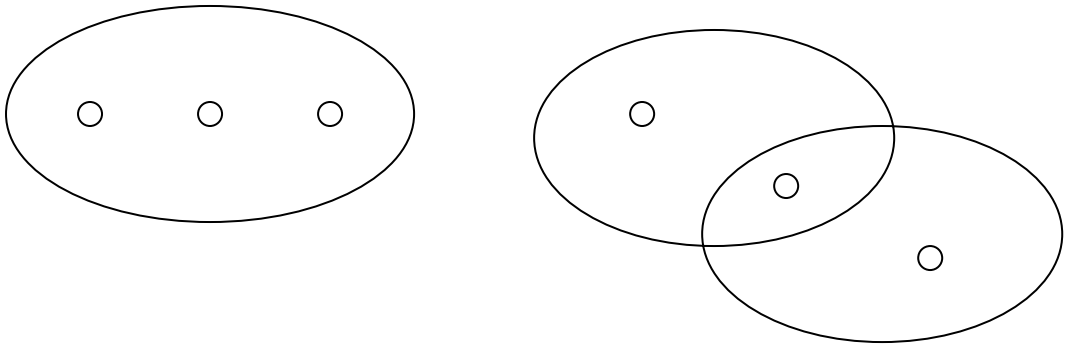


Figure 12 The ovals symbolize methods of a class. The circles symbolize the instance variables the methods use. LCOM = P – Q = 2 – 1 = 1

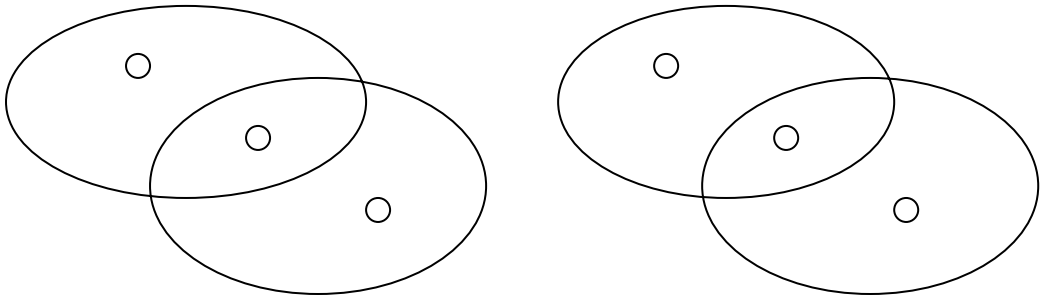


Figure 13 LCOM = P – Q = 4 – 2 = 2

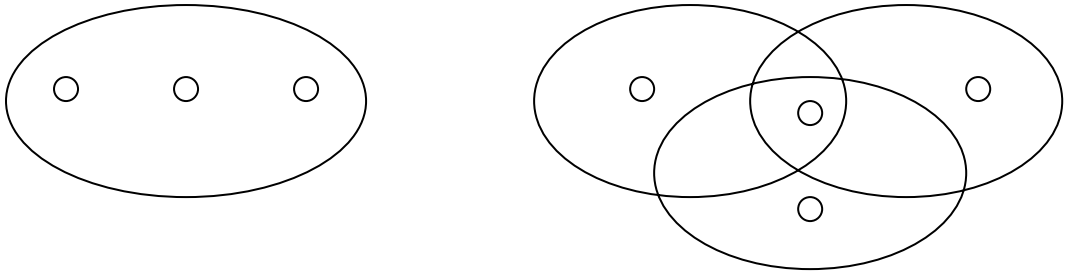


Figure 14 LCOM = P – Q = 3 – 3 = 0

Hitz and Montazeri's LCOM [53] also violate the property of having a common maximum value for cohesion. This is, however, compensated by their connectivity metric that always have a value between 0 and 1. As such, Hitz and Montazeri's measure is an improvement in terms of being able to tell the structural difference between methods with the same value of LCOM.

Bieman and Kang's Class Cohesion Measure [64] is our measure of choice. The drawback with this measure is that it can result in low values for classes with a large number of methods. This is because the denominator value for the calculation increases quadratically as the number of methods in a class increases. However, based on the work and experiences of Gupta [65], most methods only use one or two data tokens. This would result in that most classes with many methods would not achieve good cohesion values even with an ideal metric. Therefore we feel that this is a smaller drawback than not having a ratio scale for our cohesion values; as would have been the case with the alternative measures. With a ratio scale classes can be compared and are more informative than unbounded values.

Bieman and Kang use two cohesion measures, Tight Class Cohesion and Loose Class Cohesion. TCC measures only direct use of instance variables; while LCC also calculate indirect use of instance variables (the method calls a method that uses the instance variable). TCC can be prone to give too low cohesion values as it can not be expected that all methods use most or all instance variables directly. On the other hand, LCC might give too high cohesion values for classes were many methods might not use instance variables at all. Because of this we have implemented both cohesion measures, as the middle ground between their values should be a good indication of the class' cohesion.

4.3.3 Coupling

Coupling is used to establish the dependencies between classes or packages. There are numerous ways to calculate coupling. We have based our choice on the discussion presented in [30] and [50]. The candidates for coupling measure were presented in the prestudy, four of them as part of different metric suites.

Chidamber and Kemerer's Coupling Between Objects [32] is a simple and straightforward measure for coupling. The argument against the measure is its lack of sensitivity, in that it considers all couplings to be of equal strength [37].

Li and Henry's Message-Passing Coupling and Data-Abstraction Coupling [40] [41] were also considered. The drawback with Li and Henry's measure is that it does not take attribute references into consideration at all; except in the event of one class being another class' instance variable.

Brito e Abreu's Coupling Factor [48] [51] is calculated as the class' number of coupling not imputable by inheritance divided by the maximum possible number of coupling in the system. Both methods and attributes are calculated in the measure. The drawback of this measure is that most instances will get very low coupling factor values, because of the large amount of possible couplings in a system. We also regard it as more informative to know the exact amount of coupling an instance has, than to know a decimal value that will change from system to system.

Abbott, Korson and McGregor's measure for "interactions permitted" [52] calculate the opportunities an object provide for interaction and means for information flow through its interface. This measure can be used in the design phase to predict an "expert's opinion" of design alternatives. Therefore the measure has limited use for measurements on source code.

Hitz and Montazeri [53] have created a framework specifically for coupling in OO systems. The measure takes into account factors that are especially important to evolving OO systems, such as stability, type of access and scope of access. For instance, if an unstable package has a lot of incoming dependencies, then it is a high risk part of the system – and even more so if it is likely to undergo interface changes. In the same manner, a calling class that refers directly to an instance variable, instead of accessing through an interface, results in a higher coupling value as it is a breach of encapsulation.

Hitz and Montazeri's Class Level Coupling has been the inspiration for our coupling implementation. However, we have decided to count coupling inside and outside of package separately. This is because coupling to other classes inside the same package indicate that the classes belong together. On the other hand, excessive coupling to classes outside the package should raise the question if the class, or some of the functionality in that class, really belong in this package.

4.3.4 Fan-in

Fan-in measures the number of methods that call the method in question. Changes to the method could have extensive ripple effects throughout the system. Fan-in is a basic measure and does not take into account direct reference to class attributes. For our purpose fan-in will mostly be used in conjunction with other metrics.

4.3.5 Fan-out

Fan-out measures the number of method calls within a method. The measure serves as an indication for how dependent the method is on methods in other classes. As for fan-in, it does not include direct references to class attributes and will mostly be used in conjunction with other metrics.

4.3.6 Advice-in

The advice-in and advice-out measures have been influenced by the concept for fan-in and fan-out. Advice-in measures how many advices are connected to the given source, as seen in Figure 15. This informs the developer that there could be additional or overriding behaviour located in aspects that he needs to consider before changing the code.

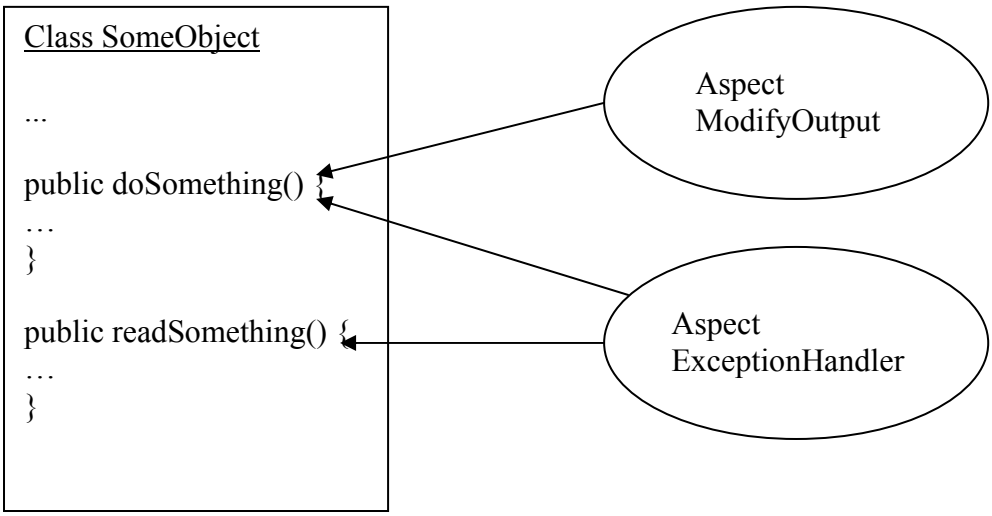


Figure 15 Aspects can change the behaviour of regular methods. This must be taken into account when making changes in methods that are affected by aspects.

Depending on the behaviour of the aspect, changes in the regular source code will make no difference to the running of the program; if the aspect overrides the regular source code anyway. By using the advice-in measure the developer can be made aware of the possibility of unintended effects of pointcuts hitting a joinpoint that it was not supposed to hit.

A more minor use for the advice-in measure is to check that only the correct classes and methods are hit by aspects. If a class or method has a value for advice-in when it should not, then it is obvious that an aspect has a too broad hit range. This is of more use in a development setting, using an IDE, than for our purpose of a system wide analysis.

4.3.7 Advice-out

Advice-out measures how many joinpoints an advice hits on. This gives an indication of how much of the system the aspect affects. If an aspect affects large parts of the system it is

important that it faultless. Extra effort should probably be used to ensure that it follows its intended behaviour. This is especially important if the aspect modifies the system behaviour and not only performs orthogonal tasks like logging.

4.4 Possible effects on metrics with AOP

Several studies have been performed on what effects introducing AOP has on a system in terms of various system properties, i. e. size, modularity and complexity. In addition we have derived two "new" metrics, advice-in and advice-out. We will discuss the expected effect they are designed to capture.

4.4.1 Size

Up to this point all studies, [1] [26] [27], have experienced a reduction in code size when introducing aspects to a system. This is a consequence of reduced code scattering, one of the advertised benefits of using AOP.

4.4.2 Cohesion

There is little correlation between moving functionality to aspects and changes in cohesion. In some instances it will increase cohesion, while cohesion will decrease in other circumstances. An increase will happen if functionality that had little or no connection with the rest of the class is moved. A decrease will happen if the crosscutting functionality has a lot of connectivity with the rest of the class. This does not necessarily indicate that moving the functionality was a bad decision. The code might have been tangled in such a way that it would be difficult to extend or maintain the class without removing the crosscutting parts.

4.4.3 Coupling

Coupling is expected to decrease when introducing aspects to a system. As can be seen in Figure 16, aspects are used to detangle code and as such re-route dependencies.

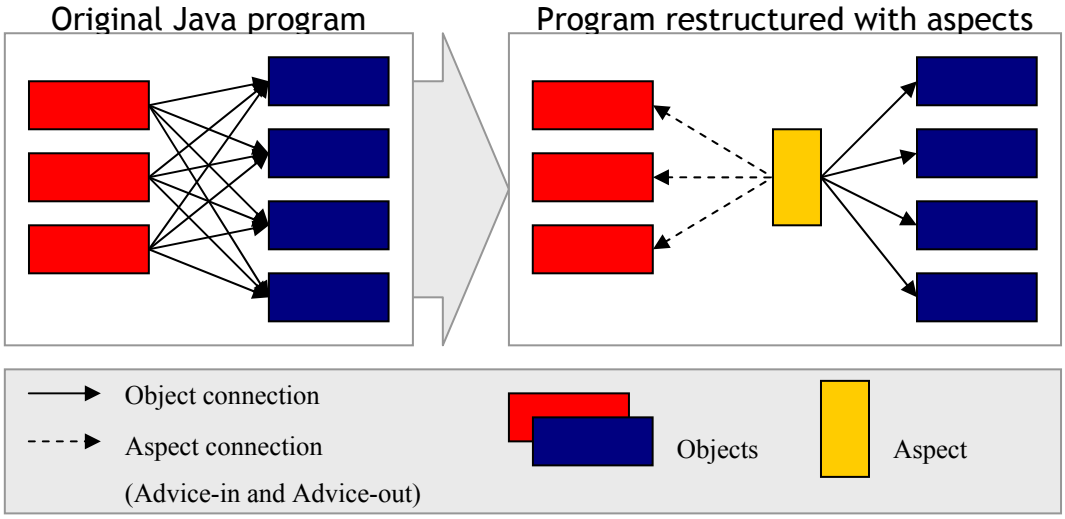


Figure 16 Coupling between objects in Java vs. AspectJ program

The effective decrease of coupling will in a few cases be less than what this illustration shows. In some instances, aspects will have to make a connection back to the pointcut source in order to fetch object information. However, on a system of significant size, this decrease will be hard to notice.

4.4.4 Fan-in

Fan-in is expected to decrease for the same reasons as coupling. As illustrated in Figure 16, more calls will go through aspects and there will be less calls overall.

4.4.5 Fan-out

Fan-out is expected to decrease because of the reasons stated above. Instead of method calls from the classes, we have pointcuts. The aspects will produce some fan-out, but not as much as the classes did.

4.4.6 Advice-in

This measure is used as a control instance, to check that the correct number of advice affect a particular class. It is also used as an indicator, used to show that a class is affected by additional or overriding behaviour.

4.4.7 Advice-out

Advice-out measures how much of the system an advice affects. This can serve as a good indicator for prioritizing test resources between aspects. It is more important that a simple advice that impacts a large part of the source is faultless than an advice with complex code that only impacts a small part of the source. This is illustrated in Figure 17.

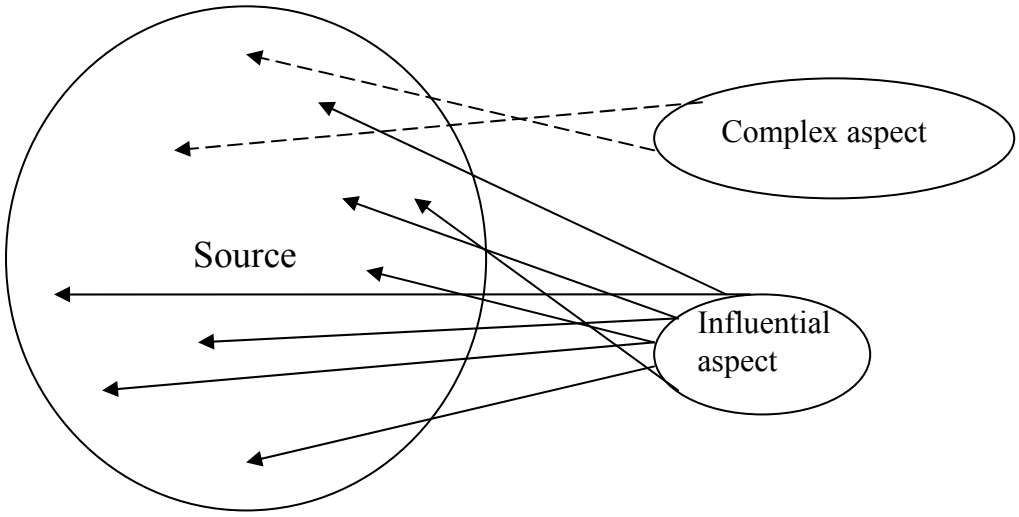


Figure 17 An aspect with a big impact area on the system needs to be faultless.

4.4.8 Fan-out + Advice-out

An advice (in an aspect) with high fan-out that is connected to many join points is a dangerous module. The join points are only loosely connected with the code since pointcuts are based on regular expressions. During operations like renaming or adding new modules it will be difficult to have the overview and system knowledge needed to predict if you will create unwanted hits for these aspects. Aspects with too general join point criteria can cause unintended behaviour in the system.

This measure is an effort to calculate the impact an aspect has on the system. Aspects with a big impact should probably undergo extra quality assurance to ensure that they perform to their intended purpose. It can also be an indication of aspects that have become too powerful[16] and possibly should be divided into aspects with less functionality or less far-reaching pointcut criteria.

On the system level, the sum of fan-out + advice-out is interesting to compare to the fan-out of the original system. Ideally, the introduction of aspects should result in that the sum is less, indicating that the system now contains fewer dependencies. Note that the summation of fan-out and advice-out is somewhat artificial, since the dependencies between aspects and classes are not entirely comparable to that between classes themselves. The similar summation of fan-in + advice-in is also of interest for the same reasons. If the value has decreased compared to the original fan-in value, then it is an indication of reduced dependencies on the system level.

[16] This can resemble the God Object anti-pattern, where too much processing is performed by one single class [84].

Chapter 5. The metrics tool

We have designed and implemented a tool that is suitable for measuring the metrics we have specified in Chapter 4 and that can work on larger systems like Telenor's COS. This tool has been given the name AspectMetrics.

5.1 Requirements

The metric tool will not be a critical part of a business application, but aims to be an accurate and effective tool for developers to analyze their code work and give indications of the change in some of a system's quality factors. See Chapter 4.

We only give a small list of primary requirements for the tool. The list of requirements is divided in two; functional requirements shown in Table 21 and non-functional requirements shown in Table 22.

Functionally, there are three main tasks that the tool must execute. An illustration of these tasks is shown in Figure 18.

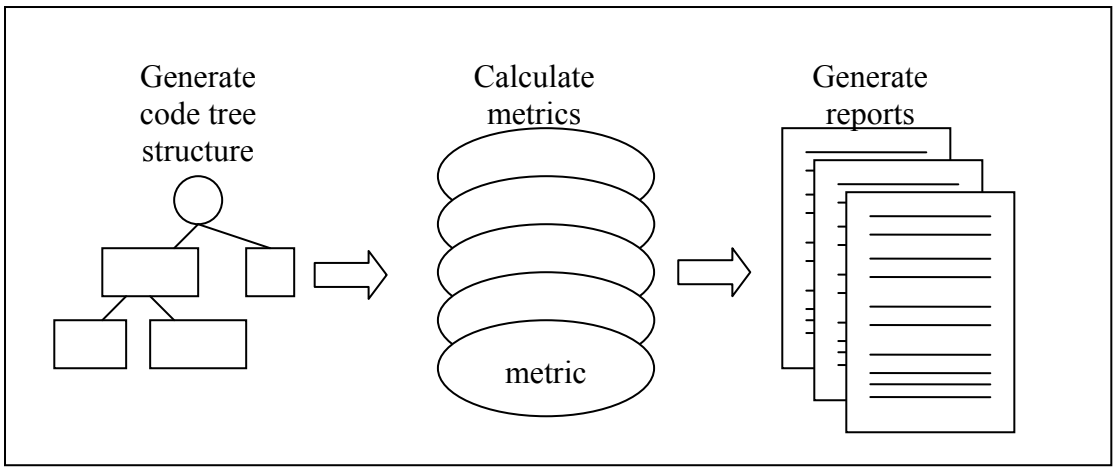


Figure 18 The three main functional tasks of the metrics tool.

5.1.1 Main functional requirements

The functional requirements are not explained in detail in this report. The purpose of the report is to provide a relation between the tool's environment, the metrics chosen and the tool as it is implemented.

Id	Requirement
FREQ1	The tool must be able to run as an ANT task to support COSMOS Radar
FREQ2	The tool must calculate metrics on several levels of code elements, defined as metrics sources. This includes methods, advices, constructors, classes, aspects, interfaces, packages and the system. Inner elements like inner classes, inner interfaces and inner aspects must also be included.
FREQ3	The tool must be able to provide information needed to calculate the metrics chosen. See Chapter 4.
FREQ4	The tool must be able to calculate the metrics chosen.
FREQ5	The tool must be able to deliver the results as an XML-report.
FREQ5	The tool must provide extension points for new calculators and reports.
FREQ6	It must be possible to choose calculators that will be used.

Table 21 Main functional requirements for the metrics tool.

5.1.2 Main non-functional requirements

As with the functional requirements, the non-functional requirements are not explained in detail.

Id	Requirement
NFREQ1	The tool must be able run for larger systems, like COS with 5000 classes, and this should not take several hours.
NFREQ2	The tool's architecture must support maintenance. Identifying incorrect code should be done within one hour if done by an experienced developer that knows the code, and its design and architecture.
NFREQ3	The tool's architecture must support extensibility, making it possible to add new calculators and reporters, and using less than one hour making code that is needed to be an accessible part of the tool. This applies for experienced developers that know the tool.

Table 22 Main non-functional requirements of the metrics tool.

5.2 Architecture

We have chosen the architecture for the metrics tool based on the functional requirements and the previously chosen technologies. An overview of this architecture is shown in Figure 19.

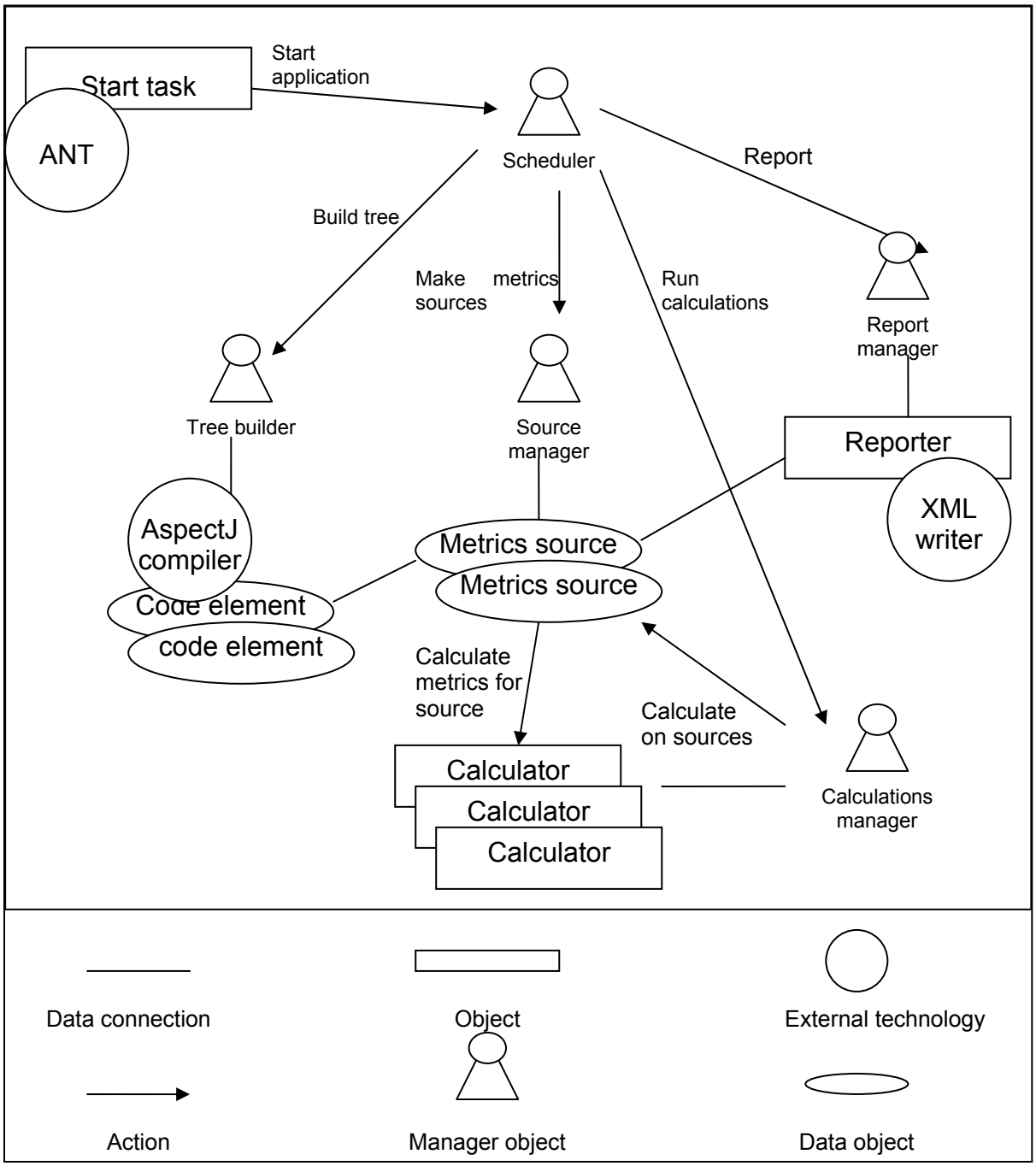


Figure 19 Architecture for the metrics tool.

A start task, shown in the upper left corner of the figure, starts up the metrics tool. The task must comply with the ANT API so that the application can be run through ANT. The scheduler administrates the whole sequence by calling on other managers, beginning with the tree builder. The tree builder must communicate with the public interface of the AspectJ compiler to compile some chosen source code and then make the elements in the code tree. The source manager is responsible for making metrics sources that must connect with the appropriate code elements in the code tree and store all values calculated.

We have chosen the following procedure for the calculation of metrics: the calculations manager calls on the metrics sources which runs the calculators defined on that source. An

alternative solution is to let each calculator work on all sources in turns. We have made our choice to restrict the amount of times a source is called on. By doing this we can let the calculators share data on one metrics source, instead of storing data for all sources or rebuilding necessary data every time a calculator is run. Example of such data is the source file which must be read to calculate the amount of statements in the code. In the end the reporter manager calls on the defined reporter which prints data stored in the metrics sources. For printing to an xml-file, an external xml-writer is used.

5.3 Design

The main goal of including design information is to describe how we have supported the calculation of our chosen metrics. Thus, we have included only the relevant parts of the design and will not discuss the design in detail.

5.3.1 Package structure

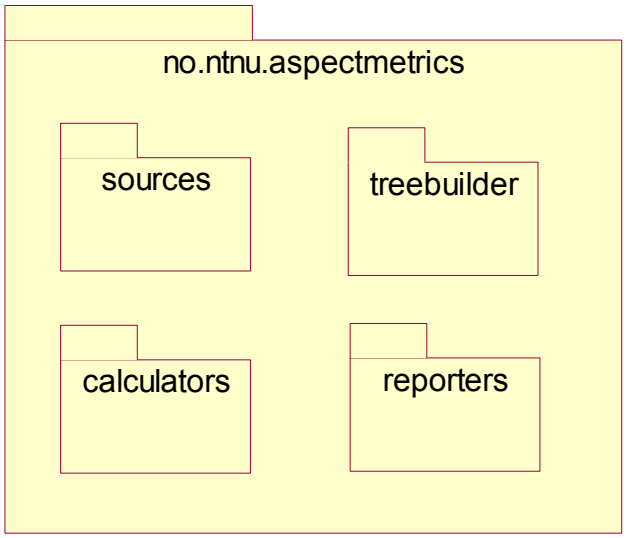


Figure 20 Package structure of the metrics tool.

Following the functionally focused architecture, the package structure is divided into four parts where the functions parsing (building code tree), calculating and reporting is placed in separate packages. The parent package no.ntnu.aspectmetrics contains the ANT start task, a manager class and a properties class.

5.3.2 Running the application

Since the application needs be run as an ANT task, the class starting the application must inherit the Task class of the ANT API. This is shown in Figure 21. The same figure shows the parameters that must or can be set when running the application. Setting sourcepath, classpath and workspacepath is required, while setting calculators and the reporter is optional.

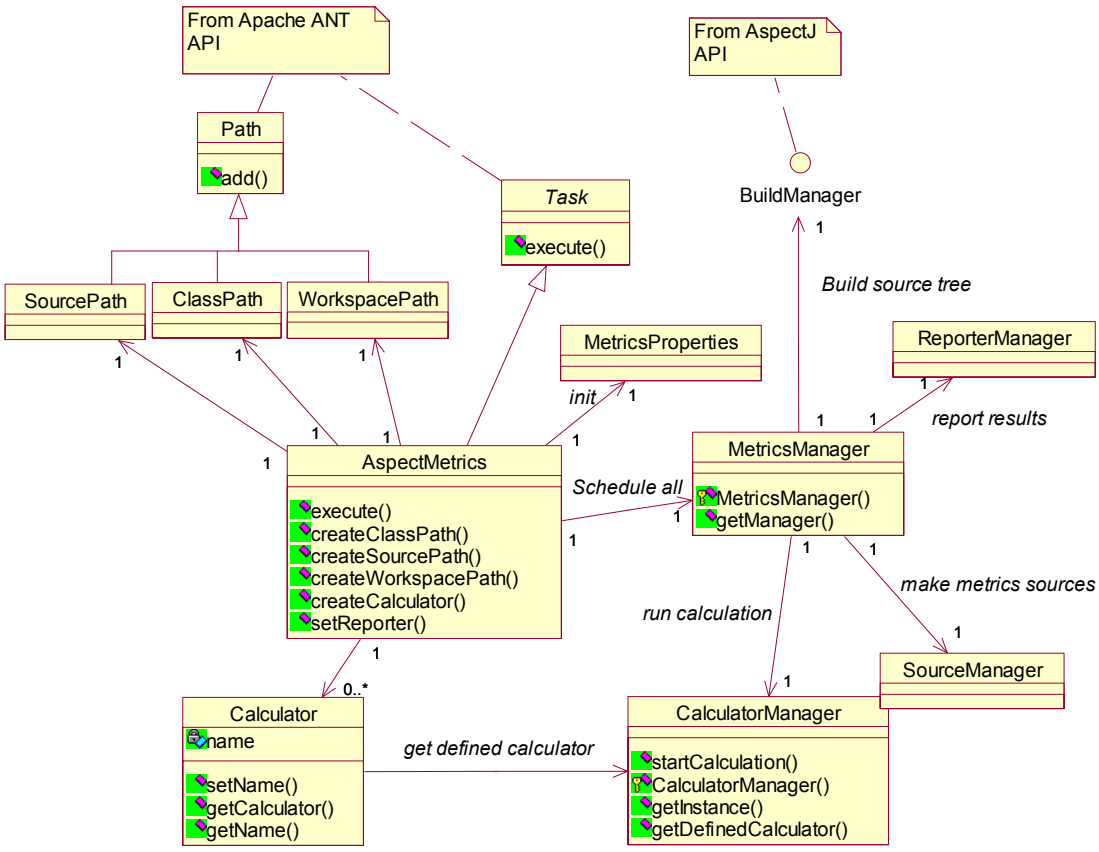


Figure 21 Design of the task that starts the metrics tool.[17]

Inheriting the Task class, the method `execute` must be implemented in the class AspectMetrics. This method is called whenever an ANT task is run. To run the application, the AspectMetrics task must be defined in an XML build file and this file must be run with the ANT application.

The `MetricsManager` is the scheduler that controls the other managers like `BuildManager` and `CalculatorManager`.

The class `MetricsProperties` holds general default properties like default calculators and default reporter.

5.3.3 Building code tree

The most important part of building the code tree is done by the AspectJ compiler. Our code must support the interface provided by the compiler. The design of this part is not included in the report.

[17] The symbol for `BuildManager` in the figure is the one defined for interface in Rational Rose [85].

After building the tree, a structure composed of program elements (`IProgramElement`) is available. Design of the program element is shown in Figure 22.

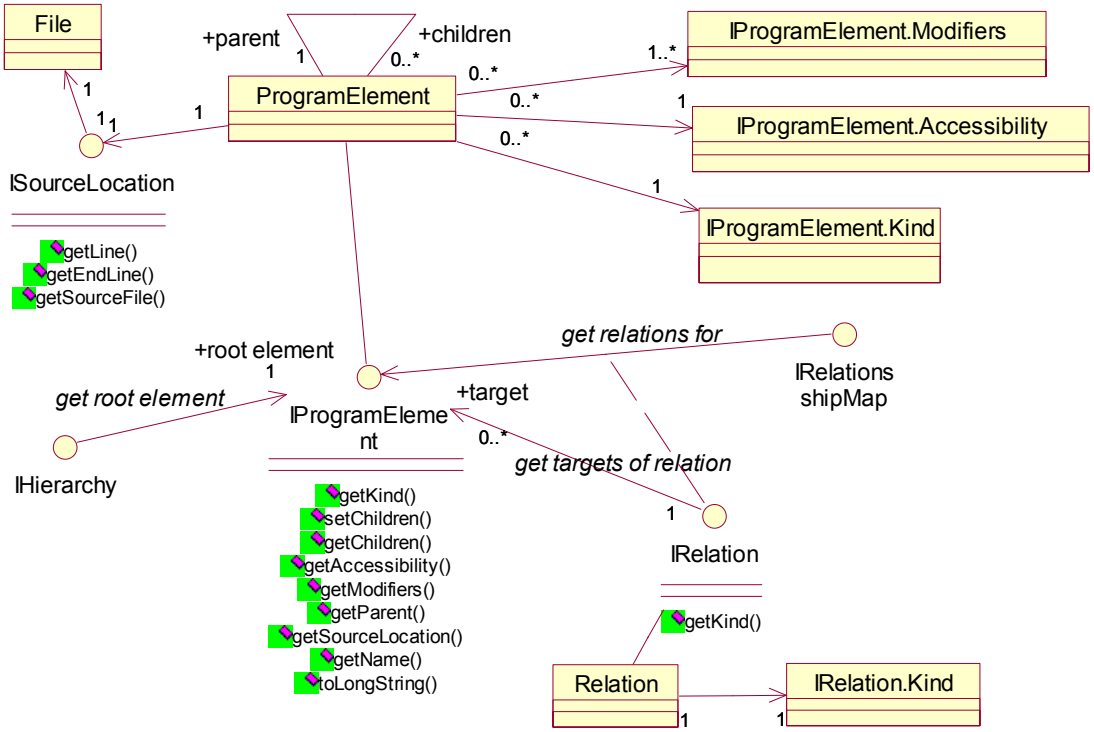


Figure 22 Design of the program element in the AspectJ compiler.

The `IProgramElement` holds important information about tokens in the code. These tokens include the following kinds: projects, packages, classes, aspects, methods, advices, and constructors. In addition, specific AspectJ code, like inter-type declarations, is also supported. Code that is defined within the above mentioned tokens, like method calls and field references, have no own tokens, but are defined as "code" tokens. The name of the element includes "method-call", "field-set" or "field-get". The modifiers of a program element include all Java modifiers like abstract, transient, strictfp etc. and accessibility can be one of the following: private, package, privileged[18], protected and public.

Through the use of `IRelationshipMap` it is possible to get relations for a program element. These relations can be one of the following types: advice, declare or declare inter-type. Any program element can be the target of such relation. See section 2.2 for an explanation of these types.

[18] Priviliged can be defined on aspects that must access fields or methods in a class that normally would not be accessable (private, protected or package).

The `ISourceLocation` connects the program element to its file and the `line` and `endLine` gives the position of the element in the file[19].

Figure 23 shows a print extract of a file named Main.java. This print is returned from the method `toLongString` in the program element and each line corresponds to the name of an `IProgramElement`. The indentation indicates which elements that are parents and children; accordingly the "import declarations" is a child of the file Main.java.

Element type:	Print extract:
[File]	Main.java
	import declarations
	figures.primitives.solid.SolidPoint
	figures.primitives.planar.Point
[Class]	Main
[Inner-class]	TestGUI
[constructor]	TestGUI
[code]	method-call(void java.awt.Component.disable())
[method]	startPoint
[method]	main(String[])
[code]	field-get(java.io.PrintStream java.lang.System.out)
[code]	method-call(void java.io.PrintStream.println(java.lang.String))
[code]	constructor-call(void figures.primitives.planar.Point.<init>(int, int))
[code]	method-call(figures.primitives.planar.Point figures.Main.makeStartPoint())
[code]	field-set(figures.primitives.planar.Point figures.Main.startPoint)
[code]	method-call(void java.lang.Throwable.printStackTrace())
	makeStartPoint()
[code]	Test
[method]	testptct()
[inner-aspect]	before(Point, int): <anonymous pointcut>..

Figure 23 Print extract from the code tree generated with the AspectJ compiler.

The source code for this file is included in Appendix D for reference.

[19] For elements of kind package, the file is the directory that the package corresponds to.

5.3.4 Metrics sources

After having generated the code tree, we need a data structure to hold references to the nodes in the code tree and the calculated values. Therefore, metrics sources are provided. In Figure 24 we have shown the design of those sources.

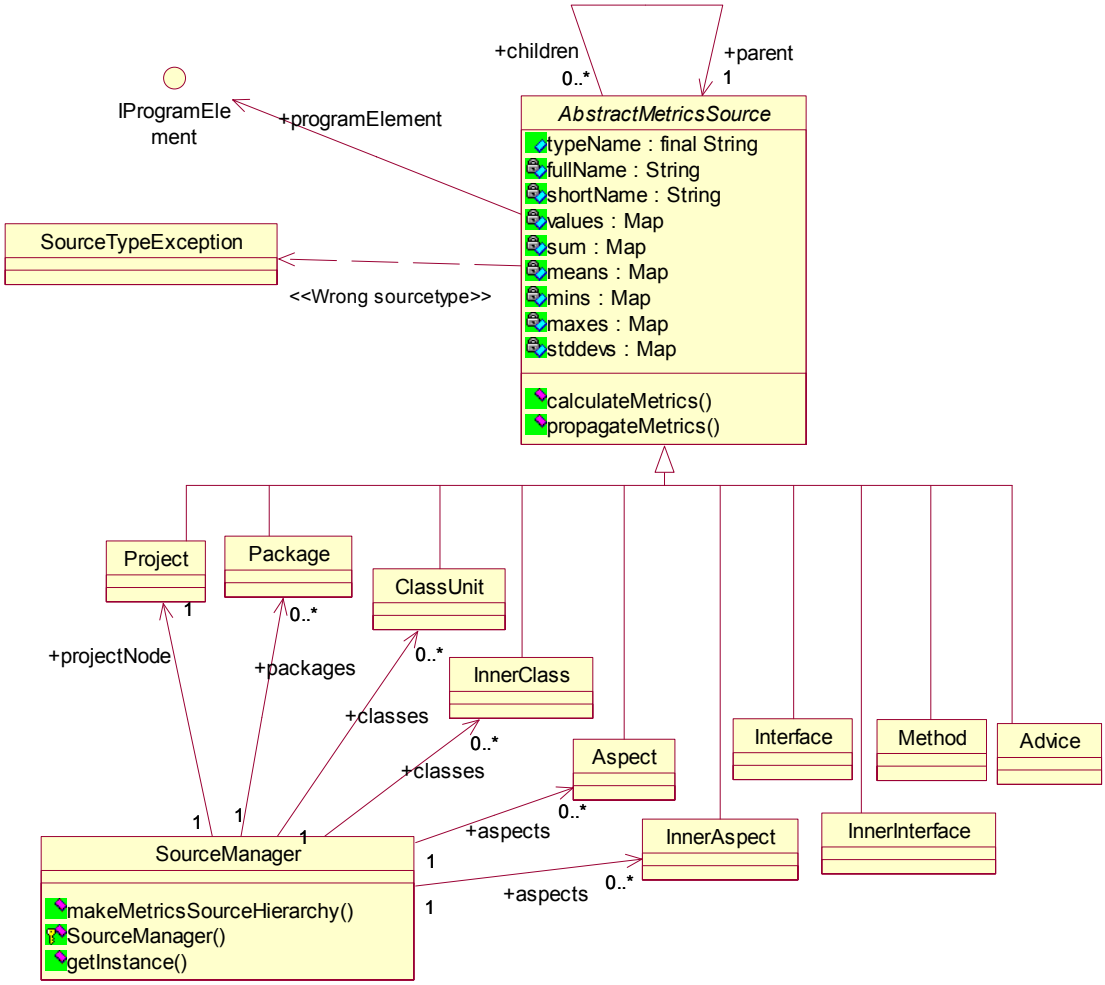


Figure 24 Structure of metrics sources used in calculations.

The abstract class `AbstractMetricsSource` holds the values that have been calculated and propagated. By propagated we mean the values that have been calculated based on nodes that are children to the current one. The propagated values given are sum, maximum, minimum, mean and standard deviation of values in children. The source is also connected to the appropriate `IProgramElement` in the code tree. Whenever it is attempted to make a source with the wrong type of `IProgramElement`, a `SourceTypeException` is thrown. The source is responsible for calling the calculators specified for its source kind (`IProgramElement.Kind`) and for propagating values. Note that one should avoid making a single calculator count on different source levels as this could clutter up the

meaning of propagated values like mean value[20]. A better solution is to provide another calculator that counts the same kind of measure on a different level.

The `SourceManager`, which is a singleton, is responsible for connecting program elements to metrics sources. It also holds maps that include all the packages, classes and aspects in the tree, using the name of the element as a key. This name is held by the source element as `fullName`.

5.3.5 Calculating

The calculation of metrics is bound to the metrics sources in such a way that it is the source which calls the correct calculators. The `CalculatorManager` shown in Figure 25 is responsible for running all calculations by calling on each source's `calculateMetrics` method.

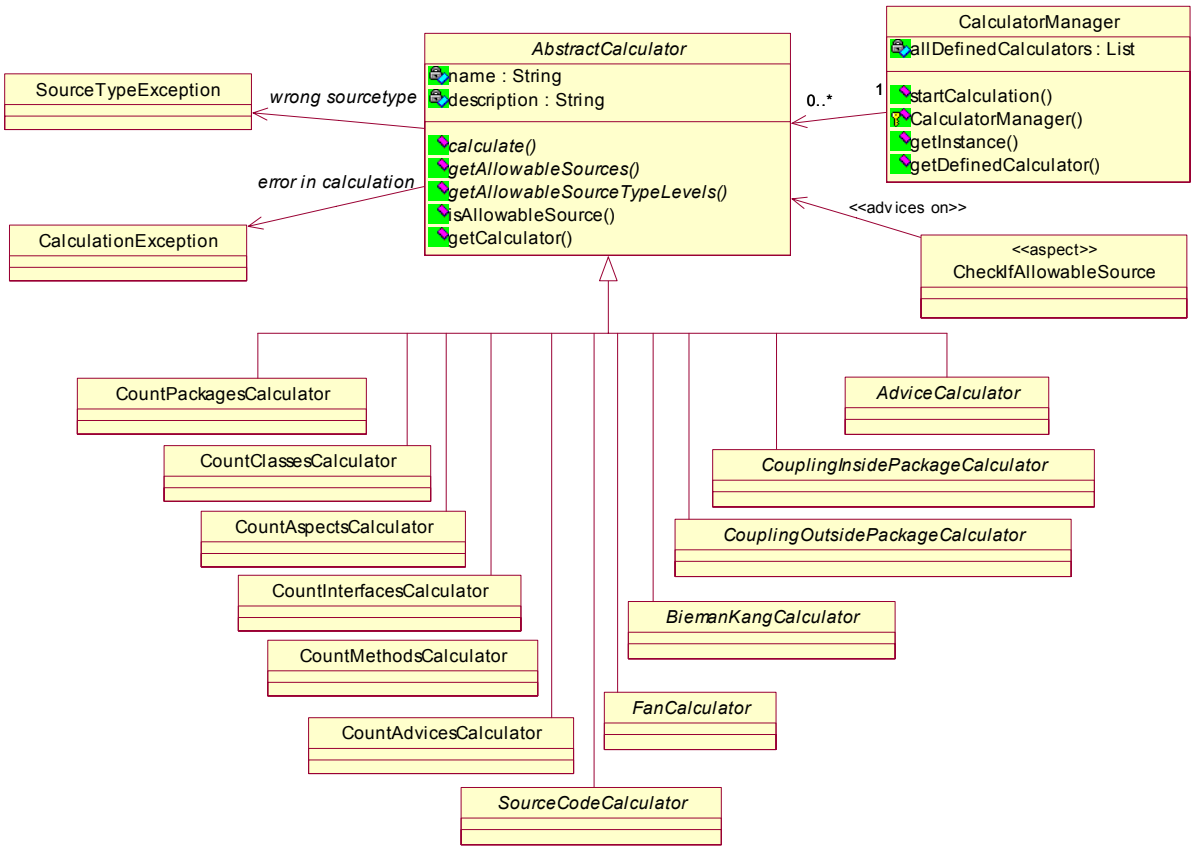


Figure 25 Calculators used on metrics sources to provide metric values.

[20] Example of cluttered sum when using same calculator on different source levels: If statements are calculated on class and method, then sum on the next level will be all the statements in the methods plus all the statements in the class; consequently wrong.

The calculation of metrics is carried out in a depth first fashion. Therefore, all children sources are calculated before a source is calculated.

All calculators must inherit `AbstractCalculator` with the abstract methods calculate, `getAllowableSources` and `getAllowableSourceTypeLevels`. Allowable sources specify which metrics sources that the calculator applies to. Since there can be both classes and inner classes, a level is also used to separate out the right source type. The aspect `CheckIfAllowableSource` is responsible for checking that the caller of the calculator is of the correct type. If not, a source type exception is thrown. Whenever something goes wrong with the calculation, e.g. a value is out of range, a `calculationException` is thrown.

Notice that some calculators are only abstract superclasses for the real calculators. This is done whenever more than one calculator uses some shared data or calculation. In Figure 25 `SourceCodeCalculator`, `FanCalculator`, `BiemanKangCalculator`, `CouplingInsidePackageCalculator`, `CouplingOutsidePackage-Calculator` and `AdviceCalculator` are such calculators.

Adding new calculators is as simple as adding new sources. The calculator needs to inherit `AbstractCalculator` and the `CalculatorManager` needs to be updated to also hold this type of calculator.

Counting calculators

All the counting calculators, listed in Table 23, are simple calculators that count the amount of source elements of one type for a higher level element.

Calculator	**Counts**	**On which level**
`CountPackagesCalculator`	# packages	System/project
`CountClassesCalculator`	# classes (including inner)	Package
`CountInnerClassesCalculator`	# inner classes	Package
`CountAspectsCalculator`	# aspects (including inner)	Package
`CountInnerAspectsCalculator`	# inner aspects	Package
`CountInterfacesCalculator`	# Interfaces (including inner)	Package
`CountMethodsCalculator`	# methods	Class or aspect
`CountAdvicesCalculator`	# advices	Aspect

Table 23 Designed counting calculators, what they count and on which level.

Figure 26 shows a simplified sequence diagram for the counting of aspects in a package. By simplified we mean that we have left out inner aspects from the counting.

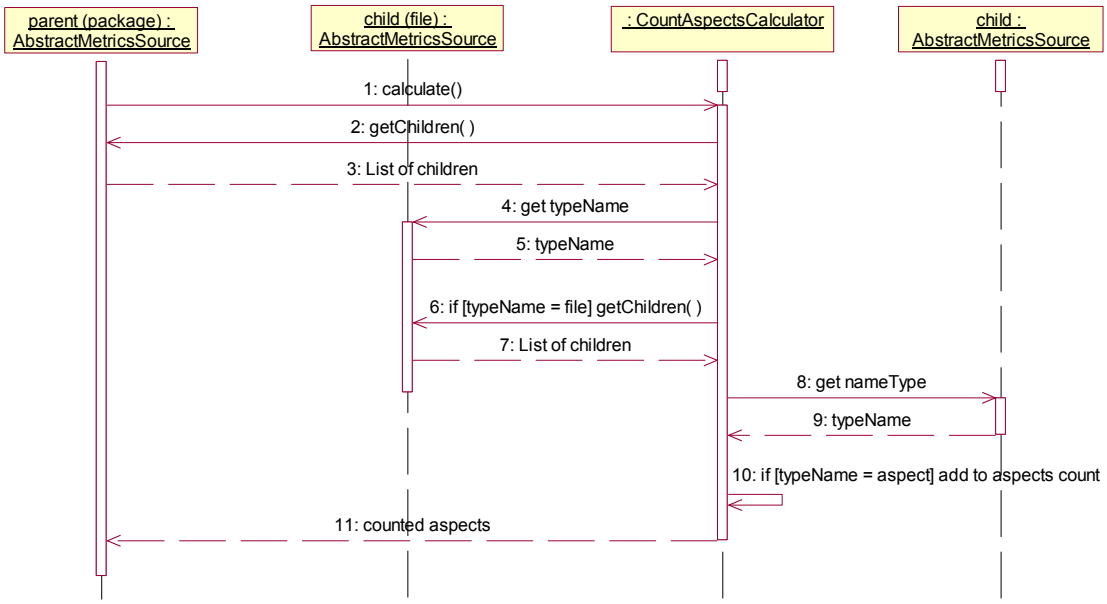


Figure 26 Sequence diagram for the counting of aspects in a package.

Whenever inner sources are also included in the count, the count must be done recursively on appropriate sources. The appropriate sources will be calculator specific. It is e.g. not appropriate to count all the classes in a sub package, whenever counting the amount of classes in a package with the `CountClassesCalculator`, only the classes in package itself.

Source code calculators

As reading code from file is time consuming, it is desirable that this is done only once for each file. There are two situations that could lead to unnecessary reading. The first is when two different calculators need to read from the same file. The second situation is when the same calculator is called for different source elements in same file. Both situations are accounted for in our design which is shown in Figure 27 as the whole source file is read by the method `readSourceLines` and stored along with the current source.

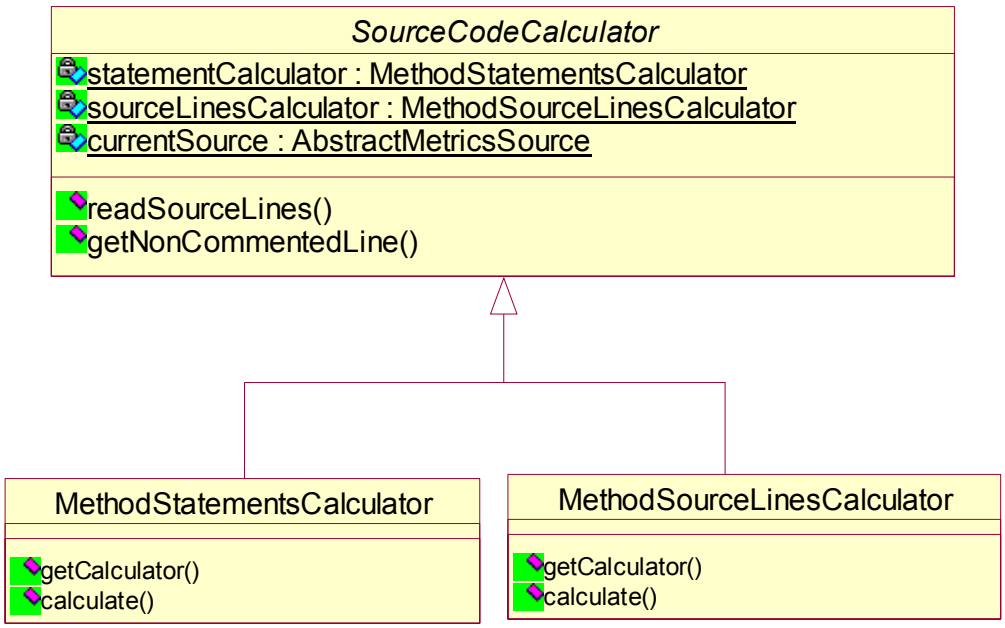


Figure 27 Design of source code counting calculators

Because commented lines are not included in our counts, the method `getNonCommentedLine` removes all characters within a comment including start and end characters like "//", "/*" and "*/".

Fan-in/fan-out calculators

The fan-in/fan-out calculators calculate the amount of method calls that a class send and receive. Outgoing method calls are straightforward to calculate since they are present in the class' own source code. It is only a matter of determining if the call is made to a local method or a method belonging to another class, and adding up in the latter case. The exception is calls to Java or AspectJ library methods, which do not count toward increased fan-out.

The counting of incoming method calls is somewhat more complicated, since the calls are obviously located in the code of the calling class. This is solved by using a map where we add the called location and increase its appurtenant value whenever we calculate an outgoing method-call. Then Fan-in for a class is calculated by looking up its full name in a map and retrieving the value linked to the name.

We need to make sure that fan-out has been calculated before we calculate fan-in. As can be seen in Figure 28 this is done by checking the boolean value `fanOutIsCalculated`. If we attempt to calculate fan-in and `fanOutIsCalculated` resolves to false, fan-out-calculation will commence before returning to complete fan-in-calculation.

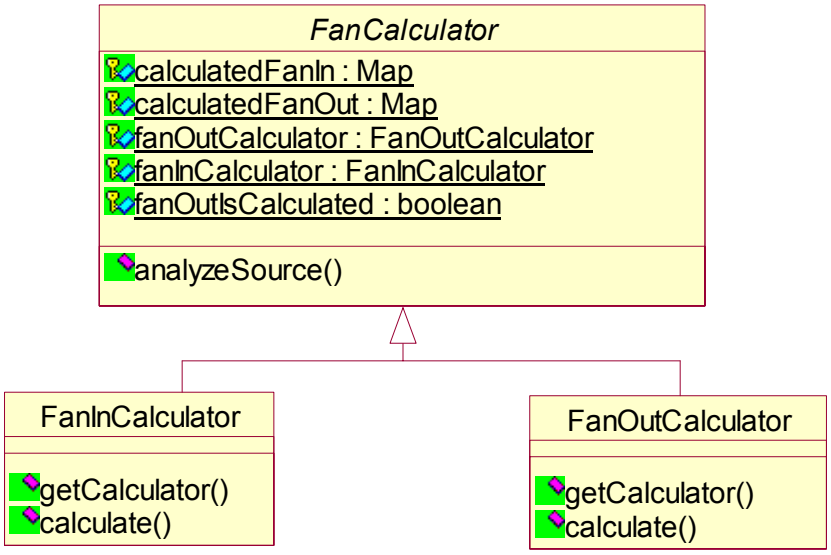


Figure 28 Design of fan-in and fan-out calculators.

Advice calculators

The UML class diagram of the advice calculators, advice-in and advice-out is similar to the one for fan-in/fan-out calculators. The main difference is that the advice-out calculator measures how many join points an advice perform advice on, and the advice-in measures how many times advices hits on joinpoints inside some element. The two advice calculators differ in which source types they are calculated on. Advice-out is calculated on the appropriate advices, while advice-in is calculated on class or aspect level. Advice-in is calculated based on the information from advice-out which is stored in a map from program element to advice-in value, similar to how fan-in/fan-out is calculated.

Coupling calculators

Coupling calculation is done in a similar fashion to fan-in/fan-in calculation, but there are two differences. The first is that only connections inside the project are counted towards coupling; calls to third-party projects are not included. Secondly, the coupling values include direct referencing of class attributes. Referencing of attributes violates the encapsulation properties typical to OO, but is often used by aspects.

In total we perform four coupling calculations, incoming and outgoing coupling inside a class' package, and incoming and outgoing coupling outside a class' package. As can be seen from Figure 29, the design of the coupling calculators is similar to the fan calculators' design.

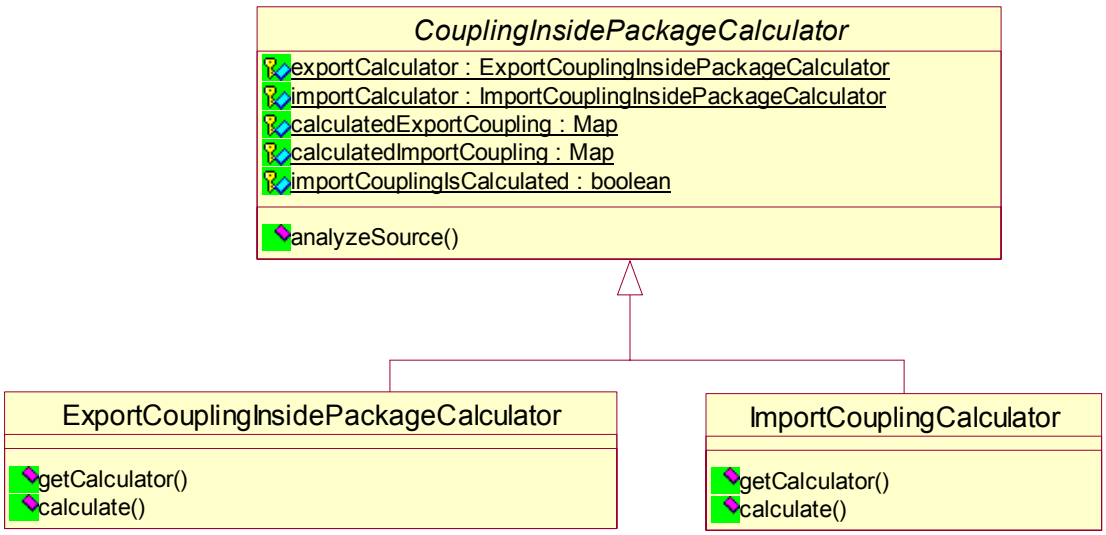


Figure 29 Design of coupling inside packages calculator.

The UML design of coupling outside packages calculators is equal to the one for the coupling inside packages calculators.

Cohesion calculators

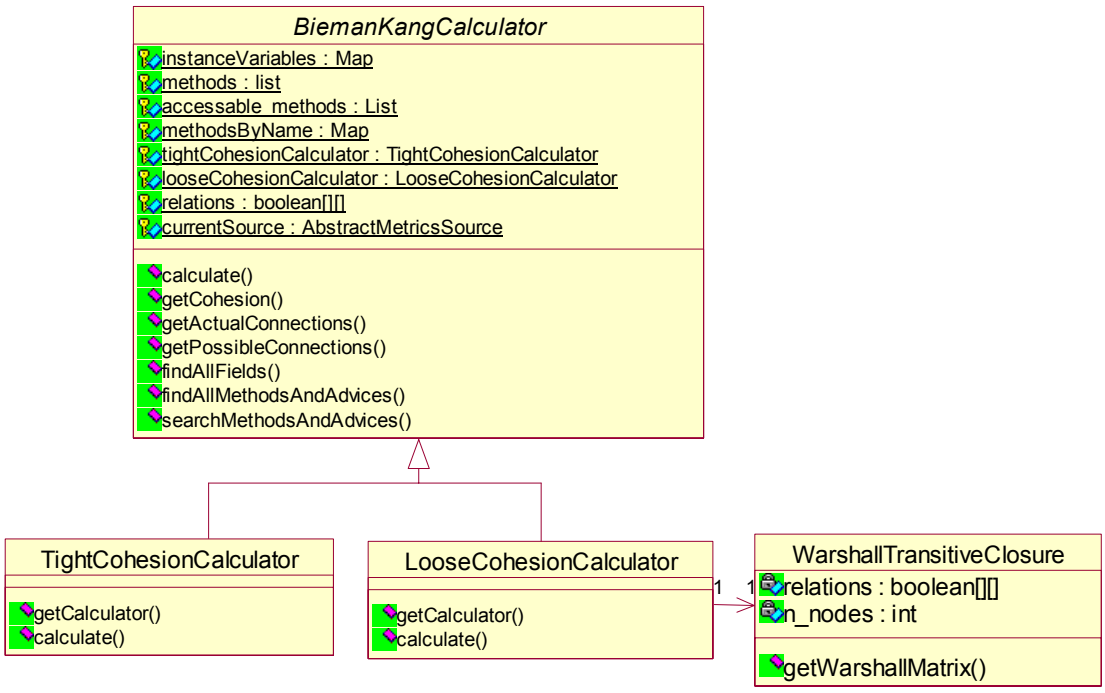


Figure 30 Design of Bieman and Kang's cohesion calculators.

Both `TightCohesionCalculator` and `LooseCohesionCalculator` use the value calculated in the abstract super class `BiemanKangCalculator`, as can be seen from Figure 30 and Figure 31.

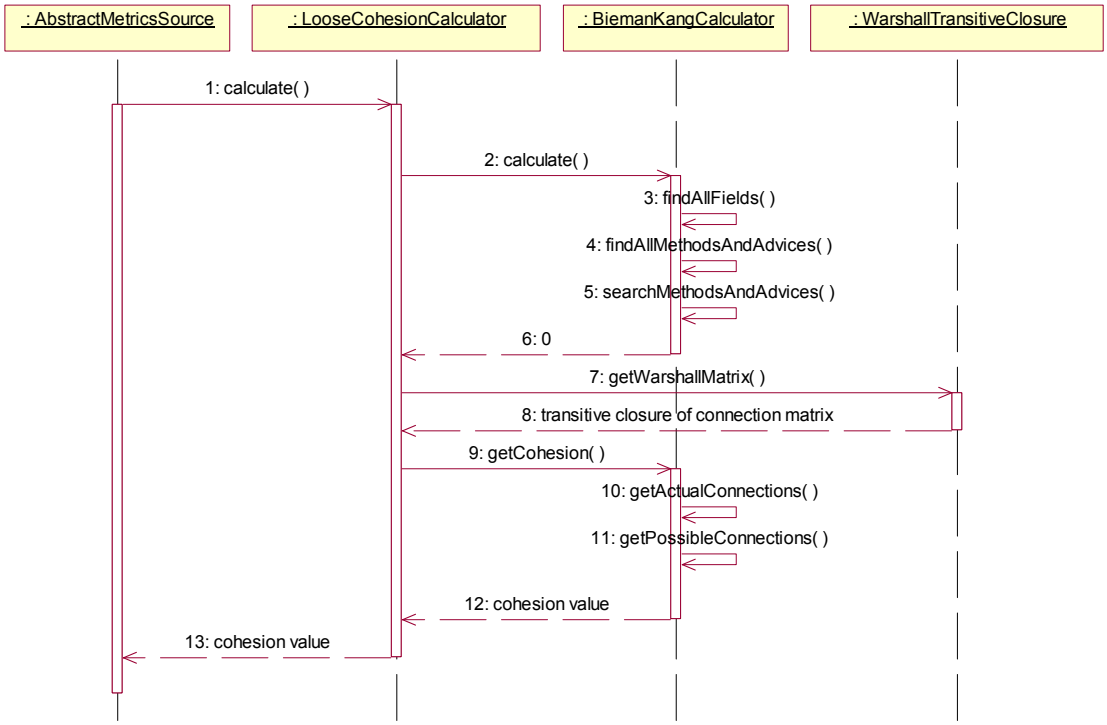


Figure 31 Sequence diagram for the calculation of loose cohesion.

This cohesion value is returned when calling method `getCohesion` which is calculated based on real connections and possible connections. To be able to calculate real connections all fields, methods and advices first must be found. Then we must identify all local method calls and field references. All connections between fields and methods are stored in a map organized by the name of the field. When all these connections are found, relations between methods that use the same field variable are stored in the 2D array "relations". Connections between methods are added directly into that array. All relations between accessible methods in the array are summed up and returned as real connections.

The above mentioned calculation corresponds to Tight Cohesion of class. `LooseCohesionCalculator` takes the relations array calculated in the super class and finds the transitive closure for it through the use of Warshall's transitive closure algorithm, as can be seen in Figure 31.

5.3.6 Reporting

The interface `Ireporter` is provided to make it possible to make any kind of reporters. The abstract class `AbstractDeapthFirstReporter` reports in a depth first matter, printing all the nodes, starting from the outer node in a subtree before moving to the next subtree. The design of the reporter classes are shown in Figure 32.

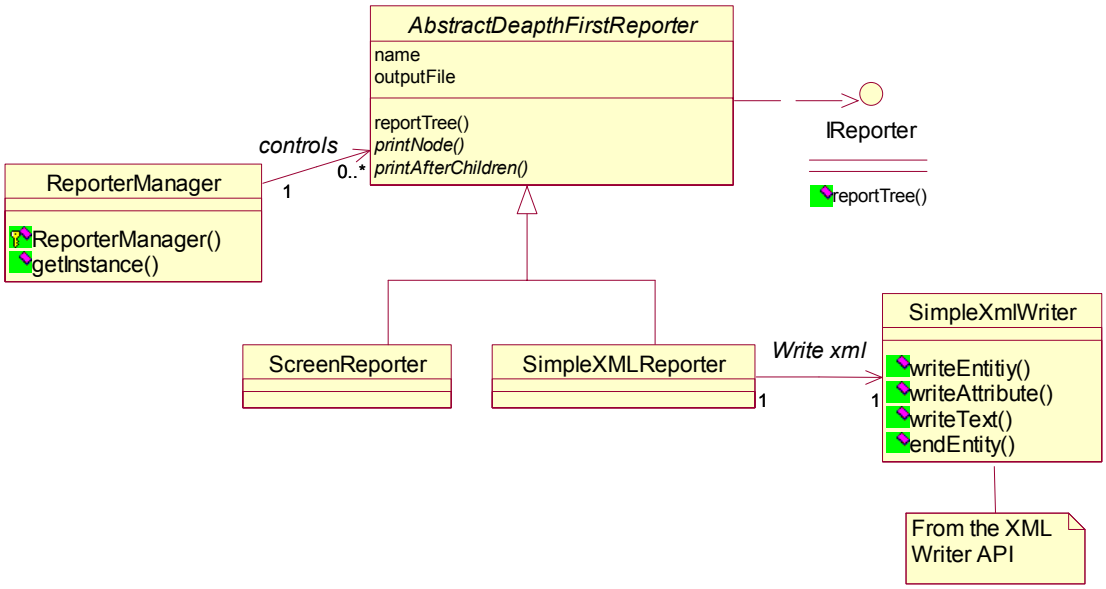


Figure 32 Design of reporters in the metrics tool.

As can be seen in Figure 32, the external class `SimpleXmlWriter` is used for the purpose of writing tags in an XML file in `SimpleXMLReporter`.

5.3.7 Logging

We have not included a design for logging status information and exceptions in this report, but this is a part of the ANT API and an important part for fulfilling the maintenance requirement (NFREQ2). Therefore all exceptions as well as some status information are to be logged to screen.

5.3.8 Extending the tool

Being extendable is a major requirement for the tool. We will thus explain how this can be done for new source types, calculators and reporters.

Adding metrics sources

The need for adding sources may occur when one two following situations appear:

1. Metrics are to be calculated on some program element that is not already defined as a metrics source, e.g. pointcuts.
2. AspectJ or Java grammar is extended.

In the first situation, there is no change to the AspectJ compiler source. Thus, we only need to update code in the `AbstractMetricsSource` and the `SourceManager`, and then add

the new source type which must extend `AbstractMetricsSource`. The `SourceManager` must be extended in order to make the metrics source for the corresponding `IProgramElement` when the metrics source tree is built.

Whenever the second situation occurs, the AspectJ compiler must be revised before correcting the metrics tool. If the new version's public interface is not incompatible with the one we use, then adding new metrics source becomes almost equal as to the first situation. The only difference is that the external libraries must be replaced with the new version.

If the new source should have some defined default calculators, this must be set in the class `MetricsProperties`.

Adding calculators

A new calculator must extend and implement abstract methods of the class `AbstractCalculator`. To be able to use the newly created calculator this must be added to the list of calculators in `CalculatorManager`. If the new calculator should be a default calculator for some source, this must be set in the class `MetricsProperties`.

Adding reporters

Adding reporters is quite similar to adding calculators. The new reporter must either implement the interface `IReporter` or extend and implement abstract methods of the class `AbstractDeapthFirstReporter`. When implementing `IReporter`, the order which the nodes are printed is not set. Changing the default reporter must be done in the class `MetricsProperties`.

5.4 Implementation

The implementation of the metrics tool consists of 72 classes, 2 aspects and 25 test classes. The compiled code is packaged in a jar file name aspectmetrics.jar, and three other jar files are included in the distribution. These are aspectjrt.jar, aspectjtools.jar and xmlwriter-2.2.jar. To run the application ANT version 1.6.1 must be installed on the machine. An ANT build file (build.xml) is also included and this must be edited to run the tool. In addition, 8 stylesheet files (XSL) and some HTML files with pictures are included in the example HTML report site.

All files in the distribution, including example code[21] and the example HTML report files are packaged in a zip file, which can be installed anywhere on the machine. This will run as long as ANT is installed properly.

[21] Two versions of the DIAS2-system is included; with and without aspects.

5.4.1 Retrieving missing information

When designing the metrics tool we had to study and experiment with the code of the AspectJ compiler itself to learn how it functions. Little documentation exists, and none is detailed. It became apparent early in the process that the code trees we generated were missing some vital information. They only indicated method calls and field references that were advised on. This is confirmed by the developers of AspectJ through their mailing list aspectj-dev@eclipse.org.

To get the missing information we add an aspect to the source which hits on all method calls and field references, but is otherwise disregarded when building metrics sources and calculating metrics.

```
public aspect HitOnAll {
      before() : set(* *) {}

      before() : get(* *) {}
```

Figure 33 An aspect that hits on all method calls and field references used as a work-around to build a complete code tree.

Figure 33 shows the aspect that is used to correct the code tree, but otherwise does not add any code to the joinpoints it hits on. The aspect is always given a name that does not already exist in the source tree.

5.4.2 ANT build file

To run the application, the ant build file, build.xml, must be set properly. In Figure 34 we have defined a target that runs the metrics tool on the DIAS 2.1 source code. 4 What is shown in the figure is that we call the task aspectmetrics with some parameters. These parameters defines the type of reporter, an output file for the report, a class path for the DIAS source, a source path and a workspace path that is used when building the source[22]. As mentioned in the code comment in Figure 34, you may also choose which calculators that you will use. If not set, a default set of calculators are used. The default set only contains metrics chosen in the GQM analysis.

[22] The source must be built and the compiled files will be placed in the workspace.

```
<target name="run-dias-metrics" description="Run the AspectMetrics
analyzer on specified code">
   <aspectmetrics reporter="${reporter.name}"
           outputFile="${dias.output.xml.file}">
         <classpath refid="dias.class.path"/>
         <sourcePath refid="dias.source.path"/>
         <workspacePath refid="dias.workspace.path"/>
         <!--  This is where you may choose calculators with eg.
               <calculator name="Aspects"/> -->
   </aspectmetrics>
</target>
```

Figure 34 Extract from the ANT build file, build.xml.

5.4.3 XML report

Results from the calculations as well as a description of the defined metrics are written in a well-formed XML file. The structure of file is defined in the DTD file aspectmetrics.dtd. This DTD is included in Appendix E.

We have included XSL transformation files which transform the data in the mentioned XML file into HTML web pages. A screen shot of the HTML site is shown in Figure 35. We have also a target in the build file which executes the transformation. This is just an example of a report as Telenor use their own transformation files in XRadar.

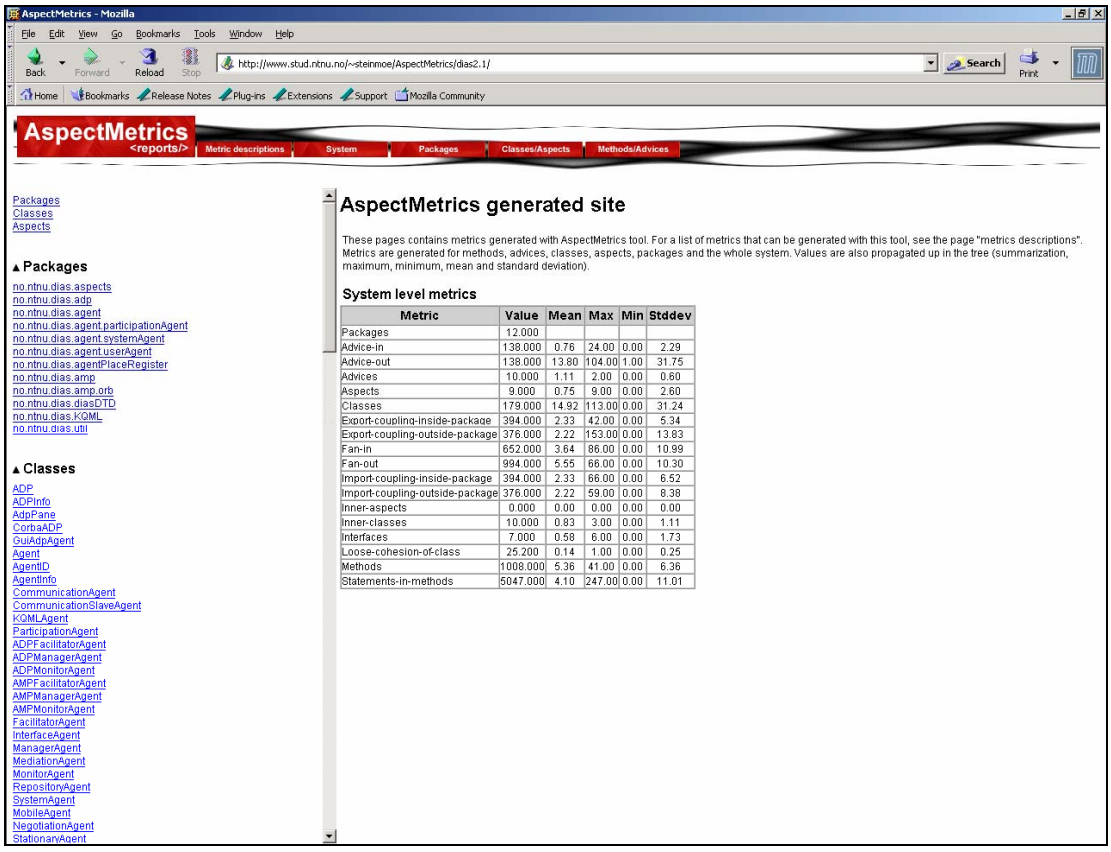


Figure 35 Report from the metrics tool in html format.

The HTML report does not distinguish between sensible and insensible propagated values. An example of this is the summation of loose cohesion of class which actually has no meaning on a higher level.

5.4.4 Testing

Testing the tool is mainly done in two separate ways, through use of specific test code and using the tool on some given code. We have manually counted metric values on the given code to make sure that the calculations are correct.

The use of test code is implemented as JUnit [77] test cases, but used as simple white-box tests for testing and debugging. The reason for using JUnit is that it is integrated with the Eclipse IDE and therefore was convenient to use.

The code has been run on the four different systems DIAS 2.0, DIAS 2.1 (DIAS 2 with aspects), Apache Tomcat open source server and Telenor COS. We have manually counted all the metrics on the DIAS 2.1 source. Six bugs were found and corrected. Test plan and test results are included in Appendix F.

5.4.5 Benchmarks

We have run the metrics tool ten times on three different sources, and the mean times from these runs are shown in Table 24.

	Statements	Has aspects	Mean time	Stddev
DIAS 2.0	5442	no	19,3	0,67
DIAS 2.1	5047	yes	21,6	0,70
Tomcat	41805	no	94	9,98
All times are in seconds				

Table 24 Benchmarks from running the metrics tool.

The following setup was used when running:

- OS: Microsoft Windows XP Proffesional, version 2002.
- Processor: Intel Pentium 4 2.40 GHz
- Ram: 512 MB
- Other applications were running concurrently on the machine.

5.4.6 Requirements fulfilled

As can be seen from architecture, design and implementation, all functional requirements are met with this program. It is, however, not possible to conclude on whether all non-functional requirements are met. The performance requirement (NFREQ1) is met in the tests we have run, but we can not show that the maintenance requirement (NFREQ2) or the extensibility requirement (NFREQ3) is met yet. We have, however, designed our tool to meet these requirements, and we have in this chapter argued why we believe these are met.

Chapter 6. Systems explored

In this chapter we explain the systems we have analyzed with the metrics tool, and give a closer look at the results gathered from the DIAS source code in versions 2.0 and versions 2.1. Even though DIAS was not the primary system target, these results are used because they are the only one that gives us the opportunity to compare DIAS versions with and without aspects. We have done an analysis on DIAS to analyze if the metrics defined may be suited for indicating change in the quality factors defined in the GQM process.

6.1 The systems

As mentioned earlier in section 5.4.5 of Chapter 5, we have run the metrics tool on the Apache Tomcat web server and two versions of Distributed Intelligent Agent System (DIAS). The running on Telenor's Custom Order Server (COS) cannot be done within the time limit of this diploma.

Even though the COS source is the base systems for the choice of quality factors and metrics, we have chosen to analyze the much smaller and different system DIAS. DIAS is originally a diploma project at NTNU. The three main reasons for using DIAS, is that we are in control of the DIAS source, that Telenor are currently only using aspects in the test source and that we have a version of DIAS with aspects that is functionally equal to the version without aspects. There are few similarities between COS and DIAS, but they are both based on communication through server applications and use technologies like XML and Corba.

6.2 Data gathered

In this section results from DIAS 2.0 and DIAS 2.1 are presented and compared.

6.2.1 General information

DIAS 2.0 consists of 11 packages, 6 interfaces and 179 classes where 110 classes are generated with an IDL compiler. In the 2.1 version of DIAS we have added 1 package, 1 interface and 9 aspects, while keeping the same number of classes.

6.2.2 Classes per package

The amount of classes is unchanged in the two versions as can be seen in Table 25.

Package	**DIAS 2.0**	**DIAS 2.1**
no.ntnu.dias.adp	8	8
no.ntnu.dias.agent	3	3
no.ntnu.dias.agent.participationAgent	6	6
no.ntnu.dias.agent.systemAgent	13	13
no.ntnu.dias.agent.userAgent	5	5
no.ntnu.dias.agentPlaceRegister	3	3
no.ntnu.dias.amp	14	14

Table 25 Classes per package for systems DIAS v.2.0 and v.2.1.

We have left out the package no.ntnu.dias.aspects of DIAS 2.1 from the calculations of classes per package because that package only contains aspects.

6.2.3 Methods per class or aspect

We have included the methods of aspects in our analysis. The calculation takes into account all methods, not only public methods.

	Max	Mean	Min	Stddev	Sum
DIAS 2.0	38	5,82	0	6,34	1042
DIAS 2.1	41	5,36	0	6,36	1008

Table 26 Methods per class or aspect for systems DIAS v.2.0 and v.2.1.

6.2.4 Statements in methods

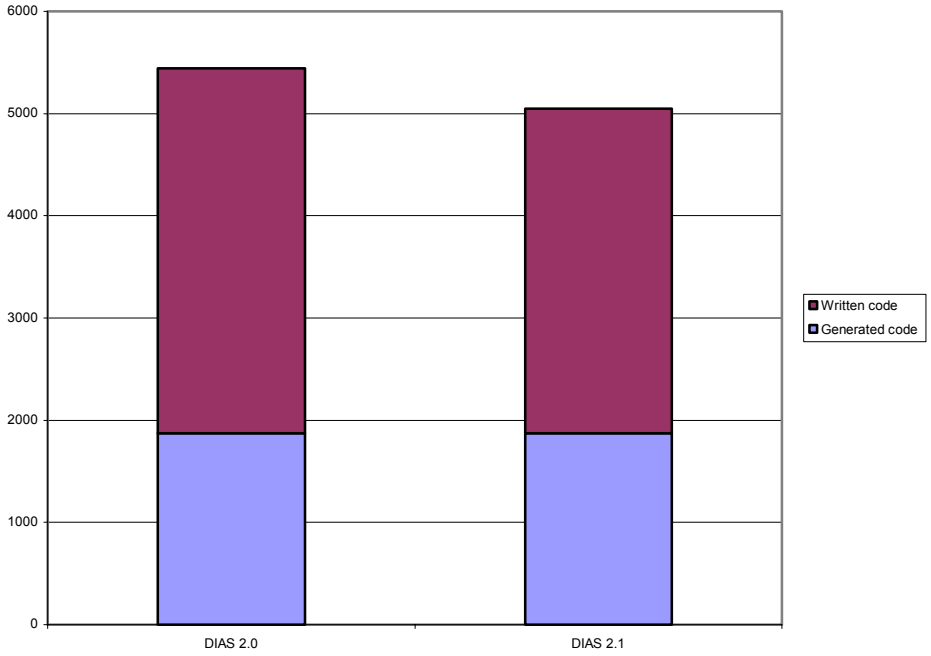


Figure 36 Statements in methods for systems DIAS v.2.0 and v.2.1.

There is 7.26 percent decrease in the amount of statements from DIAS 2.0 to DIAS 2.1. When disregarding generated code, the decrease is 11.03 percent.

6.2.5 Cohesion

We have measured the two cohesion measures of Bieman and Kang, tight cohesion of class (TCC) and loose cohesion of class (LCC). Both metrics employ a ratio scale, with lower limit 0 and upper limit 1.

	mean	**stddev**
DIAS 2.0	0,15	0,25
DIAS 2.1	0,14	0,25

Table 27 Loose cohesion of class (LCC) for systems DIAS v.2.0 and v.2.1.

Loose cohesion drops with 6.67 percent for the system, shown in Table 27. Figure 37 illustrates how the cohesion differs in the packages of the two versions. Remark that the 2.1 version contains one more package than that of 2.0.

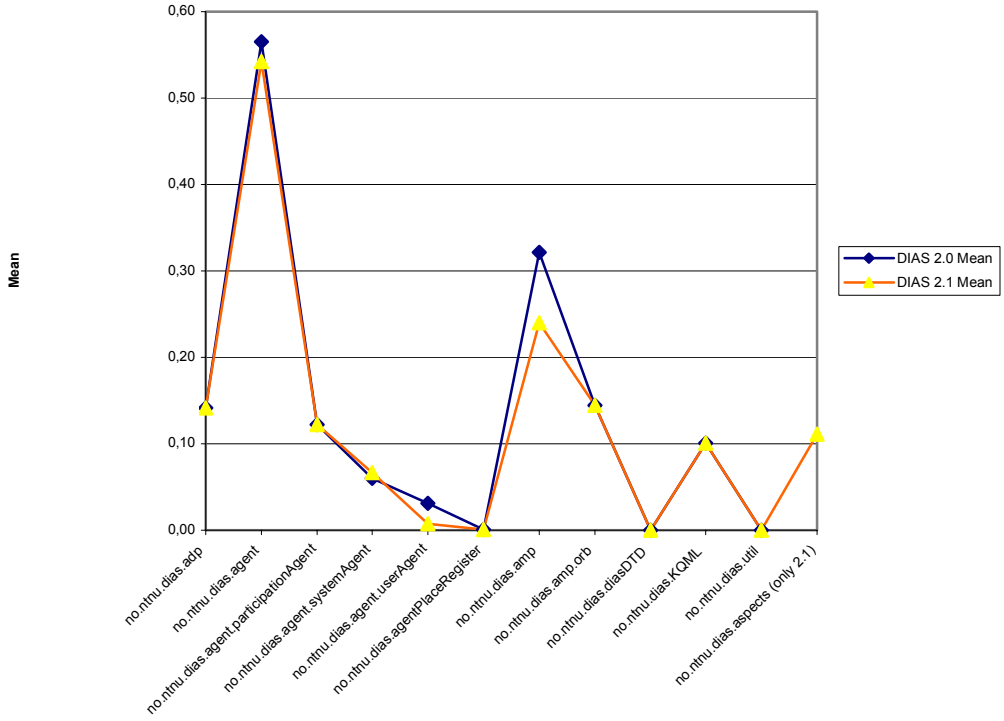


Figure 37 Loose cohesion of class (LCC) for the packages in DIAS v.2.0 and v.2.1.

There is no difference in the mean value of TCC for the two versions, but the standard deviation is increased in the 2.1 version as shown in Table 28.

	mean	**stddev**
DIAS 2.0	0,12	0,21
DIAS 2.1	0,12	0,22

Table 28 Tight cohesion of class (TCC) for systems DIAS v.2.0 and v.2.1.

6.2.6 Coupling

Two kinds of coupling are measured, coupling inside package and coupling outside package. On both kinds outgoing (import) and incoming (export) values are given, as shown in Table 29. Import and export coupling is equal on the system level, but differs in single classes or aspects.

	sum		mean		max		min		stddev	
	DIAS 2.0	DIAS 2.1	DIAS 2.0	DIAS 2.1	DIAS 2.0	DIAS 2.1	DIAS 2.0	DIAS 2.1	DIAS 2.0	DIAS 2.1
Export coupling inside package	395	394	2,34	2,21	66	66	0	0	6,52	6,37
Import coupling inside package	395	394	2,34	2,21	42	42	0	0	5,35	5,23
Export coupling outside package	422	399	2,50	2,24	191	167	0	0	14,52	14,56
Import coupling outside package	422	399	2,50	2,24	70	59	0	0	9,53	8,22

Table 29 system level coupling for systems DIAS v.2.0 and v.2.1.

The difference between the two version's coupling inside package measures is a decrease of one instance of coupling. System level coupling outside package is decreased by 23 instances, or 5.4 percent.

6.2.7 Fan-in and fan-out

The fan-in and fan-out measures are shown in Table 30 and Table 31.

	sum	mean	max	stddev
DIAS 2.0	688	3,84	113	12,47
DIAS 2.1	662	3,52	87	11,05

Table 30 System level fan-in outside package for systems DIAS v.2.0 and v.2.1.

The sum of fan-in is decreased by 3.8 percent on the system-level, while the mean value decreases by 8.3 percent.

	sum	mean	max	stddev
DIAS 2.0	1090	6,09	66	11,38
DIAS 2.1	1020	5,43	66	10,09

Table 31 System level fan-out outside package for systems DIAS v.2.0 and v. 2.1.

There is a 6.4 percent decrease in the sum of fan-out and 10.8 decrease in the mean value.

6.2.8 Advice-out and advice-in

The sums of advice-out and advice-in must be equal on the system level, but different on the lower levels and thereby have different system mean values, as seen from Table 32 and Table 33.

	sum	mean	max	stddev
DIAS 2.0	0	0	0	0
DIAS 2.1	138	0,76	24	2,29

Table 32 System level advice-in outside package for systems DIAS v.2.0 and v.2.1.

Advice-in is measured on aspects or classes, while advice-out is measured on advices, thus making mean, max and standard deviation not comparable between advice-in and advice-out.

	sum	mean	max	min	stddev
DIAS 2.0	0	0	0	0	0
DIAS 2.1	138	13,8	104	1	31,75

Table 33 System level advice-out outside package for systems DIAS v. 2.0 and v.2.1.

6.2.9 Advice-out + Fan-out

There are 8 aspects with advices in the 2.1 version of DIAS and their advice-out and fan-out values are shown in Figure 38.

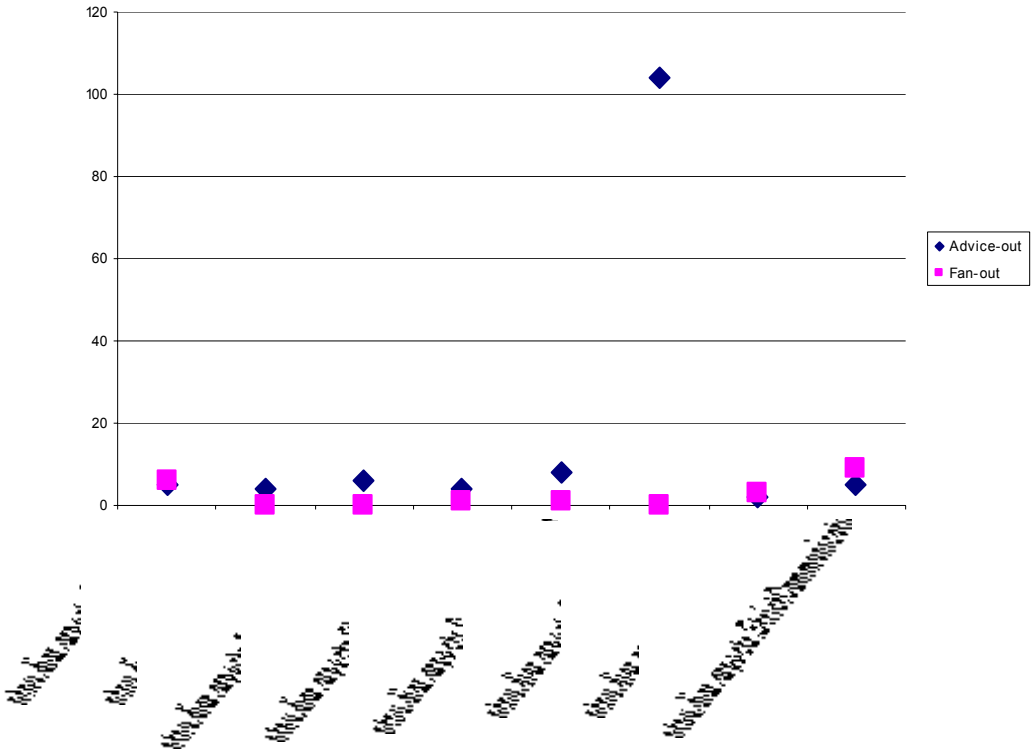


Figure 38 Advice-out and fan-out for the aspects in DIAS 2.1.

There is one aspect in particular that have a large value for advice-out. This aspect, named `ExceptionPrinter`, prints exception to screen or other chosen media. The corresponding

fan-out value for this aspect is 0. A fan-out of value of zero means that there are no external methods calls, except for calls to Java API methods and AspectJ API methods.

6.3 Analysis of the quality factors from GQM

To be able to evaluate the metric values, we will use the questions and quality factors from the GQM analysis. We want to see if the newest version of DIAS has metric values that constitute a change caused by the introduction of AOP. Radar plots containing metrics that can be compared are used to illustrate the changes in the quality criteria. There is no absolute scale in the plots, but further away from centre indicates a better value. We give an analysis of the questions in the same tables as where we answer the questions, indicate their effect on quality factors and comment on the metrics used to give the answers.

6.3.1 SG1: Modularity

Our comparison of modularity is based on coupling inside package, coupling outside package and loose cohesion of class. All metrics have been calculated on the class level. The results are illustrated in Figure 39. The questions defined for modularity are analyzed in Table 34.

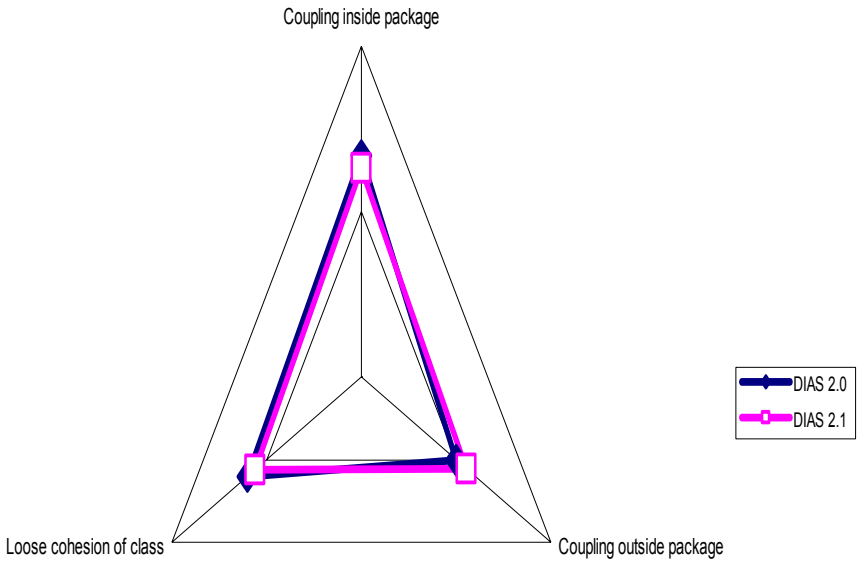


Figure 39 Modularity metrics for DIAS v. 2.0 and v. 2.1.

The comparison shows little difference in the three metrics between the two versions. In addition, there is little or no difference between the two versions concerning classes per package and methods per class or aspect.

Scientific question and answer	Change	Comment on metrics' effects
Q1.1. Are the functions in modules related? The results do not show that modules are more related in DIAS v.2.1.	+/-	As illustrated in Figure 39, the change in the coupling and cohesion metrics are small and inconclusive.
Q1.2. How many functions do modules provide? The number of functions in modules is not substantially changed.	+/-	The number of classes is unchanged, while the mean number of methods per class or aspect is slightly decreased, mainly because we have introduced aspects with few methods. The number of methods in the classes is unchanged.

Table 34 Scientific questions, how metrics results affect modularity and comments.

The results of the analysis of modularity as shown in Table 34 indicate no change in the modularity of DIAS after introducing aspects.

6.3.2 SG2: Testability

We have defined testability as a combination of metrics coupling outside package, fan-in and fan-out, and their effect is shown in Figure 40. All metrics have been calculated on the class level.

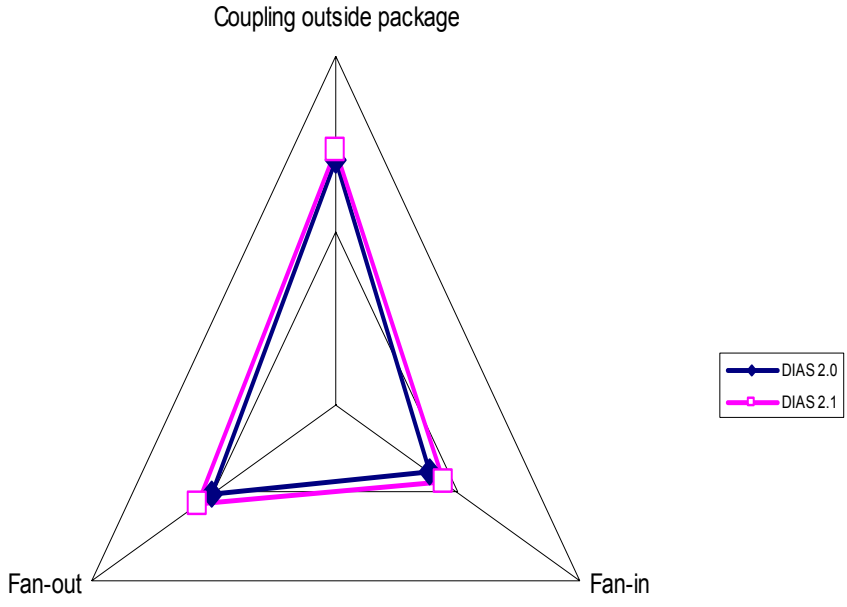


Figure 40 Testability metrics for DIAS v. 2.0 and v. 2.1.

Another metric defined for testability is the amount of commented methods, but this metric has not been implemented in the metrics tool.

Scientific question and answer	Change	Comment on metrics' effects
Q2.1 What responsibilities do the modules have?	N/A	(Information from other sources than the metrics is not included.)
Q2.2. Are the modules strongly connected to other modules? The modules are less connected.	+	Mean value of coupling outside package is slightly decreased, thereby constituting a small positive change.
Q2.2. Do the modules have many dependencies towards other modules? The modules have fewer dependencies towards other modules.	+	Mean value for import-coupling outside package is the same as the coupling outside package value, which have a small decrease. The fan-out value decreases in DIAS v.2.1. These results indicate a positive change.
Q2.3. Do many other modules depend on the modules? The modules depend (slightly) less on modules.	+	Mean and sum value for export-coupling outside package is the same as the coupling outside package value, which have a small decrease. Fan-in is decreased.
Q2.4. Are all classes and methods documented?	N/A	(Amount of commented method is not calculated.)
Q2.5. Is the documentation of high quality?	N/A	(Information from other sources than the metrics is not included.)

Table 35 Scientific questions, how metrics results affect testability and comments.

We have not analyzed all the questions defined for the quality criterion testability. The results we have shown in Table 35 indicate, however, that we have a positive change in the DIAS code from the introduction of aspects. We also know that the documentation of the classes and methods are left unchanged in the 2.1 version of DIAS.

6.3.3 SG3: Analyzability

The main metrics that affects analyzability is shown in Figure 41.

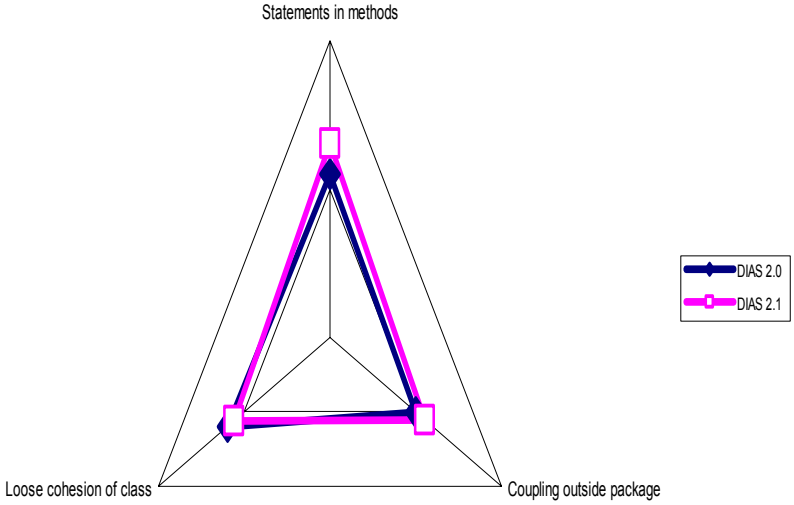


Figure 41 Analyzability metrics for DIAS v. 2.0 and v. 2.1.

Metrics like methods per class and classes per package are unchanged from v.2.0 to v.2.1 and are therefore not included in the figure. The scientific questions and corresponding metrics for analyzability are commented in Table 36.

Scientific question and answer	**Change**	**Comment on metrics' effects**
Q3.1. Are the modules large? Modules are smaller.	++	Code size is decreased in DIAS v.2.1 while classes per package and methods per class are unchanged. This indicates that modules are generally smaller in v.2.1.
Q3.2. Do the modules have clear and distinct responsibilities? We cannot conclude that modules are more or less clear and distinct.	+/-	There is small negative change in cohesion and a small positive change in mean value for import coupling outside package. These results indicate no great change.
Q3.3. Do the modules have sensible names?	N/A	(Information from other sources than the metrics is not included.)
Q3.4. Are sub modules (classes and aspects) placed in sensible modules (packages)? Classes and aspects should be more sensibly placed.	+	Mean value of coupling outside packages is decreased.

Q3.5. Are all classes and methods documented?	N/A	(Amount of commented method is not calculated.)
Q3.6. Is the documentation of high quality?	N/A	(Information from other sources than the metrics is not included.)
Q3.7. Are the connections between modules sensible?	N/A	(Information from other sources than the metrics is not included.)

Table 36 Scientific questions, how metrics results affect analyzability and comments.

Four of the seven questions, defined for analyzability, have not been answered in this analysis. We know, however, that the naming and documentation of the classes in DIAS are unchanged and then only one question remains unanswered. The notable decrease in module size should indicate a positive change in the analyzability factor for DIAS.

6.3.4 SG4: Changeability

Changeability is affected by loose cohesion of classes, fan-in, fan-out and coupling outside package, as shown in Figure 42.

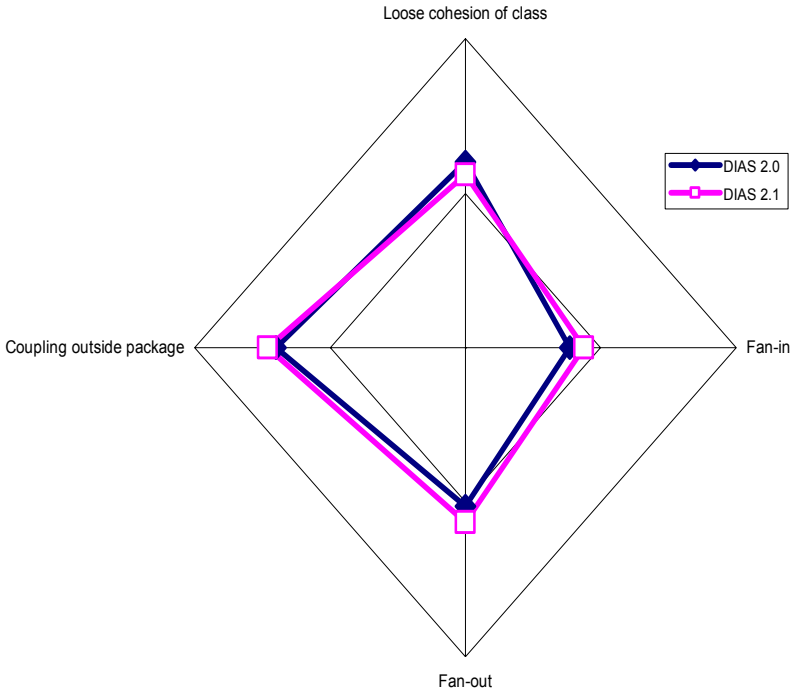


Figure 42 Changeability and stability metrics for DIAS v. 2.0 and v. 2.1.

The metrics advice-in and 'fan-out + advice-out' are not included in Figure 42, but commented on in Table 37.

Scientific question and answer	Change	Comment on metrics' effects
Q4.1. Do we trust that the modules perform the tasks they are supposed to? We cannot conclude that the modules perform more or less of the correct tasks.	N/A	(Information from other sources than the metrics is not included.)
Q4.2. Do we have trust in the developer's abilities?	N/A	(Information from other sources than the metrics is not included.)
Q4.3. Do we understand the code? Even though we cannot say anything about the comments, the code should be more understandable.	+	The amount of commented methods is not calculated, but coupling and fan-out is positively changed. Advice-in add some negative change.
Q4.4. Do the modules have clear and distinct responsibilities? We cannot conclude that modules are more or less clear and distinct.	+/-	There is small negative change in cohesion and a small positive change in mean value for import coupling outside package. These results indicate no great change.
Q4.5. Are the interfaces of the modules precisely defined?	N/A	(Information from other sources than the metrics is not included.)
Q4.6. Are the modules heavily used? The modules are less heavily used.	+	Fan-in decreases, thus modules are less used.

Table 37 Scientific questions, how metrics results affect changeability and comments.

Three questions are left unanswered in Table 37, but these can be answered through our subjective opinion and knowledge. The interfaces of DIAS are unchanged in the new version of DIAS, and it is reasonable to believe that the quality of our code work is of a similar level

to that of the students developing the previous version of DIAS. All in all, the results indicate a positive change in changeability for DIAS.

6.3.5 SG5: Stability

The stability metrics are roughly the same as the changeability and the difference between the two versions are therefore illustrated by the same figure, Figure 42. All the questions and corresponding metrics for stability are commented in Table 38.

Scientific question	**Change**	**Comment on metrics' effects**
Q5.1. Do the modules have clear and distinct responsibilities? We cannot conclude that modules are more or less clear and distinct.	+/-	There is small negative change in cohesion and a small positive change in mean value for import coupling outside package. These results indicate no great change.
Q5.2. Do the modules have many dependencies towards other modules? The modules have fewer dependencies towards other modules.	+	Mean value for import-coupling outside package is the same as the coupling outside package value, which have a small decrease. The fan-out value decreases in DIAS v.2.1. These results indicate a positive change.
Q5.3. Do many other modules depend on some modules? Fewer other modules depend on the modules in general.	+	Both import-coupling and fan-out is decreased indicating a positive change.
Q5.4. Have the modules been changed over time, without failures and complications?	N/A	(Information from other sources than the metrics is not included.)

Table 38 Scientific questions, how metrics results affect stability and comments.

Question number 4 in Table 38 has not been answered with the metrics calculated, but there is little or no difference in this factor between the two versions. The other results in Table 38 indicate a positive change in stability for DIAS.

6.3.6 G1. Reusability

We show the change indicated in the subquality factors of reusability in Table 39.

Quality factor	Change indicated
SG1: Modularity	+/-
SG2: Testability	+
SG3: Analyzability	+
SG5: Stability	+

Table 39 Change indicated in the sub quality factors of reusability.

All in all, the changes in the sub quality factors of reusability indicate a positive change in the reusability factor of DIAS.

6.3.7 G1. Maintainability

We have indicated change for all sub quality factors other than adaptability of maintainability for DIAS in Table 40.

Quality factor	Change indicated
SG2: Testability	+
SG3: Analyzability	+
SG4: Changeability	+
SG5: Stability	+
Adaptability	N/A

Table 40 Change indicated in the sub quality factors of maintainability.

We have a positive change in four of five sub quality factors of maintainability and thus an indication of improved maintainability for DIAS.

Chapter 7. Discussion

7.1 GQM analysis

The GQM analysis has given us a set of quality factors and metrics pertaining to improve product quality and to indicate change in them. We have focused on the two system goals reusability and maintainability which are important for the development of middleware systems like COS.

The definition of the chosen quality factors and questions is based on ISO 9126, discussions with advisors at Telenor and NTNU, experience from the preceding project and common sense. The ISO 9126 is a respected and frequently used framework for software quality. All in all, the choice of quality factors should be well-founded. Even so, there is room for improvement. Experience from system studies, more support from theory and more comments from developers could have given more reliability to the process.

7.2 Metrics

When choosing metrics to answer the questions from the GQM analysis we have for the most part chosen well-known metrics. The exceptions are advice-in and advice-out, but they have been inspired by and have much in common with the well-known fan-in/fan-out metrics.

7.2.1 Size

We have made our own statement calculator and a calculator for non-commented non-empty source lines inside methods. As expected the statement calculator gives a somewhat smaller value for code size than the more commonly used number of lines calculator. We argue that the statement calculator gives the more accurate indication of actual code size; and thus the more accurate indication of how complexity increases as code size increases. The line calculator indicates a higher value whenever the developer has divided a statement over two or more lines.

7.2.2 Cohesion

Bieman and Kang's cohesion measure [64] is not a perfect cohesion measure, even though it was the best we could find. Its major weakness is for calculation of classes with many methods. The result will often be a low cohesion value, even though the class is quite cohesive. This is because of the quadratic increase in denominator value as the number of methods increase.

There are doubts about the usefulness of cohesion in its current form under the object-oriented paradigm. Our experience takes this further, by also including AOP. Moving functionality to aspects can have both positive and negative effect on the local class cohesion value. When inspecting cohesion values for individual classes in the DIAS system that have been influence by aspects, we found that the cohesion values sometimes indicated improvement and sometimes indicated deterioration.

A system may become more modular as a whole, but this is not necessarily reflected in the new cohesion values. Thus, we are not sure if cohesion, at least in its current form, is a useful metric when using AOP.

7.2.3 Coupling

Ideally, the coupling metrics should weight calls to methods and direct referencing of attributes differently. Direct referencing of attributes violates the encapsulation and information hiding properties associated with OO. In addition, direct referencing is often used by aspects to gather information about the object, and thus the relative strength between method calls and attribute referencing is a highly interesting one.

Weighting is originally part of Hitz and Montazeri's coupling framework, but there is some ambiguity about the actual weights. In [50] a table for weighting is presented, but to use this you need to know in advance if the class or package is considered stable or not. This is problematic, since the calculation of package stability includes coupling. Further, Briand [30] states that "different measurement goals can require different (partial) orders". If you are measuring in relation to regression-testing[23], then import-coupling is the most interesting part as you are interested in the flow of control. Referencing of attributes would not be of interest as they do not influence this. On the other hand, if you are measuring understandability, then attribute referencing is just as interesting as method calls.

7.2.4 Fan-in / Fan-out

We experienced a small decrease for both fan-in and fan-out. This was as expected as illustrated in Figure 16, page 51.

7.2.5 Advice-in / Advice-out

Advices impact the system 138 times, with one aspect, the `ExceptionPrinter`, being responsible for 104 of those. This aspect is a good example on how much a particular functionality can be scattered throughout the system. The advice-out value only tells us the amount of locations the advice affects, not the locations. Thus, the advice-out value alone cannot tell us if the advice is used to remove functionality that was scattered through large parts of the system. Other tools can be used for this, e.g. AJDT [87], as shown in Figure 43.

[23] "Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes" [86].

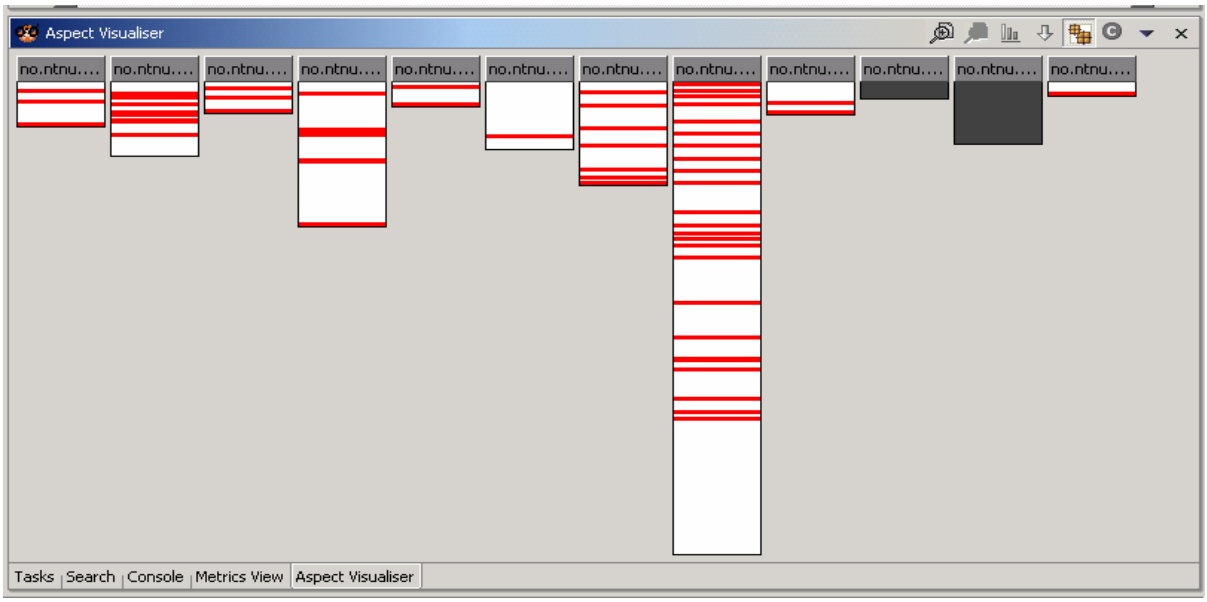


Figure 43 Example of scattering with the ExceptionPrinter aspect.

The system advice-in value should be evaluated together with system fan-in. A decrease in fan-in is expected, but if we add the advice-in value to the fan-in value, this decrease will be smaller. Thus, the amount of connections in the system is higher than the one given by fan-in and fan-out alone. The fan-in/fan-out and advice-in/advice-out values could have been combined in a fan-in and a fan-out metric. We have chosen not to combine them because method calls and advices are different in the way they are used. A method is called on, while an advice connects itself to the locations of use (joinpoints), as illustrated in Figure 16, page 51.

7.2.6 Advice-out + Fan-out

We have earlier expressed concern for aspects that have too big an impact on the system. It would be logical to think that the `ExceptionPrinter` advice would fall into this category. However, this advice does not alter the behaviour of the code it impacts on; its fan-out is 0. All it does is to undertake the responsibility of printing exceptions. The advice with the biggest functional impact on the system is in fact the `StrictCommunication` advice. It has 5 outgoing advices and in addition 9 instances of fan-out. This is by no means large values, but for this particular system it is the aspect that should probably receive the largest test effort.

7.3 The metrics tool

We have implemented and tested the tool according to the requirements. The requirements fulfil almost all metrics defined in the GQM process, see Chapter 4. We have not implemented the counting of commented methods since we did not prioritize this. A dependency graph report is also mentioned in the GQM, but not implemented in the tool. In both cases, the time available was too small.

As we have tested the tool on a small number of systems, there is a risk that some bugs have not been found. Use and maintenance over a longer period of time is required to be confident confident of the reliability of the tool.

A property in the tool which might limit the usefulness is the fact that the code must be compiled. For large systems setting up source path and classpath can be a complicated task as this requires knowledge about how the system code is built. On the positive side, the fact that the code must be compiled has as a consequence that code with errors will be left out of the analysis. Code with errors should not be a part of the system, and thus not measured.

7.4 Results from the systems and AOP effects

In our opinion, the results from DIAS v.2.0 vs. DIAS v.2.1 (with aspects) must be used with care because the system is small and different from Telenor Mobile COS. It is also worth noting that the system is a part of a student diploma and not a business system.

The results from the analysis indicate that reusability and maintainability are improved in the 2.1 version. Concerning maintainability, we see a positive change because the aspects implemented can be reused and because some of the classes are smaller and the system more readable [1]. There is, however, no difference in the public interfaces, thus indicating no change in reusability. The maintainability of DIAS is, in our opinion, improved because of smaller classes and better localization of side functionality, e.g. logging.

A surprising result of the analysis is the fact that there is no indication of positive change in modularity. Modularity is said to be one of the major benefits with AOP as crosscutting concerns are localized in aspects and not spread in classes. This might indicate that the metrics chosen are not good indicators or that modularity is in fact not improved in DIAS v.2.1. Other possible reasons are that the modularity has in fact improved, but that the metrics have not been correctly calculated.

The reason we find most likely, however, is that the criteria we have specified for improved modularity are not good enough. We have already mentioned that we have little confidence in the cohesion metrics' ability to demonstrate change in AOP-systems. Considering that functionality have been moved out to the aspect package, an increase in coupling inside the packages cannot be expected. A better specification for this criterion would have been to keep the same level of coupling inside packages. It is the coupling towards other packages that AOP mainly seek to improve, which is also reflected in the measurement results we have gathered through coupling outside packages. With improved criteria we would thus have achieved an indication of improved modularity after introducing aspects to the system.

The other quality factors, such as testability, analyzability and stability, are influenced by the moving and merger of crosscutting concerns which reduces the code size and reduce the amount of connections between modules. When considering crosscutting code that is moved to an aspect, it is positive that we only need to analyze and test the functionality once. On the other hand, all modules that are affected by an advice must take that advice into consideration when being changed or updated.

A part that is not taken into account in the metrics is that AspectJ is a more powerful language than Java. By more powerful, we mean more tokens to use, e.g. pointcuts and introductions. The new tokens add both new possibilities and new dangers that the developers must handle. There are some subjective factors that will have influence on how big impact the language has on the chosen quality factors for instance:

- The abilities of the developers.
- The experience with the language.

Chapter 8. Conclusion

We have studied Aspect-Oriented Programming and the impacts of implementing aspects with AspectJ and Java as languages. We have identified the system quality factors for the COS system, and studied how we could measure changes in the quality factors when using AOP.

This leads us to the first part of our problem definition:

We will find useful metrics for measuring AOP-based programs.

From our work and measurement analysis we have found the following metrics to be useful for measuring AOP-based programs:

- Number of statements
- Fan-in / Fan-out
- Coupling
- Advice-in / Advice-out

From our work and measurement analysis we have found the following metric not to be useful for measuring AOP-based programs:

- Cohesion

The second part of our problem definition has been:

We will develop a prototype metric tool for use with AspectJ.

We have designed and implemented a metrics tool, AspectMetrics, which is able to measure our metrics selection on AspectJ and Java source code. The tool fulfills all the planned functional requirements and is able to calculate and present results for a medium sized system, i.e. Apache Tomcat (800 classes), in less than two minutes.

If the other two non-functional requirements, maintainability and extendability, have been fulfilled can only be determined through continued use of the tool.

Chapter 9. Further work

During our work with this diploma we found several interesting areas that we could not pursue within the time frame. These areas, alone or in conjunction, should be goals for future projects.

9.1 Analyze results from systems in a period of time

We have not had the chance to analyze a business system over a period of time in our study. As we have based our analysis on Telenor Mobile's middleware systems COS, it would be favourable if this system was studied. A similar system could also be used; preferably a system that has implemented aspects through AspectJ. In COS, aspects are at the time of writing only used for test code.

A study of the above mentioned type is, in our opinion, the most important area for further work. The goal for such study would be to watch if the chosen metrics have notable effects on maintainability and reusability. Thus, to find these effects other data besides the metrics also needs to be gathered. Relevant data is the amount of reported errors, correction cost, and the qualitative opinion of developers and users. GQM could be used to elaborate on the data needed[24].

9.2 Refine the quality model

The quality model that we have defined in this thesis is not a complete model and is partly based on anecdotal information and experience. Refining the model is therefore of interest; based on empirical information as well as theoretical knowledge. The work could produce new scientific questions and metrics as well as alterations of our work.

9.3 Analyze other system goals

We have chosen to focus on a subset of the relevant system goals in our study. This is done because of the time limit for the project. Other relevant system goals defined in ISO 9126 are functionality, reliability, usability, efficiency and portability.

[24] As we have focused on software metrics in our use of GQM, we have not investigated the need for other information like bugs and qualitative opinions.

9.4 Cost-effectiveness analysis

We have indicated improved reusability and maintainabililty for the DIAS system by introducing aspects. But we have not established a relationship between the values gathered from the metrics and the amount of reduced maintenance time. This could be studied as a student project, e.g. like this: “Every student, or group of students, accounts the amount of time they spend on maintenance for a given project. This is then compared to the results their systems’ achieve by using the metrics tool.”

9.5 Develop new metrics

In addition to refining the quality model, new metrics suitable for measuring AspectJ code should be developed and implemented in the metrics tool developed in our study. The metrics tool is designed so that it is simple to add new metrics. In our opinion, new metrics that capture the effects of introducing aspects are still needed.

There is a need for research to identify which metrics, and combination of metrics, that best describes the properties of AOP-systems and their differences compared to regular OO-systems. The aim for such a study would be to define a more complete and tailored metrics suite than the one used in this thesis.

There are a collection of metrics that we elected not to implement in this version of the tool due to time constraints. They range from basic size metrics, such as the number of public methods and the number of commented methods, to more advanced metrics that aim to estimate code or system complexity. A selection of complexity metrics is presented in Appendix C.

9.6 Extend metrics tool with call graphs

An interesting extension for the metrics tool is a report containing call graphs. Such graphs can be based on method calls, it can be based on field use or it can be based on aspect affectations. This can easily be implemented as a metrics calculator in the tool and reported in a suitable new reporter. To visualize the graph, the XML child languages Scalable Vector Graphics (SVG) can be used. The SVG reports can also be incorporated as a plugin to XRadar [80].

Appendix A. References

All web addresses have been confirmed not broken at the date indicated between parentheses.

[1] Rønningen and Steinmoen. Increasing readability with Aspect-Oriented Programming. TDT4735 Software Engineering, Specialization project. Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), November 2003.

[2] Editors, Ten emerging technologies that will change the world. MIT Technology review, January/February 2001.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier and J.Irwin. Aspect-Oriented Programming. Proceedings of European Conference on Object-Oriented Programming Finland. Springer-Verlag, 1997.

[4] Marcelo Sihman and Shumuel Katz. Superimpositions and Aspect-oriented Programming. The Computer Journal, British Computer Society, issue 46(5) 2003.

[5] Norman E. Fenton and Shari Lawrence Pfleeger. Software Metrics, A Rigorous & Practical Approach, 2nd edition. PWS Publishing Company, 1997.

[6] Kristoffer Kvam and Daniel Bakkelund. Cynical Reengineering. Telenor Mobil, IT Core Systems, COS Middleware. Presented at XP2004.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. ACM Press, 1972.

[8] G. Kiczales, T. Elrad, T. Aksit, K. Lieberherr and H. Ossher. Discussing Aspects of AOP. Communications of the ACM 44(10), October 2001.

[9] AccessSience: McGraw-Hill's online encyclopaedia for science and technology. http://www.accessscience.com/ (15.03.2004)

[10] JDepend, a Java Metrics tool. http://www.clarkware.com/software/JDepend.html (15.03.2004)

[11] Ian Sommerville. Software Engineering. Addison-Wesley, 2000.

[12] Sallie M. Henry and Dennis G. Kafura. Software Structure Metric based on Information Flow. IEEE Transaction on Software Engineering, 7(5), September 1981.

[13] Arthur H. Watson and Thomas J. McCabe. Structured Testin: A Testing Methodology Using the Cyclomatic Complexity Metric. Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, September 1996.

[14] Sencer Sultanoglu. Software Measurement Page, Software Complexity. http://yunus.hun.edu.tr/~sencer/complexity.html. Department of Computer Science & Engineering. Hacettepe University, Turkey. (15.03.2004)

[15] Metrics 1.3.4 and Metrics 1.3.5, a metrics plug-in for Eclipse. http://metrics.sourceforge.net/ (15.03.2004)

[16] Zakaria and Hosny. Metrics for Aspect-Oriented Software Design. The American University in Cairo. Workshop on Aspect-Oriented Modelling, International Conference on Aspect-Oriented Software Development, March 2003.

[17] Norman Walsh. What is XML? http://www.xml.com/pub/a/98/10/guide1.html#AEN58 (15.03.2004)

[18] Apache Ant. http://ant.apache.org/ (15.03.2004)

[19] Stephen R. Schach. Classical and Object-Oriented Software Engineering. McGraw-Hill, 1999.

[20] Capers Jones. Applied Software Measurement: assuring productivity and quality. McGraw-Hill, 1996.

[21] ISO9126 Information Technology – Software Product Evaluation – Quality characteristics and guidelines for their use. International Organization for Standardization, Geneva, 1992.

[22] Jianjun Zhao. Change Impact Analysis for Aspect-Oriented Software Evolution. Proceedings of the 5th International Workshop on Principles of Software Evolution, Orlando, 2002.

[23] Jianjun Zhao. Towards A Metrics Suite for Aspect-Oriented Software. Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), 2002.

[24] Jianjun Zhao and Baowen Xu. Measuring Aspect Cohesion. Proc. International Conference on Fundamental Approaches to Software Engineering, Springer Verlag, 2002.

[25] Clas Wohlin et al. Experimentation in software engineering. Kluwer Academic Publishers, 2000.

[26] Charles Zhang and Hans-Arno. Jacobsen. Quantifying Aspects in Middleware Platforms. Department of Electrical and Computer Engineering and Department of Computer Science, University of Toronto, 2000.

[27] Magnus Mickelsson. Aspect-Oriented Programming compared to Object-Oriented Programming when implementing a distributed, web-based application. Department of Information Technology, Uppsala University, 2002.

[28] Yvonne Coady and Gregor Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. University of British Columbia, 2003.

[29] Shiu Lun Tsang, Siobhan Clarke, Elisa L. A. Baniassad. Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity. http://www.cs.tcd.ie/Elisa.Baniassad/OO-AOMetrics.pdf. Submitted to ECOOP 2004.

[30] L. Briand, J. Daly, and A. Wurst. Unified Framework for Coupling Measurement in Object Oriented Systems. IEEE transactions on Software Engineering, Vol. 25, No. 1, pp 99-121, 1999.

[31] L. Briand, J. Daly, and A. Wurst. A Unified Framework for Cohesion Measurement in Object Oriented Systems. Technical Report, ISERN-97-05, 1997.

[32] Shyam R. Chidamber and Chris F. Kemerer. A Metric Suite for Object Oriented Design. IEEE Transactionas on Software Engineering, Vol.20, No.6, June, pp. 476-49, 1994.

[33] Elaine J. Weyuker. Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, Volume: 14, No. 9, pp. 1357 – 1365, 1988.

[34] J.C. Cherniavsky, C.H. Smith. On Weyuker's Axioms for Software Complexity Measures. IEEE Transactions on Software Engineering, Volume: 17, No. 6, pp. 636-638, June 1991.

[35] N.I. Churcher and M.J. Shepperd. Comments on : ‘a metric suite for object oriented design’. IEEE Transactions on Software Engineering, Volume: 21, No. 3, pp. 263-265, March 1995.

[36] N.I. Churcher and M.J. Shepperd. Towards a conceptual framework for object-oriented software metrics. Technical report, Dept of Applied Computing and Electronics, Bournemouth University, UK, 1995. internal report.

[37] M. Hitz and B. Montazeri. Chidamber and Kemerer's metrics suite : A measurement theory perspective. IEEE Transactions on Software Engineering, 22, 1996.

[38] B. Henderson-Sellers. Software Metrics. Prentice Hall, Hemel Hempstead, UK, 1996.

[39] V. Basili, L. Briand and W. Melo. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, Volume : 22, No. 10, October 1996.

[40] W. Li and S. Henry. Object-oriented metrics that predict maintainability. Journal of systems and software, Volume: 23, No. 2, pp. 111-122, 1993.

[41] W. Li and S. Henry, D. Kafura and R. Schulman. Measuring object-oriented design. Journal of Object Oriented Programming, pp. 48-55, July-August 1995.

[42] Victor R. Basili, Gianluigi Caldiera and H. Dieter Rombach. Goal Question Metric Paradigm. In: "Encyclopedia of Software Engineering", Volume 1, pp. 528-532, edited by John J. Marciniak, John Wiley & Sons, 1994.

[43] T. Gilb. Competitive Engineering: A Handbook for Systems and Software Engineering Management Using Planguage. Pearson Education, 2001.

[44] Young-Jin Lee and Kai H. Chang. Developing Quality Measurement Model for Object-Oriented System. Referred full paper, 39th Annual ACM Southeast Conference, March 2001.

[45] Jana Dospisil. Measuring Code Complexity in Projects Designed with Aspect/J™. Informing Science + IT Education (InSITE) Conference, Finland, June 2003.

[46] Jana Dospisil and Arin Khemngoen. Measuring the Complexity of Mobile Agents Designed with Aspect/J™. Informing Science + IT Education (InSITE) Conference, Finland, June 2003.

[47] J.-Y. Chen and J.F. Lu. A new metric for object-oriented design. Information and Software Technology, Volume 35, No. 4, pp 232-240, April 1993.

[48] F. Brito e Abreu. Object-oriented software design metrics. Proc. of the OOPSLA'92 workshop on OO metrics, 1992.

[49] M. Hitz and B. Montazeri. Measuring product attributes of object-orientes systems. In W. Schfer and P. Botella, editors, Proc. ESEC'95 (5th European Software Engineering Conference), pp 124-136. Springer Verlag, September 1995.

[50] Joe Raymond Abounader and David Alex Lamb. A Data Model for Object-Oriented Design Metrics. External Technical Report, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, October 1997.

[51] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In Proc. METRICS'96, Berlin, Germany, March 1996. IEEE.

[52] D.H. Abbott, T.D. Korson and J.D. McGregor. A proposed design complexity for object-oriented development. Technical report, Clemson University, South Carolina, April 1994.

[53] M. Hitz and B. Montazeri. Measuring coupling in object-oriented systems. Object Currents. Volume 1, No. 4, 1996.

[54] J. Zhao and M. Rinard. System Dependence Graph Construction for Aspect-Oriented Programs. MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.

[55] J. Zhao. Slicing Aspect-Oriented Software" Proc. 10th IEEE International Workshop on Program Comprehension (IWPC'2002), pp.251-260, Paris, France, June 2002.

[56] G. Kovacs, F. Magyar and T. Gyimothy. Static slicing of Java Porgrams. Technical Report TR-96-108, Research Group on Artifical Intelligence, Hungarian Academy of Sciences, December 1996.

[57] M. W. Neil Walkinshaw and Marc Roper. The Java system dependence graph. In Third IEEE International Workshop on Source Code Analysis and Manipulation, page 55, Sept. 2003.

[58] D. Balzarotti and M. Monga. Using Program Slicing to Analyze Aspect Oriented Composition. In Foundations of Aspect-Oriented Languages Workshop, AOSD 2004, Lancashire UK, March 2004.

[59] F. Tip. A survey of program slicing techniques. Journal of programming languages, Volume 3, pp. 121-189, 1995.

[60] J.S. Davis and R.J. LeBlanc. A study of the applicability of complexity measures. IEEE Transactions on Software Engineering, Volume 14 (9), pp. 1366-1371.

[61] M.J. Shepperd and D.C. Ince. The use of metrics in the early detection of design errors. Proceeding of the European Software Engineering Conference'90. 1990.

[62] Arthur H. Watson and Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, September 1996.

[63] L. Briand, P. Devanbu and W. Melo. An Investigation into Coupling Measures for C++. Technical Report ISERN 96-08, IEEE ICSE'97, Boston, USA, May 1997.

[64] J.M. Bieman and B.-K. Kang. Cohesion and Reuse in an Object-Oriented System. In Proc. ACM Symp. Software Reusability (SSR'94), pp 259-262. 1995.

[65] Bindu S. Gupta. A Critique of Cohesion Measures in the Object-Oriented Paradigm. Master of Science Thesis, Department of Computer Science, Michigan Technological University. March 1997.

[66] Verifysoft CMTJava, a Java Complexity Measure tool. http://www.verifysoft.com/en_cmtjava.html (11.05.2204)

[67] Eclipse, an open extensible IDE. http://www.eclipse.org/ (11.05.2004)

[68] JMetric, a Java metric tool. http://www.it.swin.edu.au/projects/jmetric/products/jmetric/ (11.05.2004)

[69] Package Design Principles. http://javacentral.compuware.com/pasta/concepts/packageDesign.html (13.05.2004)

[70] The Open/Closed Principle. http://www.objectmentor.com/resources/articles/ocp.pdf (13.05.2004)

[71] Andrew Hunt and David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Oct 1999.

[72] Bison, a general-purpose parser generator. http://www.gnu.org/software/bison/bison.html (16.05.2004)

[73] JavaCC, a parser generator for use with Java™ applications. https://javacc.dev.java.net/ (16.05.2004)

[74] AspectJ, a seamless aspect-oriented extension to the Java™ programming language. http://www.aspectj.org (16.05.2004)

[75] Heather Williamson. XML: The Complete Reference. Osborne/McGraw-Hill 2001.

[76] XmlWriter, an XML outputter. http://www.osjava.org/xmlwriter/index.html (16.05.2004)

[77] JUnit, a framework to write repeatable tests. http://www.junit.org/ (20.05.2004)

[78] I want my AOP! Part 2. http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html (20.05.2004)

[79] G. Booch. Object Oriented Design with Applications. Benjamin/Cummings, Redwood City, CA. 1991.

[80] Xradar, an open extensible code analysis framework and report tool for Java. http://xradar.sourceforge.net (01.06.2004)

[81] M. Shepperd. A critique of cyclomatic complexity as a software metric. Software Engineering Journal, 30-36, Mar 1988.

[82] Karl J. Ottenstein , Linda M. Ottenstein, The program dependence graph in a software development environment, ACM SIGPLAN Notices, v.19 n.5, p.177-184, May 1984.

[83] Mark Weiser, Program slicing, Proceedings of the 5th international conference on Software engineering, p.439-449, March 09-12, 1981, San Diego, California, US.

[84] God Object anti-pattern. http://perldesignpatterns.com/perldesignpatterns.html#GodObject (02.06.2004)

[85] Raional Rose, a UML designer tool. http://www-306.ibm.com/software/awdtools/developer/rosexde/ (07.06.2004)

[86] WhatIs.com, an encyclopedia of information technology. http://www.whatis.com (07.06.2004)

[87] AJDT AspectJ plug-in for Eclipse. http://www.eclipse.org/ajdt/ (07.06.2004)

Appendix B. Glossary

Advice	Used in order to define the behaviour of the aspects.
ANT	An open-source Java-based build tool from Apache.
Aspect	Design decision that are difficult to address in acutal code
AspectJ	A programming language that is a seamless aspect-oriented extension to the Java programming language
Aspect-Oriented Programming (AOP)	A style of programming that attempts to abstract out features common to many parts of the code beyond simple functional modules and thereby improve the quality of software.
Chidamber and Kemerer (C&K) metrics suite	A suite of measures for object-oriented systems.
Cohesion	A software attribute representing the degree to which the components are functionally connected within a software module.
Complex	Plaited together, intervowen. A whole comprehending in its compass a number of parts, especially of interconnected parts or involved particulars, a complex or complicated whole. Opposite to simple.
Complexity	The quality of being *complex*.
Coupling	The degree to which components depend on one another.
Crosscutting concerns	Properties or are of interest in a system that can not be cleanly encapsulated in a generalized procedure.
Custom Order Server (COS)	Telenor Mobile's middleware platform; one of Norway's largest Java based systems. Designed to give front-end applications a consistent view across multiple backend systems.
Dependency tree	See *program dependence graph*

DIAS (2)	Distributed Intelligent Agent System ver. 2.0 and ver. 2.1 (with aspects). Central issues in this system are inter-agent communication, interoperability, dispatching and disposing of agents.
Document Type Definition (DTD)	A description how an XML document is structured.
Export coupling	Number of classes outside a specific module that depend on a specific module.
Fan-in	A measure of the number of functions that call some other function (say X). A high value for fan-in means that X is thightly coupled to the rest of the design.
Fan-out	A measure of the number of functions which are called by function X. A high value for fan-out suggests that the overall complexity of X may be high due to the control logic needed to coordinate the called components.
Gilb	Introduced a method can be thought of as “design by measurable objectives” where the goal is to quantify all requirements.
Goal-Question-Metrics (GQM)	Aims at tying measurement to the overall goals of projects and process.
Import coupling	Number of dependencies a specific module has to classes outside the module
Inter-type declaration	Declarations that cut across classes and their hierarchies.
ISO 9126	A framework for evaluating software quality
Javacc	A tool for parsing the code by breaking down sentences and statements into acceptable tokens in a given language.
Joinpoint	Location which is affected bu one or more crosscutting concerns. Another way of looking at this is that joinpoints are the locations where we can hook on new actions before or after the original code is executed.
Lack of Cohesion in Methods (LOCM)	A metric referring to cohesion, proposed by Chidamber and Kemerer.
Loose cohesion of class (LCC)	A specific cohesion metric, which is based on the transitive closure of the matrix from TCC.

Metric	Although metric generally refers to the decimal-based metric system of weights and measures, software engineers often use the term as simply "measurement."
Object-oriented programming (OOP)	Programming that supports object technology. It is an evolutionary form of modular programming with more formal rules that allow pieces of software to be reused and interchanged between programs. Major concepts are encapsulation, inheritance and polymorphism.
Open source	Free source code of a program, which is made available to the development community at large.
Plug-in	An auxiliary program that works with a major software package to enhance its capability.
Pointcut	A program element that picks out join points, as as data from the execution context of the join points. Pointcuts are primarily used by advices.
Program dependence graph	The problem of *static slicing* restated in terms of a reachability problem in a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies.
Program slicing	A program slice consists of the parts of the program that affect the values computed at some point of interest, referred to as a slicing criterion.
Quality	A characteristic property that defines the apparent individual nature of something; a construct whereby objects or individuals can be distinguished.
Quantified statements	Statements that have effect many places in the underlying code.
Reenginering	The application of technology and management science to the modification of existing systems, organizations, processes, and products in order to make them more effective, efficient, and responsive.
Regex	See regular expression
Regular expression	A formal description of a language acceptable by a finite automaton or for the behavior of a sequential switching circuit.
Tight cohesion of class (TCC)	A specific cohesion metric.

Tomcat	Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process.
XML	Extensible Markup Language
XmlWriter	An open-source tool that outputs simple XML to a file.
Xradar	An open extensible code analysis framework and report tool for Java, designed to support reengineering tasks. Originally, developed by Telenor Mobile.
XSL	XML stylesheet language.

Appendix C. Other metrics and metrics suites

In this appendix we present metrics and metrics suites we have not found place for in the Theoretical foundation.

A.1. Complexity metrics

Henry and Kafura [12] defined a complexity measure for information flow based on fan-in/fan-out:

$$\text{Complexity} = C \times (\text{Fan-in} \times \text{Fan-out})^2$$

C is any measure of a complexity factor such as LOC or McCabe's cyclomatic complexity.

From a measurement theory perspective, Fenton [5] is concerned about the lack of "complexity" for many modules. He notes that if either fan-in or fan-out is zero, then the multiplication leads to zero "complexity" for the module. This is obviously a flaw in the measure, as all modules have some degree of complexity.

Shepperd [61] identified a number of theoretical problems with Henry and Kafura's measure. Some arise from the informal definition of the model and the notion of indirect flows, which make it difficult to produce the model required to compute the measure. The distinction between local and global flows and the fact that the metric penalized module reuse in the same system are also questionable; each instance of reuse is counted as a separate information flow. Shepperd proposed a number of refinements to Henry and Kafura's model:

- Recursive module calls should be treated as normal calls
- Any variable shared by two or more modules should be treated as a global data structure. Compiler and library modules should be ignored.
- Indirect flows should be counted across only one hierarchical level. Indirect flows should be ignored, unless the same variable is both imported and exported by the controlling module.
- No attempt should be made to include dynamic analysis of module calls.
- Duplicate flows should be ignored.
- Module length should be disregarded, as it is a separate attribute.

The Shepperd refinement to Henry and Kafura's information flow measure:

$$\text{Complexity} = (\text{Fan-in} \times \text{Fan-out})^2$$

Shepperd's refinements attempt to capture a specific view of information flow structure, namely development time, and are thus consistent with measurement theory.

McCabe's cyclomatic complexity measures the amount of decision logic in a single software module [13]. The metric measures the number of independent paths through a program, thereby placing a numerical value on the code's complexity. In practice it is a count of the number of test conditions in a program. The cyclomatic complexity (CC) of a graph (G) may be computed according to the following formula [14]:

$$CC(G) = \text{Number (edges)} - \text{Number (nodes)} + 1$$

Cyclomatic complexity has been criticized by Shepperd [81]. He claims that "it is based upon poor theoretical foundation," and as such should not be used as a predictor for reliability and development effort. The cyclomatic complexity is also strongly correlated with lines code size.

Chen and Lu [47] presents a new set of metrics for OO design, mostly concentrating on complexity measures, as seen in Table 41.

Metric	**Description**
Operation complexity (OpCom)	$\sum O(i)$, where O(i) is operation i's complexity value, and is evaluated from a table ranging from 'null' to 'extra high'.
Operation argument complexity (OAC)	$\sum P(i)$, where P(i) is the value of each argument i in each operation in the class, and is evaluated from a table giving values to different operators (from Boolean or Integer to File).
Attribute complexity (AC)	$\sum R(i)$, where R(i) is the value of each attribute in the class, and is evaluated from the same table as OAC.
Reuse (Re)	Measures whether a class is a reused one, either from the current or a previous project.

Table 41 Chen and Lu's OO metrics

Henderson-Sellers [38] argues that "metrics with subjective weighting in which not only are Likert scales used, but the mapping from that scale to a numerical scale is itself fuzzy, have no scientific validity, and should be avoided if at all possible.

A.2. Metrics suites

A collection of metrics suites are given in the following subsections.

A.1.1 Li and Henry's metrics suite

Li and Henry [40] [41] present ten metrics in their system. They include five of the C & K metrics, namely WMC, DIT, NOC, RFC and LCOM. In addition, they define five metrics of their own, as shown in Table 42.

Metric	**Description**
Message-Passing Coupling (MPC)	MPC measures the complexity of message passing among objects. MPC is the (static) number of send statements defined in a class, where a send statement is a message sent out from a method in a class to a method in another class.
Data Abstraction Coupling (DAC)	A class can be viewed as an implementation of an abstract data type (ADT). A variable declared within a class may have a type of ADT which is another class definition. DAC for a class is its number of instances of ADTs, or the number of its variables having an ADT type.
Number of Methods (NOM)	The number of local methods.
Number of semicolons (SIZE1)	A LOC traditional metric.
Number of properties (SIZE2)	The number of attributes plus the number of local methods.

Table 42 Li and Henry's proposed metrics.

Li and Henry claim that "there is a strong relationship between [their] metrics and maintenance effort in object-oriented systems", and that "maintenance effort can be predicted from combinations of metrics collected from source code". The metrics were, however, not validated in a statistically sound manner, so it is impossible to conclude any correlation between measurement results and maintenance effort [31].

Besides the critique that the C & K metrics receive, Li and Henry's own metrics was also criticised. In SIZE1 the number of semicolons is used, which is language-dependent and not derivable until the source code is available. Hitz and Montazeri[49] argue that trying to minimize DIT, in order to decrease complexity, leads to the guideline "do not use inheritance at all", while inheritance is one of the major advantages of the OO paradigm.

A.1.2 MOOD metrics

Britoe e Abreu [48] [51] give a set of six metrics known as the MOOD (Metrics for Object Oriented Design) metrics. The MOOD metrics are seen in Table 43.

Metric	Description
Method Hiding Factor (MHF)	The invisibility of a method is the percentage of the total classes from which this method is not visible.
Attribute Hiding Factor (AHF)	The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible.
Method Inheritance Factor (MIF)	MIF is the sum of inherited methods divided by the total number of available methods (locally defined plus inherited).
Attribute Inheritance Factor (AIF)	AIF is the sum of inherited attributes divided by the total number of available attributes (locally defined plus inherited).
Polymorphism Factor (POF)	POF is the actual number of possible different polymorphic situations divided by the maximum number of possible distinct polymorphic situations.
Coupling Factor (COF)	COF is the actual number of coupling not imputable to inheritance divided by the maximum possible number of coupling in the system.

Table 43 Brito e Abreu's MOOD metrics.

The metrics were applied to eight projects representing eight variations of the design for the same requirements document [51]. A correlation is established between the metrics and defect density, failure density, and normalized rework. Most of the metrics were good predictors of these three quality measures.

A.1.3 Interaction metrics

Abbott, Korson and McGregor [52] propose metrics for measuring the number and strength of the object "interactions permitted" by an object oriented design. Classes in OO designs exhibit various levels of complexity along two dimensions, as seen in Table 44.

Metric	Description
Interaction level (IL)	The degree to which an object is likely to interact with other objects by providing opportunities for such interactions. IL = K1 * (value based on number of interactions) + K2 * (value based on strength of interactions); where K1 and K2 are tentatively set to 1.0 each.
Interface size (IS)	The degree to which classes provide means for information flow in and out of their encapsulation. IS = K3 * (value based on number of interface items) + K4 * (value based on size of interface items); K3 and K4 are tentatively set to 1/8 and ¼ respectively.

Table 44 Abbott, Korson and McGregor's proposed OO metrics.

Interface size measures the surface complexity of a class, while the interaction level measures the opportunities for interaction between its surface and its interior.

Appendix D. Example code

```
import figures.primitives.planar.Point;
import figures.primitives.solid.SolidPoint;

class Main {

    private static Point startPoint;

    public static void main(String[] args) {
        try {
            System.out.println("> starting...");
            Point p = new Point(0,0);
            startPoint = makeStartPoint();
        } catch (RuntimeException re) {
            re.printStackTrace();
        }
        System.out.println("> finished.");
    }

    public static Point makeStartPoint() {
         return null;
    }

    static class TestGUI extends javax.swing.JFrame {
        TestGUI() {
           this.disable();
        }
    }
}

privileged aspect Test {
    pointcut testptct(): call(* *.*(..));

    before(Point p, int newval): target(p) && set(int Point.xx) &&
args(newval) {
        System.err.println("> new value of x is: " + p.x + ", setting to: "
+ newval);
    }

      before(int newValue): set(int Point.*) && args(newValue) {
          if (newValue < 0) {
              throw new IllegalArgumentException("too small");
          }
      }
}
```

Appendix E. DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT AspectMetrics (Metrics-descriptions?, build-configuration-file)>
<!ELEMENT Metrics-descriptions (metric-description*)?>
<!ELEMENT metric-description (id, description, source-kind+)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT source-kind (#PCDATA)>
<!ELEMENT build-configuration-file (statistics, package*, class*, aspect*)>
<!ATTLIST build-configuration-file
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT package (statistics, package*, class*, aspect*)>
<!ATTLIST package
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT class (statistics, class*, aspect*, constructor*, method*)>
<!ATTLIST class
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT aspect (statistics, class*, aspect*, method*, advice*)>
<!ATTLIST aspect
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT method (statistics, class*)>
<!ATTLIST method
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT advice (statistics, class*)>
<!ATTLIST advice
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT constructor (statistics, class*)>
<!ATTLIST constructor
      name CDATA #REQUIRED
      short-name CDATA #REQUIRED
>
<!ELEMENT statistics (metric*)>
<!ELEMENT metric EMPTY>
```

```
<!ATTLIST metric
      id CDATA #REQUIRED
      value CDATA #IMPLIED
      mean CDATA #IMPLIED
      sum CDATA #IMPLIED
      min CDATA #IMPLIED
      max CDATA #IMPLIED
      stddev CDATA #IMPLIED
>
```

Appendix F. Test plan and execution of the metrics tool

This appendix gives a short introduction to our testing of the metrics tool.

There is two major parts in the test plan: white box and black box testing. White box or structural testing is done with the help JUnit framework which is incorporated in the Eclipse IDE. Black box testing or functional testing is done on DIAS 2.1 source code, where we have manually handcounted some metrics to see if they are consistent with the results from the tool.

A.3. White box testing

We have implemented 24 JUnit test classes which are placed in separate tree from original source code, but in the same packages as the original source. These tests are mainly used for debugging purposes, especially for complex functions. They are implemented in paralell with the original source. Along with the tests, we have also included some test data (classes) that are originally from the AspectJ source and made especially for testing the AspectJ compiler. This makes them suitable for testing that we treat the tokens in the language correctly.

Test name	**Description**
no.ntnu.aspectmetrics	
AspectMetricsTest	This is a setup class for most tests where test data are chosen and the AspectJ compiler is run. (Most test classes inherit from this class which is abstract)
EmptyTest	Just runs the super class AspectMetricsTest.
TaskTest	A simple ANT task test.
no.ntnu.aspectmetrics.calculators	
AdviceCalculatorsTest	Runs test on the advice-out and advice-in calculators.

CalculatorManagerTest	Runs the calculator manager and prints the results to some file.
CohesionTest	Runs tests on the two cohesion calculators.
CouplingInsidePackageTest	Runs tests on the two “coupling inside package” calculators.
CouplingOutsidePackageTest	Runs tests on the two “coupling outside package” calculators.
FanTest	Runs tests on the fan-in and fan-out calculators.
GetCalculatorNames	A test that retrieves all the calculator’s names.
StatementsTest	A test for the calculation of statements.
TestStringForSourceCalculator	A test to see if the removal of comments works for some particular code lines.
no.ntnu.aspectmetrics.reporters	
RunAllTest	A test that runs calculations and a reporter.
SimpleReporterTest	Similar to RunAllTest, but with implemented in a different way.
no.ntnu.aspectmetrics.sources	
SourceManagerTest	A test to make sure all the right sources is made.
TestAdviceInformation	A test that retrieves information about advices and their relations.
no.ntnu.aspectmetrics.treebuilder	
BuilderTestCase	Some simple building tests for the AspectJ compiler, like succession of build and that the model is created correctly.
PrintTree	Prints the code tree for the test data chosen in AspectMetricsTest.

TestAsmManager	This class tests the AsmManager which holds the code tree and offer some different ways of accessing it.
TestHitOnAll	A test to see if the adding of aspect HitOnAll is done correctly.
TestMakeListFile	A test to see if the build configuration file (list of source files) is made correctly.
TestModel	Finds certain nodes in the code tree through search and prints information about them.
TestRelations	A test that checks aspect relations (advices and introductions).

Table 45 Test classes

A.4. Black box testing

We found at least five sources for each of the element types: method, advice, class, aspect, file and package. The sources are chosen by random, but some specific sources are added because they are borderline cases. We also controlled the system values. On these sources we manually counted those metrics that were relevant and compared the the manual results with those of from the tool. Table 45 shows how the calculations are made on different source types and the tests are carried out in consistence with this table.

	Method	**Advice**	**Class**	**Aspect**	**File**	**Package**	**System**
Advice-in	0	0	calculation	calculation	propagation	propagation	propagation
Advice-out	0	calculation	propagation/0	propagation	propagation	propagation	propagation
Advices	0	0	propagation/0	calculation	propagation	calculation	propagation
Aspects	0	0	0	0	0	calculation	propagation
Classes	0	0	0	0	0	calculation	propagation
Export-coupling-inside-package	0	0	calculation	calculation	propagation	propagation	propagation
Export-coupling-outside-package	0	0	calculation	calculation	propagation	propagation	propagation
Fan-in	0	0	calculation	calculation	propagation	propagation	propagation
Fan-out	0	0	calculation	calculation	propagation	propagation	propagation

Import-coupling-inside-package	0	0	calculation	calculation	propagation	propagation	propagation
Import-coupling-outside-package	0	0	calculation	calculation	propagation	propagation	propagation
Inner-aspects	calculation	calculation	calculation	calculation	propagation	propagation	propagation
Inner-classes	calculation	calculation	calculation	calculation	propagation	propagation	propagation
Interfaces	0	0	0	0	0	calculation	propagation
Loose-cohesion-of-class	0	0	calculation	calculation	propagation	propagation	propagation
Methods	0	0	calculation	calculation	propagation	propagation	propagation
NCSS-in-methods	calculation	calculation	propagation	propagation	propagation	propagation	propagation
Packages	0	0	0	0	0	0	calculation
Statements-in-methods	calculation	calculation	propagation	propagation	propagation	propagation	propagation
Tight-cohesion-of-class	0	0	calculation	calculation	propagation	propagation	propagation

Table 46 calculations overview