Universiteit
Antwerpen

Measuring Aspect-Oriented Software In Practice

Hakan Özler

Principal Advisor: Dirk Janssens

Assistant Advisor: Tim Molderez

Dissertation Submitted in June to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Ansymo
Antwerp Systems and Software Modelling

Acknowledgements

I have thought that the acknowledgements are a very valuable part to begin the thesis text, since all the expressed thoughts must reach to the right people before delving into it. You will only see me while reading the whole thesis, whereas there are a few invaluable people who deserve certainly my thanks –without them I would never get the fruits of my work.

First of all, many thanks to my supervisor, *Tim Molderez* for his continuous guidance, support and encouragement throughout the year that make this work possible. Also, my thanks go to my promoter, *Prof. Dr. Dirk Janssens* for initiating this work in order to be enlightened some missing parts of this area. I would like to thank *my family* for their continuous support and encouragement. Although they are far from me, I felt them next to me while working on it. A special word of thanks goes to *my girlfriend* for her continuous tolerance, encouragement and support.

Finally, I would like to dedicate this thesis to *my mother* and end up with a quote of *Hz. Mevlâna (Rumi)* that summarizes my one-year effort:

"Patience is the key to joy"

Abstract

Aspect-oriented programming (AOP) is a way of modularizing a software system by means of new kind of modules called *aspects* in software development. To this end AOP helps in alleviating crosscutting concerns of system modules by separating into several aspect modules, thereby aiming to improve separation of concerns. On the other hand, aspects can bring unexpected behaviour to a system while attempting to alter the system's concerns. They can modify the behaviour of the base system without warning. Following to this, such impact can limit to achieve modular reasoning in an aspect-oriented system properly.

Obtaining the valuable data, we try to get an idea of how difficult it is to achieve modular reasoning. In this thesis, we analyse the existing ten aspect-oriented systems by answering six research questions. These six questions were derived from our general question: *"how AspectJ is used in practice?"*. In order to answer each one of them, we have implemented a metrics suite including both aspect-oriented and object-oriented features using *Ekeko*. Next to modular reasoning, we also acquire other usefulness about AOP constructs and coupling between classes and aspects. These results can then be used to influence the design of existing or new AOP languages, or to improve existing analysis tools.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Aspect-oriented programming (AOP) [28] aims to improve separation of concerns by extending existing languages with a new kind of module called *aspects*. For example, the AspectJ language [27] is an aspect-oriented extension of the Java language. Aspects are allowed to alter the behaviour of the system's members as well as influence the system's control flow. While aspects aim to improve separation of concerns, and thus aim to make a system better maintainable, adaptable , and readable, they can also inadvertently modify the system in unexpected ways. Since aspects can permeate an entire project, we might never know how we harm the manner of the project itself because aspects can modify the behaviour of multiple existing modules in the project such that it might be difficult to determine whether these changes are harmful or not. In addition, we integrate aspects into the project to overcome the design problems that object-oriented programming (OOP) techniques cause. The side effects of aspects can also vary depending on the requirements of a system.

Another related difficulty is that even AspectJ developers might not realize as to what they attempt when completing an aspect-oriented project. For example, we need to change the behaviour of a system by introducing new members within existing modules, accessing many different modules, and changing some members' flow by means of aspects. Therefore, such impacts may affect the aspects to cause unexpected behaviour.

This paper aims to study how AOP is used in practice. One of the main motivations is related to the ability to enable modular reasoning in aspect-oriented languages. In AOP, modular reasoning aims to help in understanding a module, whose behaviour need to be considered clearly. In other words, a module must be aware of related aspects that interfere with the module's behaviour. To make such exposed modules aware of aspects, aspects should adhere to principles like the Advice Substitution Principle (ASP) [32] revised using design by contract (DbC) perspective for the notion of aspects. For example, if a module is affected by aspects, but it is unaware of those aspects, those aspects should not interfere with the module's behaviour in an unexpected way. We thus need to examine existing aspect-oriented projects to try to get an idea of how difficult it is to achieve modular reasoning, which leads to several questions such as *how many aspects initiate to bypass the behaviour of members?*, *how often are advanced AOP features used?*, and *how often are basic AOP features used?*. Next to modular reasoning, other motivations for the work in this paper include several ideas, which might help in enhancing existing implementations of AOP (e.g. AspectJ), designing a new aspect-oriented language as well as improving well-established analysis tools of AOP for better performance. In our case, we are interested in the AspectJ language as it is a general-purpose and well-known aspect-oriented language. Furthermore, AspectJ has established itself over the past years. Next to the AspectJ language, we need to manifest all the motivations above by answering particular research questions upon existing AspectJ projects including small and big applications. We make use of Ekeko [20], which is a meta programming library and facilitates the implementation of metrics as queries on AspectJ source code. That is, in practice, we can convert our questions into valuable metrics using Ekeko in order to combine specific features about an aspect-oriented project. Another point worth mentioning is that analysis results of such queries can lead to bring notable characteristics of AOP implementations such as observing less or common used AspectJ features.

In this paper, we have created meaningful queries depending on our research questions such as *what percentage of a system is exposed by aspect-oriented?*, *how many members are exposed by aspect-oriented?*, *how often are aspect-oriented features used?* and so on. To answer such questions, we examine how AOP features are used upon existing AspectJ projects and how related principles are straightforwardly achieved. As consequence, we reveal our assumptions and results according to the projects in order to verify how AOP is being stated and whether or not AspectJ developers did really fulfil our points of view.

In summary, the paper is organized as follows:

1. The following subsections will introduce the background information of aspect-oriented programming, the AspectJ language, and the Ekeko plug-in in more detail, respectively.

2. We present several research questions, which help to answer the general question of how AOP is being used in practice, a running example and selected systems.

3. We present the corresponding 41 metrics that aim to answer each of these research questions.

4. We will then discuss the outcomes of the ten AspectJ projects measured with regard to our metrics.

5. Finally, we conclude the paper and suggest potential future work.

1.1 Introducing Aspect-Oriented Programming

Object-oriented software systems are expressed as functional modules or classes. In other words, the functional modules combine more than one service to manage their behaviour in order to form the final system. We must merely keep in mind that if a module is grown that means the job of the module is going to be more complex. Therefore, the main problem of the software systems is complexity. In order to reduce it, one essential way that every developer must do is to break it into operable pieces.

Object-oriented programming (OOP) thus facilitates to breaking complex systems into small modules by means of a *class*. In OOP, a class addresses several features for the software systems. One of the features that is encapsulation provides each module to combine its own functionalities. The functionalities of each module are called as *concerns*. Concerns can be split between *core concerns* and *crosscutting concerns*. Core concerns refer to the main functionalities of an application such as indexing in a library system or customer management in a banking system. Crosscutting concerns indicates the others that do not match with the core concerns of the applications such as logging, monitoring, caching, and security. These concerns cut across many modules and affect adversely the quality of code reuse, code maintenance, and evaluation of code. Accordingly, software designs and developments are being impacted by crosscutting concerns.

Since in OOP we implement both core concerns and crosscutting concerns in one module , the module deliberately initiates two software design problems which are as follows - *code tangling* and *code scattering*. These design issues can spread over the modules according to the amount of implemented crosscutting concerns in a system. Most software systems submit themselves to code tangling as they have multiple concerns in order to finalize their process.

Code tangling refers to the fact that, the code of a core concern and one (or more) crosscutting concern(s) can be tangled. That is, multiple concerns can be implemented in the same module, which can make it more difficult to read and understand. Code scattering is reflected in the fact that the implementation of a concern can be scattered across multiple modules. Moreover, the affected modules can be difficult to refactor according to systems' size. An example of both code tangling and code scattering is shown in Fig. 1.1. The orange lines representing a crosscutting concern scatter over the three modules in the left side of the figure. When we take a look at the module b, four different colors (yellow, red, orange, and turquoise) depicting four unique crosscutting concerns tangle the dark grey colors referring to the core concerns. Consider the first and last dark gray lines work with these colors and all the lines indicate a complete design of the module b.

In order for software systems to avoid the occurrence of the scattering and tangling problems, we must modularize each crosscutting concern from the related modules, plus the separation of concerns (SoC) principle should be used for them in order to facilitate to maintain a system. All in all, aspect-oriented programming (AOP) aims to enable the modularization of crosscutting concerns by means of a new module called an *aspect*. Aspects have been designed to organize each crosscutting concern separately in order to develop a system that is easier to design. AOP improves encapsulation by using aspects, as each crosscutting concern is now implemented as a crosscutting module in a system. More details about aspects will be introduced in Sec. 1.2.5.

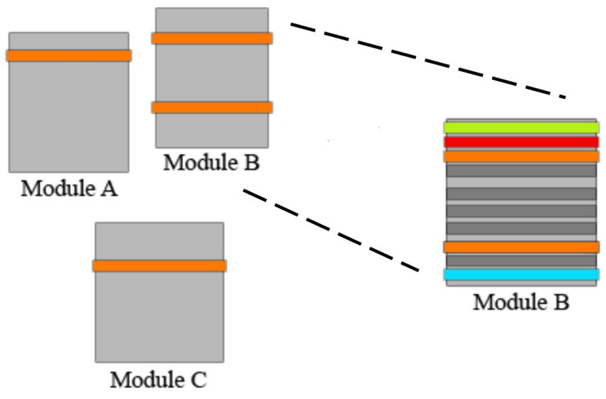


Figure 1.1: Showing code scattering and code tangling

Another point worth mentioning is that the complexity of software systems might be diminished using AOP as the technique promotes developers who are willing to use the benefits of the separation of crosscutting concerns in their software systems, to revise their implementations in order to relieve their systems from the aforementioned problems that the common implementation techniques of OOP cause. AOP specifically reinforces OOP that has difficulty in some points of a system during implementation. Another example shown in Fig. 1.2 depicts how AOP tries to help in solving scattered and tangled concerns. In the figure, each module is now unaware of the required crosscutting concerns mentioned four different colors. What is important to note is that the absence of information about these crosscutting concerns is denoted as the obliviousness of AOP. In addition, the total size of the modules can vary after using AOP. However, such crosscutting concerns implemented in separate aspect modules are marked as shadows on the related core concerns of each module since each module still unites with them so as to complete its operation.

Although we discussed the usefulness of the technique, there also are some drawbacks. Comprehending AOP concept needs more practice. That is, we must first apprehend the notion of AOP concept upon various aspect-oriented applications.

AOP itself is designed as an extension to a base language such as Java, and C++ in order to implement its core mechanisms. For example, Java developers create their systems using Java as base language that is important for AOP technique as well to adapt the keywords of an aspect-oriented language. For this reason, the various implementations of AOP are released deliberately and the community of aspect-oriented programming reaches in strength. Consequently, software systems are going to be more interac-

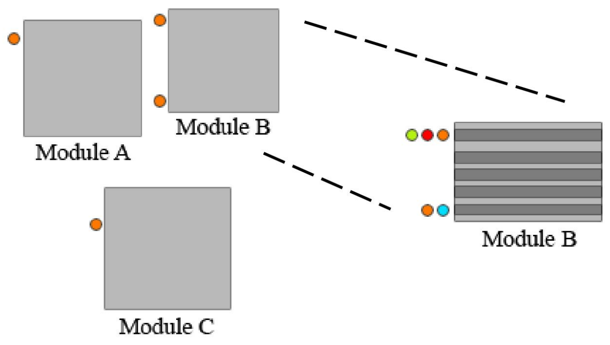


Figure 1.2: Solving the problems using AOP techniques

tive by combining OOP and AOP collectively. In the next section, we will illustrate the core concepts of AOP by means of the AspectJ language.

1.2 AspectJ

AOP provides various constructs to enable the modularization of crosscutting concerns. AspectJ is an aspect-oriented extension to the Java programming language as the base language. AspectJ supports all the benefits of the Java language so that Java developers can easily adapt and only deal with the new keywords of the AspectJ language. As long as we use the extending language, our modules will be shaped as more collaborative structures. Although AspectJ is a well-known language and is utilised in our analysed projects, there is also a variety of implementations of AOP such as PostSharp [11] developed for .NET systems, AspectC++ [4] extended C++, AspectJS [5] developed for JavaScript, JBoss [7] AOP extended Java, Aquarium [3] developed for Ruby.

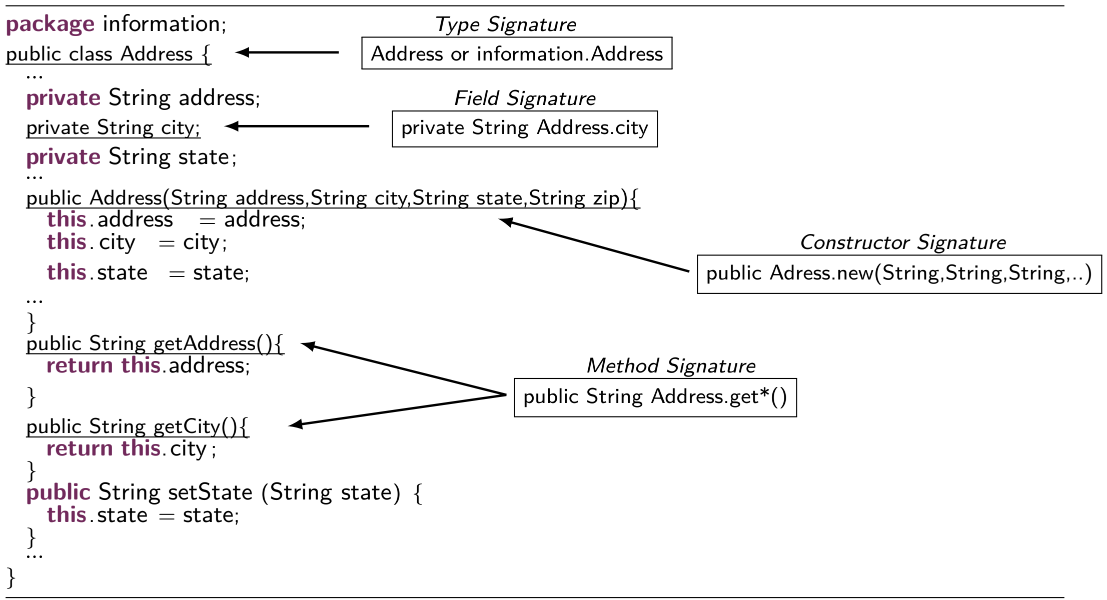
After obtaining the main information about the language, we must recall the concept of AOP. Using AspectJ brings with essential constructs such as *join point*, *pointcut*, *advice*, *intertype declaration*, and *aspect*.

1.2.1 Join Points

A *join point* is a core concept used by aspect-oriented languages and it is the starting point of the AspectJ language as well. Without any join point we cannot refer to points where we want to connect our crosscutting concerns. A join point is a particular identifiable point in a program's control flow. Calling a method, executing a constructor, and reading a field are all identified as join points. For example, each method declared in a system is a potential place for crosscutting concerns. The same behaviour is probable for each method-like (constructor) declaration. A call to a method/constructor and an execution of a method/constructor are the most commonly used join points. In addition, a field representing a particular property in a system has two join points namely read and write access. A join point possesses runtime information according to program elements such as the arguments of a specified method. Fig. 1.3 shows an elaborated example depicting the most common join points.

In the figure, the program elements highlighted in blocks have their own code signatures in order to be identified the join points at runtime. What is important to note about join points is that selecting an exact join point depends on its information such as type, name, arguments, and modifiers. Signature patterns are used to describe a set of desired join points. Type signature is the one of them and denotes *classes*, *primitive types*, *interfaces*, and *aspects*. Although a specific type signature can be specified, it also is possible to use wildcards. Using three wildcards (*,+,..) supported by AspectJ we can decide to reach several join points or omit some redundant types.

Star (*) indicates any number of characters. In a type signature, it refers to a part of the type or package name and in other signatures such as method and field, it denotes a part of the name. For example, in Fig. 1.3, the `Address` class is specified as the `Address` type. Consider we have another class called `AddressBook` implemented in the same package with the `Address` class and replace the current type signature with the `Address*` type, the new type would denote a type with a name starting with Address and thus we select both the `Address` and `AddressBook` classes as our two join points. Instead of obtaining one method with method signature, star provides widely selections of elements in modules. The method signature illustrated in the figure denotes any public method in the `Address` class with the name starting with *get* that returns *String* type and takes no arguments. In this way, we regard the two method declarations namely `getAddress` and `getCity` as they comport with the explained signature.



```
package information;
public class Address {
  ...
  private String address;
  private String city;
  private String state;
  ...
  public Address(String address,String city,String state,String zip){
    this.address    = address;
    this.city   = city;
    this.state    = state;
  ...
  }
  public String getAddress(){
    return this.address;
  }
  public String getCity(){
    return this.city;
  }
  public String setState (String state) {
    this.state = state;
  }
  ...
}
```

Type Signature
Address or information.Address
Field Signature
private String Address.city
Constructor Signature
public Adress.new(String,String,String,..)
Method Signature
public String Address.get*()

Figure 1.3: Connection between program elements and signatures

Two dots (..) has different meanings between a type and method signature are common signature patterns. In a type signature it denotes any number of characters that possibly collect any subpackage or all inner types. For example, assume we set our type signature as `information..*`, so that the new signature can simply scans all join points where any types inside of the `information` package or any related subpackages starting with "`information.`" exist. Provided that we had other packages including various types such as `information.data.Connection`, `information.customer.CustomerService`, we would count them as well. About the method signature, the wildcard ".." refers to an arbitrary number of arguments. However, in the `Address` class, the constructor signature differs from the method signature as constructors do not have a return type, and a specific name. In this case, by default constructor of a class seems with no arguments like `Classname.new()`. In our example, the constructor signature is expressed `public Adress.new( String, String, String,..)` and makes use of two dots in order to omit the rest of the arguments.

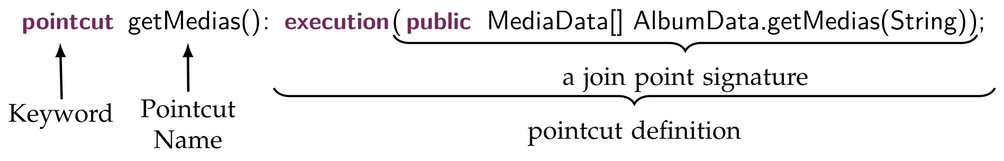
The last wildcard called plus (+) indicates any subtype of a given type. For example, if we have the aforementioned class - `AddressBook` inherited by the `Address` class. In other words, if the `AddressBook` is subclass of the `Address` class, as well as our type signature is `Address+`, we would collect both of them for our crosscutting actions.

Finally, a concept closely related to join points is *join point shadow* (JPS). A join point shadow itself is a projection of a join point in the source code. In Fig. 1.3, the arrows of both the method and constructor signatures link to the representative shadows. Additionally, AspectJ developers can comprehend those kinds of join point shadows by means of AJDT [1] which describes a visual representation of where crosscutting concerns are executed in detail. For example, if we return Fig. 1.2 in order to understand more precisely what the points are, each colored point refers to its join point shadow showing more specific information pertaining to its crosscutting action. Note that all crosscutting actions are not included in the modules in this way the characteristic of the modules obtains another behaviour which is oblivious.

After specifying the join points by means of signature patterns, we have to select each of them in order to give functionality. In the following section, we will examine pointcuts helping to select marked program elements.

1.2.2 Pointcuts

A *pointcut* specifies a set of join points. A pointcut itself can be *named* or *anonymous*. Named pointcuts are the most commonly used and preferred pointcuts. Fig. 1.4 shows an example of a named pointcut taken from the MobileMedia project.



```
pointcut getMedias(): execution(public MediaData[] AlbumData.getMedias(String));
```

Figure 1.4: The definition of a named pointcut

The named pointcut shown in the figure collects a specific join point. The pointcut named `getMedias()` collects the execution of the `public` method named `getMedias` with one argument in the `AlbumData` class and returned an array list. Pointcut consists of three main elements namely the pointcut keyword, a name, and a pointcut definition using a pointcut type and a join point signature. Pointcut types also known as kinded pointcuts consist of various categories such as *method-call*, *method-execution*, *field-get*, *field-set*, and so on. The difference between named and anonymous pointcuts is that we can specify an anonymous pointcut directly within an advice declaration and we cannot use it with other advice declarations. The method signature is shown as an anonymous pointcut in Fig. 1.5.

anonymous pointcut definition

```
after () : execution( public MediaData[] AlbumData.getMedias(String)) {
    // advice body
}
```

Figure 1.5: The definition of an anonymous pointcut

Although we use signature patterns in order to map each join point in a pointcut, we need an additional element to determine the pointcut definition. Pointcut types therefore contribute to any signature to finalize the behaviour of pointcuts. Each pointcut type works with a specific signature. For example, the *execution* pointcut shown in Fig. 1.5 only matches with both method and constructor signatures. The *call* pointcut as well matches with both method and constructor signatures. Additionally, the *get* and *set* pointcut work with field signature.

However, pointcut types do not only consist of the execution, call, get and set, there are a few other types as well such as *handler()*, *adviceexecution()*, *withincode()*, *within()*, *cflow()*, *cflowbelow()*, *this()*, *target()*, *args()*, and *if()*. These pointcut types facilitate to reach exact join points forming the first step of building separate crosscutting concerns.

A pointcut is not sufficient alone for implementing a crosscutting action. We need another construct called an *advice* in order to perform the crosscutting action. We will delve into the advice constructors to learn their purpose in the next section.

1.2.3 Advice

An *advice* is a body of code associated with a pointcut declared as a(n) named and/or anonymous to construct the *pointcut-advice* mechanism. Advice, is a method-like declaration, has an advice body which we need to implement crosscutting actions at the join points collected by a pointcut to affect the behaviour of a system. Additionally, each advice may throw exceptions just like a method using *throws* clause. We can use `this` keyword inside of an advice body in order to refer to its aspect instance. What is important to note about advice is that they can take arguments in terms of an exposed join point in order to facilitate to implement behaviour. Moreover, we can make use of a few special keywords within an advice to reach more details about an advised pointcut: *thisJoinPoint*, *thisJoinPointStaticPart*.

The first variable is *thisJoinPointStaticPart*, which contains the join point's static information, such as its source location, signature, name, and kind. The *thisJoinPoint* object also contains the same information just like the static part. Additionally, it collects the target object and this object and arguments at the join point. These two special variables provide more details for specific crosscutting concerns such as logging and tracing.

What is important to mention as well is that we can specify an exact time for an advice such that it performs crosscutting actions at exposed join points. Therefore, AspectJ itself classifies advice into 5 kinds which provide more flexible manner to build crosscutting concerns namely *before advice*, *after advice*, *after returning advice*, *after throwing advice*, and *around advice*. Before advice is simply triggered prior to the execution of a selected join point. An example of before advice taken from the AJHSQLDB project is shown in Fig. 1.6

anonymous pointcut definition

```
before() : execution(* javax ..*(..)) {
    System.out. println ("Executing: "
        + thisJoinPointStaticPart .getSignature()
        + "\n\t" + Thread.currentThread());
}
```

Advice body

Figure 1.6: Showing an example of before advice

In the figure, the before advice with no arguments connecting to the anonymous pointcut denoting the execution of any method in the `javax` package or any related subpackages regardless of return type and argument size runs prior to the execution of the advised join points and adds a new behaviour at each point. The next advice as provided is *after advice* which runs following the execution of a join point regardless of a return value. After advice works as the finally block because it is executed even if an exception occurs.

AspectJ also divides after advice into two separate advice to obtain more expressiveness for developers' tasks. After throwing runs only when an advised join point throws an exception. An *after throwing advice* snippet taken from the AJHotDraw project is shown as follows:

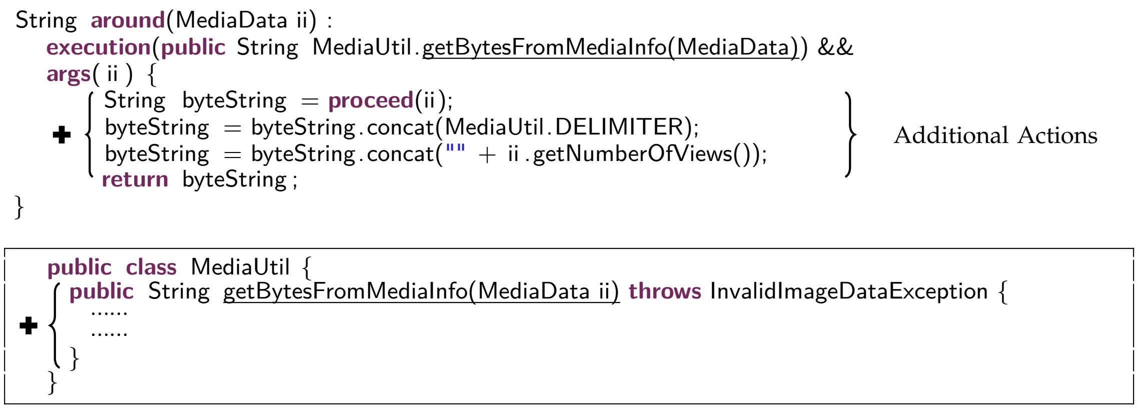
```
after (String dataStr) throwing(Exception e) throws InvalidDateException :
    stringToDataHandler() &&
     args(dataStr ,..){
   throw new InvalidDateException(dataStr);
}
```

The after throwing advice defined with a named pointcut called `stringTo DataHandler` and an args pointcut construct takes one argument and catches `InvalidDate Exception` using a throws clause. The args pointcut construct reaches the parameters of the advised join point. In this case, the after throwing advice takes into account only the first declared parameter type from the args pointcut construct in order to use it in the body of the advice. In addition, we use the && operator to select all the declared pointcuts, for example, combining the `stringToDataHandler()` and the `args()` pointcut construct. The advice captures an exception thrown by the advised point by means of throwing keyword. We can however need another advice not throwing any exception and therefore *after returning advice* executes following the execution of an advised join point completes without any exception. Note that after returning advice executes unless an advised join point throws an exception. Additionally, after returning advice reaches the return type of an advised join point. The source code of an after returning advice taken from the MobileMedia project is shown as follows:

```
after () returning (String context) :
    withincode(* util . Library . readFile (..)) &&
     call (String util . Library . internalSetContext (..)) {
  contextSet.put(String.valueOf(Thread.currentThread().getId ()), context);
}
```

The code above depicts that the after returning advice matching with any `internalSetContext` method returning String type in the `util.Library` class regardless of the number of arguments to the method within any `readFile` method in the `util.Library` class regardless of the return type and number of arguments to the method accesses the returned value by means of *returning()* keyword.

The last kind of advice is *around advice* surrounds an advised join points to perform additional behaviour before and/or after the advised join point, or it can entirely bypass the behaviour of the advised join point. Hence around advice can be used to alter the actual behaviours of join points. While around advice surrounds a join point, we can also say that it executes instead of the join point. In addition, in order to pursue the features of around advice, a special keyword called `proceed()` is provided in the body of around advice. Using the *proceed* keyword we can trigger to run the advised join point. On the other hand, we can avoid running the join point by not making a proceed call at al. Fig. 1.7 illustrates an around advice (taken from the MobileMedia project) introducing new actions after its join point executes.



```
String around(MediaData ii) :
   execution(public String MediaUtil.getBytesFromMediaInfo(MediaData)) &&
   args( ii ) {
      String byteString = proceed(ii);
   +  byteString = byteString.concat(MediaUtil.DELIMITER);        Additional Actions
      byteString = byteString.concat("" + ii.getNumberOfViews());
      return byteString;
}
```

```
   public class MediaUtil {
     public String getBytesFromMediaInfo(MediaData ii) throws InvalidImageDataException {
        ......
+       ......
     }
   }
```

Figure 1.7: Relationships between an around advice and its join point

In the figure, the around advice is simply substituted for the exposed join point. In other words, the around advice is executed instead of the actual join point. The figure depicts that the advice body of the around advice refers to the `getBytesFromMediaInfo` method in the `MediaUtil` class enclosed with the block. As long as the proceed call is used within the body, the behaviour of the selected method is reached in order to declare other functionality. Moreover, the additional actions in the around body refers to an after advice manner such that after the execution of the proceed call (i.e. join point) these actions run respectively.

While we can select join points through pointcuts and alter the behaviour of the selected points in terms of advice, we obviously do affect the dynamic behaviour of a system. On the other hand, we may need to affect the static structure of the system by using intertype declarations.

1.2.4 Intertype Declarations

In order to change the static structure of a system, we can use intertype declarations (introductions – ITD) introduced by AspectJ. Using intertype declarations, classes and interfaces obtain new capabilities in Java. Additionally, the occurrence of a new declaration of an element is proceeded at compilation time. Intertype declarations come in three forms: Intertype declarations for methods, fields and constructors. Moreover, intertype members are only associated with their existing types and hierarchies and are declared in aspect modules. The use of intertype members adheres to the AOP concept for solving an intricate system as well. In order to reify the concept of intertype declarations, an example of intertype members including field and method (taken from the SpaceWar project) are shown in Fig. 1.8.

```
private Color Ship.color;              ←——— Intertype field declaration

public void Bullet.paint(Graphics g) {  ←—— Intertype method declaration
        g.setColor(Color.green);
      g. fillOval ((int)this.getXPos() - 1,(int)this.getYPos() - 1, 3,3);
}
```

Figure 1.8: Showing intertype method and field declarations in an aspect module

In the figure, a member named `color` of type `Color` into the `Ship` class is declared in the aspect and has private modifier, which means the member can only be accessed in its own aspect. Even if the `Ship` class possesses another field with the same name called `color`, they will not conflict one another. Concerning the last declaration, the definition refers to a new method named `paint` with one parameter and `public` modifier. Therefore, the method is part of the `Bullet` class and we are able to access the method in every module due to its modifier. What is important to note about intertype declarations is that we cannot use three wildcards as mentioned earlier for declaring a member, as the wildcards are only used to identify join points and their context.

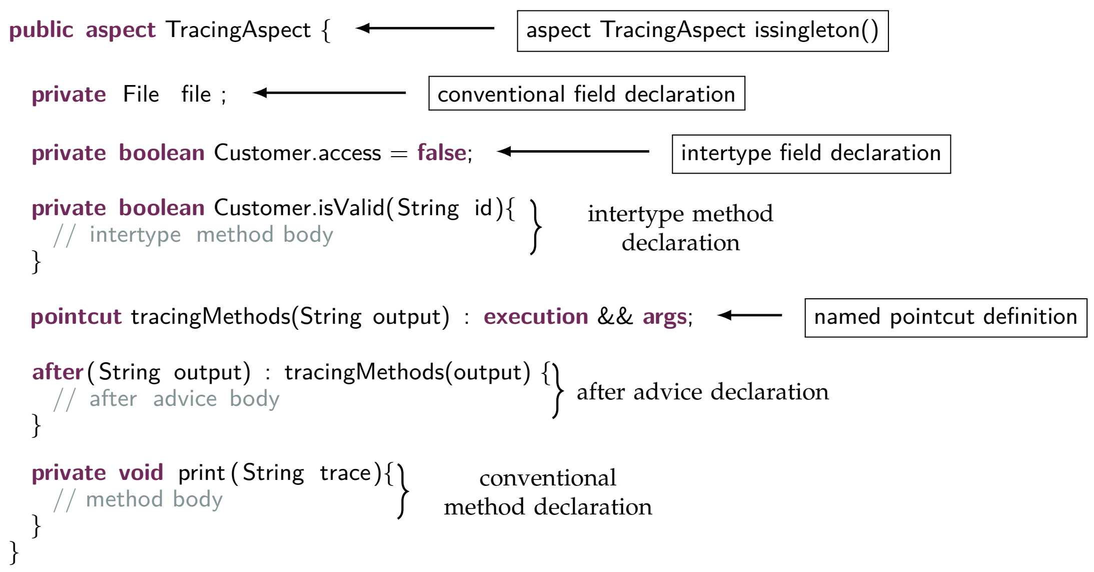
We all know that without using any single constructor as explained above we cannot implement different actions including both dynamic and static structures in order to modularize a software system. Unlike OOP, The implementing of a crosscutting concern needs to be done in another module which we will examine with comparing the conventional technique.

1.2.5 Aspects

While we implement a system by means of OOP techniques, we use classes in order to modularize our systems' concerns. However, classes may not help for good modularity in order to avoid essential design problems according to a system and thus another unit is needed to control crosscutting concerns in a system. An *aspect* provides help in the modularization of crosscutting concerns. In other words, an aspect is now part of the modules in the system in terms of the AspectJ language and works as a separate crosscutting module. An aspect is similar to a class, but it can also contain advice, pointcuts, and intertype declarations. We can define an aspect as an abstract like an abstract class. Additionally, aspects (concrete and abstract) can extend to a class as well as implement interfaces. The differences between aspects and classes are as follows:

i. An aspect does not support the *new* keyword to be instantiated.
ii. An aspect including concrete and abstract can only be inherited by an abstract aspect
iii. An aspect cannot be defined as a local or anonymous aspect.
iv. An aspect can be specified as *privileged*, which helps the aspect to see the private attributes and behaviour of advised classes.
v. An aspect cannot be extended by a class.
vi. An aspect can be declared as an inner aspect in a class or an aspect.

In addition to aspects, there are a few additional instantiations in order to affect the aspect declarations for various design factors and allow altering aspect creations depending upon situations. The aspect instantiations such as *singleton*, *per object*, *per control-flow*, and *per type* are to modify the execution of an aspect. The rest of the aspect associations provide a mechanism to overcome different situations between aspect and program flow. The basic instantiation is singleton structure. We can use this structure by typing the *issingleton* keyword in the aspect declaration. A singleton aspect is also a default instantiation that means without any modifier the aspect has singleton behaviour as well. Another basic feature is that default association (or singleton) create only one instance of an aspect that crosscut an entire system. Finally, we can conclude the main section by illustrating the overall concept of an aspect module shown in Fig. 1.9 along with crosscutting constructors and conventional implementations.



```
public aspect TracingAspect {

  private File file ;

  private boolean Customer.access = false;

  private boolean Customer.isValid(String id){
    // intertype method body
  }

  pointcut tracingMethods(String output) : execution && args;

  after(String output) : tracingMethods(output) {
    // after advice body
  }

  private void print(String trace){
    // method body
  }
}
```

aspect TracingAspect issingleton()
conventional field declaration
intertype field declaration
intertype method declaration
named pointcut definition
after advice declaration
conventional method declaration

Figure 1.9: Crosscutting module named `TracingAspect` along with its members

1.3 Introducing Ekeko

The Ekeko plug-in (Ekeko) [2] is suitable for analysing Java projects in Eclipse workspace. Thanks to *Coen De Roover* and his contributors that we make use of Ekeko for our purpose. Ekeko itself is a meta-programming library, because it has been built on top of Clojure' core.logic library and thus it simply provides higher-order functions [40]. Ekeko is also defined as the successor to the SOUL logic program query language [20, 21].

In order to analyse Java code, Ekeko provides us to write satisfying logic queries. However, an additional tool is required to find our questions that will reveal valuable details from existing AspectJ projects. More specifically, we use Ekeko's AspectJ extension plug-in [19] named General-purpose Aspectual Source code Reasoner (GASR). Since AspectJ itself increases its own community progressively, GASR will always suit for AspectJ developers' objective in order to estimate the design and behaviour of an AspectJ project. Moreover, Ekeko enables writing your question in the form of a query written using Clojure's core.logic library.

Consider there are two proposed questions about which we all want to know the outcomes in an AspectJ project. The two ekeko queries simply answer to *"how many aspects does the project have?"* and *"how many advice are singleton in the project?"* questions respectively as follows:

```
1 (defn NOAspects [?aspects  ?source]
2   "the number of aspects in a given project"
3   ( l/all
4       (w/aspect ?aspects)
5       (equals ?source (.getSourceLocation ?aspects))))

1 (defn SingetonAspects [?aspectname ?singleton]
2    "the number of singleton aspects"
3    (l/fresh  [?aspect ?source]
4      (NOAspects ?aspect ?source)
5      (equals   ?singleton (.getName (.getKind (.getPerClause (.getDelegate ?aspect)))))
6      (succeeds (= " issingleton " ?singleton))))
7
8 (count (ekeko [?aspectname ?sing] (SingletonAspects ?aspectname ?sing)))
```

The Ekeko snippets above written in the Clojure programming embody the two questions. Each variable name in the queries starts with a question mark (?) which indicates a logic variable. The first query is substituted for our first question that denotes *"the number of aspects in a given project"*. On line 3 of both the first and second queries, we make use of two different logic macros namely *fresh* and *all*, which are part of Clojure's core.logic library. Using the logic macros in a query do not allow to use any functional macros like *doseq*, *for*. In addition, while *fresh* allows to introduce new fresh variables into our logic predicates like `SingetonAspects`, *all* does not allow to create a new variable. What is important to mention about the use of *fresh* is that we can only use new variables within the lexical scope of a query (i.e. related parentheses marked as *red* in the second query). Consider the first logic predicate named `NOAspects` defined with 2 arguments namely *?aspects* and *?source*, on line 4, we fetch all aspect declarations from a project by means of the `(aspect ?aspect)` predicate which enables to be used along with other well-used predicates such as `(type-field ?t ?field)`, `(pointcutdefinition ?point)`, and `(advice ?a)` forming the predicate library of GASR. Once we obtain the aspects, we can follow the next line in order to collect their source locations by means of the `getSourceLocation` method of the aspects as each aspect is of an instance of the `ResolvedType` class and this enables to unify specific characteristics of an aspect. That is to say, as long as we use the `NOAspects` query, we undoubtedly acquire the number of aspects and their locations. We now change our view to the second query named `SingetonAspects` which has two arguments namely *?aspectname* and *?singleton*. The query collects the number of singleton aspects which denotes as *"the number of singleton aspects in a given project"*. The goal of the `NOAspects` query on line 4 is to retrieve all aspects of a project. In addition, the new fresh arguments are associated with the result the `NOAspects` query acquires. Since, we have all the aspects, we then examine what their associations are by invoking the `getPerClause` method and finally on line 6 we filter default aspects from the collection by comparing with *"issingleton"*. Although the Clojure language eases to define a query in one line, writing a query sequentially just as the mentioned queries could be more definite for understanding the implementation of the query.

Unlike Clojure, Ekeko has a special keyword called *ekeko*, which is used to run a query in read-eval-print loop (REPL) [41]. Additionally, there is a few auxiliary logic keywords (only two of them shown in the queries) such as *equals*, *succeeds* and *contains* and *fails* of which Ekeko itself facilitates us to make use during the implementation of a query. On the other hand, these two queries are also parts of our existing metrics. The source code of all metrics presented in this thesis are available on GitHub[1].

[1]`http://ozlerhakan.github.io/aop-metrics/`

CHAPTER 2

Research Questions

In this section, we will present the research questions that aim to provide a better idea of how AOP is used in practice. As our aims about the research questions were explained in the introduction section, we want to analyse how AOP relates to coupling, how AOP is used in existing projects, how AOP projects take into account existing principles, and how often aspect-oriented constructs are used in comparison to object-oriented constructs. Our general question has been separated into several research questions organized in order to enlighten particular characteristics as for given aspect-oriented projects. What is important to note is that each research question consists of particular metrics. That is, each specified metric only answers to its assigned research question. The hierarchy of the questions is constructed as follows:

1. How large is the system?
   - *Lines of Code (LOC)**
   - *Vocabulary Size (VS)**
   - *Number of Attributes (NOA)**
   - *Number of Methods (NOM)*

* 3 metrics were taken directly from [34]

2. How often are AOP constructs used compared to OOP features?
   - *Number of Intertypes (NOI)*
   - *Number of Advice (NOAd)*
3. Which AOP constructs are typically used?
   - *Inherited Aspects (InA)*
   - *Singleton Aspects (SA)*
   - *Non-Singleton Aspects (nSA)*
   - *Advice-Advanced Pointcut Dependence (AAP)*
   - *Advice-Basic Pointcut Dependence (ABP)*
   - *Number of Around Advice (NOAr)*
   - *Number of Before/After Advice (NOBA)*
   - *Number of After Throwing/Returning Advice (NOTR)*
   - *Number of Call (NOC)*
   - *Number of Execution (NOE)*
   - *Adviceexecution-Advice Dependence (AE)*
   - *Number of Wildcards (NOW)*
   - *Number of non-Wildcards (NOnW)*
   - *Argument size of Args-Advice (AAd)*
   - *Argument size of Args-Advice-args (AAda)*
4. How many types and members of a system are advised by AOP?
   - *Percentage of Advised Classes (AdC)*
   - *Percentage of non-Advised Classes (nAdC)*
   - *Number of Advised Methods (AdM)*
   - *Number of non-Advised Methods (nAdM)*
   - *Classes and Subclasses (CsC)*
   - *Average of Subclasses of Classes (ScC)*
5. Is there a relation between the amount of coupling in an aspect, and how many shadows it advises?
   - *Advice-Join Point Shadow Dependence (AJ)*
   - *Number of thisJoinPoint/Static (tJPS)*
   - *Number of Modified Args (MoA)*
   - *Number of Accessed Args (AcA)*
   - *Around Advice - non-Proceed Call Dependence (AnP)*
6. How many dependencies are there between classes and aspects?
   - *Attribute-Class Dependence (AtC)**
   - *Advice-Class Dependence (AC)**

* 2 metrics were taken directly from [44]

- *Intertype-Class Dependence (IC)**
- *Method-Class Dependence (MC)**
- *Pointcut-Class Dependence (PC)**
- *Advice-Method Dependence (AM)**
- *IntertypeMethod-Method Dependence (IM)**
- *Method-Method Dependence (MM)**
- *Pointcut-Method Dependence (PM)**

2.1 Running Example

Before describing each of the metrics in detail, we will first present our running example that we can use to illustrate each metric. Fig. 2.1 has been constructed as two separate parts including class and aspect modules. The black arrows identified in the left-hand side of the figure depict the related join point shadows in the project. If we look into the right-hand side of the figure, the numbers enclosed with circles likewise state how many join point shadows the advice have. In presenting the example, we mention the values of each metric listed below Fig. 2.1. The test project is also available on GitHub[1] in order that you can try out the metrics through this example.

* 7 metrics were taken directly from [44]

[1] `https://github.com/ozlerhakan/aop-metrics/`

```java
package ua.thesis.test;
public abstract class Media {
  private String filePath;
  public String getFilePath(){
    return filePath;
  }
  public void setFilePath(String filePath){
➤   this.filePath = filePath;
  }
}
class ImageMedia extends Media {
  protected int width,height;
}
class JpegImage extends ImageMedia {
}
class PngImage extends ImageMedia {
}
abstract class GraphicObject {
  abstract double getArea();
}
class Circle extends GraphicObject {
  double radius;
  @Override
➤ double getArea() {
    return radius * radius * Math.PI;
  }
}
class Rectangle extends GraphicObject {
  double length, width;
  @Override
  double getArea() {
    return length * width;
  }
  public void setLength(double length) {
    this.length = length;
  }
  public void setWidth(double width) {
    this.width = width;
  }
  public static void main(String[] args){
    Rectangle rec = new Rectangle();
    rec.setWidth(4);
    rec.setLength(5);
➤   rec.getArea();
    Media media = new Media();
➤   media.setFilePath("images/figure.png");
    System.out.println(media.getFileName());
     Circle circle = new Circle();
➤    circle.getArea();
  }
}
class Clipboard {
  private String content;
  public String getContent(){
    return content;
  }
➤ public void setContent(String content){
    this.content = content;
  }
}
```

```aspectj
package ua.thesis.test;
import java.io.File;
import java.util.ArrayList;
public aspect UtilAspect {
 interface Log{
   void logPath(String message);
 }
 String Media.filename;
 String Media.getFileName(){
   return filename;
 }
 private ArrayList<String> Clipboard.listofContent =
                    new ArrayList<String>();
 void addList(Clipboard clip, String content){
   clip.listofContent.add(content);
 }
 // divide the selected content
 void splitContent(Clipboard clip, String content){
   String [] list = content.split("\n");
   for (int i = 0; i < list.length; i++) {
     addList(clip, list[i]);
   }
 }
 pointcut printArea() : call(double GraphicObject+.getArea());
❷after() returning(double area) : printArea() {
   System.out.println("Execution of the "
   + thisJoinPoint.getSignature().getName() +" method\n"
   + "Area is "+area);
 }
 pointcut showContent(Clipboard clip,String content) :
   execution(public void Clipboard.setContent(String)) &&
   args(content) &&
   this(clip);
❶before(Clipboard clip, String content):
   showContent(clip,content){
     splitContent(clip,content);
 }
❶after(Log log, String path) :
     call(* Media.setFilePath(..)) &&
     args(path) &&
     target(log){
       log.logPath(path);
 }
❶before(String filePath, Media media):
   set(private String Media.filePath) &&
   if(media.getFilePath() != null) &&
   args(filePath) &&
   this(media){
     media.filename = (new File(filePath).getName());
 }
 pointcut Area(Circle circle) :
   execution(double getArea()) &&
   this(circle);
❶double around(Circle circle): Area(circle){
   if(circle.radius <= 0) { return -1; }
   else { return proceed(circle); }
 }
}
```

Figure 2.1: Illustration of an AspectJ project

LOC : 117	*SA* : 1	*NOC* : 2	*AdC* : 0.5	*tJPS* : 1	*MC* : 4
VS : 9	*nSA* : 0	*NOE* : 2	*nAdC* : 0.5	*MoA* : 0	*PC* : 3
NOA : 7	*AAP* : 1	*AE* : 0	*AdM* : 4	*AcA* : 3	*AM* : 6
NOM : 11	*ABP* : 4	*NOW* : 2	*nAdM* : 7	*AnP* : 0	*IM* : 0
NOI : 1	*NOAr* : 1	*NOnW* : 3	*CsC* : 1	*AtC* : 0	*MM* : 3
NOAd : 5	*NOBA* : 3	*AAd* : 0.6	*ScC* : 1.5	*AC* : 6	*PM* : 1
InA : 0	*NOTR* : 1	*AAda* : 1	*AJ* : 1.2	*IC* : 3	

2.2 Selected Systems

In this subsection, we briefly describe the selected systems. Our data are retrieved in terms of ten aspect-oriented systems. In addition, we selected the following systems as they have widely been used by other researchers for several purposes like software measurements in the area of AOP [12, 13, 14, 15, 17, 22, 29, 31, 33].

- *HealthWatcher* [1] (HW), which is a web-based information system, unifies various health issues collected from citizens of which health care institutions can promptly take care. Using the utilities of the Java language, HW is composed of a different variety of techniques such as GUI, Servlets, RMI, and JDBC. HW [25, 36] was revised in several steps in order to modularize the existing exception handlers with new refactored exception handler aspects such as distribution, concurrency, persistence, synchronization, and transaction using AspectJ.

- *HyperCast* [2] (HC) is a software system developed by the Multimedia and Networks Group at the University of Virginia to build protocols and application programs for application-layer overlay networks. It supports a variety of overlay protocols, delivery semantics and security schemes, and has a monitor and control capability. Sullivan et al. [39] have refactored the original version of the system using AspectJ. The system uses aspects to implement the following six crosscutting concerns: socket, protocol, monitoring, service, adapter, and lastly logging. We chose this system because of its larger code size.

- *AJHotDraw*[3] (AJH) has been developed by Marin et al. [30] by refactoring the JHotDraw [8] framework that is an open-source 2D graphics editor written in Java. Therefore, the migrated system (i.e. AJHotDraw) facilitates the same functionalities of the OO version, even though the quality of the system varies due to a few aspect refactoring capabilities. Crosscutting concerns such as persistence, the command and undo functionality have been modularized using aspects. According to the system size, we also selected this system into our pool.

[1]`http://www.kevinjhoffman.com/tosem2012/`
[2]The source code were kindly released by A. Przybyłek
[3]`http://ajhotdraw.sourceforge.net/`

- *AJHSQLDB*[4] is the aspect-oriented refactoring of HSQLDB [6] system, an open-source Java system implementing a relational database system. Störzer et al. [38] played a role in migrating the existing system to the aspect-oriented system using the Feature Exploration and Analysis tool (FEAT) to find relevant crosscutting code. The current AO system consists of a variety of relevant concerns detected by the semantic-guided approach such as logging, tracing, exception handling, caching, and pooling, and authentication/authorization. This system is the largest one of the selected systems. That is, we believe that this system will applicatively contribute most to the main research questions of this paper.

- *Contract4J5*[5] is an open-source system that supports Design by Contract programming for Java using Java 5 annotations to define contracts. In other words, the Contract4J5 project was developed by Aspect Research Associates [37] by means of AspectJ in order to support the design by contract approach for the Java programming. Exception handling, invariant conditions are used as crosscutting concerns in the whole system.

- *MobileMedia*[6] (MM) was implemented by Figueiredo et al. [23] for the manipulation of multimedia data such as photo, music, and video on mobile devices (i.e. mobile phones). MM was derived based on a software product line (SPL) [42] called MobilePhoto [43]. This dependence makes this project as a software product line as well. MM was released two versions, one in AspectJ, the other in Java. In each release of both Java and AspectJ versions, several features were depicted due to plausible scenarios, plus both design were determined by the use of the Model-View-Controller (MVC) architecture pattern

- *iBATIS*[7] (now called as *MyBATIS* [10]) is a Java-based open-source framework that maps relational databases to object-oriented programs. Essentially, it was started by Clinton Begin and now it has more than ten contributors [9]. iBATIS supports the two most popular platforms: Java and C# for .Net. We have found the related repository including a few releases shared for the purpose of a research experiment by the Computing Department of Lancaster University. In that point, we have only considered one refactored version of the project namely

[4]`http://www.sourceforge.net/projects/ajhsqldb/`

[5]`https://github.com/deanwampler/Contract4J5`

[6]`http://sourceforge.net/projects/mobilemedia/`

[7]`http://sourceforge.net/projects/ibatislancaster/`

iBatis01.5.

- *Telestrada*[8] [16] is a web-based traveler information system developed for a Brazilian national highway administrator. It allows its users to register and visualize information about Brazilian roads. Telestrada was refactored in order to clearly construct the exception-handling mechanisms through new exception handler aspect modules. That is, the crosscutting modules of the system deal with exception handler depending on several processing.

- *SpaceWar* is an implementation of the classic video game Spacewar. It is an example that comes bundled with the AJDT. We chose this project as it is the largest one among AJDT example projects and facilitates to understand how to use the features of AspectJ.

- *TetrisAJ*[9] , a small AspectJ system, was developed by Gustav Evertsson by refactoring the existing Tetris program in order to observe how AspectJ can be used and some problems that occur during the existing one. The overall architecture consists of three sections which are Main including the game rules, Logic including all the data manipulations, and Gui displaying everything over the screen.

[8] `http://www.kevinjhoffman.com/tosem2012/`

[9] `http://www.guzzzt.com/coding/aspecttetris.shtml`

CHAPTER 3

The Metrics

In this section, the definition of each metric is presented. The metric framework that we utilise has been developed by means of the Ekeko plug-in from scratch. We now start delving into the metrics corresponding to the list of research questions presented in chapter 2.

How large is the system?

- *Lines of Code (LOC)*. LOC counts the number of source code lines. Additionally, this metric is the most commonly used measurement for object-oriented projects. We thus use the metric for both classes and aspects code. What is important to note about the metric is that single-line and multi-line comments as well as documentations known as java docs are eliminated. Lastly, blank lines and test classes are not interpreted within the metric. That is, we only count the source code of a given project. Table 3.1 shows the exact number of source code lines per AspectJ project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
LOC	62692	51132	21046	11902	3740	5898	3870	1383	3947	849

Table 3.1: The results of LOC

- *Vocabulary Size (VS)*. VS counts the number of class and aspect declarations in a given project. Each module referring either class or aspect is considered as a part of the system vocabulary. Note that class itself involves various types such as abstract class, subclass, inner class, static inner class, local class and anonymous class. In our case, the metric only takes into account subclass, abstract class and default class definitions. About the aspect declarations, we consider all aspect types including abstract, inner, and default. Another important detail about the metric is that we exclude classes which have test behaviour by eliminating the test folder of a given project manually and examining their name starting or ending with "test" in order to collect more expressive result. Each column represented in table 3.2 below denotes a number of class and aspect declarations.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
VS	255	291	284	215	50	99	92	29	167	14

Table 3.2: The results of VS

- *Number of Attributes (NOA)*. NOA counts the number of fields in classes and aspects. In order to acquire the fields in a given project, the VS metric is used by NOA. The implementation of the metric in Ekeko corresponds to the query above that takes advantage of the VS metric on line 4 in order that it enables us to analyse a system after eliminating some unnecessary parts from the collection (e.g., line 8 eliminates internally generated AspectJ compiler code). The snippet likewise yields the output of the NOA metric for each specified AspectJ project shown in table below.

```
1 (defn NOA [?module ?field]
2   "count the number of attributes of both classes and aspects"
3   (l/fresh [ ?fields ?modules]
4          (equals      ?modules (VocabularySize))
5          (contains    ?modules ?module)
6          (equals      ?fields   (.getDeclaredFields ( first ?module)))
7          (contains    ?fields    ?field )
8          (equals      false  (IndexOfText (.getName ?field) "$"))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOA	3146	2290	629	475	76	195	150	104	138	57

Table 3.3: The number of attributes

- *Number of Methods (NOM)*. NOM counts the number of method declarations declared in both classes and aspects in a given projects. The metric makes use of the VS metric in order to collect its data like the NOA metric. It does not count abstract methods and constructors of classes and aspects. The following Ekeko snippet illustrates our concrete function called NOM. The snippet excludes redundant methods which belong to internal AspectJ compiler code in order to achieve a precise result. Essentially, the metric refines the declared methods in a project by checking specific method features. For example, if a method equals one of the static methods of the AspectJ language called "`hasAspect`" and "`aspectOf`", the query excludes the method from the collection on line 12. Table 3.4 shows the results of method declarations collected from class and aspect modules that the metric obtains.

```
1 (defn NOM [?module ?method]
2   "counts the number of method declarations of  classes  and aspects"
3   (l/fresh  [?types    ?modules ?methods]
4              (equals    ?modules (VocabularySize))
5              (contains ?modules ?module)
6              (equals    ?methods (.getDeclaredMethods (first ?module)))
7              (contains ?methods ?method)
8              (equals false  (.isAbstract  ?method))
9              (equals false  (= 3 (.getKey (.getKind ?method))))
10             (equals false  (= 8 (.getModifiers ?method)))
11             (equals false  (IndexOfText (.getName ?method) "$"))
12             (equals false  (or (= "hasAspect" (.getName ?method))
13                                (= "aspectOf"    (.getName ?method))))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOM	3915	2907	2278	1132	279	394	175	134	281	34

Table 3.4: The number of method declarations

How often are AOP constructs used compared to OOP features?

- *Number of Intertypes (NOI)*. NOI counts the number of intertype methods of aspects. For the consistency of the metric, abstract intertype methods are not considered in results. Our metric shown below collects all the declared intertype method declarations of aspects, plus acquires the outcome of the projects as shown in table 3.5.

```
(defn NOIntertype [?intertype]
  "counts the number of intertype method declarations"
  ( l/all
          (w/intertype|method ?intertype)
          (equals false (.isAbstract (.getSignature ?intertype )))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOI	0	0	50	4	0	17	46	6	0	0

Table 3.5: Showing intertype method declarations per project

- *Number of Advice (NOAd).* NOAd counts the number of advice of aspects in a given project. The function below is concerned with combining five different advice declarations namely before advice, after advice (default), after throwing advice, after returning advice, and around advice. The table below represents about how many advice declarations the projects have.

```
(defn NOAdvices [?aspect ?advice ?pointcut]
 "counts the number of advice"
  ( l/all
    (w/advice ?advice)
    (equals ?aspect    (.getDeclaringType (.getSignature ?advice)))
    (equals ?pointcut  (findpointcut ?advive))
    (equals false      (IndexOfText (.getName (.getClass ?advice)) "Checker"))
    (equals false      (or
                        (or (.isCflow     (.getKind ?advice))
                            (.isPerEntry (.getKind ?advice)))
                        (= "softener"     (.getName (.getKind ?advice)))))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOAd	106	25	47	120	17	73	77	13	46	21

Table 3.6: Showing the number of advice

Which AOP constructs are typically used?

- *Inherited Aspects (InA).* InA counts the number of inherited aspects in a given project. That is to say, it counts all aspects that implement an abstract aspect. The following snippet is a representative implementation of the metric. At the beginning of the query there are two different types of auxiliary elements indicating another query called `NOAspects` and an Ekeko's predicate which is `aspect-declaredsuper` of which we make use for finding an aspect that extends to an abstract aspect. How many inherited aspects the projects have is shown in the following table.

```
(defn InheritedAspets [?aspect ?super]
        (l/fresh [?source]
                (NOAspects ?aspect ?source)
                (w/aspect-declaredsuper ?aspect ?super)
                (succeeds (.isAbstract ?super))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
InA	3	4	0	5	7	11	11	2	2	0

Table 3.7: The number of inherited aspects

- *Singleton Aspects (SA)*. SA counts the number of aspects that use the singleton instantiation mechanism. To implement this metric we examine the PerClause form of each aspect. The query below as explained in more detail in Sec.1.3 retrieves all aspects that provide singleton behaviour. The total number of singleton aspects is depicted in table 3.8 through this metric.

```
(defn NOSingletonAspects [?aspect ?association ]
"the number of singleton aspects"
 (l/fresh [?source]
   (NOAspects ?aspect ?source)
   (equals ?association (.getName (.getKind (.getPerClause (.getDelegate ?aspect)))))
   (succeeds (= "issingleton" ?association ))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
SA	28	5	31	41	14	25	42	6	18	8

Table 3.8: Singleton aspects over each project

- *Non-Singleton Aspects (nSA)*. nSA counts the number of aspects that do not have singleton behaviour. Similar to the SA metric, the `PerClause` form plays a role in order to collect desired aspects. In other words, determined aspects have different aspect associations such as per object and per type. Table 3.9 designates a concrete result of the metric for each determined project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
nSA	2	5	0	1	1	5	0	2	0	0

Table 3.9: The number of nSA

- *Advice-Advanced Pointcut Dependence (AAP)*. AAP counts the number of advice associated with a pointcut using more advanced constructs. More precisely, if an advice is constructed with at least one of the advanced pointcuts such as *cflow*, *cflowbelow*, *if* and *adviceexecution*, we consider it an advanced advice. The representative scheme shown in table 3.10 shows the number of the dependencies per project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AAP	32	0	0	0	15	3	0	0	0	0

Table 3.10: Advanced advice

- *Advice-Basic Pointcut Dependence (ABP)*. ABP counts the number of advice – basic pointcut dependence if a basic pointcut construct is advised by an advice. In other words, ABP collects the basic advice of aspects in a given project. An advice is considered basic if its pointcut does not make use of any advanced constructs (as defined by the AAP metric). What is essential to note about the metric is that as long as an advice is flagged as an advance by AAP, ABP eliminates this advice in its examination. The table below depicts advice-pointcut dependencies as basic advice declarations.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
ABP	74	25	47	120	2	70	77	13	46	21

Table 3.11: Basic advice

- *Number of Around Advice (NOAr)*. NOAr counts the number of around advice in a given project. The following query extracts all the declared around advice of a project by checking each advice kind. The around advice of the projects are demonstrated over the table's columns.

```
(defn NOAroundAdvice[]
 "count the number around advice in a system"
   (count (ekeko [?aspect ?advice ?pointcut]
          (NOAdvices ?aspect ?advice ?pointcut)
          (succeeds (= (.getName (.getKind ?advice)) "around")))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOAr	35	3	18	14	7	42	29	1	13	6

Table 3.12: Around advice

- *Number of Before/After Advice (NOBA)*. NOBA counts the number of before/after advice in a given project. The related table of the metric emphasises the result over the projects. The metric called *NOAfter* and *NOBefore* are combined into the main query, as follows:

```
(defn NOAfter[]
  "count the number after advice in a system"
   (count (ekeko [?aspect ?advice ?pointcut]
          (NOAdvices ?aspect ?advice ?pointcut)
          (succeeds (= (.getName (.getKind ?advice)) "after")))))

(defn NOBefore[]
  "count the number before advice in a system"
   (count (ekeko [?aspect ?advice ?pointcut]
                 (NOAdvices ?aspect ?advice ?pointcut)
                 (succeeds (= (.getName (.getKind ?advice)) "before")))))

(defn NOBeforeAfterAdvices[]
  "count the number before and after advice in a system"
         (+ (NOAfter) (NOBefore)))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOBA	60	20	25	43	5	8	28	4	2	15

Table 3.13: Showing before/after advice

- *Number of After Throwing/Returning Advice (NOTR)*. NOTR counts the number of after throwing/returning advice in a given project. In the same way as the earlier metric, this metric consists of two different metrics in order to combine both after returning and after throwing advice. The combined metrics' results are shown, depending on each project in table 3.14.

```
(defn NOAdviceReturning[]
  "count the number before and after returning advice in a system"
    (count (ekeko [?aspect ?advice ?pointcut]
                  (NOAdvices ?aspect ?advice ?pointcut)
                  (succeeds (= (.getName (.getKind ?advice)) "afterReturning")))))

(defn NOAdviceThrowing[]
   "count the number after throwing advice in a system"
    (count (ekeko [?aspect ?advice ?pointcut]
                 (NOAdvices ?aspet ?advice ?pointcut)
                 (succeeds (= (.getName (.getKind ?advice)) "afterThrowing")))))

(defn NOAfterThrowingReturningAdvice[]
  "count the number after throwing/returning advice in a system"
    (+ (countNOAThrowing) (countNOAReturning)))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOTR	11	2	4	63	5	23	20	8	31	0

Table 3.14: Showing after throwing/returning advice

- *Number of Call (NOC)*. NOC counts the number of call pointcut constructs declared in pointcut definitions in a given project. The metric obtains the result by examining all pointcuts consisting of method-call and constructor-call. The following query called *NOCall* uses an auxiliary logic predicate named *CallsExecutions* including four parameters in order to collect both call and execution pointcut constructs including methods and constructors from the systems and eliminates unnecessary pointcuts to show only call pointcut constructs, thus retrieving the number of call pointcut constructs is shown the table below.

```
(defn NOCall [?shortAspect ?call ?advicekind]
  (l/fresh [?pointdefs]
        (CallsExecutions ?shortAspect ?advicekind  ?call  ?pointdefs)
        (succeeds (or
                     (= "method-call"      (.toString (getKind ?call)))
                     (= "constructor-call" (.toString (getKind ?call)))))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOC	55	19	23	39	1	17	20	24	7	12

Table 3.15: The number of call pointcut constructs

- *Number of Execution (NOE)*. NOE counts the number of execution pointcut constructs in a given project. We control each pointcut declaration in order to find execution pointcut constructs. Just like the previous metric, the *NOExecution* query uses the same logic predicate named *CallsExecutions* and collects only the execution pointcut constructs including methods and constructors of a project from the collection. Execution pointcut constructs of each project have been accomplished through the metric shown the table below.

```
(defn NOExecution [?shortAspect ?execution ?advicekind]
 (l/fresh [?pointdefs]
   (CallsExecutions ?shortAspect ?advicekind  ?execution  ?pointdefs)
    (succeeds (or (= "method-execution"    (.toString (getKind ?execution)))
                     (= "constructor-execution"(.toString (getKind ?execution)))))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOE	124	7	24	128	12	86	58	0	55	7

Table 3.16: The number of execution pointcut constructs

- *Adviceexecution-Advice Dependence (AA)*. AA counts adviceexecution – advice dependence if an advice has an adviceexecution pointcut construct in its pointcut definition. More specifically, we obtain the number of adviceexecution-advice dependencies in a given project using this metric. In order for the metric to obtain exact values, we explore *adviceexecution* pointcut construct in each advice' pointcut definitions including named and anonymous pointcuts in a project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AE	0	0	0	0	0	0	0	0	0	0

Table 3.17: Showing adviceexecution-advice dependencies

- *Number of Wildcards (NOW)*. NOW counts the number of pointcuts using wildcards. If any of the three wildcards (*,+,..) are mentioned in specific pointcuts consisting of *method-execution*, *method-call*, *constructor-execution*, *constructor-call*, *field-get*, and *field-set*, the pointcut is included in the metric. What is important to clarify is that the wildcards facilitate matching with multiple different classes or methods. In our case, the use of the wildcards is important to find incompletely named classes in the mentioned pointcuts. More clearly, if we take a look at Fig. 1.6 in order to comprehend NOW. The before advice connects with an anonymous pointcut consisting of an execution pointcut construct. The metric examines this pointcut's signature (i.e. method signature) whether any wildcard has been used in order not to mention a complete class name. In this case, this `javax..*` part of the method signature denotes a declared class name. Since the declared class name is represented as two dots (..) combining with the `javax` package, the metric does not spot any particular class name and thus the execution pointcut construct is marked as a member of NOW. Another example for NOW shown in Fig. 2.1 yields 2 pointcuts using wildcards in two named pointcut definitions namely `printArea` and `Area`. If we look into the `Area` declaration, the named pointcut is comprised of `this()` and an execution pointcut construct whose method signature denotes any `getArea` method that returns double regardless of modifiers (i.e. protected, private). In a way the method signature can be rewritten like `execution(double *.getArea())`. That is, the execution pointcut construct makes use of star (*) in order to reach any `getArea` method in the entire project whereas the boundary of the method narrows using `this()` in our example. As a result, since we still do not know the exact class name of the method and NOW does not too, the execution pointcut construct pertains to this metric. The visual representation of the results of NOW for each mentioned AspectJ project is shown in the table below.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOW	198	0	9	31	30	20	0	7	10	0

Table 3.18: The number of declared pointcuts using wildcards

- *Number of non-Wildcards (NOnW).* NOnW counts the number of pointcuts not using wildcards. More clearly, the metric only counts the number of classes clearly named without any wildcards' (*,+..) help in the pointcuts which can refer to *method-call*, *method-execution*, *constructor-call*, *constructor-execution*, *field-get*, or *field-set*. Unlike NOW, this metric is to gather completely named classes from the particular pointcuts in a given project. If we return Fig. 1.5, the `AlbumData` class is named along with its member `getMedias` in the execution pointcut construct of the after advice without using any wildcards. Thus, we directly choose this execution pointcut construct according to the proper description of the class name. Additionally, Fig. 2.1 conveys the result of the metric in terms of the example. Table 3.19 shows a number of all matching pointcuts not facilitating any wildcards for each AspectJ project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
NOnW	21	19	25	134	0	76	78	17	47	21

Table 3.19: Representing pointcuts not using wildcards

- *Argument size of Args-Advice (AAd).* AAd calculates the average of the argument size of args pointcut constructs per advice in a given project. In other words, this metric first fetches all the advice of aspects and then calculates the average of the argument size of args pointcut constructs per advice declaration. Note that if an advice does not have an args pointcut construct, the argument size of this construct is counted as zero (0) for the advice. Lastly, the wildcards especially star (*) and two dots (..) are eliminated from selected args pointcut constructs in order to count the exact argument size of args. In showing the metric in Fig. 2.1, AAd computes the total number of argument size of args and calculates their average by dividing by the total number of advice (i.e. 3÷5). In addition, table 3.20 shows the average of arguments per advice declaration.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AAd	1.11	0.44	0.23	0.38	0.18	0.16	0.58	0.23	0.17	0.38

Table 3.20: Average of argument size of args pointcut constructs

- *Argument size of Args-Advice-args (AAda).* Unlike the previous metric, this metric collects advice that must include an args pointcut construct, and then calculates the average of argument size of args pointcut constructs per advice. That is to say, the metric only considers an advice that has an args pointcut construct, otherwise the metric excludes this advice from the number of advice for calculating the average. Note that the wildcards including star (*) and two dots (..) also are not counted as argument size. Fig. 2.1 shows the result of the metric as 1, depending on the example since the number of the advice that have an args pointcut construct equals to the number of argument size of args pointcut constructs (i.e. 3÷3). The table below covers a particular result about the specified projects of which make use this metric.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AAda	1.68	1.1	1.22	3.29	1	1.09	1.36	1	1	1.14

Table 3.21: Outcome of AAda per project

How many members of a system are advised by AOP?

- *Percentage of Advised Classes (AdC).* AdC calculates the percentage of classes that are advised, among all classes in the system.. In other words, if a class in which a join point shadow exists is determined, the class is automatically considered as *"advised"*. Note that a class is only counted once, even if it contains multiple join point shadows. In order to implement the metric, we look at the most used pointcut types namely *method-execution*, *method-call*, *constructor-call*, *constructor-execution*, *field-get*, and *field-set* pointcuts. In addition, the total number of classes comprising the list of specific class declarations that the VS metric collects is also part of the metric's implementation as we use it to verify the advised classes by checking them throughout the list of classes. The percentage of advised classes divided by the number of classes is shown in the table below for each project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AdC	0.95	0.16	0.12	0.41	0.6	0.57	0.38	0.43	0.13	0.83

Table 3.22: Advised classes per project

- *Percentage of non-Advised Classes (nAdC).* nAdC performs the counterpart of the *AdC* metric. It calculates the percentage of non-advised classes in a given project. It considers all selected classes do not have any join point shadows referred by advice. The percentage of classes which do not contain any join point shadows is shown in the table below.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
nAdC	0.05	0.84	0.88	0.59	0.4	0.43	0.62	0.57	0.87	0.17

Table 3.23: Classes that are not influenced by advice

- *Number of Advised Methods (AdM)*. AdM counts the number of advised methods of classes. It also depicts that each advised method is normally a non-inherited method declaration. Specific pointcut types namely *method-call* and *method-execution* are considered in order to construct the metric. Once the metric finds several join point shadows, depending on method-call/execution in classes, the selected methods are directly marked as *"advised"*. What is important to mention about *AdM* is that selected methods are counted only once regardless of how many times the same methods are mentioned –i.e. a method might be advised by both call and execution pointcuts. Non-inherited method declarations of classes have been collected by the metric as advised methods shown in table 3.24.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AdM	3801	10	26	142	124	181	27	10	37	9

Table 3.24: Showing advised methods

- *Number of non-Advised Methods (nAdM)*. nAdM counts the number of non-advised methods of classes. More specifically, the metric eliminates methods (i.e. non-inherited methods) that are not advised, from each class in a given project. Just as the AdM metric, nAdM takes into account *method-call* and *method-execution* pointcut types. The following table displays methods of which are not part the advised collection.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
nAdM	90	2897	2252	990	155	213	144	108	244	25

Table 3.25: Representing methods to which do not belong advised collection

- *Classes and Subclasses (CsC)*. CsC counts the number of advised classes that have at least one subclass. That is to say, the metric first looks at each method signature of *method-call* and *method-execution* pointcuts in order to collect completely named classes, such that it can fetche the subclasses of the collected classes, likewise the subclasses of already collected subclasses are traversed recursively until the metric does not find any other related subclasses of the base classes (i.e. first collected classes). The differences between CsC and AdC are that CsC only examines two pointcut types rather than using other pointcut types with which AdC deals. CsC also considers completely named classes in the mentioned pointcuts; this means classes using wildcards are not included. Additionally, AdC does not deal with signatures of pointcuts, it only considers whether a join point shadow exists in a class or not. Lastly, after finding proper base classes by checking two pointcuts, CsC uses a recursive function called `getsubclasses` shown in the snippet below in order to find related subclasses of the base classes. Note that, if a first collected class does not include any related subclasses (i.e. empty list), the class is excluded from the collection as a class should have at least one related subclass. The query below explaining what the aforementioned intention is on line 9 verifies that the return list called `?listofsub-classes` must not be empty. For example, if we delve into our running AspectJ example, we realise that the result of the metric is 1 such that there is one base class that contains at least one related subclass. The metric obtains this result after achieving a few steps. First, it collects two completely named classes (i.e. `Media` and `Clipboard`) by checking all the method-call/execution pointcut constructs of the `UtilAspect`. Second, once the metric has the two classes, it recursively examines other related subclasses of the classes. In this case, only the Media class has three related subclasses namely `ImageMedia`, `JpegImage`, and `PngImage`. In consequence, the Media class is counted as it has a number of subclasses. The next table shows the result of CsC according to the selected systems.

```
1 (defn CsC [?class  ?listofsubclasses ]
2    (l/fresh  [?type ?types  ?subclassofclass ]
3              (equals ?types  (declaringTypesMethodConstruct))
4              (contains ?types ?type)
5              (equals  ?class  (.getName (.getType (first ?type))))
6              (succeeds  ( nil?  (getInterface  ?class )))
7              (equals  ?subclassofclass  ( class|subclasses  ?class ))
8              (getsubclasses  ?subclassofclass   ?listofsubclasses )
9              (succeeds (false?  (empty?  ?listofsubclasses )))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
CsC	1	0	4	5	0	0	4	0	1	0

Table 3.26: Showing the advised classes including at least one subclass

- *Average of Subclasses of Classes (ScC)*. ScC calculates the average of the subclasses per base class. What is important not to forget about the metric is that if a base class does not have any subclasses, the subclasses of the base class are counted zero (0) such that only the base class contributes to the number of base classes in order to be calculated the average. If we recall the example mentioned earlier, CsC has excluded the `Clipboard` class from the result, whereas ScC by contrast incorporates the class into the number of base classes and set its empty list as zero for calculating the average of related subclasses. Thus, the result shapes differently according to Fig. 2.1 (i.e. 3÷2 = 1.5). The following table also shows the average of values for each AspectJ project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
ScC	0.33	0	1.62	0.55	0	0	0.42	0	0.05	0

Table 3.27: The averages over each project

Is there a relation between the amount of coupling in an aspect, and how many shadows it advises?

- *Advice-Join Point Shadow Dependence (AJ)*. AJ calculates the average of the advice-join point shadow dependencies. More precisely, the metric works by calculating the average of the join point shadows per advice in a given project. We can see below that the averages of the advice-join point shadow dependencies are shown over each project respectively.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AJ	51.45	6.8	2.7	2.88	44.65	7.71	1.14	4.69	1.5	1.48

Table 3.28: Advice-join point shadow dependencies

- *Number of thisJoinPoint/Static (tJPS)*. tJPS counts how often the `thisJoinPoint` and `thisJoinPointStatic` objects are used in advice bodies. As long as an advice uses at least one of the special objects, the metric directly incorporates the advice. In a way, we find how many advice have these objects in their body. The following table shows how many advice contain any of two specific objects in their body.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
tJPS	22	0	3	2	13	3	0	5	0	3

Table 3.29: Number of tJPS

- *Number of Modified Args (MoA)*. MoA counts the number of potentially modified arguments (bound by args pointcut constructs) in the body of advice. The metric only considers well-defined arguments in args pointcut constructs of advice. For example, altering the value of an argument of an args pointcut construct in an advice body denotes a modified behaviour. More clearly, imagine an args pointcut construct of an advice has an argument `x` of the `User` type. Creating a new instance of the `User` object using the `x` argument or changing a field's value of `x` in the advice body denote that such procedures are counted as a modification in this metric. The table below visually explains the number potentially modified arguments of args pointcut constructs for each project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
MoA	0	0	0	0	0	7	0	0	3	0

Table 3.30: Arguments potentially modified

- *Number of Accessed Args (AcA)*. AcA counts the number of accessed arguments (bound by args pointcut constructs) in the body of advice. In addition, an accessed argument can be specified as a(n) parameter type of another method call or assignment variable within the body of an advice. Consider the same argument (i.e. `x`) expressed in the MoA metric while we give several examples. Accessed arguments can be identified in several ways such as calling a method of an argument (e.g. `x.methodCall();`), using in another method call as a parameter (e.g. `doSomething(x);`), using in a simple assignment statement (e.g. `String s = x.name;`), and displaying in a message (e.g. `println("user " +x);`). What is also important to note is that aliasing is not taken into account for this metric. Accessed arguments within advice in each project are described as numerical data in the table below.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AcA	152	11	11	25	4	39	64	8	10	20

Table 3.31: Accessed arguments within advice

- *Around Advice - non-Proceed Call Dependence (AnP)*. AnP counts the number of around advice that do not contain any proceed calls in their body. The implementation of the metric consists of two stages. First, we collect all around advice in a given project. Second, the metric traces all statements implemented in the body of the around advice line by line in order not to find a proceed call in their body. The following table also represents the number of around advice not including any proceed calls per project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AnP	22	3	2	3	0	5	0	0	0	2

Table 3.32: Showing the AnP metric over each project

How many dependencies are there between classes and aspects?

An important detail that we should keep in mind about the undermentioned metrics: AtC, AC, IC, PC, and MC is that all these metrics eliminate the primitive data types. Besides, for IM, AM, and MM, they are comparable metrics, whereas their behaviour are slightly different.

- *Attribute-Class Dependence (AtC)*. AtC counts the number of different classes that can occur as the type of a field declared in an aspect. More precisely, consider a class called *Address* is declared as the type of a field named *personalAddress* of an aspect. In this case, we would simply count the field as a member of the AtC metric as the `Address` class is the part of the field declaration. The following metric explicitly shows the number of field-class dependencies after filtering the collection which can contain primitive data types and interfaces as a type of fields that we do not want them in the result. For this reason, on line 8 and 10, primitive data types are excluded plus each selected type is checked whether it is an interface or not, respectively. As a result, the metric performs the number of field-class dependencies per project shown in the table below.

```
1 (defn AtC [?aspectName ?fieldName ?fieldTypeName]
2     "count the number of types of fields of aspects"
3       (l/fresh [ ?field ?aspect ?isSameInterface ?fieldType]
4             (w/type-field ?aspect ?field )
5             (succeeds (.isAspect ?aspect))
6             (equals ?aspectName (str "Aspect {"(.getName ?aspect)"}"))
7             (equals ?fieldType  (.getType ?field ))
8             (equals false       (.isPrimitiveType (.getType ?field )))
9             (equals ?fieldTypeName    (.getName ?fieldType))
10            (equals true ( nil? ( isInterface ?fieldTypeName)))
11            (equals ?fieldName (str "<Field name: " (.getName ?field) ">"))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AtC	15	4	0	12	1	4	19	2	0	8

Table 3.33: The number of field-class dependencies

- *Advice-Class Dependence (AC)*. AC counts the number of classes that can occur as the parameter or return type of an advice. For example, the `Address` class as mentioned in the AtC metric is in a role of the parameter type of a before advice and at the same time is defined as the return type of an around advice. For this reason, we would include the before advice and the around advice in the AC metric according to the connection of the `Address` class. Each project has been examined by the AC metric as shown in the table below.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AC	173	17	31	112	22	44	123	10	44	5

Table 3.34: Showing the results of AC over each project

- *Intertype-Class Dependence (IC)*. IC counts the number of intertype method - class dependencies if a class is defined as the parameter type of an intertype method or the return type of the intertype method in a given project. Imagine that an intertype method consists of one parameter type and one return type. The return type of the intertype method is the `Address` class and the parameter type of the method is `String`. Therefore, the intertype method suits for the IC metric and it counts as two members because two different classes are mentioned in the intertype method declaration. Inter type method-class dependencies are explicitly depicted by the following table.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
IC	0	0	19	6	0	0	43	6	0	0

Table 3.35: Showing the results of IC over each project

- *Method-Class Dependence (MC)*. MC is similar to the IC metric. The only difference from the IC metric is that the dependencies of MC correspond to method-class interaction in aspect declarations. We only examine method declarations declared within aspect modules. The number of dependencies are shown, depending on each column as follows:

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
MC	50	1	0	2	29	6	4	17	0	1

Table 3.36: Showing the results of MC over each project

- *Pointcut-Class Dependence (PC)*. PC counts the number of pointcut-class dependencies if a class is defined as the parameter type of a *named* pointcut declaration. We know that a named pointcut declaration syntactically forms with a pair of parentheses "( )". The metric examines inside of these parentheses in order to examine the pointcut's parameters. For example, see Fig. 2.1, there are three named pointcut declarations, one of which is named `Area` whose parameter type is declared as the `Circle` class. PC incorporates this pointcut-class (i.e. *Area-Circle*) dependency into its collection. Note that the metric seeks all parameter types declared in every named pointcut declaration regardless of the parameters coming from other pointcut constructs such as `args()`, `this()`, and `target()`. The dependencies are clearly described over each column in the table below. In order to find the dependencies the following snippet has been constructed as follows:

```
(defn PC [?typename ?pointcutname ?aspect]
  "counts pointcut-class dependencies"
    (l/fresh [?pointcut ?types ?type ?isInterface ]
      (w/pointcutdefinition ?point)
      (equals ?types (.getParameterTypes ?pointcut))
      (contains ?types ?type)
      (equals false (.isPrimitiveType ?type))
      (equals ?typename (.getName ?type))
      (equals ?isInterface (getInterface ?typename))
      (succeeds (nil? ?isInterface ))
      (equals ?aspect (.getName (.getDeclaringType ?pointcut)))
      (equals ?pointcutname (str "<Pointcut Name :"(.getName ?pointcut)">"))))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
PC	12	16	12	39	18	12	93	1	0	2

Table 3.37: The number of pointcut-class dependencies

- *Advice-Method Dependence (AM)*. AM counts the number of methods directly invoked within the body of advice in a given project. More clearly, consider a method named `getAccountID` declared in the `Customer` class is called within the body of an after advice in order to implement a new behaviour. Such an example is counted by the AM metric. Additionally, consider again this method called `getAccountID` overridden by another subclass of the `Customer` class is called in the body of an around advice, such overriding methods are also counted regardless of methods' parent. What is important to as well is that all library calls including internal and external libraries of each projects such as `System.out.println`, `String.concat` also are counted. The Ekeko query called AM describes the total number of called methods by invoking 2 different queries and one predicate named `advice-soot|method`. Next to the query, we show the dependencies occurred in the projects illustrated in Table 3.38.

```
(defn AM [?aspectName ?calledmethods ?soot|methodName]
"the number of method calls per advice body"
 (l/fresh [?soot|method ?advice ?aspect ?pointcut]
    (NOAdvices ?aspect ?advice ?pointcut)
    (ajsoot/advice-soot|method ?advice ?soot|method)
    (NOMethodCalls ?soot|method ?aspectName ?calledmethods ?soot|methodName)))
```

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
AM	323	71	96	100	192	179	345	37	52	34

Table 3.38: The results of AM

- *IntertypeMethod-Method Dependence (IM)*. IM counts the number of methods of classes directly invoked within the body of intertype methods in a given project. The difference between the AM and IM metric is that while the AM metric counts called methods in the body of advice, the IM metric performs only within the body of intertype methods. What is important not to forget is that the language (i.e. Java) and external libraries are included. The table below shows that the number of called methods is contained in each project.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
IM	0	0	105	1	0	17	72	40	0	0

Table 3.39: The results of IM

- *Method-Method Dependence (MM)*. MM counts the number of methods of classes directly called within the body of methods in a given project. What is important not to ignore about the metric is that it only takes into account methods declared in aspects. In addition, library calls are also counted in this metric. The number of called methods is represented in each cell according to project as follows:

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
MM	117	117	0	17	140	19	5	54	0	9

Table 3.40: Showing the results of method calls within methods

- *Pointcut-Method Dependence (PM)*. PM counts pointcut-method dependencies if a pointcut of an aspect refers to a method or constructor of a class. In other words, we obtain how many join point shadows of pointcuts of aspects relate to a method or constructor of a class using this metric. For example, we have a method named `getZip` in the `Address` class and one pointcut of an aspect which refers to this method once in the `Address` class (i.e. the method's class) through the pointcut's join point and thus PM simply subsumes this relationship as a pointcut-method dependence. Note that the method `getZip` could be an overriding method. That is, such methods are also taken into account by the metric as well. Fig. 2.1 shows another example for PM whose score is 1. The reason why the result is 1 is because only one method called `getArea` of the `Rectangle` class is referred by the pointcut of the after returning advice in the main method of the *same* class (i.e. `Rectangle`). The results of PM shown the table below demonstrates the dependencies according to specified projects.

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
PM	21	4	2	15	276	11	2	24	0	2

Table 3.41: Depicting the results of PM over each project

After detailing the meaning of each metric, we also illustrate the complete table as a summary as follows:

	AJHSQLDB	HyperCast	AJHotDraw	iBatis	Contract4J5	HeathWatcher	MobileMedia	SpaceWar	Telestrada	TetrisAJ
How large is the system?										
LOC	62692	51132	21046	11902	3740	5898	3870	1383	3947	849
VS	255	291	284	215	50	99	92	29	167	14
NOA	3146	2290	629	475	76	195	150	104	138	57
NOM	3915	2907	2278	1132	279	394	175	134	281	34
How often are AOP constructs used compared to OOP features?										
NOI	0	0	50	4	0	17	46	6	0	0
NOAd	106	25	47	120	17	73	77	13	46	21
Which AOP constructs are typically used?										
InA	3	4	0	5	7	11	11	2	2	0
SA	28	5	31	41	14	25	42	6	18	8
nSA	2	5	0	1	1	5	0	2	0	0
AAP	32	0	0	0	15	3	0	0	0	0
ABP	74	25	47	120	2	70	77	13	46	21
NOAr	35	3	18	14	7	42	29	1	13	6
NOBA	60	20	25	43	5	8	28	4	2	15
NOTR	11	2	4	63	5	23	20	8	31	0
NOC	55	19	23	39	1	17	20	24	7	12
NOE	124	7	24	128	12	86	58	0	55	7
AE	0	0	0	0	0	0	0	0	0	0
NOW	198	0	9	31	30	20	0	7	10	0
NOnW	21	19	25	134	0	76	78	17	47	21
AAd	1.11	0.44	0.23	0.38	0.18	0.16	0.58	0.23	0.17	0.38
AAda	1.68	1.1	1.22	3.29	1	1.09	1.36	1	1	1.14
How many types and members of a system are advised by AOP?										
AdC	0.95	0.16	0.12	0.41	0.6	0.57	0.38	0.43	0.13	0.83
nAdC	0.05	0.84	0.88	0.59	0.4	0.43	0.62	0.57	0.87	0.17
AdM	3801	10	26	142	124	181	27	10	37	9
nAdM	90	2897	2252	990	155	213	144	108	244	25
CsC	1	0	4	5	0	0	4	0	1	0
ScC	0.33	0	1.62	0.55	0	0	0.42	0	0.05	0
Is there a relation between the amount of coupling in an aspect, and how many shadows it advises?										
AJ	51.45	6.8	2.7	2.88	44.65	7.71	1.14	4.69	1.5	1.48
tJPS	22	0	3	2	13	3	0	5	0	3
MoA	0	0	0	0	0	7	0	0	3	0
AcA	152	11	11	25	4	39	64	8	10	20
AnP	22	3	2	3	0	5	0	0	0	2
How many dependencies are there between classes and aspects?										
AtC	15	4	0	12	1	4	19	2	0	8
AC	173	17	31	112	22	44	123	10	44	5
IC	0	0	19	6	0	0	43	6	0	0
MC	50	1	0	2	29	6	4	17	0	1
PC	12	16	12	39	18	12	93	1	0	2
AM	323	71	96	100	192	179	345	37	52	34
IM	0	0	105	1	0	17	72	40	0	0
MM	117	117	0	17	140	19	5	54	0	9
PM	21	4	2	15	276	11	2	24	0	2

Table 3.42: The outcome of the metrics, depending on the ten AspectJ projects

CHAPTER 4

Discussion

In this section, we go one step further to present the findings of the measurement process. The absolute data have been formed based on the set of our metrics considering ten targeted AspectJ projects (Sec. 3). The discussion part is organized in six individual parts due to our research questions to observe each question in depth.

4.1 How large is the system?

The corresponding metrics of the question are:

- *Lines of Code (LOC)*
- *Vocabulary Size (VS)*
- *Number of Attributes (NOA)*
- *Number of Methods (NOM)*

We now evaluate the activities of how the architecture of each aspect-oriented system combines OOP and AOP features in concert. Fig. 4.1 visually depicts the size of the source code relying on the systems respectively. The distillation of the metric is about counting all the implementations that shape the systems' structure. In addition, software complexity can be estimated using LOC in a correlative manner which means if a system has a large quantity of source code, the overall complexity of this system can increase as well. According to the obtained values per project, half of the systems are below 5000 lines of code. Two of the systems (i.e. AJHSQLDB, HyperCast) have high values which can be inferred that both systems might have higher complexities than the others, plus developers might focus both aspect-oriented systems on more code in order to understand the implementation of their functionalities.

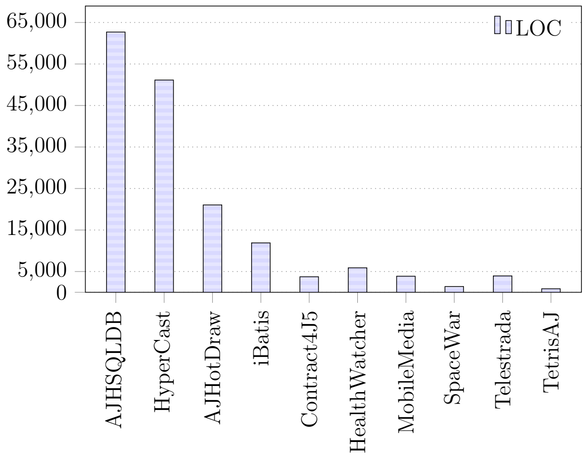


Figure 4.1: Collected values of the LOC metric per project

Fig. 4.2 shows the number of modules (classes and aspects) of each system. Even though, AJHotDraw is in the third place due to the result of LOC, it is the second most complex project in terms of VS. On the other hand, the six applications, iBatis, Contract4J5, HW, SpaceWar, Telestrada, and TetrisAJ have approximately the same activities with respect to LOC. In other words, the volume of their modules does not show an unexpected size, compared to Telestrada and AJHotDraw and HyperCast. Fig. 4.3 depicts the number of attributes (fields) and methods. On average, the four largest projects are AJHSQLDB, HyperCast, AJHotDraw, and iBatis. They might likewise include more complex functionalities in terms of the three graphs. Such impact might affect external attributes of the systems as well such as maintainability, and reusability if perceiving of the entire systems are tedious. Our general question can be varied due to the total components of the systems. For example, while four out the ten targeted systems (i.e. AJHSQLDB, Hy-

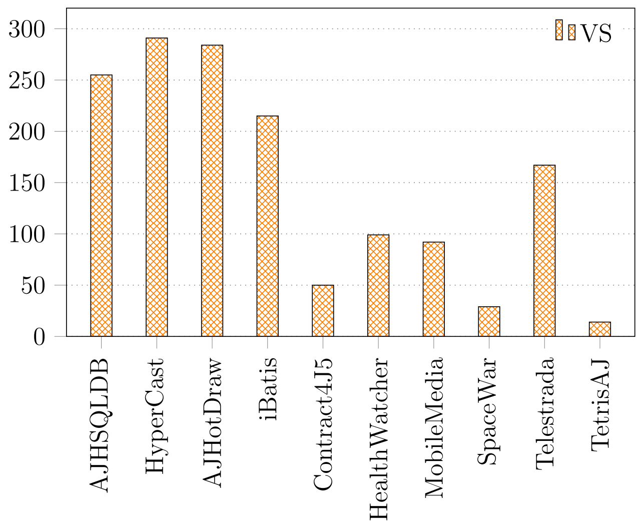


Figure 4.2: The number of modules per system

perCast, AJHotDraw, and iBatis) consist of large amount of capabilities, the answer can be interpreted as these systems potentially have a realistic size, plus they have more responsibilities between their modules. On the other hand, the Contract4J5, HW, MobileMedia and Telestrada systems include the average size of components, their essence of the systems' behaviour are yet non-trivial, whereas their functionalities are not as complex as against the former systems.

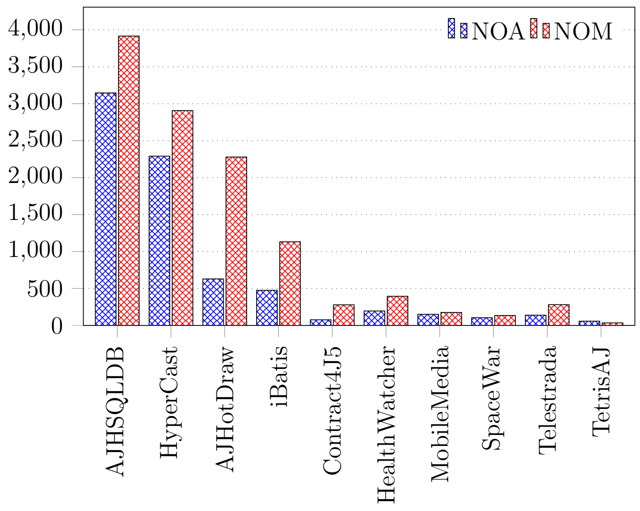


Figure 4.3: The number of NOA and NOM per system

4.2 How often are AOP constructs used compared to OOP features?

The metrics (1, 2) assigned with this section and related metrics (3, 4, 5, 6, 7) from other research questions are as follows:

- *Number of Intertypes (NOI)*[1]
- *Number of Advice (NOAd)*[2]
- *Number of Attributes (NOA)*[3]
- *Number of Methods (NOM)*[4]
- *Singleton Aspects (SA)*[5]
- *Non-Singleton Aspects (nSA)*[6]
- *Inherited Aspects (InA)*[7]

To answer this question, we make use of the NOA, NOM, SA, nSA, and InA metrics. In other words, the second part of the discussion combines several metrics besides intertype method declarations and advice declarations. The results of the corresponding metrics including NOI, NOAd, and the number of aspects (i.e. SA + nSA) are presented in Fig. 4.4. Only half of the systems include intertype methods which means only these systems were affected by intertype methods imposing additional actions on some types as a new member. Next to NOI, advice were widely used in every system especially in AJHSQLDB, iBatis, HW, and MobileMedia. A large number of advice declarations refer to a large number of new behaviours for a system. In our case, either of the systems (AJHSQLDB, iBatis) merely contains more than 100 advice which indicate these systems can be potentially most exposed ones.

While NOA and NOM give a rough idea about how large a system is, using these metrics we can also estimate AOP constructs as both AOP and OOP construct the results of these metrics through our targeted systems. In addition, the combination of SA and nSA give us an exact number of aspect declarations depending on each system. Half of the systems (AJHSQLDB, AJHotDraw, iBatis, HW, and MobileMedia) contain more than or equal to 30 aspects. The growth activities of advice vary regardless of how much aspect modules a system contains. That is, it cannot be implied that the small number of aspects in a system among systems do not potentially have a small number of advice. Note that both the systems' functionalities and size are not identical. While one can solve a crosscutting concern with a few advice, other one might need more advice for more than one concern. On the other hand, InA can give an approach about concluding our question because we now know which aspects are subaspects and which are not. This feature might be useful in order to find different AOP features over OOP. In consequence, if we compose the results of our metrics, we can conclude that AOP constructs are used much less than OOP constructs because OOP is still overwhelming in the entire systems.

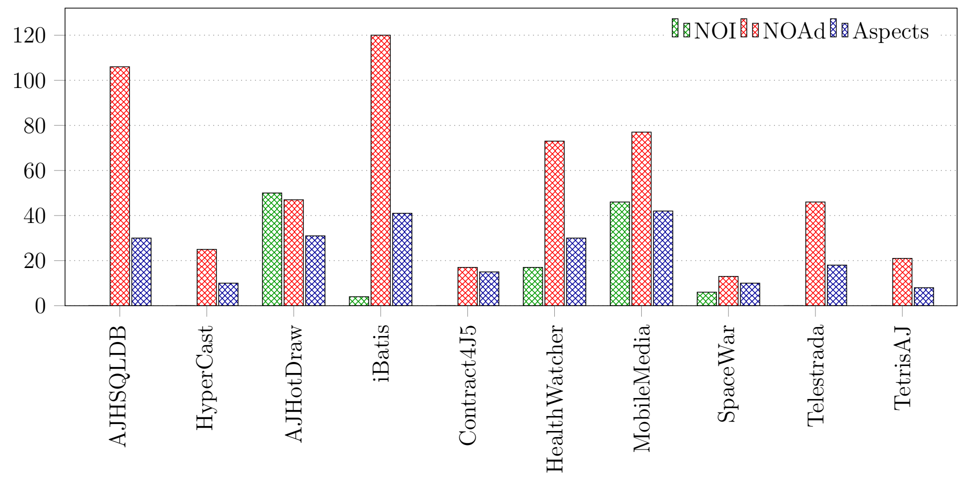


Figure 4.4: The number of AOP constructs

4.3 Which AOP constructs are typically used?

The corresponding metrics of this question are:

- *Inherited Aspects (InA)*
- *Singleton Aspects (SA)*
- *Non-Singleton Aspects (nSA)*
- *Advice-Advanced Pointcut Dependence (AAP)*
- *Advice-Basic Pointcut Dependence (ABP)*
- *Number of Around Advice (NOAr)*
- *Number of Before/After Advice (NOBA)*
- *Number of After Throwing/Returning Advice (NOTR)*

- *Number of Call (NOC)*
- *Number of Execution (NOE)*
- *Number of AdviceExecution (AE)*
- *Number of Wildcards (NOW)*
- *Number of non-Wildcards (NOnW)*
- *Argument size of Args-Advice (AAd)*
- *Argument size of Args-Advice-args (AAda)*

In the third section, we identify which crosscutting constructs are commonly used and which are rarely used. Following this, the most and least dependable ones will be clarified among AOP constructs. Additionally, other information about a variety of pointcut designators, wildcards usage, and states of aspect associations are observed step by step. We also start obtaining some rough ideas as to how modular reasoning was achieved, depending on certain values in this section. Fig. 4.5 shows the number of concrete (or sub) aspects extending to an abstract aspect. When a system includes abstract aspects, its crosscutting units can be more reusable. The majority of our systems have at least two concrete aspects. We might mention that the most reusable crosscutting units of abstract aspects are used in both HW and MobileMedia. The two systems, AJHotDraw and TetrisAJ do not include any abstract aspect such that their systems do not contain a reusable pointcut declaration (i.e. abstract pointcut).

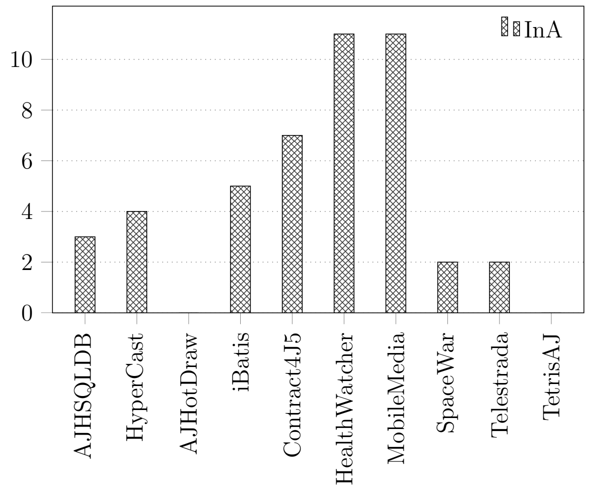


Figure 4.5: Inherited aspects

Fig. 4.6 designates the changes of singleton aspects and non-singleton aspects for each system. The majority of aspects are singleton in the whole figure. Only certain systems (AJHSQLDB, HyperCast, iBatis, Contract4j5, HW, and SpaceWar) provide less than or equal to five different kinds of aspect associations. The majority of associations were defined as default aspects. Such that, there may not be any unique or certain crosscutting functionality for different purpose on these systems.

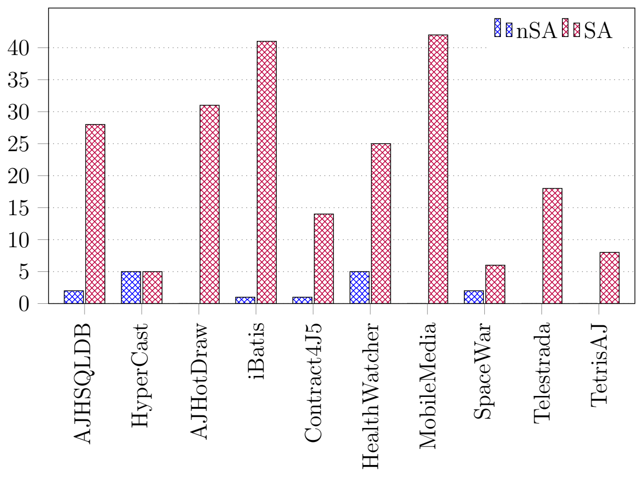


Figure 4.6: Showing each aspect association

We now compare advanced and basic crosscutting constructs (i.e. pointcuts) that are connected with the obtained advice per system. We expected at least half of the systems include advanced pointcut constructs, whereas only three aspect-oriented systems: AJHSQLDB, Contract4J5, and HW do. A larger number of advanced pointcut constructs makes the modularity of these systems more complex because it is more difficult being aware of specified advice through members of exposed modules. On the other hand, the latter systems that do not contain any advanced pointcut construct use basic pointcut constructs in order to interrupt program elements directly. Fig. 4.7 suggests that AspectJ's basic pointcut constructs are sufficient for the majority of projects we studied. Using basic declarations facilitate modules to realise each of the related crosscutting actions.

Each triple of bars shown in Fig. 4.8 merely exhibits the distribution of advice based on their advice kind (*before/after (returning/throwing)/around*). At first glance, we could say that *before/after* advice have high values than the other two metrics in total. This means before/after advice are often used in order to perform extra *pre-* and/or *post-* actions. It seems that each system contains one or more around advice (the highest one includes in HW) in order that those advice might bypass the original action and applies al-

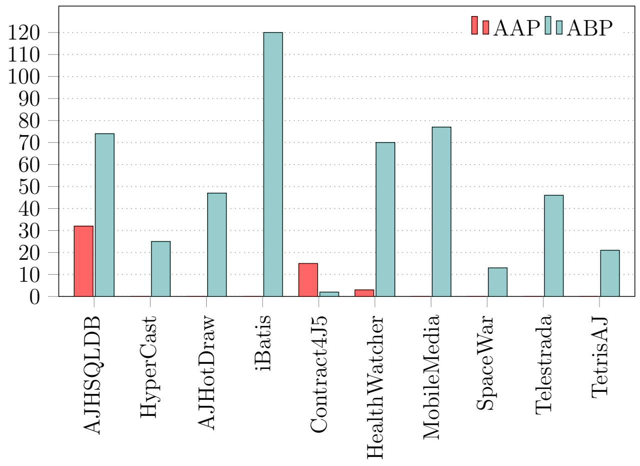


Figure 4.7: Quantity of pairs including advanced and basic pointcut constructs

ternative behaviour, might execute an alternative action before or after the selected join point, or might enclose the base operation with an exception handler. The only metric, NOTR, did not find a related advice in TetrisAJ. The system either does not connect with an advice that captures an exception or does not need a return value of an advised behaviour.

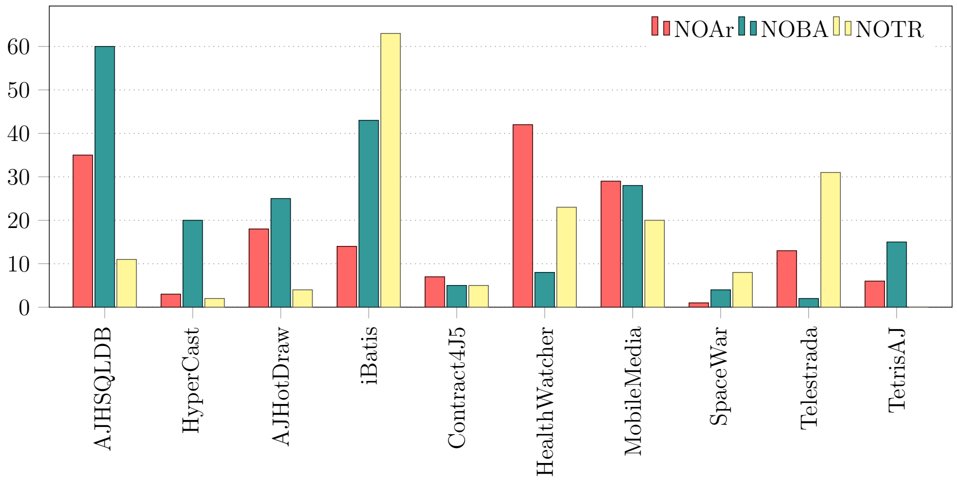


Figure 4.8: Showing a diversity of advice in each triple of bars

After observing the volume of advice, we now discuss about *call* and *execution* pointcut constructs. Fig. 4.9 compares the two types of pointcut constructs per system respectively. Provided that the number of pointcuts consisting of call and execution are high in a system, a wide range of the system constructs (i.e. methods or/and constructors) might be interfered in an unexpected way. In our scenario, the second potential interfering constructs are contained in AJHSQLDB after iBatis in terms of NOE. Conversely, AJHSQLDB is superior with respect to the call size. Both iBatis and AJHSQLDB have similar quantity. Plus, both systems have an abundance of call and execution pointcut constructs. We also expect that majority of the advised units such as members and classes will arise from both. In addition, besides both projects, HW is a potential third system that can be expected reasonable values too.

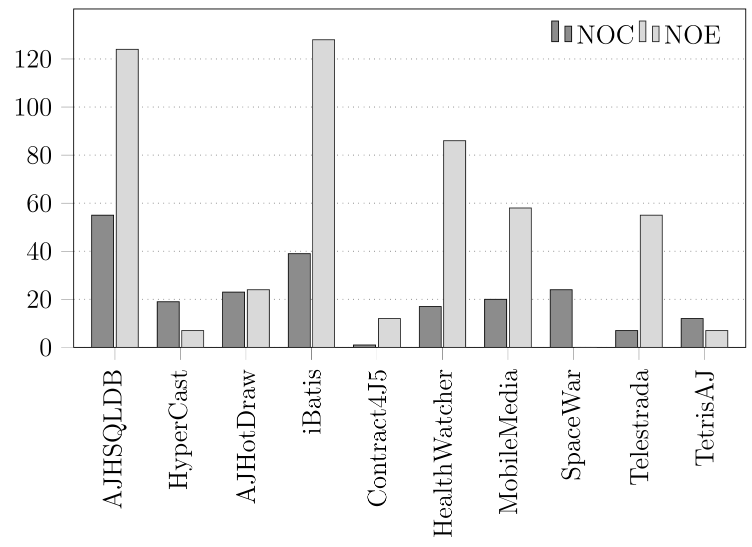


Figure 4.9: Comparing call and execution pointcut constructs

Additionally, while we measured all the metrics based on the systems, we surprisingly noticed that none of the systems uses *adviceexecution* pointcut in their crosscutting modules.

We can now infer some rough ideas about the use of wildcards when comparing NOW and NOnW, respectively. Fig. 4.10 shows the growth and decline of the metrics in terms of the systems. Recall that both metrics only examine the class name of the determined signature patterns. It is obvious that the majority of systems using wildcards are found in AJHSQLDB in terms of NOW. That is to say, in this system, the invocations of aspect modules are spread over the different modules and its join points might include multiple behaviour containing different advice. In a way, this system might involve more common characteristics of program elements than the systems that include less NOW. About NOnW systems generally simply capture a set of join points directly matched with completely referred types (i.e. without using wildcards in types). Accordingly, iBatis has an high quantity of NOnW than the other results of the systems. HW and MobileMedia follow the highest one respectively about completely named classes. Only one system, Contract4j5, contains full of signatures declared with uncompleted names which it can be a plausible way for the system propose in order to fully construct this framework to support DbC. On the other hand, NOnW establishes the three systems' pointcuts that do not include any wildcard implementation inside namely HyperCast, MobileMedia, and TetrisAJ.

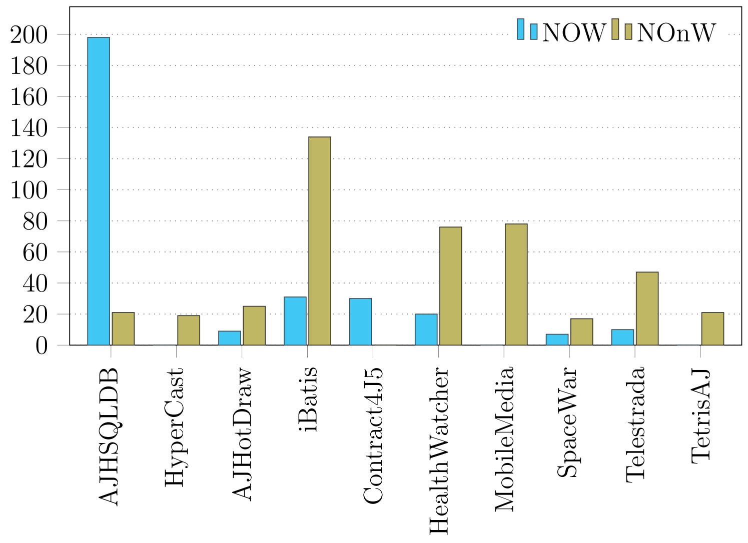


Figure 4.10: Depicting NOW and NOnW

Fig. 4.11 shows the obtained values of both AAd and AAda by examining args pointcut constructs of each system's advice. In order to compare the two metrics as their results base on args pointcut constructs, we put them together. There are interesting results of the average of the argument size of AAd and AAda that they both have low values. The majority of advice with args pointcut constructs give unpromising values in terms of AAd. That is, most advice are not conducted with args and only AJHSQLDB passes 1 which could mean that most args pointcut constructs consist of at least one argument or some args pointcut constructs have more than three or four arguments. If we take a look at AAda as indicated with the yellow bars, we obtain that the majority of advice with args pointcut constructs bind 1 argument. AJHSQLDB is also above 1 over AAda which proves our assumption about the argument size of args pointcut constructs partially. Following this, iBatis demonstrates itself about the average that its args pointcut constructs are the enormous one compared with the other systems, plus it is obvious that its most advice included in the calculation have more arguments.

Ultimately, our metrics try to help in perceiving a close response to the question. It is obvious the question has no any unique answer as the ten systems' functionalities and purpose are different from one another. However, the answer to this research question could be that in total, singleton aspects (i.e. *default* aspects), *around/after/before* advice, basic pointcut constructs as well as one of the basic pointcut constructs: *execution* pointcut are typically used among AOP constructs. This result brings equally some ideas about the process of AOP. For example, an implementation of AO language can only focus on a particular necessity with regard to the precedence of constructs, or the existing analysis tools of AOP can be improved to evaluate aspect-oriented systems based on the most accurate constructs.

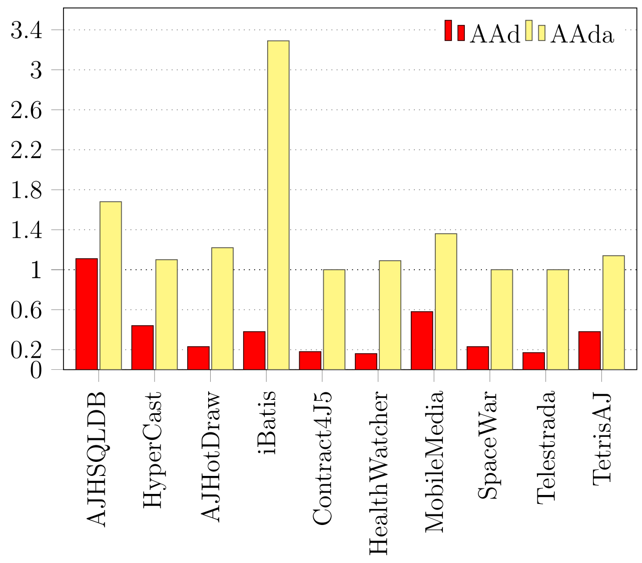


Figure 4.11: Analysing args pointcut constructs with different perspectives per system

4.4 How many types and members of a system are advised by AOP?

The corresponding metrics of this question are:

- *Percentage of Advised Classes (AdC)*
- *Percentage of non-Advised Classes (nAdC)*
- *Number of Advised Methods (AdM)*
- *Number of non-Advised Methods (nAdM)*

- *Classes and Subclasses (CsC)*
- *Average of Subclasses of Classes (ScC)*

In this section we discuss about our fourth research question. The following figures focus on members including fields and methods and types of the systems that are advised by AOP. Fig. 4.12 shows how many classes are advised and how many classes are not. Each stacked bar shows the fluctuation of how aspects interrupt the conventional modules using crosscutting units. As we expected, the most invocations would take place in AJHSQLDB. Almost all class declarations of this database system connect with at least one aspect declaration. Recall that this system itself supports more crosscutting concerns than the rest systems (Sec. 2.2). The major crosscutting concern of this system is exception-handling mechanism among the concerns. The aspect module that is responsible for exception handling is spread over the entire system. What we can also infer is that the class declarations themselves are potentially more prone to be abnormal termination and more clarity the majority of program elements attached to this handler aspect module. In addition, not just exception handler but tracing and pooling units on the other hand follow the exception handler aspect with regard to the size of join point shadows in order to connect with different or same classes. These results nearly shape about how AJHSQLDB becomes aspect-oriented and how its classes become an advised type.

We can also estimate other information based on AdC. In terms of the results of AdC, the affected classes have at least one join point shadow where a new action rises. The six-tenths of the systems are below 50%. This proportion might denote the included systems are not completely fully surrounded with aspect-oriented behaviour. For the rest, the four-tenths are composed of AJHSQLDB, Contract4j5, HW and TetrisAJ. Although TetrisAJ is the one of the small applications among the collection, it holds advised classes as the second most encompassed system by AOP.

Apart from this, Fig. 4.13 depicts how many methods correlate with join point shadows. Due to the large number of methods of AJHSQLDB, HyperCast, and AJHotDraw, the figure is not accurate enough as to systems whose data are less than 300. These output might affect the average of AJ with a high or low number pertaining to the systems. We merely spot that the following systems: AJHSQLDB, iBatis, Contract4J5, and HealthWatcher have a distinguishable data compared to the latter systems. The returning values of AJHSQLDB do not surprise us as it hold the superiority of AdC among the targeted AO systems. Based upon the advised classes of Contract4J5 and MobileMedia, their advised methods reach an acceptable level

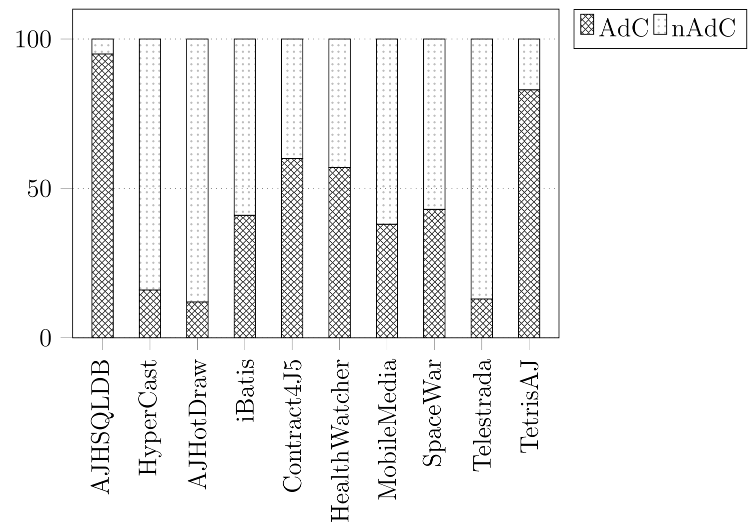


Figure 4.12: Percentage of advised against non-advised classes

in their own systems. Hence, their system might be nearly controlled by means of aspects in terms of AdC and AdM.

On the other hand, half of the systems do not include any class not including any subclass in terms of CsC. It is obvious that the majority of the systems do not have a super class. The highest one, iBatis, brings only 5 classes might affect their related subclasses. This is a rough result that the obtained classes may attach weight to the hierarchy of their connections. Following to CsC, we observe what the average of subclasses of classes among the findings is. Most systems do not give much more details about the average of subclasses (ScC). The obtained proportions could say that subclasses obtained by ancestors in the selected pointcuts do not participate too much over the hierarchy of the specified systems.

What is important to realise is that we mention the above-mentioned results in order to clarify the research question. Based on Fig. 4.12, we can only designate what the amount of advised types could be in practice. The answer indicates that AOP is attached to the systems' types in average manner. If we combine Fig. 4.12 and 4.13, our answer also combines the systems' methods roughly. In that case, it seems to be the systems' methods and constructors are affected less by AOP and our final idea might be that AOP is not dominant yet with regard to interrupting the OOP features when we unify all the specified metrics. Last but not least, one system out of ten (i.e. AJHSQLDB) has a privilege which means only this system is technically under the influence of AOP concept.

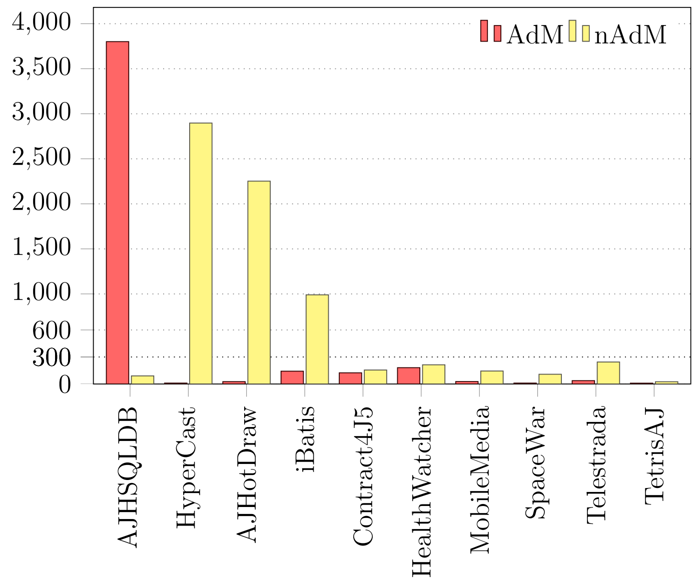


Figure 4.13: The effect of aspects over methods and constructors

4.5 Is there a relation between the amount of coupling in an aspect and how many shadows it advises?

The following metrics are considered during this part:

- *Advice-Join Point Shadow Dependence (AJ)*
- *Number of thisJoinPoint/Static (tJPS)*
- *Number of Modified Args (MoA)*
- *Number of Accessed Args (AcA)*
- *Around Advice - non-Proceed Call Dependence (AnP)*

The interactions of aspects between the amount of coupling are obtained in this part. Depending on our metrics we will observe different perspectives only for the specific properties included the implemented aspect modules per system. Fig. 4.14 shows what portion of advice-join point shadow dependencies of each system involves. We can notice how tight the coupling is between advice and their join points in terms of the figure. The minority of join point shadows are above 40% barely including AJHSQLDB and Contract4j5 such that their advice reach a wide range of members of classes. As we mentioned a few details about the system called AJHSQLDB in the previous section, the system reaches a wide variety of program elements. This result does not surprise us. Even the medium-size system, Contract4J5, has high values as we also saw that the previous table found sufficient numbers based on exposed methods. As long as a system provides high output of AJ, its structure is interfered with more pointcuts as well as its program element might be exposed with multiple pointcuts at determined points in which several interruptions impose on such element as it eventually contains additional behaviours. The small portions must not be always interpreted as invocations are not spread too much over the systems (i.e. SpaceWar and TetrisAJ). Note that these two systems have few lines of source code. That is, their output can be yet effective in their own way. Additionally, the highest values of certain systems especially for AJHSQLDB, HyperCast, Contract4J5, HW, and SpaceWar can denote the contracts of their advice might be difficultly adhered with the related method calls.

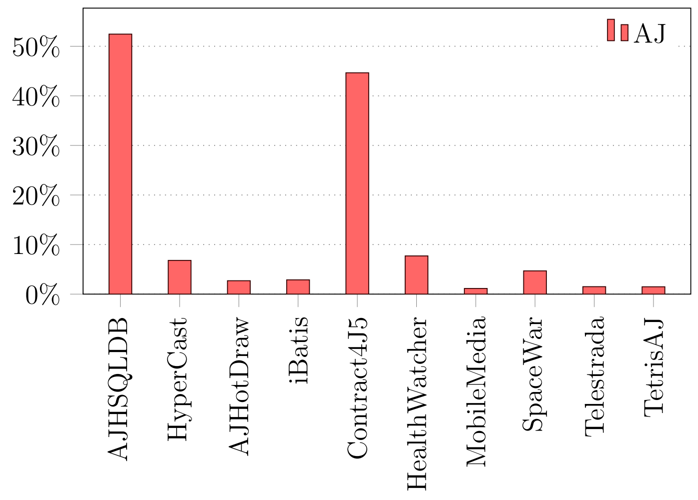


Figure 4.14: Advice-join point shadow dependencies

We next observe in Fig. 4.15 whether each advice declaration uses any of the two special objects, `thisJoinPoint` and `thisJoinPointStatic`, in order to access the dynamic or/and static information of an advised join point. Either two objects are rarely used in the body of advice of the systems. Even HyperCast, MobileMedia, and Telestrada do not incorporate these objects. They do not make use of these utilities that AspectJ offers. Moreover, they might use different ways like different pointcuts to obtain some collection for specific purposes or they never have a specific approach. Using any of two terms in advice bodies can cause such advice to be less reusable, as well as their advice can be cumbersome (e.g. AJHSQLDB and Contract4J5).

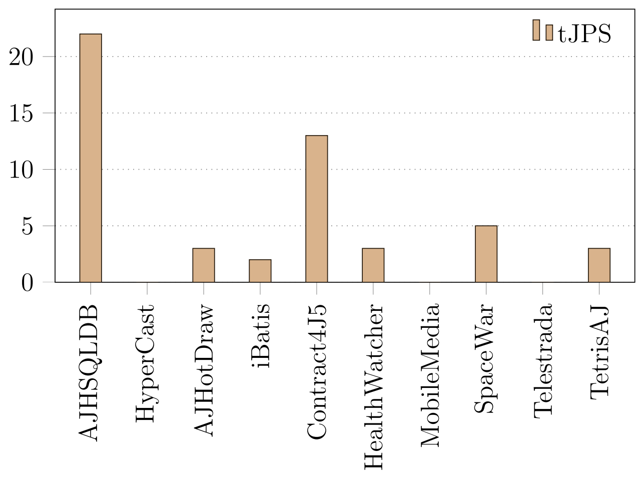


Figure 4.15: The uses of `thisJoinPoint/Static`

Fig. 4.16 compares two metrics through the systems respectively. The majority of args pointcut constructs are used to access context information. Only several specified args pointcut constructs are used to modify their own system's context. HW and Telestrada are part of the mentioned concept closely. A modified argument can influence the system flow in unexpected way or affect the core modules' functionality if it is returned from an advice (i.e. an around advice) in a changed manner. Most arguments by contrast are made use of initiating a new behaviour by accessing their values.

Our last observation is concerned with AnP, the number of around advice-non-proceed call dependencies. The output is quite low in terms of AnP, although some systems (AJHSQLDB, HW, and MobileMedia) include more than 20 around advice. Around advice not including a proceed call are used often in AJHSQLDB. These advice never pass the original behaviours, all of which are bypassed directly without making the original ones aware of. They are used as a new behaviour in aspects as well as in systems. These numbers in terms of the systems especially for AJHSQLDB affect the core actions to be unaware of any changes.

Finding approximate answers to the fifth question can be done with a few evidence. If we overlap the corresponding metrics per system, we can depict that aspect modules are likely used for different activities. One system use only aspects to access join points' arguments, not to bypass any behaviour, and to mention special objects (e.g. for logging), other system uses them for both accessing and modifying arguments and for bypassing the original elements multiple time in order to fully perform new crosscutting behaviour. Note that, the metrics except for AJ are considered in advice bodies. Especially, large number of results can convey high relations as well as high coupling within advice. In a way, we also spot which advice of systems are more correlated with rarely used AOP features. As a result, we can say that there is a connection between the amount of coupling in aspects in certain systems such as AJHSQLDB, Contract4J5, and HW since their data are more promising.

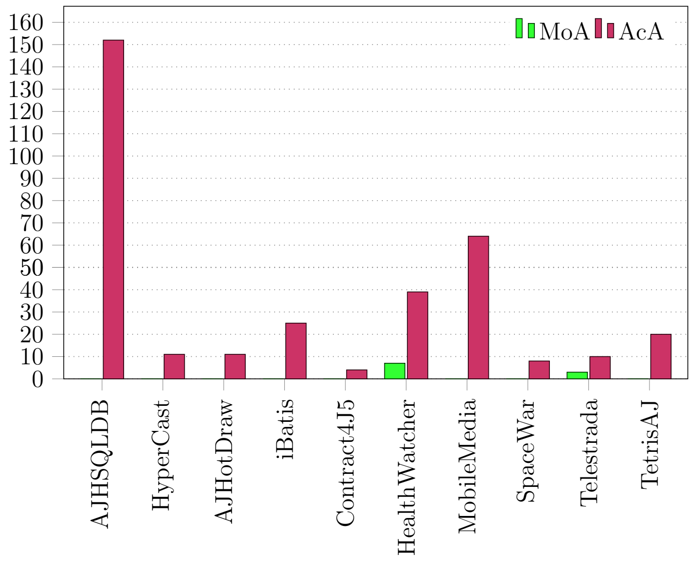


Figure 4.16: Comparing MoA and AcA

We also examined the advice with a large[1] amount of join point shadows by measuring the tJPS, MoA, AcA, and AnP values for these advice individually. Based on these results, AJHSQLDB, Contract4J5, HyperCast, and iBatis are the only survivers with this choice. We hardly ever catch a small number of advice including 10 advice (i.e. AJHSQLDB(5), Contract4J5(2), HyperCast(1), and iBatis(2)). More clearly, 3 advice out of 10 only contain any of the two special objects. Plus, while none of the advice uses arguments for modifying their context, we found 7 accessed arguments in total. About the AnP values, the majority of advice declarations consist of 6 around advice and only 4 around advice out of 6 ignore their connected join points. At long last, We can conclude that the determined advice show a very small degree of coupling.

On the other hand, we obtain the number of join point shadows per system with regard to its advice size. Specifically, we found that AJHSQLDB

[1]In our case, a large amount of join point shadows refer to the advice with more than 50 join point shadows.

includes 5454 join point shadows which means this system has the most largest amount of join point shadows among the whole selected systems. Next to this very large number, we retrieve the join point shadows for Contract4J5, HW, iBatis, HyperCast, AJHotDraw, MobileMedia, Telestrada, SpaceWar, and lastly TetrisAJ which are 759, 564, 346, 170, 127, 88, 69, 61, and 31 respectively.

4.6 How many dependencies are there between classes and aspects?

The corresponding coupling metrics are as follows:

- *Attribute-Class Dependence (AtC)*
- *Advice-Class Dependence (AC)*
- *Intertype-Class Dependence (IC)*
- *Method-Class Dependence (MC)*
- *Pointcut-Class Dependence (PC)*
- *Advice-Method Dependence (AM)*
- *IntertypeMethod-Method Dependence (IM)*
- *Method-Method Dependence (MM)*
- *Pointcut-Method Dependence (PM)*

This is the last subsection of the discussion that is concerned with coupling between classes and aspects. There are 9 metrics that convey what the scale of coupling is among both conventional and crosscutting modules under this research question. Achieving of internal software attributes in a good way is the desirable goal of every software system. One of the internal attributes called coupling is one of the key concepts of software development. In addition, if the rule of coupling is precisely applied in a system, this system can now obtain better values for external software attributes such as maintainability, reusability, and reliability. Moreover, reducing maintenance effort can lead to improving software quality.

In order to project all the aforementioned facilities in a system, the principle of low coupling must be taken into consideration. The tradeoff must be low between aspects and classes in order to minimize their data flow. References of classes in field declarations of aspects are not intensive in the whole systems in terms of AtC. Following this, AC expresses the advice-class dependencies in terms of our systems. The majority of dependencies are either greater than or roughly equivalent to the number of advice. For example, the obtained coupling values of AJHSQLDB, Contract4J5, and MobileMedia are greater than their advice declarations. They contain more tight dependencies than the others. The next metric, IC, does not apply to most systems, as they do not contain any intertype method declarations. Recall that intertype methods represent additional capabilities for the attached types. The increasing of the IC dependencies should be low as their transactions are part of the existing types as well.

Differently, the MC metric only examines the structure of method declarations within aspects. Only AJHSQLDB has more than 30 method-class dependencies. The rest are not bothered with a large number of dependencies. Besides MC, we look into the volume of pointcut-class dependencies within aspects. The large/small number of PC affect the attached advice declarations adversely because we look only named pointcuts. That is, all the mentioned classes are likewise mentioned within the parameter types of advice declarations when connected one another. Fig. 4.17 present the 3 comparable metrics in order to discover the number of interfering methods among advice, intertype methods and methods. According to the data, each system more or less contains method calls which moderately identify how complex the overall advice bodies of systems are. The quantity of high dependencies obstruct the understandability of existing advice, plus it may require more effort to change and correct interrelated statements within advice. Apparently, the most functional advice come with AJHSQLDB, MobileMedia, Contract4J5, and HW based on the AM metric. We can also think about adding the small systems (i.e. SpaceWar and TetrisAJ) into this pool. The green bar does not catch useful evidence while most systems lack intertype method declarations. However, we should not underestimate the values of IM which five systems still contain some information about exposed method calls. The most intricate intertype methods reside in AJHotDraw. We also acquire an idea about method-method dependencies by examining the last compared metric. The data of MM comes after the AM metric as a second. The dependencies are inferior for method declarations, plus might not be a detrimental effect for aspect themselves.

On the other hand, we have the proper determined results including about the PM metric. Three systems out of ten have slightly the identical values, only one system, Telestrada, did not include any information. Only Contract4J5 has a notable feedback. In particular, it depicts that in Con-

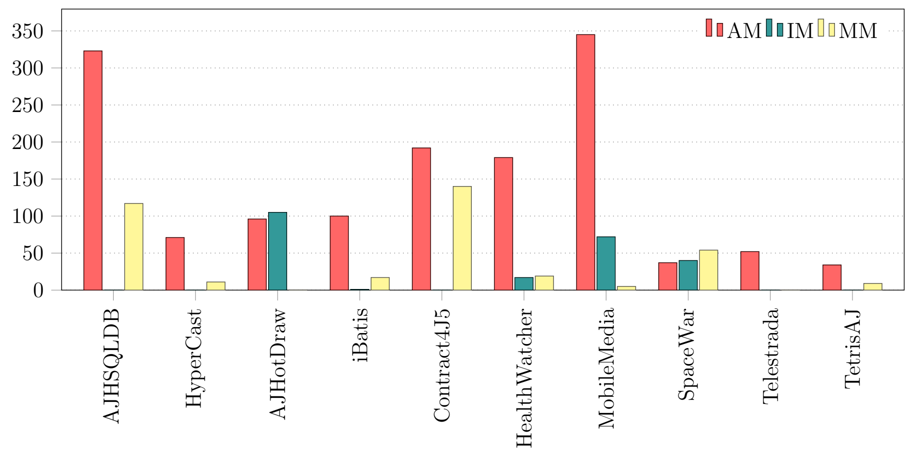


Figure 4.17: Coupling consisting of advice, intertype methods and methods

tract4J5, most join points refer to methods or constructors of classes within the same related classes, compared to other systems' join points. A high value of pointcut-method dependency might designate that the corresponding classes involve more interruption between their child and the attached advice. Consequently, since we have different size of aspect-oriented systems along with their different aim, plus low coupling is more acceptable for a system, we observe that the results of the coupling metrics for each system differ considerably. The number of coupling reach high numbers (over 650) in total for AJHSQLDB, Contract4J5, and MobileMedia projects. The coupling of the third-ten targeted systems namely AJHotDraw, iBatis, and HW are between 200 and 300. Finally for the rest of the systems: HyperCast, SpaceWar, Telestrada, and TetrisAJ remain below 200.

CHAPTER 5

Related Work

While AOP aims to improve separation of concerns, it might complicate the software analysis and development. In order to evaluate and examine the general internal attributes of AOP systems, various researchers have analysed aspect-oriented systems by comparing them to the object-oriented version of the same systems, by refactoring the existing object-oriented systems, or by assessing widely-accepted metrics. In this section we mention a number of papers that fulfil these efforts.

The research by Sant'Anna et al. [34] has included an empirical assessment which indicates the usefulness of the proposed metrics as displayed the strengths and weaknesses of two case studies in respect of separation of concerns, coupling, cohesion and size. In other words, the authors' metrics depict to reveal the reusability and maintainability of software systems. We have revised the three main metrics, LOC, VS, and NOA in order to measure the overall quality of software system's architecture and code. A paper by Zhao [44] has proposed a metric suite consisting of various metrics so as to compute the coupling dependencies between aspects and classes in aspect-oriented systems. In a way, his research is to uncover the basic idea of coupling in terms of aspects and classes by defining coupling measures in response to different types of dependencies. Although introducing the new coupling measures for aspect-oriented systems, the paper does not conduct an empirical evaluation. However, in our paper, we have fulfilled one of his future works on applying all the metrics defined by the author to our aspect-oriented projects. Plus, we have only revised the behaviour of three metrics out of nine, namely the AM, IM, and MM metrics in our metric framework in that we can obtain an idea about a number of the most exposed method calls as well as a number of interfering methods among advice by comparing different systems.

Ceccato and Tonella [18] have mentioned their study by building a set of metrics, most of which have been adapted from existing OOP metrics so as to observe the effects of AOP approach. That is to say, their purpose is based on investigating several kinds of coupling by means of aspect-oriented systems compared to object-oriented systems over the metrics in more detail. One of their metrics called CIM has a similar behaviour with our metric called NOnW. Although CIM collects completely named classes in the pointcuts as NOnW does, it also considers interfaces, plus we did not incorporate it into our metrics, because we normally have the NOW counterpart in order to perceive the differences between one another. Similarly to us, they also achieve the output of the metrics by writing a query, but the query is implemented on a relational database tool. Burrows et al. [15] have devoted their work for a new coupling measure called BAC that assesses specific dependencies between aspect and classes from several releases of an aspect-oriented system, Moreover, they investigate the results of several dependencies, depending on existing coupling metrics defined in [18] and their new metric. The BAC metric has been also evaluated by comparing it against the existing measures in order to approve its presence. Similarly to us, we incorporate the relevant aspect-oriented project called *iBatis* into our aspect-oriented project pool for analysing our entire research questions.

Kulesza at al. [29] have presented a quantitative case study that analyses some internal software attributes: coupling, conciseness and separation of concerns. The quantitative case study relies on a set of metrics mentioned in [34]. Their work relies on assessing the benefits and drawbacks of AOP techniques by comparing between the object-oriented and aspect-oriented version of the same project called *HealthWatcher*. Next to the latter paper, Ali et al. [12] have introduced a compelling systematic literature review of comparative evidence of AOP based development. They discussed the benefits and limitations of aspect-oriented systems over 22 papers in order to clarify some criteria that software systems face such as modularity, code size, performance, resolvability, cognition, and language mechanism. In their review, in order to collect such criteria, they made use of the data of several AOP metrics suites [18, 34, 44] adapted by the reviewed papers. Research about detecting program comprehension was performed by Feigenspan et al. [22] in order to evaluate how software measures and program comprehension correlate in terms of two separate versions (i.e. AspectJ and Java) of the *MobileMedia* project. Their experiment was conveyed in terms of three categories: complexity measures, size measures and concern measures. These categories made for understanding the efficacy of source code such as complexity, lines of code, attributes and operations (i.e. methods) was delineated depending on the research approach. Another study is about modularizing exception handling code of a system called *Telestrada*, demonstrated by Filho and Rubira [17]. They assessed whether AOP reinforces such modularizing exception handling code over the set of metrics proposed by [34] so as to obtain a good enhancement in internal software attributes such as separation of concerns, coupling, cohesion and conciseness. After evaluating both the original and refactored version of the system, they obtained that AOP facilitates handler reuse and improves separation of concerns.

The work on analysing AspectJ projects in [13] is closely related to our work. The quantifiable questions are asked to answer the main question, which is how the AspectJ language is used in eleven aspect-oriented programs. The representative values were illustrated with certain graph diagrams pertaining to the selected aspect-oriented projects, some of which are included in our paper namely *HealthWatcher*, *Tetris*, *AJHotDraw*, *AJHSQLDB*, and *Hypercast*. The empirical results are collected using a metrics suite including three metrics separating crosscutting concerns into three different sections as homogeneous/heterogeneous, static/dynamic, and basic/advanced for their objectives. Godil and Jacobsen [24] have applied AOP to the *Prevayler* system (i.e. the Prevayler main memory database management system) by refactoring the existing object-oriented version of this system. Using the number of metrics depicted in [34], the two systems consisting of OO and AO versions were studied in order for a question that is *"can modularity be achieved through refactoring"* to reveal quantitative results which denote the most important attributes (e.g. cohesion and coupling) can be adjusted over refactoring a conventional system. The next paper [33] depicted by Przybłek is also about comparing the OO and AOP paradigm, whereas differently, the paper examines two versions of 10 different software systems with regard to only two of the desirable software traits called cohesion and coupling over the GQM approach. The experimental results of the author exhibits that the aspect-oriented systems technically do not have an accurate proportion so as to support better modularity (i.e. cohesion and coupling), compared to the object-oriented systems. Thus, in the absence of good modularity in AOP, AO systems might not be readable, maintainable, as well as understandable. Guyomarch'h at al. [26] have extended the work of [34] in order to fully focus on obtaining the effect of AOP features over object-oriented metrics. Their methodology was established in 3 steps in order to find the impact of AOP because of its close relations to OOP. Their last decision was to implement a plug-in in Eclipse IDE depending on a few external libraries such as AJDT tools.

On the other hand, a case study from Bartsch et al. [14] has explained how they evaluate the effects of aspect-oriented system over maintainability. Assessing maintainability of both aspect-oriented and object-oriented versions of the same system shows that there is no clearly visible difference between AOP and OOP systems in terms of the results. Another empirical study of maintainability was proposed by Shen et al. [35]. The paper presents a fine-grained coupling metrics suite for AO systems so as to measure software changes during system evaluation. After the metrics have been performed upon eight AspectJ projects, the analyses of the projects have illustrated meaningful and presentable information in terms of their correlation model between metrics and system maintainability. Lastly, Mguni et al. [31] have presented an empirical assessment of maintainability of an aspect-oriented system by comparing to the original code (i.e. the OO version). In other words, refactoring the original code of an open-source software used in retail business system is part of the study. They have thus examined the output of the two different versions of the software system. The maintainability was compared using metrics from [18, 34]. The AOP version of the system has shown better maintainability values over the results of metrics.

CHAPTER 6

Conclusion and Future Work

This thesis focused on carrying out empirical research with regard to the six research questions that yield some characteristics of ten aspect-oriented systems in terms of our metric framework. We have fulfilled the thesis aim by first explaining our metric framework consisting of 41 metrics and second discussing each question as well as each metric result over the obtained values throughout this paper. We have briefly addressed what our assumptions were, what we expected or what we did not expect based on the results of the metrics.

In terms of future work, apart from our metrics, there are a number of metrics that need to be discovered to reveal other missing parts of AOP, such as what can be seen if we include the excluded classes: inner class, static inner class, local class and anonymous class, along with their members, if we incorporate interface declarations, or if we trace the control flow of the systems when control-flow based pointcuts occur, or to extend this particular achievement with new fresh data because dealing with more numbers affect the goal of work in a good manner. In consequence, while considering a system's functionalities and working on aspect-oriented features, we need to be more careful, because we may cause other unintended consequences while solving a system problem since it is only depends on how AspectJ is used in practice.

APPENDIX A

Glossary of acronyms

OOP	Object-Oriented Programming (See section 1.1)
AOP	Aspect-Oriented Programming (See section 1.1)
SoC	Separation of Concerns
ASP	Advice Substitution Principle
DbC	Design by Contract
JPS	Join Point Shadow (See subsection 1.2.1)
AJDT	AspectJ Development Tools
ITD	Intertype Declaration (See subsection 1.2.4)
GASR	General-purpose Aspectual Source code Reasoner (See section 1.3)
REPL	Read-eval-print Loop (See section 1.3)

Bibliography

[1] AJDT: AspectJ Development Tools. `https://www.eclipse.org/ajdt/`.

[2] Applicative logic meta-programming using Clojure's core.logic against an Eclipse workspace. `https://github.com/cderoove/damp.ekeko`.

[3] Aquarium, Aspect-Oriented Programming for Ruby. `http://aquarium.rubyforge.org/`.

[4] AspectC and AspectC++. `http://www.aspectc.org/`.

[5] AspectJS, Method-Call Interception in JavaScript. `http://www.aspectjs.com/`.

[6] HSQLDB. `http://hsqldb.org/`.

[7] JBoss AOP, Framework for Organizing Cross Cutting Concerns. `http://jbossaop.jboss.org/`.

[8] JHotDraw Start Page. JHotDraw as Open-Source Project. `http://jhotdraw.org/`.

[9] MyBatis GitHub. `http://mybatis.github.io/mybatis-3/`.

[10] MyBatis Homepage. `http://blog.mybatis.org`.

[11] PostSharp. `http://www.postsharp.net/`.

[12] Muhammad Sarmad Ali, Muhammad Ali Babar, Lianping Chen, and Klaas-Jan Stol. A Systematic Review of Comparative Evidence of Aspect-oriented Programming. *Inf. Softw. Technol.*, 52(9):871–887, September 2010.

[13] Sven Apel and Don Batoryz. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology*, 9(1):117–142, 2010.

[14] Marc Bartsch and Rachel Harrison. An Exploratory Study of the Effect of Aspect-oriented Programming on Maintainability. *Software Quality Control*, 16(1):23–44, March 2008.

[15] Rachel Burrows, Fabiano Cutigi Ferrari, Alessandro Garcia, and François Taïani. An Empirical Evaluation of Coupling Metrics on Aspect-oriented Programs. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '10, pages 53–58, New York, NY, USA, 2010. ACM.

[16] Fernando Castor Filho, Paulo Asterio de C Guerra, Vinicius Asta Pagano, and Cecília Mary F Rubira. A Systematic Approach for Structuring Exception Handling in Robust Component-based Software. *Journal of the Brazilian Computer Society*, 10(3):5–19, 2005.

[17] Fernando Castor Filho, Cecılia Mary F Rubira, and Alessandro Garcia. A quantitative study on the aspectization of exception handling. In *ECOOP Workshop*, page 137, 2005.

[18] Mariano Ceccato and Paolo Tonella. Measuring the Effects of Software Aspectization. In *1st Workshop on Aspect Reverse Engineering*, volume 12. Citeseer, 2004.

[19] Coen De Roover. Damp.ekeko.aspectj. `https://github.com/cderoove/damp.ekeko.aspectj/`.

[20] Coen De Roover. Damp.ekeko.plugin. `https://github.com/cderoove/damp.ekeko/blob/master/damp.ekeko.plugin/README.md`.

[21] Johan Fabry, Coen De Roover, and Viviane Jonckers. Aspectual source code analysis with GASR. In *SCAM*, pages 53–62. IEEE, 2013.

[22] Janet Feigenspan, Sven Apel, Jörg Liebig, and Christian Kastner. Exploring Software Measures to Assess Program Comprehension. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 127–136. IEEE, 2011.

[23] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM.

[24] Irum Godil and Hans-Arno Jacobsen. Horizontal Decomposition of Prevayler. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 83–100. IBM Press, 2005.

[25] Philip Greenwood, Thiago Tonelli Bartolomei, Eduardo Figueiredo, Marcos Dósea, Alessandro Garcia, Nélio Cacho, Cláudio Sant'Anna, Sérgio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin, Germany, 07/2007 2007.

[26] Jean-Yves Guyomarc'h and Yann-Gaël Guéhéneuc. On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics. In *Proceedings of the 9th ECOOP Workshop on Quantitative Approaches to Object-Oriented Software Engineering (QAOOSE 2005), Glasgow, UK*, 2005.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[28] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-Oriented Programming. pages 220–242. Springer-Verlag, 1997.

[29] Uirá Kulesza, Cláudio Sant'Anna, Ro Garcia, Roberta Coelho, Arndt Von Staa, and Carlos Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. of the Intl Conf. on Software Maintenance (ICSMâĂŹ06*, 2006.

[30] Marius Marin, Arie Deursen, Leon Moonen, and Robin Rijst. An Integrated Crosscutting Concern Migration Strategy and Its Semi-automated Application to JHotDraw. *Automated Software Engg.*, 16(2):323–356, June 2009.

[31] Kagiso Mguni and Yirsaw Ayalew. An Assessment of Maintainability of an Aspect-Oriented System. *ISRN Software Engineering*, 2013.

[32] Tim Molderez and Dirk Janssens. Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective. *Transactions on Aspect-Oriented Software Development*, 2014 In Press.

[33] Adam Przybyłek. Where The Truth Lies: AOP and its impact on Software Modularity. *Proceedings of the 14th international conference on Fundamental approaches to software engineering:*, pages 447–461, 2011.

[34] Claudio Nogueira Sant'Anna, Alessandro Fabricio Garcia, Christina von Flach Garcia Chavez, Carlos Jose Pereira de Lucena, and Arndt von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[35] Haihao Shen, Sai Zhang, and Jianjun Zhao. An Empirical Study of Maintainability in Aspect-Oriented System Evolution Using Coupling Metrics. In *Proceedings of the 2008 2Nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '08, pages 233–236, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 174–190, New York, NY, USA, 2002. ACM.

[37] Software Technology Firm. Aspect Research Associates (ARA). `http://aspectresearchassociates.com/`, 2003-2013.

[38] Maximilian Störzer, Uli Eibauer, and Stefan Schöffmann. Aspect Mining for Aspect Refactoring: An Experience Report. In *Towards Evaluation of Aspect Mining*, Nantes, France, July 2006. at ECOOP 2006.

[39] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai,

Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information Hiding Interfaces for Aspect-oriented Design. *SIGSOFT Softw. Eng. Notes*, 30(5):166–175, September 2005.

[40] Wikipedia. Higher-order Function. `http://en.wikipedia.org/wiki/Higher-order_function`.

[41] Wikipedia. Read eval print loop. `http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`.

[42] Wikipedia. Software Product Line. `http://en.wikipedia.org/wiki/Software_product_line`.

[43] Trevor J. Young. Using AspectJ to Build a Software Product Line for Mobile dDvices. Master's thesis, University of British Columbia, August 2005.

[44] Jianjun Zhao. Measuring Coupling in Aspect-Oriented Systems. In *Information Processing Society of Japan (IPSJ*, pages 14–16, 2004.