Science of Computer Programming 78 (2012) 117–144

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# An empirical study of aspect-oriented metrics

Eduardo Kessler Piveta [a,*], Ana Moreira [b], Marcelo Soares Pimenta [c], João Araújo [b], Pedro Guerreiro [d], R. Tom Price [c]

[a] Depto. de Eletrônica e Computação, Universidade Federal de Santa Maria (UFSM), Av. Roraima, 1000, Cidade Universitária, 97105-900, Santa Maria – RS, Brazil
[b] CITI/FCT - Departamento de Informática, Universidade Nova de Lisboa (UNL), Monte da Caparica, 2829-516, Caparica, Portugal
[c] Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Av. Bento Gonçalves 9500, 91501-970, Porto Alegre – RS, Brazil
[d] Departamento de Eng. Electrónica e Informática, FCT, Universidade do Algarve (UAlg), Campus de Gambelas, 8005-117, Faro, Portugal

ARTICLE INFO

*Article history:*
Received 25 July 2010
Received in revised form 7 June 2011
Accepted 7 February 2012
Available online 26 February 2012

*Keywords:*
Metrics
Aspect-oriented software development
Empirical evaluation
AspectJ

ABSTRACT

Metrics for aspect-oriented software have been proposed and used to investigate the benefits and the disadvantages of crosscutting concerns modularisation. Some of these metrics have not been rigorously defined nor analytically evaluated. Also, there are few empirical data showing typical values of these metrics in aspect-oriented software. In this paper, we provide rigorous definitions, usage guidelines, analytical evaluation, and empirical data from ten open source projects, determining the value of six metrics for aspect-oriented software (lines of code, weighted operations in module, depth of inheritance tree, number of children, crosscutting degree of an aspect, and coupling on advice execution). We discuss how each of these metrics can be used to identify shortcomings in existing aspect-oriented software.



## 1. Introduction

Aspect-Oriented Software Development (AOSD) aims at providing abstraction and composition mechanisms to better modularise crosscutting concerns [17,10]. These concerns often cannot be clearly decomposed from the rest of the software artefacts, and their modularisation using object-oriented techniques usually results in either scattering or tangling in the resulting software application.

The use of software metrics can help to evaluate various quality attributes of aspect-oriented software, such as modularity, reusability, and size. For example, size metrics can support the identification of modularisation problems: large modules can be broken into smaller ones with fewer responsibilities or have their features merged into other modules.

Metrics adapted from widely known and used metrics for object-oriented software [9] have already been used in experimental studies on AOSD [5,7,15], where the original object-oriented metrics were extended to be paradigm-independent, generating comparable results [6]. Up to date, these metrics have been informally described [8], their properties have not been analysed, and typical values of these metrics for actual and practical software are not available in the literature.

We complement these previous works by providing, for a sub-set of those metrics, rigorous definitions, analytical evaluation, empirical data collected from a set of widely available aspect-oriented (AO) projects, detailed discussion about these obtained values, and also a set of usage guidelines. In summary, the main contributions of this paper are:

* Corresponding author.
*E-mail addresses:* piveta@inf.ufsm.br, piveta@gmail.com (E.K. Piveta), amm@di.fct.unl.pt (A. Moreira), mpimenta@inf.ufrgs.br (M.S. Pimenta), ja@di.fct.unl.pt (J. Araújo), pjguerreiro@ualg.pt (P. Guerreiro), tomprice@terra.com.br (R.T. Price).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.02.003

- Rigorous definitions and usage scenarios for a set of selected metrics. These definitions can be used to improve the accuracy of quantitative assessment of aspect-oriented software by reducing the ambiguity normally present in informal descriptions. The usage scenarios can show the relation of the metrics with quality attributes.
- An interpretation of collected empirical data, discussing the scope of values (minimum, maximum, etc.), comparing the values in aspects and in classes, and examining variations between the metric values of the selected projects. We show and discuss a set of examples of high and low values for each of the selected metrics, illustrating the value of these metrics on practical applications. The correlation between metrics is also shown and discussed.
- An analytical evaluation of the selected metrics against established criteria for validity. We show that the aspect-oriented adapted metrics also satisfy the criteria originally satisfied by the Chidamber and Kemerer [9] object-oriented metrics, recommending that they can be used to assess aspect-oriented software.

Two sets of metrics are considered:

1. Metrics adapted from Chidamber and Kemerer [9] by Zakaria and Hosny [32], Santanna et al. [26], Ceccato and Tonella [8]: lines of code (*locc*), weighted operations in module ($wom$), depth of inheritance tree (*dit*), and number of children (*noc*);
2. Metrics specifically defined for aspect-oriented software: crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*) [8].

This paper is organised as follows. Section 2 presents a brief summary of AOSD and the AspectJ language.[1] Section 3 describes the selected metrics, the selected projects, the computed statistics, the data collection process, and the evaluation criteria for the selected metrics. Section 4 describes a rigorous definition of the metrics, usage scenarios, analytical evaluation of the metrics, and empirical data. Section 5 presents an interpretation of the empirical data, discusses the correlation between the metrics, and provides some lessons learned. Section 6 presents related work, and Section 7 concludes the paper.

## 2. Aspect-oriented software development

Traditional software development methods are unable to effectively modularise *crosscutting concerns*. The implementation of these concerns is usually scattered among several base modules [17], making the final result difficult to understand, to reuse, and to maintain. AOSD aims at providing means for their systematic identification, modularisation, and composition. AOSD modularises crosscutting concerns in separated modules, called *aspects*, and uses composition mechanisms to later compose them with the base modules.

Aspects define which points in the software application will be affected and what happens with the application execution whenever these points are reached. Although the main ideas of this paper can be adapted to other aspect-oriented languages, we focus on the AspectJ language, as our set of cases is based on this language.

AspectJ is a widely used aspect-oriented language, based on the Java programming language. In AspectJ, aspects are similar to classes in several ways: they may contain fields, methods, and implement interfaces. However, unlike classes, they cannot be instantiated, and their inheritance mechanism is more limited (only classes or abstract aspects can be extended). Aspects encapsulate join points, pointcuts, pieces of advice, and inter-type declarations in a single abstraction mechanism.

Join points are well-defined points in the execution flow of a program. Examples of join points are: method and constructor calls and execution, field access, and initialisations. Consider, for example, an *Account* class, containing a method named *withdraw* (representing the withdrawal of money) and a field named *balance* (to store the balance of the account).

```
1  public class Account {
2    double balance;
3    public void withdraw(double value){
4      setBalance(getBalance() − value);
5    }
6    public static void main(){
7      Account c = new Account();
8      c.setBalance(100);
9      c.withdraw(50);
10   }
11   ...
12 }
```

In this context, join points would be, for instance, the execution of the *withdraw* method (line 4), its call on line 9, the *Account* object instantiation (line 7), among others. In AspectJ, there are syntactic elements that allow the developer to describe join points representing the points in the code that are affected by the aspects.

[1] http://www.eclipse.org/aspectj/.

Pointcuts group join points by the definition of a predicate that, whenever satisfied, causes the actions associated to it to be executed. These join points can be composed through the logical operators *and*, *or* and *not* (&&, || and !, respectively). Pointcuts can be named and may receive parameters. These represent the arguments that are received by the pointcut, for example, the object that receives the message, the current object, and the actual message parameters. These parameters can be inspected and modified according to the semantic of the given aspect.

For example, to define an aspect that performs some actions whenever a call to the *Account.withdraw* method is made, a pointcut can be defined as follows: *call(void Account.withdraw(double))*. To define that the aspect affects the creation of new objects one could use: *initialization(public Account.new(..))*. The AspectJ pointcut language is powerful, and enables us to describe join points both in static and in dynamic structures.

An advice is an action usually associated with a pointcut. It may occur *before*, *after* or *around* a join point. The *after* advice can still have two variations: it may be executed after the successful run of the code associated to the pointcut, or in cases where an exception occurs while executing the code associated with the current join point.

## 3. Empirical metric data collection and evaluation criteria

When measuring aspects, the developer can focus on the relation of the aspect with other modules (aspects, classes, or interfaces) in terms of use of inheritance (number of children, depth of inheritance tree), associations (coupling metrics), affected modules (crosscutting degree of an aspect), etc. Also, the developer can measure aspects members (pieces of advice, pointcuts, inter-type declarations). Pointcuts can be measured to evaluate the size and the complexity of pointcut expressions and also to determine the number of join points the expression affects. Inter-type declarations can be used as a part of the measurements to express the complexity of an aspect (such as the weighted operations in module metric).

In this section, we discuss the metrics selected for this study, the projects used to collect empirical data, the computed statistics, the data collection process properties, and the evaluation criteria for the metrics.

### 3.1. Selected metrics

For this study, we selected six metrics for aspect-oriented software [8]: lines of code (*locc*), weighted operations in module (*wom*), depth of inheritance tree (*dit*), number of children (*noc*), crosscutting degree of an aspect (*cda*), and coupling on advice execution (*cae*). The reasons behind these particular metrics are two-fold:

- The four complexity/size metrics (*locc*, *wom*, *dit*, and *noc*) allow us to compare them with the classical object-oriented metrics presented by Chidamber and Kemerer [9], in terms of how large are the aspects and the classes of a given software system, and how the modules are related regarding inheritance.
- The two coupling metrics (*cda* and *cae*) provide overall estimates regarding the effects of aspects in other modules (both classes, or other aspects), in terms of how many modules an aspect affects and how many aspects affect a given module.

To help in the comparison between aspects and classes, we also use the ratio between the mean value of a $\mu$ metric for aspects and the mean value of the same metric for classes. Values of this metric higher than one indicate that the mean value of the $\mu$ metric is higher in the aspects than in the classes. On the other hand, values below one denote that the metric values are higher in the classes. A ratio value of one indicates that the mean values for this metric are equal in aspects and classes. This metric is used to compare the values of the metric in aspects and in classes (which projects have a higher value of a chosen metric for aspects than for classes or which ones have lower values for aspects). Such a ratio can be defined as follows:

*Let* $\overline{x}(\mu(aspects))$ *be the mean value for a metric* $\mu$ *for all the aspects and* $\overline{x}(\mu(classes))$ *be the mean value for a metric* $\mu$ *for all the classes, the ratio between the mean value of the metric* $\mu$ *for aspects and the mean value of the metric* $\mu$ *for classes (denoted by* $\overline{x}$ *rat.) is:* $\overline{x}(\mu(aspects))/\overline{x}(\mu(classes))$.

Other metrics would be also useful for drawing a comparison between aspects and classes in aspect-oriented software systems, such as the ones provided by Castor Filho et al. [7], and Ceccato and Tonella [8]. The suite of Castor Filho et al. [7] has already been used in some experimental studies [5,7,15]. Further research is needed to assess and evaluate these metric suites. Given that it is not feasible to analyse all these metrics in one paper, we consider these metrics as being the focus for future work, more specifically metrics for coupling and cohesion, including: coupling on intercepted modules (*cim*), coupling on method call (*cmc*), coupling on field access (*cfa*), response for a module (*rfm*) and lack of cohesion in operations (*lco*). Related work 6 discusses some of these metrics in more detail.

### 3.2. Selected projects

We selected ten projects from open source repositories to provide empirical data used as examples in Section 4. We considered the number of users and the variety of the domain, aiming to select projects that are reasonably stable and that have a significant user base. Table 1 shows summary information (name, description, version, size, and URL) of the selected projects.

**Table 1**
Summary of selected projects.

| Name | Description | Version (locc) |
|---|---|---|
| 1. AspectJ Design Patterns<br>URL: http://www.cs.ubc.ca/~jan/AODPs/ | Implementation of the GoF Patterns. | v1.1 (2344) |
| 2. AspectJ Examples<br>URL: http://www.eclipse.org/aspectj/ | Examples of the AspectJ distribution. | AJ5 (2878) |
| 3. AspectJ Hot Draw<br>URL: http://sourceforge.net/projects/ajhotdraw/ | An aspect-oriented version of the JHotDraw graphics framework. | v0.3 (23,051) |
| 4. aTrack<br>URL: https://atrack.dev.java.net/ | Bug Tracking Application. | CVSHead (2221) |
| 5. Jakarta Cactus<br>URL: http://jakarta.apache.org/cactus/ | Test framework for server-side java code. | v1.3 (5244) |
| 6. Glassbox<br>URL: http://www.glassbox.com/ | Troubleshooting agent for Java applications. | v1.0a2 (1562) |
| 7. GTalkWap<br>URL: http://sourceforge.net/projects/gtalkwap | GoogleTalk access from WAP-enabled devices. | v1.0b (1013) |
| 8. Infra Red<br>URL: http://sourceforge.net/projects/infrared | Performance Monitoring Tool for Java/J2EE. | v2.3 (13,888) |
| 9. My SQL Connector J<br>URL: http://www.mysql.com/products/connector/j/ | MySQL Native Java driver. | v5.0 (40,755) |
| 10. Surrogate<br>URL: http://sourceforge.net/projects/surrogate | Unit testing framework. | v1.0RC1 (806) |

### 3.3. Computed statistics

We used a third-party open source tool named *aopmetrics*[2] to collect the metric values for the selected projects. AopMetrics is a tool that provides the implementation of metrics for both aspect-oriented and object-oriented software written in Java/AspectJ, including the implementation for the following suites of metrics: Chidamber and Kemerer [9], Martin [19], and Li and Henry [18].

For each metric, we selected the *mean* as a measure of central tendency and the *standard deviation* as a measure of dispersion. We grouped the values by project and by module type (aspect or class). For each metric we created histograms for the values for aspects and for classes. As the sample data is different for each histogram, the Shimazaki [27] method was used to select the bin size.[3]

### 3.4. Data collection properties

We considered the following properties for the data collection process: accuracy, precision, replicability, correctness, and consistency. These properties were defined by Fenton and Pfleeger [11] and are described as follows:

- *Accuracy:* is related to the notion that there can be differences between the actual data and the measured data. The smaller the difference, the higher is the accuracy.
- *Precision:* deals with the number of decimal places used to express data. More decimal places usually indicate a higher accuracy.
- *Replicability:* means that the experiments can be performed by different people, at different times, using the same setting provided in the experiment (data, equipment, etc.).
- *Correctness:* means that the data was collected accordingly to the metrics definition.
- *Consistency:* deals with differences with the metric values when collected by different people using different tools. To be replicable, the consistency property of a process must be high.

The *accuracy* property is satisfied, as there are no differences between the collected data and the real data. The *precision* property is satisfied because the metrics are of integer domain and we use Java *Integer* objects to represent them. The

2 Available at http://aopmetrics.tigris.org.

3 The bin size of an histogram represents the size of each category in the histogram.

experiment can be *replicated*, as the metrics can be collected from the same projects using the version numbers provided in this paper.

We did our best to come out with clear and unambiguous definitions, that are the basis for *correctness* and *consistency*, but these properties can only be satisfied through formalisation and proof that the software application used to collect the metrics is derived from the formal specification (which is outside the scope of this paper). We partially assessed the *correctness* property through test cases. Although the use of test cases does not ensure correctness, it provides a certain degree of confidence in the results. In the *aopmetric* tool, there are JUnit test cases covering each kind of module that can be targeted by the metric. The tests for all the selected metrics are described in five test classes and 42 test cases to explore different combinations of modules.

### 3.5. Evaluation criteria

Chidamber and Kemerer [9] state that several researchers recommend properties that software metrics should possess to increase their usefulness. They choose the Weyuker [31] criteria, to evaluate a set of size and coupling metrics for object-oriented software because Weyuker's criteria is a widely known formal analytical approach and also because her formal analytical approach subsumes most of the earlier, less well-defined and informal properties.

Since this research is evaluating metrics adapted from Chidamber and Kemerer [9], the same criteria to evaluate the original metrics are used. Note that the criteria are paradigm-independent [31] and can be used to evaluate both object-oriented and aspect-oriented software. The two additional metrics (crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*)) are also evaluated using the same criteria. As Weyuker's criteria are also used to evaluate coupling and cohesion object-oriented metrics, there are no issues associated with the use of the criteria to evaluate these additional coupling metrics.

The Weyuker [31] criteria are summarised and expressed using predicate logic as follows. For all the properties, let us consider the modules[4] $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the metric $\mu$:

- *Non-coarseness (property 1):* This property verifies that the metric value can be different among modules, otherwise the metric is not meaningful. Given $\mathcal{A}$ and $\mathcal{B}$, the predicate $\forall \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ must hold.
- *Non-uniqueness (property 2):* This property expresses that two different modules can have the same value for the metric (i.e. the modules are equally complex). Given $\mathcal{A}$ and $\mathcal{B}$, the predicate $\exists\ \mathcal{A}\ \exists\ \mathcal{B}\ \mu(\mathcal{A}) = \mu(\mathcal{B})$ must hold.
- *Design details are important (property 3):* This property leads to the notion that different design alternatives can produce different values for the metrics. Given $\mathcal{A}$ and $\mathcal{B}$ providing the same functionality, the predicate $\mu(\mathcal{A}) = \mu(\mathcal{B})$ is not necessarily true.
- *Monotonicity (property 4):* This property states that the value of the metric for the composition of two modules can never be less than the metric values of each individual module. The predicate $\forall \mathcal{A}, \mathcal{B}\ (\mu(\mathcal{A}) \leq \mu(\mathcal{A}+\mathcal{B})) \wedge (\mu(\mathcal{B}) \leq \mu(\mathcal{A}+\mathcal{B}))$ must hold, where $\mathcal{A} + \mathcal{B}$ denotes the composition between $\mathcal{A}$ and $\mathcal{B}$.
- *Non-equivalence of interaction (property 5):* This property considers that the composition of $\mathcal{A}$ and $\mathcal{B}$ can result in different values for the same metric that the composition of $\mathcal{A}$ and $\mathcal{C}$. In this case, $\mu(\mathcal{A}) = \mu(\mathcal{B})$ does not imply that $\mu(\mathcal{A} + \mathcal{C}) = \mu(\mathcal{B} + \mathcal{C})$.
- *Interaction increases complexity (property 6):* This property states that when two modules are composed, the metric value can increase. $\exists\ \mathcal{A},\ \mathcal{B}$ such as: $\mu(\mathcal{A} + \mathcal{B}) > \mu(\mathcal{A}) + \mu(\mathcal{B})$.

The composition considering both aspects and classes is primarily made by the use of:

1. *Inheritance*: an aspect can extend a class or an aspect (but not vice-versa).
2. *Association*: an aspect can hold a reference to an object of a class (through an attribute).
3. *Type Binding*: an aspect can have generic types which are composed with classes (or other aspects).
4. *Inter-type declarations*: an aspect can add state or behaviour to a class.
5. *Pointcut expressions*: aspects composed with classes through expressions that define a composition.

These properties (one to six) will be used for the analytical evaluation of each metric in the next section.

## 4. Rigorous definitions, empirical data and analytical evaluation

In this section, we provide a rigorous definition using set theory, a set of usage scenarios, empirical data, and an analytical evaluation for each of the six selected metrics: lines of code (*locc*), weighted operations in module ($wom$), depth of inheritance tree (*dit*), number of children (*noc*), crosscutting degree of an aspect (*cda*) and coupling on advice execution (*cae*).

Table 2 summarizes the number of modules per selected project, presenting an overall feeling of the size of the selected projects (in terms of modules). Section 5 provides a detailed interpretation for the values of each metric.

[4] When applicable, we use the generic term *module* to denote a class or aspect.

**Table 2**
Modules in the project.

| Project name | #Classes | #Aspects |
|---|---|---|
| AspectJ Design Patterns | 104 | 40 |
| AspectJ Examples | 56 | 27 |
| AspectJ Hot Draw | 357 | 10 |
| aTrack | 53 | 28 |
| Jakarta Cactus | 93 | 1 |
| Glassbox | 28 | 24 |
| GTalkWap | 25 | 2 |
| Infra Red | 158 | 11 |
| My SQL Connector J | 149 | 1 |
| Surrogate | 19 | 3 |
| Total | 1092 | 147 |

Each metric is described using the following structure:

- *Informal definition:* In the introduction of each metric, an informal introduction is provided to describe the meaning of the metric;
- *Rigorous definition:* Set theory is used to describe the metric in a rigorous way;
- *Usage:* The usage scenarios for the metric are discussed, together with scenarios for the combination with other metrics;
- *Empirical data:* A set of summary statistics for the values of the metric in the selected projects is provided. Also, a brief discussion of the metric values in the selected projects is conducted;
- *Analytical evaluation:* An analytical evaluation, using predicate logic, of the metric according to the selected set of evaluation criteria.

### 4.1. Lines of code

This metric counts the number of *lines of code* (*locc*). We used the Java and AspectJ grammars[5] to define the components needed to compute this metric. In AspectJ, aspects can be composed of several elements, including those that can be also elements of classes. Aspects can contain declare constructions, pieces of advice, inter-type method/constructor declarations, inter-type field declarations, inner classes/aspects/interfaces, enumerations, constructors, fields, and methods. The *locc* of a module (aspect or class) can be computed as the sum of the *locc* of its components.

#### 4.1.1. Rigorous definition

Let $\mathcal{O}$ be the set of declare constructions, inter-type field declarations, enumerations, and fields of a module, $\mathcal{A}$ be the set of inner classes/aspects/interfaces of a module and $\mathcal{MS}$ be the set of pieces of advice, methods, constructors and inter-type method/constructor declarations of a module. Consider that the set $\mathcal{A}$ is composed of several elements $(a_1, \ldots, a_n)$ where $n = |\mathcal{A}|$ and the $\mathcal{MS}$ set is composed of several elements $(ms_1, \ldots, ms_m)$ where $m = |\mathcal{MS}|$. The $locc(m) : Module \rightarrow \mathbb{N}$ of a module $m$ can be computed by:

$$locc(m) = |\mathcal{O}| + \sum_{i=0}^{n} locc(a_i) + \sum_{j=0}^{m} locc(ms_j) \tag{1}$$

Considering that an operation is composed of a set of statements $(\mathcal{S} = (s_1, \ldots, s_w))$, with its size denoted by $w = |\mathcal{S}|$, the *locc* of operations $(\mathcal{MS})$ can be denoted as $locc(o) : Operation \rightarrow \mathbb{N}$, and can be computed by the *locc* of the individual statements:

$$locc(o) = \sum_{k=0}^{w} locc(a_k) \tag{2}$$

In this case, declare constructions, inter-type field declarations, enumerations and fields are counted as one line of code. Pieces of advice, inter-type method/constructor declarations, constructors and methods are composed of statements (conditionals, loops, etc.[6]). The set of possible statements is version dependent and the individual *locc* computations are left open. Roughly, considering each statement in a single line, the *locc* of such constructions can then be computed by the number of carriage returns in the body of each construction.

[5] Java Grammar at https://javacc.dev.java.net/, AspectJ Grammar at http://abc.comlab.ox.ac.uk/documents/scanparse.pdf.
[6] See the full Java grammar at https://javacc.dev.java.net/ for more details of all possible statements.

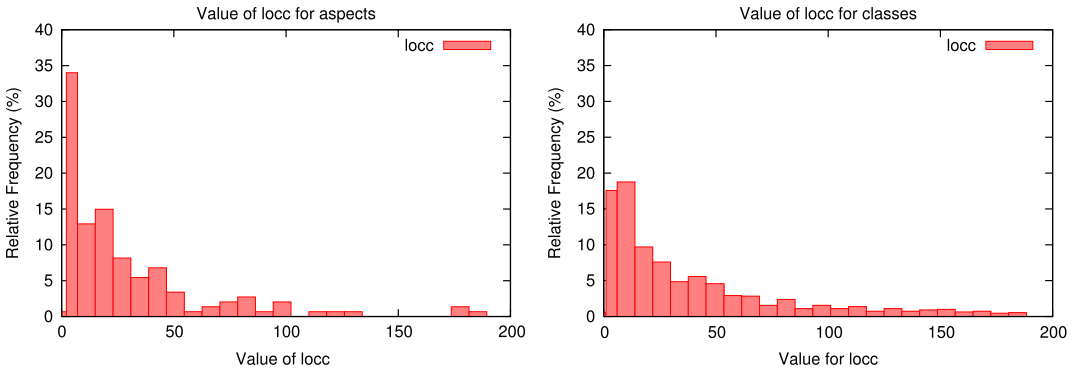


**Fig. 1.** Value of *locc* for aspects and for classes.

*4.1.2. Usage*

Metrics that count lines of code are usually used as indicators of effort, productivity, and cost [11]. In the measurement of effort, the *locc* metric is used to allow comparisons between different projects or systems. This metric is also used in productivity measurements (such as *locc*/*hour*) or costs (*cost*/*locc*), for example.

The following considerations can be made regarding *locc*, focusing on product metrics:

- The value of *locc* can be used as a rough indicator of how much effort was put into developing and maintaining an aspect. It is also used as an indicator of complexity [11].
- The *locc* metric can be also used to express rates regarding quality attributes, such as defects per *locc* [4].
- The *locc* is also used to measure the size of methods (mean *locc* per method), and the quantity of documentation of a module (comments per *locc*) [11].
- Methods with high values of *locc* are usually not so easily reusable, as methods that override their behaviour may have to redefine a lot of the original behaviour. If long methods are broken into smaller ones, the developer may have to redefine only one or a few small methods. In this case, the *locc* of a class with smaller methods can be higher, but the *locc*/*method* is smaller.
- If flexibility is an important requirement of the system being developed (when developing a framework, for example), the overall value of *locc* can be higher than a single system with few extension points. However, it does not mean that the values of *locc*/*method*, *locc*/*class* or *locc*/*aspect* of a framework are necessarily higher than the values of *locc* for a single system.

Considering the combination of the *locc* metric with the other metrics discussed in this paper, the following usage guidelines are described:

- The combination of *locc*/$wom$ can be used as an indicative of the size of operations. High *locc* values with low values of $wom$ can be a *Large Method* [12]. Modules with high *locc* values and high $wom$ values can be a *Large Aspect* [21] or a *Large Class* [12].
- Classes with high *locc* values and with low *cda* values can denote that the aspect has state or behaviour that is not dealing with crosscutting concerns. This state or behaviour can be moved to a new class or to an existing class. The aspect can use association mechanisms to access these features.
- Another possible combination is in terms of *cae* values. Aspects or classes with low *locc* values and high *cae* values can be used as an indicator of aspect interactions.

*4.1.3. Empirical data*

Table 3 summarizes the statistical data collected from the ten selected projects. The first two columns show the mean and the standard deviation for aspects, the next two columns show the same information for classes, and the last one shows the ratio between the mean value of *locc* for aspects and the mean value of *locc* for classes.

Fig. 1 shows the values of *locc* for aspects and for classes.[7] The *x*-axis shows the values of *locc* and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format [*min*, *max*[. For example, in the value of *locc* for classes, the majority of the classes have a *locc* value in the interval [0, 50[. The relative frequency states the percentage of the classes that have the size defined in the *x*-axis (value of *locc*).

In the selected projects, the core functionality is defined in classes. The aspects modularise concerns that would be otherwise scattered over the classes. The computed mean value of *locc* for classes in the selected projects is 94.3 and the

[7] The classes with a *locc* value higher than 200 (5% of the classes) are not shown to provide a better (visual) comparison between *locc* of aspects and of classes. Section 5.1 discusses such classes in detail.

**Table 3**
Summary statistics for *locc* values.

| Project name | $\bar{x}$ (Aspects) | $\sigma$ | $\bar{x}$ (Classes) | $\sigma$ | $\bar{x}$ rat. |
|---|---|---|---|---|---|
| AspectJ Design Patterns | 19.7 | 14.1 | 13.9 | 9.5 | 1.42 |
| AspectJ Examples | 33.7 | 35.9 | 34.7 | 39.3 | 0.97 |
| AspectJ Hot Draw | 18.0 | 10.4 | 62.8 | 88.7 | 0.29 |
| aTrack | 33.5 | 46.4 | 22.1 | 18.8 | 1.51 |
| Jakarta Cactus | 88.0 | 0.0 | 54.3 | 64.1 | 1.62 |
| Glassbox | 40.5 | 35.7 | 18.9 | 16.6 | 2.14 |
| GTalkWap | 11.0 | 7.1 | 38.2 | 33.6 | 0.29 |
| Infra Red | 33.7 | 38.2 | 84.5 | 97.4 | 0.40 |
| My SQL Connector J | 186.0 | 0.0 | 271.9 | 668.4 | 0.67 |
| Surrogate | 7.0 | 6.9 | 41.0 | 54.3 | 0.17 |
| Total | 30.4 | 35.5 | 94.3 | 291.8 | 0.32 |

standard deviation is 291.8. The mean value of *locc* for the aspects is 30.5 and the standard deviation is 35.5. The mean values of *locc* in the selected projects are, in general, higher for classes; also, the variability for classes is higher than for aspects. The high values of standard deviation, in terms of *locc* values, show that the size of classes and of aspects varies among them. This can be an indication that the modules are not well balanced in terms of size, there are too many *Lazy Modules* or too many *Large Modules*.

If we analyse the ratio between the mean *locc* of aspects and the mean *locc* of classes, we have projects with a ratio lower than one (classes are bigger, in terms of *locc*) and projects with ratio higher for aspects. Projects presenting a low ratio include Surrogate (0.17), AspectJ Hot Draw (0.29), GTalkWap (0.29), Infra Red (0.40), My SQL Connector J (0.67) and AspectJ Examples (0.97). In these projects, the core functionality of the application is modularised in classes and the aspects are used to encapsulate auxiliary concerns (such as logging, tracing, and policy enforcement), application of design patterns or other infra-structure aspects.

There are projects in which the ratio is bigger than one (i.e. the aspects are bigger than the classes). This usually occurs when the application is heavily based on aspects, such as the aTrack (1.51) and the AspectJ Design Patterns (1.42) projects or when the application is designed to be plugged into another using load-time weaving. In this sense, the *locc* of the affected classes is not being computed. This situation occurs in the Glassbox (2.14) project. One last project with a high ratio is the Jakarta Cactus (ratio of 1.62), which has 93 classes and only one aspect. This single aspect is responsible for logging every entry and exit of methods and is quite long. It could be simplified by reducing the duplication inside its pieces of advice.

Considering the collected data, 95% of the modules have a *locc* smaller than 250 and 85% are smaller than 100 lines of code. Developers tend to define small classes to improve the understandability of the modules and reduce the defects per module (as demonstrated by Briand et al. [4]). There are however, classes with high values for *locc*. Querying the metric values of the selected projects, the maximum value of *locc* is 4374 (which is a pretty high number for the size of a class) and the maximum value for the *locc* in the aspects is 186.

#### *4.1.4. Analytical evaluation*

Consider an aspect $\mathcal{A}$ and an aspect $\mathcal{B}$ that is an exact copy of $\mathcal{A}$, plus an additional field, and a name change. The value of $locc(\mathcal{B}) = locc(\mathcal{A}) + 1$ and Property 1[8] is satisfied, as the predicate $\forall \mathcal{A}\ \exists \mathcal{B}\ locc(\mathcal{A}) \neq locc(\mathcal{B})$ holds. Property 2 is satisfied, as for each $\mathcal{A}$, an exact copy $\mathcal{B}$ can be created (with a different name), and therefore the predicate $\exists \mathcal{A}\ \exists \mathcal{B}\ locc(\mathcal{A}) = locc(\mathcal{B})$ holds. The *locc* of each construction in a class is a design decision and it is not determined by the functionality of the aspect, therefore Property 3 is satisfied.

Consider two modules $\mathcal{A}$ and $\mathcal{B}$, which are composed by inheritance ($\mathcal{A}$ extends $\mathcal{B}$). The *locc* of the composition of $\mathcal{A}$ and $\mathcal{B}$ can be defined as $locc(\mathcal{A} + \mathcal{B}) = locc(\mathcal{A}) + locc(\mathcal{B}) + \kappa$, where $\kappa$ is the number of lines needed to express the composition relationship (inheritance, association, type binding, inter-type declaration, or pointcut expression), which is non-negative. The value of $\kappa$ is $\geq 0$. Either if ($\kappa = 0$) or ($\kappa \geq 0$), ($locc(\mathcal{A} + \mathcal{B}) \geq locc(\mathcal{A})$) and ($locc(\mathcal{A} + \mathcal{B}) \geq locc(\mathcal{B})$). Therefore, Property 4 is satisfied.

Let $\mathcal{A}$ and $\mathcal{B}$ be two different modules, in which $locc(\mathcal{A}) = locc(\mathcal{B})$. Consider a third different aspect $\mathcal{C}$, containing a set of methods $\mathcal{K}$ in common with $\mathcal{A}$ and $\mathcal{K}'$ in common with $\mathcal{B}$. For example, if we use *inheritance* to compose $\mathcal{C}$ with $\mathcal{A}$ ($\mathcal{A}$ extends $\mathcal{C}$) and $\mathcal{C}$ with $\mathcal{B}$ ($\mathcal{B}$ extends $\mathcal{C}$) the common methods are removed. If $\sum_{i=0}^{|\mathcal{K}|} locc(k_i) \neq \sum_{i=0}^{|\mathcal{K}'|} locc(k'_i)$, $locc(\mathcal{A}+\mathcal{C}) \neq locc(\mathcal{B} + \mathcal{C})$. Property 5 is satisfied as the existence of the same value of *locc* for both aspects ($locc(\mathcal{A}) = locc(\mathcal{B})$) does not imply that the $locc(\mathcal{A} + \mathcal{C}) = locc(\mathcal{B} + \mathcal{C})$.

[8] Section 3.5 describes each of the properties used in the analytical evalution.

If two modules $\mathcal{A}$ and $\mathcal{B}$ are composed through *association*, for example, $locc(\mathcal{A} + \mathcal{B}) > locc(\mathcal{A}) + locc(\mathcal{B})$, as the association must be expressed either in $\mathcal{A}$ or in $\mathcal{B}$. The same occurs using the inter-type declaration as the composition mechanism. Therefore, Property 6 is satisfied, as $\exists\ \mathcal{A},\ \mathcal{B}$ such as $locc(\mathcal{A} + \mathcal{B}) > locc(\mathcal{A}) + locc(\mathcal{B})$.

### 4.2. Weighted operations in module

The *weighted operations in module* ($wom$) metric counts the number of operations in a given module [8]. Chidamber and Kemerer [9] define the $wom$ of classes as the number of methods of a given class. When dealing with aspects, there is the need to also consider pieces of advice and inter-type declarations. So, in this paper, we define a slightly different formula for the *wom* of aspects. This metric is similar to the metric named *Weighted Methods Per Class* ($wmc$) [9]. The $wom$ metric relates directly to the complexity of modules similarly to $wmc$, since an advice is a method-like construct that provides a way to express crosscutting actions at the join points that are captured by a pointcut [30]. Informally, the value of $wom$ for a module is given by the number of its methods, pieces of advice, inter-type method declarations and inter-type constructor declarations.

#### 4.2.1. Rigorous definition

Consider a 4-tuple $\mathcal{W} = (\mathcal{M}, \mathcal{A}, \mathcal{MD}, \mathcal{CD})$ where $\mathcal{M}$ is the set of methods, $\mathcal{A}$ is the set of pieces of advice, $\mathcal{MD}$ is the set of inter-type method declarations and $\mathcal{CD}$ is the set of inter-type constructor declarations of a $m$ module. The $wom$ of the module $m$ is given by a function $wom(m) : Module \rightarrow \mathbb{N}$:

$$wom(m) = |\mathcal{M}| + |\mathcal{A}| + |\mathcal{MD}| + |\mathcal{CD}| \tag{3}$$

#### 4.2.2. Usage

We adapted the following viewpoints from Chidamber and Kemerer [9], applicable to the $wom$ metric:

- The number and the complexity of the pieces of advice in a class indicate how much time and effort is needed to develop and to maintain the aspect.
- Aspects with large numbers of pieces of advice are likely to be more application specific, limiting the possibility of reuse.
- One of the original viewpoints from Chidamber and Kemerer [9] is that the larger the number of methods in a class, the greater the potential impact on children, as children will inherit all the methods defined in a class. The impact of this viewpoint is not as high for aspects (as in AspectJ, for instance, the sub-aspects do not redefine the advices of super-aspects), as the pieces of advice of super-aspects do not influence much the complexity of sub-aspects.

Other considerations regarding the $wom$ metric are:

- Classes and aspects with low values of $wom$ can be inspected to see if they are not Lazy Aspects [20,21] or Lazy Classes [12]. This shortcoming occurs if an aspect or a class has few responsibilities, and its elimination could be beneficial. Sometimes, this responsibility reduction is related to previous refactoring or to unexpected changes in requirements (planned changes that did not occur, for instance).
- Classes and aspects with high values of $wom$ can be Large Classes [12] or Large Aspects [21]. When an aspect encapsulates more than one concern, it can be divided in as many aspects as there are concerns. This shortcoming is usually discovered when the developer finds several unrelated members (fields, pointcuts, inter-type declarations) in the same aspect [22].
- In terms of reusability, modules with high values of $wom$ can suffer from *Refused Bequest* (in which the sub-classes inherit methods that are not used) [12].
- In terms of modularity, the higher the value of $wom$, the higher is the likelihood that those methods are responsible to deal with different concerns. Further research is needed to correlate $wom$ with cohesion metrics.

When considering the combinations of this metric with the other metrics discussed in this paper, the following rules of thumb apply:

- The combination with *locc* is discussed in the previous sub-section (Section 4.1).
- The $cda/wom$ metric can be an indicative of how much influence (in terms of affected modules) an aspect has. High values of $cda/wom$ indicate that the pieces of advice affect several modules.
- The $cae/wom$ metric can be used to see how much the aspects influence the overall behaviour of a given module. The value of $cae/wom$ is directly proportional to the influence of aspects in a module.

#### 4.2.3. Empirical data

Table 4 shows summary statistics related to the $wom$ metric. Fig. 2 shows histograms for the values of $wom$ for aspects and for classes. The *x*-axis shows the values of $wom$ and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format [*min*, *max*[. For example, in the value of $wom$ for aspects, nearly 80% of the aspects have a $wom$ value between the interval [0, 5[.

**Table 4**
Summary statistics for *wom* values.

| Project name | $\bar{x}$ (Aspects) | $\sigma$ | $\bar{x}$ (Classes) | $\sigma$ | $\bar{x}$ rat. |
|---|---|---|---|---|---|
| AspectJ Design Patterns | 2.8 | 2.7 | 2.1 | 1.2 | 1.34 |
| AspectJ Examples | 5.3 | 5.3 | 4.9 | 4.6 | 1.08 |
| AspectJ Hot Draw | 2.7 | 2.5 | 9.2 | 12.1 | 0.29 |
| aTrack | 4.1 | 6.1 | 3.8 | 4.2 | 1.08 |
| Jakarta Cactus | 5.0 | 0.0 | 7.2 | 9.2 | 0.69 |
| Glassbox | 5.3 | 6.0 | 2.8 | 2.7 | 1.91 |
| GTalkWap | 1.5 | 0.7 | 5.5 | 4.4 | 0.27 |
| Infra Red | 2.4 | 2.0 | 10.7 | 15.1 | 0.22 |
| My SQL Connector J | 17.0 | 0.0 | 19.4 | 47.8 | 0.96 |
| Surrogate | 0.3 | 0.6 | 5.8 | 7.0 | 0.05 |
| Total | 3.9 | 4.8 | 9.3 | 21.0 | 0.42 |

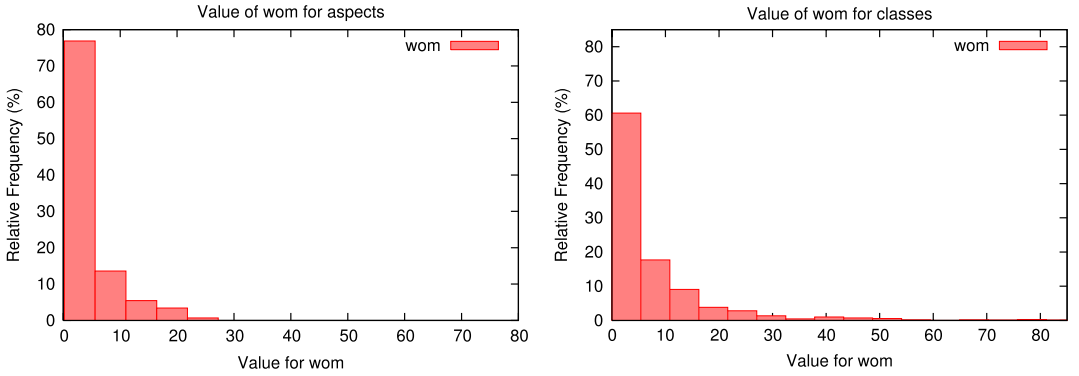


**Fig. 2.** Value of *wom* for aspects and for classes.

The values for the *wom* metric are highly correlated to those of *locc* (Section 5.8 describes this in more detail). The mean value of *wom* for classes is 9.3 and for aspects is 3.9, denoting that the number of operations in classes is larger than in aspects in the selected sample. The maximum value of *wom* for classes is 313 and for aspects it is 23. Also, the variability of this metric in the classes is higher than in the aspects. In the selected projects, the computed standard deviation of classes is 21 and in aspects it is 4.8. High values of standard deviation, in terms of *wom* values, can indicate a set of *Large Modules*, as the number of methods in a module, as shown by the empirical data, is quite small. There are, however, classes with more than 300 methods, which clearly indicate design issues.

The majority of modules (81.3%) have a maximum of ten operations (methods, pieces of advice, inter-type method declarations, or inter-type constructor declarations), while 11% have between 11 and 20 operations, and 7.6% have more than 20 operations. Considering only classes, there are 78.3% of them with less than 10 methods, 12.4% with 11–20 methods and 9.3% with more than 20 methods. The *wom* of aspects is usually smaller: 90.5% of the aspects have up to 10 operations, 8.8% have between 11 and 20 and only 0.7% (one aspect) has more than 20 operations.

The ratio between the mean of *wom* in classes and the mean of *wom* in aspects provides an indication of the proportion between the number of operations in classes and in aspects. The ratio per project is below one for Surrogate (0.05), Infra Red (0.22), GTalkWap (0.27), AspectJ Hot Draw (0.29), Jakarta Cactus (0.69), and My SQL Connector J (0.96). The last project presents a high ratio value because there is only one aspect with 17 operations. Ratio values higher than one are found in the aTrack (1.08), AspectJ Examples (1.08), AspectJ Design Patterns (1.34), and Glassbox (1.91) projects. Note that these ratio values are smaller than the ones presented for the *locc* metric.

#### *4.2.4. Analytical evaluation*

Consider two identical aspects $\mathcal{A}$ and $\mathcal{B}$ (except for their name). Adding an advice to $\mathcal{B}$ implies that $wom(\mathcal{A}) = wom(\mathcal{B}) - 1$. Property 1 is satisfied (the predicate $\forall \mathcal{A}\ \exists \mathcal{B}\ wom(\mathcal{A}) \neq wom(\mathcal{B})$ holds). Property 2 is satisfied, as for any $\mathcal{A}$ aspect one can create another aspect with the same number of operations. The number of pieces of advice, inter-type declarations, declare constructions and methods is a design decision and is not dependent of the functionality of the aspect, therefore Property 3 is satisfied.

If the modules $\mathcal{A}$ and $\mathcal{B}$ are composed through *inheritance*, the *wom* of the composition of $\mathcal{A}$ and $\mathcal{B}$ can be defined as $wom(\mathcal{A} + \mathcal{B}) = wom(\mathcal{A}) + wom(\mathcal{B}) - \omega$, where $\omega$ is the number of common operations (methods, pieces of advice, inter-type declarations) between $\mathcal{A}$ and $\mathcal{B}$. The value of $\omega$ can vary from $[0, min(wom(\mathcal{A}), wom(\mathcal{B}))]$. Either if $(\omega = 0)$

**Table 5**
Summary statistics for *dit* values.

| Project name | $\bar{x}$ (Aspects) | $\sigma$ | $\bar{x}$ (Classes) | $\sigma$ | $\bar{x}$ rat. |
|---|---|---|---|---|---|
| AspectJ Design Patterns | 0.4 | 0.5 | 0.5 | 1.4 | 0.8 |
| AspectJ Examples | 0.2 | 0.4 | 0.7 | 0.9 | 0.3 |
| AspectJ Hot Draw | 0.0 | 0.0 | 1.5 | 1.6 | 0.1 |
| aTrack | 0.4 | 0.7 | 1.1 | 1.7 | 0.4 |
| Jakarta Cactus | 0.0 | 0.0 | 0.9 | 1.1 | 0.0 |
| Glassbox | 1.1 | 1.1 | 1.4 | 1.1 | 0.8 |
| GTalkWap | 0.0 | 0.0 | 0.7 | 0.7 | 0.0 |
| Infra Red | 0.4 | 0.5 | 0.4 | 0.7 | 1.0 |
| My SQL Connector J | 0.0 | 0.0 | 1.0 | 1.3 | 0.0 |
| Surrogate | 0.7 | 0.6 | 0.8 | 1.0 | 0.9 |
| Total | 0.4 | 0.7 | 1.1 | 1.4 | 0.36 |

or ($\omega = min(wom(\mathcal{A}), wom(\mathcal{B}))$), $wom(\mathcal{A}+\mathcal{B}) \geq wom(\mathcal{A})$ and $wom(\mathcal{A}+\mathcal{B}) \geq wom(\mathcal{B})$, satisfying Property 4. If the modules are composed through association, type-binding, pointcut expression, or inter-type declarations, the value of $\omega$ is zero, as no common method is removed, leading to the same conclusion that $wom(\mathcal{A}+\mathcal{B}) \geq wom(\mathcal{A})$ and $wom(\mathcal{A}+\mathcal{B}) \geq wom(\mathcal{B})$. Therefore, Property 4 is satisfied.

Let $\mathcal{A}$ and $\mathcal{B}$ be two different modules, in which $wom(\mathcal{A}) = wom(\mathcal{B})$. Consider a third different aspect $\mathcal{C}$, containing a set of methods $\mathcal{K}$ in common with $\mathcal{A}$ and $\mathcal{K}'$ in common with $\mathcal{B}$. If we use inheritance to compose $\mathcal{C}$ with $\mathcal{A}$ ($\mathcal{A}$ extends $\mathcal{C}$) and $\mathcal{C}$ with $\mathcal{B}$ ($\mathcal{B}$ extends $\mathcal{C}$) the common methods are removed. If $|\mathcal{K}| \neq |\mathcal{K}'|$, $wom(\mathcal{A}+\mathcal{C}) \neq wom(\mathcal{B}+\mathcal{C})$. Property 5 is satisfied as the existence of the same value of $wom$ for both aspects ($wom(\mathcal{A}) = wom(\mathcal{B})$) does not imply that the $wom(\mathcal{A}+\mathcal{C}) = wom(\mathcal{B}+\mathcal{C})$.

If two modules $\mathcal{A}$ and $\mathcal{B}$ are composed through inter-type method declaration, $wom(\mathcal{A}+\mathcal{B}) > wom(\mathcal{A}) + wom(\mathcal{B})$, as the inter-type declaration must be expressed either in $\mathcal{A}$ or in $\mathcal{B}$. Therefore, Property 6 is satisfied, as $\exists\ \mathcal{A},\ \mathcal{B}$ such as: $wom(\mathcal{A}+\mathcal{B}) > wom(\mathcal{A}) + wom(\mathcal{B})$.

### 4.3. Depth of inheritance tree

The value for *depth of inheritance tree* (*dit*) is given by the longest path from a module to the class/aspect hierarchy root [9,8]. It is computed by counting the number of inheritance levels, from the module to the root class/aspect.

#### 4.3.1. Rigorous definition

Considering a function $s(x) : Module \rightarrow Module$ that computes the super-class or super-aspect of a given module, the value of *dit* is given by:

$$dit(m) = \begin{cases} dit(s(m)) + 1 & : \ m \neq rootClass \\ 0 & : \ otherwise \end{cases}$$

#### 4.3.2. Usage

The following viewpoints are adapted from Chidamber and Kemerer [9] and Ceccato and Tonella [8]:

- The higher the *dit* of an aspect, the more inherited methods it has and usually the more complex the aspect is, as the developer may have to understand not only the aspect but also the super-aspects or super-classes.
- Aspects with high values for *dit* are commonly project specific, whilst abstract aspects are usually more reusable across different projects.
- The *dit* metric does not have perceptible relevant pairwise combination scenarios with the other metrics described in this paper.

#### 4.3.3. Empirical data

Table 5 shows the values for *dit* in the selected projects. Fig. 3 shows histograms for aspects and classes. Note that, in the implementation, we are considering classes inheriting from the Java's `Object` class as root classes to compute the value of *dit* (as such inheritance from `Object` is mandatory, implicit, and transparent to the user). The *x*-axis shows the values of *dit* and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format [*min*, *max*[. For example, in the value of *dit* for aspects, nearly 90% of the aspects have a *dit* value between the interval [0, 1[ (as the *dit* metric is discrete, these aspects have a *dit* value of zero).

The value of the mean of *dit* for classes is 1.1 whilst for aspects the equivalent value is 0.4. Also, the dispersion in the values of this metric is higher for classes (1.4) than for aspects (0.7). The maximum value for *dit* of classes is eight and for

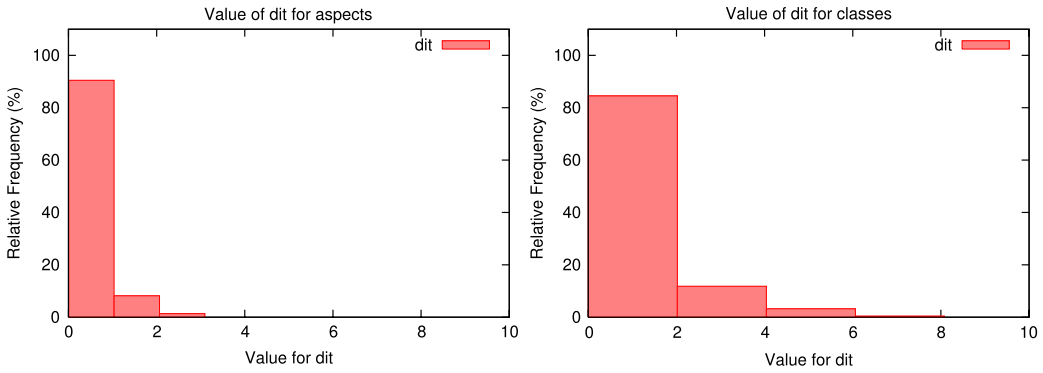


**Fig. 3.** Value of *dit* for aspects and for classes.

aspects is three. The inheritance trees in the observed projects are deeper in classes than in aspects. This is expected to be true, as aspects in AspectJ can only extend classes or abstract aspects. As there are no benefits of inheriting pieces of advice, the reuse using inheritance is mainly due to the definition of abstract aspects with abstract pointcuts and pieces of advice. The sub-aspects override the abstract pointcuts to provide the concrete join points that the aspect will affect. Low values of standard deviation in the selected projects indicate that the classes have similar *dit* values. There is no much dispersion regarding the values of the metric. Indeed, it is uncommon to have too many classes or aspects with deep inheritance trees.

The ratio between the mean *dit* of aspects per mean *dit* of classes varies from zero to 0.8. In all projects, the inheritance tree is deeper for classes than for aspects. In the selected projects, the value of *dit* of classes is less than two in 85.5% of the cases, within two and four in 11.8% of the classes, and higher than four in 3.7% of the classes. In the case of *dit* for aspects, the maximum value of *dit* is three. In fact, only two aspects are three levels down in the inheritance tree. In the other cases, 90.5% of the aspects are root aspects or inherit from one abstract aspect or class, and 8.2% have a *dit* equals to two.

Classes and aspects with high *dit* values can be inspected to search for misuse of inheritance or instances of the *Refuse Bequest* shortcoming (modules that do not use inherited features). In the selected projects, only 3.7% of the classes have a *dit* value higher than four. Inspecting the classes with a *dit* higher than four, we see that 75% of the classes are in the AspectJ Hot Draw project, where inheritance is heavily used. In this case, the project can be analysed to see if too much emphasis is given to inheritance instead of association, for example.

#### 4.3.4. Analytical evaluation

Consider two aspects $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{B}$ is a sub-aspect of $\mathcal{A}$. In this case, the value of $dit(\mathcal{B}) = dit(\mathcal{A}) + 1$. Property 1 is satisfied, as the predicate $\forall \mathcal{A}\ \exists \mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds. The *dit* for any sibling of $\mathcal{B}$ is also $dit(\mathcal{A}) + 1$, so Property 2 is satisfied, as any module can have siblings (in terms of inheritance). Property 3 is satisfied as the use of inheritance is a design dependent issue and is independent of the aspects functionality.

The composition of $\mathcal{A}$ and $\mathcal{B}$, in terms of *inheritance*, depends on the following situations: (a) $\mathcal{A}$ and $\mathcal{B}$ are super and sub-aspects, (b) $\mathcal{A}$ and $\mathcal{B}$ are siblings, and (c) $\mathcal{A}$ and $\mathcal{B}$ are unrelated in the inheritance tree. For situation (a), if we compose $\mathcal{A}$ and $\mathcal{B}$, the $dit(\mathcal{A} + \mathcal{B})$ is equal to the *dit* of the super-aspect. In this case, the predicate $dit(\mathcal{A} + \mathcal{B}) \geq dit(superAspect) \wedge dit(\mathcal{A} + \mathcal{B}) \geq dit(subAspect)$ does not hold, and therefore, for this specific case, Property 4 is not satisfied. For situation (b), $dit(\mathcal{A}) = dit(\mathcal{B})$ and therefore $dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A}) \wedge dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A})$ holds. In this case, Property 4 is satisfied. For situation (c), if the direct common ancestor of $\mathcal{A}$ and $\mathcal{B}$ is the super-aspect or the super-class of $\mathcal{A}$, the combination of both aspects is located in $\mathcal{B}$ actual location. In this case, Property 4 is satisfied, as $dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A}) \wedge dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A})$. If the common ancestor of both aspects is not a direct ancestor of both $\mathcal{A}$ and $\mathcal{B}$, there is the need to use multi-inheritance to address the situation (which is not a common feature in aspect-oriented languages). For a discussion of the *dit* metric for object-oriented software, refer to Chidamber and Kemerer [9]. The composition through association, type binding, pointcut expression, and inter-type declarations do not change the inheritance tree of the composed modules $\mathcal{A}$ and $\mathcal{B}$ and, therefore the predicate $dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A}) \wedge dit(\mathcal{A} + \mathcal{B}) \geq dit(\mathcal{A})$ holds. In these particular cases, Property 4 is satisfied.

Note that the *dit* for aspects fails to satisfy Property 4 only when two aspects are in a parent-descendent relationship. The same situation happens with the metrics for object-oriented software [9]. Not satisfying this property does not invalidate the use of *dit* as a metric to assess the use of inheritance mechanisms (the same occurs to the metric for object-oriented software).

Let $dit(\mathcal{A}) = dit(\mathcal{B})$. Let $\mathcal{C}$ be a sub-aspect of $\mathcal{A}$. As $dit(\mathcal{A} + \mathcal{C}) = dit(\mathcal{A})$ and $dit(\mathcal{B} + \mathcal{C}) = dit(\mathcal{A}) + 1$, Property 5 is satisfied as $dit(\mathcal{A}) = dit(\mathcal{B})$ does not imply that $dit(\mathcal{A} + \mathcal{C}) = dit(\mathcal{B} + \mathcal{C})$. Considering that the $dit(\mathcal{A} + \mathcal{B})$ is equal to $max(dit(\mathcal{A}), dit(\mathcal{B}))$, $dit(\mathcal{A} + \mathcal{B}) \leq dit(\mathcal{A} + \mathcal{B})$ for all $\mathcal{A}$ and $\mathcal{B}$. Property 6 is not satisfied. Section 5.7 discusses the effects of not satisfying this property.

### 4.4. Number of children

The *number of children* (*noc*) represents the number of direct sub-classes or sub-aspects for a given module [9,8].

**Table 6**
Summary statistics for *noc* values.

| Project name | $\bar{x}$ (Aspects) | $\sigma$ | $\bar{x}$ (Classes) | $\sigma$ | $\bar{x}$ rat. |
|---|---|---|---|---|---|
| AspectJ Design Patterns | 0.4 | 0.7 | 0.02 | 0.1 | 19.5 |
| AspectJ Examples | 0.2 | 0.5 | 0.2 | 0.7 | 1.0 |
| AspectJ Hot Draw | 0.0 | 0.0 | 0.6 | 2.5 | 0.0 |
| aTrack | 0.3 | 0.7 | 0.2 | 0.8 | 1.4 |
| Jakarta Cactus | 0.0 | 0.0 | 0.4 | 0.8 | 0.0 |
| Glassbox | 0.5 | 1.4 | 0.7 | 1.4 | 0.7 |
| GTalkWap | 0.0 | 0.0 | 0.2 | 0.7 | 0.0 |
| Infra Red | 0.4 | 1.2 | 0.2 | 0.7 | 2.1 |
| My SQL Connector J | 0.0 | 0.0 | 0.3 | 1.1 | 0.0 |
| Surrogate | 0.7 | 1.2 | 0.2 | 0.5 | 4.2 |
| Total | 0.3 | 0.8 | 0.4 | 1.9 | 0.9 |

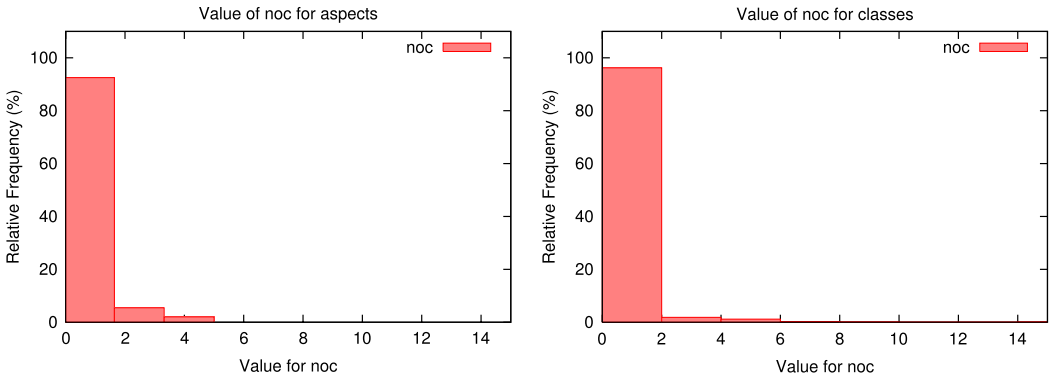


**Fig. 4.** Value of *noc* for aspects and for classes.

#### 4.4.1. Rigorous definition

Consider a function $s(x) : Module \rightarrow Module$ that computes the super-class or super-aspect of a given module and a set $\mathcal{M}$ representing the set of all modules of a given project. To compute the value of the number of children for a module $m$, let $\mathcal{S}$ be the set of all modules that satisfy the predicate $\forall y \in \mathcal{M}, s(y) = m$. Thus, the metric value is given by the cardinality of the $\mathcal{S}$ set:

$$noc(m) = |\mathcal{S}| \tag{4}$$

#### 4.4.2. Usage

We adapted the following viewpoints for the *noc* metric from Chidamber and Kemerer [9] to the context of aspect-oriented software:

- The higher the values of *noc*, the higher are the possibilities that the aspect has been reused, since inheritance is a reuse mechanism. However, the higher is the likelihood of a *Refused Bequest* [12], in which the aspect does not use part of the attributes or the methods defined in the super-class or in the super-aspect.
- Aspects with high values of *noc* can be more benefited from extensive testing, as sub-aspects usually depend on the behaviour of the super-aspect.
- Pairwise combinations of the *noc* metric and the other metrics described in this paper do not show relevant shortcoming scenarios.

#### 4.4.3. Empirical data

Table 6 shows summary statistics of the *noc* metric and Fig. 4 shows empirical data for *noc* metric both in aspects and in classes. The *x*-axis shows the values of *noc* and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format $[min, max[$. For example, in the value of *noc* for classes, nearly all classes have a *noc* value between the interval $[0, 2[$.

The majority of modules do not have children: in the selected projects 82% of the aspects and 87% of the classes do not have children. Aspects with more than one sub-aspect comprise 7.5% of the aspects; and classes with more than one sub-class correspond to 6.8% of the total number of classes. The values of the standard deviation for the *noc* metric indicate that

the classes have similar *noc* values. As occurs with *dit* values, there is no much dispersion regarding the values of the metric. Typically, classes and aspects have low values for this metric (i.e. it is uncommon to have a class with too many children — except for base core classes).

Six projects have a ratio between the mean value of *noc* for aspects and the mean value of *noc* for classes equals to zero or below one. On the other hand, four projects have more children in the aspects than in the classes: aTrack (1.38), Infra Red (2.13), Surrogate (4.22), and AspectJ Design Patterns (19.5). The value in the AspectJ Design Patterns project is high because the mean value for the *noc* of classes in the project is only 0.02.

#### 4.4.4. Analytical evaluation

Let $\mathcal{A}$ and $\mathcal{B}$ be leaves and $\mathcal{C}$ be the root of an an inheritance tree. Property 1 is satisfied as $noc(\mathcal{C}) \neq noc(\mathcal{A})$ and $\forall \mathcal{A}\ \exists \mathcal{B}\ noc(\mathcal{A}) \neq noc(\mathcal{B})$. As $\mathcal{A}$ and $\mathcal{B}$ are leaves, they both have $noc = 0$, so Property 2 is satisfied. Also, Property 3 is satisfied as the *noc* of an aspect is a design issue and is independent of the functionality.

Let $\mathcal{B}$ be the only sub-aspect or sub-class of $\mathcal{A}$. If we compose $\mathcal{A}$ and $\mathcal{B}$ through *inheritance*, the value of $\text{noc}(\mathcal{A}+\mathcal{B}) = 0$ and the predicate $noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A}) \wedge noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A})$ does not hold as $noc(\mathcal{A}+\mathcal{B}) < noc(\mathcal{A})$. Therefore, Property 4 is not satisfied. As happens with the *dit* metric, the *noc* for aspects fails to satisfy Property 4 when two aspects are in a parent–descendent relationship. As discussed before, not following this property does not invalidate the use of *noc* as a metric to assess the use of inheritance mechanisms. We discuss more details in Section 5.7. In terms of other composition mechanisms (association, type binding, pointcut expression, and inter-type declarations), the composition do not change the number of children of the composed modules $\mathcal{A}$ and $\mathcal{B}$ and, thus the predicate $noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A}) \wedge noc(\mathcal{A}+\mathcal{B}) \geq noc(\mathcal{A})$ holds. For these types of composition, Property 4 is satisfied.

Now, let $noc(\mathcal{A}) = noc(\mathcal{B})$ and let $\mathcal{C}$ be a sub-aspect of $\mathcal{A}$. As $noc(\mathcal{A}+\mathcal{C}) = noc(\mathcal{A}) + noc(\mathcal{C}) - 1$, $noc(\mathcal{B}+\mathcal{C}) = noc(\mathcal{B}) + noc(\mathcal{C})$ and $noc(\mathcal{A}) = noc(\mathcal{B})$, the predicate $noc(\mathcal{A}+\mathcal{C}) \neq noc(\mathcal{B}+\mathcal{C})$ holds and Property 5 is satisfied. Considering that the maximum value of $noc(\mathcal{A}+\mathcal{B})$ is equal to $noc(\mathcal{A}) + noc(\mathcal{B})$ for all $\mathcal{A}$ and $\mathcal{B}$, the Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the *dit* metric). Section 5.7 discusses additional details.

### 4.5. Crosscutting degree of an aspect

The *crosscutting degree of an aspect* (*cda*) metric counts the number of modules affected by pieces of advice, declare constructions, declared annotations, inter-type method declarations, and inter-type constructor declarations in a given aspect [8]. The modules *affected* by an aspect are in general undecidable, because some pointcut expressions can rely on information available only on runtime (using constructs such as `cflow`, `cflowbelow`, or `if`). Thus, in this paper, we consider the *cda* metric for the statically determinable joinpoints (described in the following definition).

#### 4.5.1. Rigorous definition

Consider a 3-tuple $\mathcal{CA} = (\mathcal{AA}, \mathcal{AD}, \mathcal{AI})$, containing information about affected modules by an aspect $\alpha$, where $\mathcal{AA}$ is the set of modules affected by pieces of advice of $\alpha$, $\mathcal{AD}$ is the set of modules affected by declare constructions of $\alpha$, $\mathcal{AI}$ is the set of modules affected by the inter-type declarations of $\alpha$. A function $cda(\alpha) : Aspect \rightarrow \mathbb{N}$ that computes the crosscutting degree of an aspect can be defined as the cardinality of the union of the $\mathcal{CA}$ elements:

$$cda(\alpha) = |\mathcal{AA} \cup \mathcal{AD} \cup \mathcal{AI}| \tag{5}$$

Note that the precise determination of infeasible conditions is undecidable in general, hence we must resort to the approximate notion of modules *affected* by aspects, in our case, to the set of modules which are syntactically regarded as potentially affected. This, however, depends on how the weaver works. For a formal calculus, for example, this can be formally specified and its correctness and completeness proven. For an evolving programming language, with different implementations and with an unclear semantics, it is not as precise as the former. A formal calculus (such as [25]) can be used to specify precisely how the weaver does its job, but it does not address the issue regarding current AspectJ's implementations of weavers.

In this paper, we consider that an aspect $\mathcal{A}$ *affects* a module $\mathcal{B}$ if and only if the following constructs of $\mathcal{A}$ matches $\mathcal{B}$: inter-type member declarations (i.e. $\mathcal{A}${*typeB.aField*; } or $\mathcal{A}${*B.aMethod*(){...}; }), extension and implementation through declare constructions (`declare parents`, `declare error`, `declare warning`, `declare annotation`, and `declare soft`), and pointcut expressions composed of statically determinable pointcuts (which are pointcut expressions that do not contain the following constructs: `cflow`, `cflowbelow`, `this`, `target`, `args`, and `if`).

#### 4.5.2. Usage

The crosscutting degree of an aspect metric can be used as an indicator of separation of concerns [14]. The following usages can be considered:

- High values of *cda* are desirable [8], as the *cda* metric indicates how many modules an aspect affects and how useful the aspect is.

**Table 7**
Summary statistics for *cda* values.

| Project name | $\bar{x}$ | $\sigma$ | Min | Max |
|---|---|---|---|---|
| AspectJ Design Patterns | 2.8 | 5.9 | 0 | 38 |
| AspectJ Examples | 3.5 | 4.8 | 0 | 20 |
| AspectJ Hot Draw | 3.6 | 5.4 | 1 | 18 |
| aTrack | 13.3 | 21.2 | 0 | 75 |
| Jakarta Cactus | 68.0 | 0.0 | 68 | 68 |
| Glassbox | 5.17 | 9.5 | 0 | 33 |
| GTalkWap | 3.0 | 1.4 | 2 | 4 |
| Infra Red | 2.0 | 4.7 | 0 | 15 |
| My SQL Connector J | 78.0 | 0.0 | 78 | 78 |
| Surrogate | 1.3 | 2.3 | 0 | 4 |
| Total | 6.2 | 13.9 | 0 | 78 |

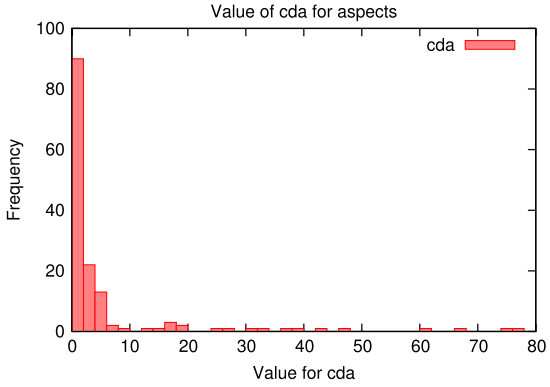


**Fig. 5.** Value of *cda* for aspects.

- Ceccato and Tonella [8] point out that while high values of *cda* are desirable, the number of explicitly named modules in the pointcuts of an aspect must be kept low (this metric is named coupling on intercepted modules (*cim*) [8]).
- If the *cda* value is equal to one, the developer has to evaluate if it is better to inline the aspect or to use inheritance or association mechanisms to separate the concerns encapsulated by the aspect.
- Section 4.1 discusses combinations with the *locc* metric, and Section 4.2 discusses combinations with the $wom$ metric.

### 4.5.3. Empirical data

Table 7 shows summary statistics of the *cda* metric. Fig. 5 shows the values of *cda* in the selected projects. Note that this metric only applies to aspects, not to classes. The *x*-axis shows the values of *cda* and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format [*min*, *max*[. For example, in the value of *cda* for aspects, nearly 90% of the aspects affect a few classes (between the [0, 6[ interval).

The values of *cda* are low in general (72% of the aspects have a *cda* value of three or less), but the values can be high and can vary according to the nature of the concerns being encapsulated by the aspects (with a maximum of 78 in the selected projects), thus the high values for the standard deviation. Logging and tracing aspects are more likely to have high values of *cda* than other aspects. Higher values of *cda* indicate that the aspect is a valuable entity. This happens because, if the concern is being implemented as a class, calls to its methods must be scattered over other classes.

### 4.5.4. Analytical evaluation

Consider two aspects $\mathcal{A}$ and $\mathcal{B}$ with $cda(\mathcal{A}) = cda(\mathcal{B})$. It is always possible to create a new module $\mathcal{C}$ and insert an inter-type declaration in $\mathcal{B}$ module, which targets the $\mathcal{C}$, for example. Thus, Property 1 is satisfied, as the predicate $\forall\mathcal{A}\ \exists\mathcal{B}\ \mu(\mathcal{A}) \neq \mu(\mathcal{B})$ holds. Two empty aspects $\mathcal{A}$ and $\mathcal{B}$ have equal values of the *cda* metric. In this case, Property 2 is satisfied. Property 3 is also satisfied, as the number of modules affected by aspects is dependent of design decisions and not of functionality.

If we compose aspects $\mathcal{A}$ and $\mathcal{B}$ through *inheritance*, the $cda(\mathcal{A} + \mathcal{B}) = cda(\mathcal{A}) + cda(\mathcal{B}) - \nu$, where $\nu$ is the number of modules affected by both $\mathcal{A}$ and $\mathcal{B}$. In this case, $\nu$ is given by a value in the interval $[0, min(cda(\mathcal{A}), cda(\mathcal{B}))]$, if $\nu = 0$, $cda(\mathcal{A} + \mathcal{B}) = cda(\mathcal{A}) + cda(\mathcal{B})$, and Property 4 is satisfied. If $\nu = min(cda(\mathcal{A}), cda(\mathcal{B}))$ then $cda(\mathcal{A} + \mathcal{B}) = max(cda(\mathcal{A}), cda(\mathcal{B}))$ and Property 4 remains satisfied.

If we compose $\mathcal{A}$ and $\mathcal{B}$ through *association*, one of them must be a class (as aspects cannot be composed together through association). Let $\mathcal{B}$ be that class and $\mathcal{A}$ be an aspect. In this case, let us also consider that $cda(\mathcal{A}+\mathcal{B}) = cda(\mathcal{A})+cda(\mathcal{B})-\nu$, where $\nu$ is the number of modules affected by both $\mathcal{A}$ and $\mathcal{B}$. Given that classes do not affect modules, $cda(\mathcal{B}) = 0$ and $\nu = 0$. Therefore, $cda(\mathcal{A}+\mathcal{B}) = cda(\mathcal{A})$ and Property 4 is satisfied.

If both $\mathcal{A}$ and $\mathcal{B}$ are classes, $cda(\mathcal{A}) = cda(\mathcal{B}) = cda(\mathcal{A}+\mathcal{B}) = 0$. If $\mathcal{A}$ and $\mathcal{B}$ are aspects (ore one of them is a class), and they are composed through *type binding*, it means that the $\mathcal{B}$ is a type parameter for some $\mathcal{A}$ construct (or vice-versa). As pointcuts can target type parameters, this can increase the number of affected modules of $\mathcal{A}$, as it can target itself if the pointcut expression matches the particular $\mathcal{B}$ type. Thus, $cda(\mathcal{A}+\mathcal{B}) = cda(\mathcal{A}) \vee cda(\mathcal{A}) + 1$, and the predicate $cda(\mathcal{A}+\mathcal{B}) \geq cda(\mathcal{A})$ and $cda(\mathcal{A}+\mathcal{B}) \geq cda(\mathcal{B})$ holds.

If modules $\mathcal{A}$ and $\mathcal{B}$ are composed through a *pointcut expression*, then the affected modules can increase in one unit (if it was not being affected before the composition). Thus, the $cda(\mathcal{A}+\mathcal{B}) = max(cda(\mathcal{A}), cda(\mathcal{A})+1)+max(cda(\mathcal{B}), cda(\mathcal{B})+1)$. Therefore, $cda(\mathcal{A}+\mathcal{B}) \geq cda(\mathcal{A}) + cda(\mathcal{B})$ holds and Property 4 is satisfied.

Consider that $cda(\mathcal{A}) = cda(\mathcal{B})$ and let $\mathcal{C}$ be another aspect. Consider that the number of affected modules in $\mathcal{C}$ that are common with $\mathcal{A}$ is $\upsilon$ and in common with $\mathcal{B}$ is $\omega$ and also that $\upsilon \neq \omega$. As $cda(\mathcal{A}+\mathcal{C}) = cda(\mathcal{A}) + cda(\mathcal{C}) - \upsilon$ and $cda(\mathcal{B}+\mathcal{C}) = cda(\mathcal{B}) + cda(\mathcal{C}) - \omega$, the predicate $cda(\mathcal{A}+\mathcal{C}) \neq cda(\mathcal{B}+\mathcal{C})$ holds and therefore Property 5 is satisfied as $cda(\mathcal{A}) = cda(\mathcal{B})$ does not imply that $cda(\mathcal{A}+\mathcal{C}) = cda(\mathcal{B}+\mathcal{C})$. Consider that the number of common modules affected by pieces of advice, inter-type method declarations or inter-type constructor declarations between two aspects $\mathcal{A}$ and $\mathcal{B}$ is given by $\nu$. For all $\mathcal{A}$ and $\mathcal{B}$, $cda(\mathcal{A}) + cda(\mathcal{B}) - \nu \leq cda(\mathcal{A}) + cda(\mathcal{B})$ and therefore Property 6 is not satisfied (the implications are the same of those for the *dit* metric — see Section 5.7).

### 4.6. Coupling on advice execution

The *coupling on advice execution* (*cae*) metric weights the number of aspects affecting a given module [8]. Again, consider that the information regarding if aspects affect classes are in general statically undecidable, because the weaver can rely on information available only on runtime (using constructs such as `cflow`, `cflowbelow`, or `if`, for example). Thus, for this metric, we also consider the *cae* metric for statically determinable joinpoints (as defined in Section 4.5).

#### 4.6.1. Rigorous definition

Consider a 3-tuple $\mathcal{CE} = \{\mathcal{EA}, \mathcal{ED}, \mathcal{EI}\}$ containing information about the modules that affect a module $m$, where $\mathcal{EA}$ is the set of aspects that advises $m$, $\mathcal{ED}$ the set of aspects that add declare constructions to $m$, and $\mathcal{EI}$ the set of aspects that define inter-type declarations to $m$. The $cae(\mathrm{m}) : Module \rightarrow \mathbb{N}$ function can be defined as the cardinality of the union of the $\mathcal{CE}$ elements:

$$cae(m) = |\mathcal{EA} \cup \mathcal{ED} \cup \mathcal{EI}| \tag{6}$$

#### 4.6.2. Usage

The values of the *cae* metric can be used as an indicator of aspect interaction. The following considerations can be made:

- Low values of *cae* are good, as the higher the *cae* value, the more coupled is the class to the aspects that affect it [8]. If a module has a *cae* with a zero value, it means that the module is not affected by aspects.
- More affecting aspects can denote aspect interactions and possible precedence conflicts or incompatibilities between the applied aspects.
- We discuss combinations with the *locc* metric in Section 4.1 and combinations with the *wom* metric in Section 4.2.

#### 4.6.3. Empirical data

Table 8 shows summary statistics for the values of *cae* in aspects and in classes. Fig. 6 shows histograms for the *cae* values for aspects and for classes. The *x*-axis shows the values of *cae* and the *y*-axis shows the relative frequency of each category in the projects. Each bar shows the relative frequency in the format [*min*, *max*[. For example, in the value of *cae* for aspects, nearly 80% of the aspects are not affected by any aspects, or are affected by only one aspect (between the [0, 2[ interval). Please, note that the total $\bar{x}$ rat. value (2.2) is greater than all individual values (including the maximum value) because it considers all the aspects and all the classes in the selected projects (the other results are computed *per project*). This may seem awkward, but it is correct.

In terms of standard deviation, the values of *cae* are as expected, given that it is not common to have a large number of aspects affecting a given module. This occurs because of the nature of the aspects (they tend to modularise a few concerns, mainly related to non-functional requirements) and the focus on domain design with classes. In the selected projects, the aspects affect classes more than affect other aspects. The only exception is the Glassbox project, that uses load time weaving to attach its aspects to a target Java application, so the values for the *cae* metric in the affected classes can only be computed at load-time or later.

Considering the ratio between the mean value of *cae* for aspects and the mean value of *cae* for classes, all the projects – except the Glassbox – have a ratio lower than one. Six projects have a zero ratio (GTalkWap, AspectJ Hot Draw, Jakarta Cactus, Surrogate, Infra Red and My SQL Connector J), three projects have a ratio lower than one (classes are more affected

**Table 8**
Summary statistics for *cae* values.

| Project name | $\bar{x}$ (Aspects) | $\sigma$ | $\bar{x}$ (Classes) | $\sigma$ | $\bar{x}$ rat. |
|---|---|---|---|---|---|
| AspectJ Design Patterns | 0.1 | 0.4 | 0.5 | 0.6 | 0.2 |
| AspectJ Examples | 0.5 | 0.6 | 1.4 | 1.5 | 0.3 |
| AspectJ Hot Draw | 0.0 | 0.0 | 0.1 | 0.3 | 0.0 |
| aTrack | 3.3 | 1.5 | 4.4 | 1.9 | 0.7 |
| Jakarta Cactus | 0.0 | 0.0 | 0.7 | 0.4 | 0.0 |
| Glassbox | 3.0 | 1.4 | 1.5 | 0.9 | 2.0 |
| GTalkWap | 0.0 | 0.0 | 0.2 | 0.6 | 0.0 |
| Infra Red | 0.0 | 0.0 | 0.1 | 0.3 | 0.0 |
| My SQL Connector J | 0.0 | 0.0 | 0.5 | 0.5 | 0.0 |
| Surrogate | 0.0 | 0.0 | 0.2 | 0.4 | 0.0 |
| Total | 1.2 | 1.7 | 0.6 | 1.2 | 2.2 |

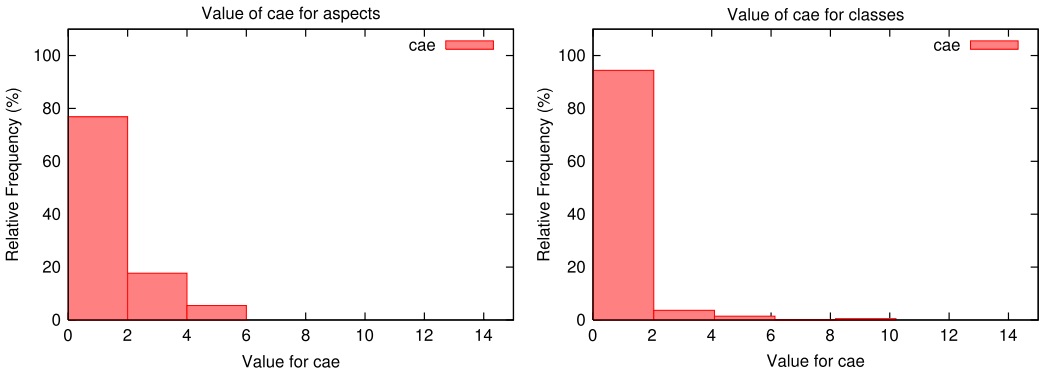


**Fig. 6.** Value of *cae* for aspects and for classes.

than aspects), including the AspectJ Design Patterns (0.23), AspectJ Examples (0.34) and aTrack (0.74) whereas the Glassbox has a ratio of 1.97.

Note that the values for this metric are quite low, which indicates that interactions among aspects are not very common. Considering all the selected projects but Glassbox and aTrack, only 2.1% of the aspects have a *cae* value higher than one. However, for the Glassbox and the aTrack projects, interactions can introduce precedence issues as 91% of the aspects in the Glassbox project and 81% of the aspects in the aTrack project have $cae > 1$. The mean value of the *cda* metric for the selected projects is higher that the values of *cae*, indicating that the aspects are being used to modularise concerns that the corresponding object-oriented alternative would be otherwise scattered among several classes.

#### *4.6.4. Analytical evaluation*

Consider two equal aspects $\mathcal{A}$ and $\mathcal{B}$ (except for their name) with $cae(\mathcal{A}) = cae(\mathcal{B})$. It is always possible to create a new aspect $\mathcal{C}$ that affects only $\mathcal{B}$, for example. In this case, $cae(\mathcal{A}) = cae(\mathcal{B}) - 1$. Property 1 is satisfied, as $\forall \mathcal{A}\ \exists\ \mathcal{B}\ cae(\mathcal{A}) \neq cae(\mathcal{B})$. Property 2 is also satisfied, as two unaffected modules have the same value for the *cae* metric. The number of aspects that affects a class is design dependent and not mandated by the functionality of the classes and aspects, therefore Property 3 is also satisfied.

Consider the $\mathcal{A}$ and $\mathcal{B}$ modules, not related by inheritance, and their respective super-classes or super-aspects $\mathcal{A}'$ and $\mathcal{B}'$, where $\mathcal{A}' \neq \mathcal{B}'$. As there are certain pointcut expressions that affect a class and all its descendants, consider that $\mathcal{B}'$ has one of such pointcuts affecting it and its sub-classes or sub-aspects (in this case, including $\mathcal{B}$). If the inheritance tree is changed (i.e. the $\mathcal{B}'$ is not the superclass of $\mathcal{B}$ any more), the $\mathcal{B}$ can be no longer affected by such pointcut, and the predicate $\forall \mathcal{B}\ \ (\mu(\mathcal{B}) \leq \mu(\mathcal{A} + \mathcal{B}))$ does not hold, and thus Property 4 is not always satisfied for this particular case (a pointcut expression affecting descendants of the $\mathcal{B}'$ module). Section 5.7 discusses more details about this. Composition through association or through type binding can increase the number of classes affecting a given module, as a reference to a new type is introduced in the module, thus $cae(\mathcal{A} + \mathcal{B}) \geq cae(\mathcal{A}) + cae(\mathcal{B})$ and Property 4 is satisfied for this case. If a $\mathcal{A}$ module is composed with $\mathcal{B}$ through a pointcut expression (i.e. a pointcut expression of $\mathcal{A}$ uses $\mathcal{B}$ in its expression), the $cae(\mathcal{A} + \mathcal{B}) \geq cae(\mathcal{A}) + cae(\mathcal{B})$, as the $cae(\mathcal{B})$ after the composition is increased by one if the $\mathcal{B}$ module was not already being affected by $\mathcal{A}$. The same situation occurs with the composition through inter-type declarations. In both cases, Property 4 is satisfied.

Consider that $cae(\mathcal{A}) = cae(\mathcal{B})$ and let $\mathcal{C}$ be another module. Consider that the number of aspects affecting $\mathcal{C}$ that are common with the aspects that affect $\mathcal{A}$ is $\upsilon$ and in common with $\mathcal{B}$ is $\omega$ and also that $\upsilon \neq \omega$. As $cae(\mathcal{A} + \mathcal{C}) =$

$cae(\mathcal{A}) + cae(\mathcal{C}) - \upsilon$ and $cae(\mathcal{B} + \mathcal{C}) = cae(\mathcal{B}) + cae(\mathcal{C}) - \omega$, the predicate $cae(\mathcal{A} + \mathcal{C}) \neq cae(\mathcal{B} + \mathcal{C})$ holds and therefore Property 5 is satisfied. Let $\theta$ be the number of common modules affecting two modules $\mathcal{A}$ and $\mathcal{B}$. For all $\mathcal{A}$ and $\mathcal{B}$, $cae(\mathcal{A}) + cae(\mathcal{B}) - \theta \leq cae(\mathcal{A}) + cae(\mathcal{B})$ and therefore Property 6 is not satisfied (the effects of not satisfying this property are the same of those for the *dit* metric — as described in Section 5.7).

## 5. Discussion

In this section, we show a series of examples of classes and aspects showing high and low values for each metric and discuss the implications of such situations.

### 5.1. Lines of code

Classes with high values of *locc* can be analysed and, if needed, be broken into two or more classes. For example, the top ten classes in terms of *locc* values, in the selected projects, are from the My SQL Connector J project. The *locc* of these classes vary from 1317 to 4374 and they can be considered *Large Classes* [12].

The highest value of *locc* for aspects in the selected projects is the `Tracer` aspect, in the My SQL Connector J project. Inspecting this aspect, we note that two private methods are not being used[9] and can be deleted. Also, there are two duplicated methods.[10] The *Extract Method* refactoring [12] can be used to extract the common behaviour. The same situation occurs with the `entry` and `exit` methods and with the `methods` and `constructors` pointcuts. This aspect can be broken into two different aspects: one containing the core behaviour of tracing and other with the binding between this behaviour and the My SQL Connector J classes.

Classes with low values of *locc* can also bring shortcomings. For example, in the AspectJ Design Patterns project, there is an empty class named `Panel` and in the aTrack project, an empty aspect named `Observing`. Other empty classes in the selected projects are the `Sorter` and `ButtonCommand2` classes in the AspectJ Design Patterns, the `HTMLTextAreaFigure` inner-class named `InvalidAttributeMarker` in the AspectJ Hot Draw project and the `MockMethodTestCase` inner-class named `TestException` in the Surrogate test framework.

There are 25 classes and aspects with a *locc* value below five and 129 below six, for example. The developer can inspect these small classes and aspects to evaluate if their existence is justified or if they can be merged with existing classes.

### 5.2. Weighted operations in module

We analysed two classes with the highest values for the *wom* metric in the selected projects: `Connection` and `ConnectionProperties` (from the My SQL Connector J project). The `Connection` class has an inner class, named `UltraDevWorkAround`, with 154 methods.

Inspecting the `ConnectionProperties` class, the first thing to note is that it has a lot of inner classes. Fig. 7 shows the `ConnectionProperties` class with the attributes and methods compartments hidden to better visualize both the inner classes and two of its sub classes: `PropertiesDocGenerator` and `DocsConnectionPropsHelper`.

The dependency between `ConnectionProperties` and the selected sub-classes is very weak. The sub-classes have only a `main` method. As the source code to both classes is available, the `PropertiesDocGenerator` class (Listing 1) was inspected. This class does not need to extend the `ConnectionProperties` class.

```
1  public class PropertiesDocGenerator
2    extends ConnectionProperties {
3    public static void main(String[] a) throws SQLException {
4      System.out.println(new PropertiesDocGenerator().exposeAsXml());
5    }
6  }
```

**Listing 1.** `PropertiesDocGenerator` class

Listing 2 shows that the inheritance dependency can be removed and an instance of the `ConnectionProperties` class (line 3) can be created instead.

```
1  public class PropertiesDocGenerator{
2    public static void main(String[] a) throws SQLException {
3      System.out.println(new ConnectionProperties().exposeAsXml());
4    }
5  }
```

**Listing 2.** `PropertiesDocGenerator` class — Modified

9 The `getStream` and `setStream` methods.

10 The `printEntering` and `printExiting` methods.

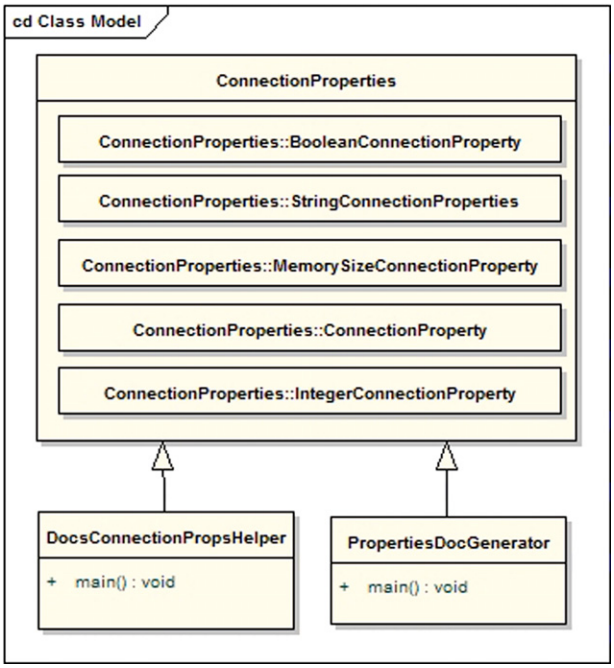


**Fig. 7.** Class Diagram for the `ConnectionProperties` class and some of its sub-classes.

Also, the `DocsConnectionPropsHelper` is equal to the `PropertiesDocGenerator` class, except for its name. None of them are being used by other classes and they can be both deleted. If they are being used indirectly, through reflection, only one of them is needed. Also, the inner classes are quite big and can be extracted to new classes. This can reduce the size of the `ConnectionProperties` class and some of its complexity.

In terms of high values of *wom* in aspects, consider the `ExecutionTracer` aspect in the Glassbox project, for example. It has 20 operations dealing with trace printing, pattern matching and advising trace points. It can be refactored into a set of aspects dealing each one with a different concern. Large aspects can benefit from the use of a guideline named *one concern per aspect* [23], that proposes that each aspect should encapsulate a single concern.

Low values of *wom* in classes appear, for example, in the `Version` class (Listing 3) of the Jakarta Cactus project. This class has only a constant field (line 2). This constant can be moved to another class or stored in a resource bundle.

```
1  public class Version {
2    public static final String VERSION = "@version@";
3  }
```

**Listing 3.** `Version` class

Another class with a low *wom* value is the `EscapeProcessorResult` class, from the My SQL Connector J project (Listing 4). Usually, it is interesting to encapsulate the access to the attributes using accessors. For example, the `escapedSql` attribute (line 3) is used by six different methods in three classes. The developer can use the *Encapsulate Attribute* refactoring [12] to provide a `getEscapedSql` and a `setEscapedSql` methods to access the protected data. This allows the structure of the `escapedSql` to be changed over time (from a `String` to a `StringBuffer`, for example), decoupling the `EscapeProcessorResult` class from the classes that use it.

```
1  class EscapeProcessorResult {
2    boolean callingStoredFunction = false;
3    String escapedSql;
4    byte usesVariables = Statement.USES_VARIABLES_FALSE;
5  }
```

**Listing 4.** `EscapeProcessorResult` class

Low values of *wom* in aspects occur, for example, in the abstract aspect `TemplateOperationMonitor`, in the Glassbox project. This aspect has four pointcuts that can be extended by sub-aspects, but does not have any associated behaviour. This aspect can be seen both as a *Lazy Aspect* and as an occurrence of *Speculative Generality* (i.e. when classes or aspects are created to handle hypothetical future requirements that are never materialised.) [12,21]. Other aspects with too few responsibilities to justify its existence include the `AtrackLogManager` and the `AtrackExceptionHandling` aspects, in the aTrack project.

For example, the AtrackLogManager (Listing 5) only defines a declare parents statement (line 2), that can be moved to another aspect that deals with logging.

```
1  public aspect AtrackLogManager {
2    declare parents: org.atrack..* implements Loggable;
3  }
```

**Listing 5.** AtrackLogManager aspect

### 5.3. Depth of inheritance tree

In the aTrack project, there are four classes with a *dit* of five or six. All these classes represent Java exceptions, with the following hierarchy:

```
-Object
  - Throwable
    - Exception
      - RuntimeException
        - AtrackException
          - PersistenceException
          - ControllerException
          - ModelException
            - EntityNotFoundException
```

As user defined exceptions in Java usually extend from RuntimeException or from one of its sibling classes, it is expected that they have a *dit* of four or more. In this case, however, one of the classes in the inheritance tree, the AtrackException class, is used only in the LoginAction class (Listing 6 — line 8):

```
1  public class LoginAction extends Action {
2    ...
3    private Subject authenticate(String username, String password){
4      ...
5      try {
6        lc.login();
7      } catch (LoginException e) {
8        throw new AtrackException(e);
9      }
10     return lc.getSubject();
11   }
12 }
```

**Listing 6.** LoginAction class

This class is an unnecessary *Middle Man* [12]. The PersistenceException, ControllerException and ModelException can inherit from RuntimeException and the AtrackException can be deleted, as it is used in only one place, which can be changed to a direct reference to the RuntimeException class.

In the selected projects, the maximum value for the *dit* of aspects is three. There are no problems associated with such values for *dit* in aspects. In fact, only two aspects actually have a *dit* equal to three: the AbstractXmlCallMonitor and the XMLParsingMonitor aspects in the Glassbox project. Further inspection in these two aspects does not show any misuse of inheritance.

Low values of *dit* in classes are quite common. It is only a problem if the class is bloated with a lot of responsibilities and the use of inheritance can alleviate the problem or if the class is a *Lazy Class* and has few responsibilities in the overall design.

As the inheritance in aspects plays a slightly different role, compared to the inheritance of classes, the values of *dit* in aspects are expected to be lower. Usually, inheritance mechanisms are used to decouple the behaviour defined in a super-aspect with the concrete join points specified in the sub-aspects. Examples of such use of inheritance are representative of a design guideline named *use abstract aspects* [23], that states that the developer should design towards abstract aspects, whose behaviour is defined completely by its pieces of advice, and its relationship with classes or other aspects is accomplished by specialization.

The use of abstract aspects can help in developing more reusable aspects, by postponing implementation decisions and leaving the definition of concrete pointcut definitions to the sub-aspects. Also, the behaviour defined in abstract aspects can be reused by different applications. Each application can create sub-aspects that capture the specific points that will activate the aspect behaviour.

For example, in the Observer pattern [16] (in the AspectJ Design Patterns project) there is a ScreenObserver aspect (Listing 7) that extends the reusable abstract aspect ObserverProtocol (line 1) and defines that both roles (Subject and Observer) will be played by the Screen class (lines 2 and 3). It also defines when the subject state changes (line 4)

and what should be done to update the observers (lines 5–7). This example defines an abstract aspect implementing the logic for the Observer pattern and leaves for the sub-aspects the task of binding the Subject and Observer roles and the changes in the Subject with the classes that will play these roles.

```
1  public aspect ScreenObserver extends ObserverProtocol{
2     declare parents: Screen implements Subject;
3     declare parents: Screen implements Observer;
4     pointcut subjectChange(Subject sub):
5       call(void Screen.display(String)) && target(sub);
6     void updateObserver(Subject sub, Observer obs) {
7       ((Screen)obs).display("Updated");
8     }
9  }
```

**Listing 7.** ScreenObserver aspect

### 5.4. Number of children

High values of *noc* can be seen in classes that are highly reused through inheritance. In the AspectJ Hot Draw project, for example, the AbstractCommand class has 33 children and is the base class for new Command classes. Other examples of classes with a high value of *noc* in the AspectJ Hot Draw framework are the UndoableAdapter (with 24 children), the AbstractTool (with 15 children) and the ResizeHandle, with 8 children.

Aspects with high values of *noc* usually implement the basic behaviour of a concern and use abstract pointcuts to define a contract that the sub-aspects must fulfil.

Consider the InfraREDBaseAspect aspect (Listing 8), from the Infra Red project, for example. This aspect tracks the time spent by a method call and updates a set of statistics. It defines an abstract pointcut and an abstract method as hooks that are overridden by the sub-aspects, binding the application classes with the time tracking behaviour. The condition abstract pointcut (line 2) specifies the condition based on which monitoring is performed and the getApiType method (line 3) gets the type (Session Bean/Entity Bean/JDBC etc.) of an API.

```
1  public abstract aspect InfraREDBaseAspect {
2    public abstract pointcut condition();
3    public abstract String getApiType();
4    Object around() : condition(){
5      ...
6      final String apiType = getApiType();
7      // Time tracking statements
8    }
9  }
```

**Listing 8.** InfraREDBaseAspect aspect

The InfraREDBaseAspect aspect has four sub-aspects: EntityBeanAspect, SessionBeanAspect, StrutsAspect and WebAspect. The WebAspect aspect (Listing 9), for example, overrides the condition pointcut (line 2) to define which join points are affected by the InfraREDBaseAspect behaviour and the getApiType (line 5), to specify the type of API used.

```
1  public aspect WebAspect extends InfraREDBaseAspect {
2    public pointcut condition():
3      execution (public * HttpServlet+.*(..)) ||
4        execution (public * Filter+.*(..));
5      public String getApiType() {
6        return "Web";
7      }
8    }
```

**Listing 9.** WebAspect aspect

Low *noc* values in classes and aspects is commonplace. In fact, in the selected projects, nearly 87% of all the aspects and classes do not have sub-classes or sub-aspects.

### 5.5. Crosscutting degree of an aspect

Examples of aspects with high values of *cda* include the Tracer aspect from the My SQL Connector J project (cda = 78), the LogAspect from the Jakarta Cactus project (cda = 68), the AtrackLogManager (cda = 39) from the aTrack project and the QueueStateAspect (cda = 38) from the AspectJ Design Patterns project.

Consider the Tracer aspect (Listing 10), for example. This aspect has a pointcut named methods that defines the join points using wildcards and package information, instead of simply listing all the affected points. It ensures that every method

executions within a set of packages is traced.

```
1  public aspect Tracer {
2     pointcut methods(): execution(* *(..))
3     && within(com.mysql.jdbc.* )
4     && within(!com.mysql.jdbc.trace.*)
5     && within(!com.mysql.jdbc.log.*)
6     && within (!com.mysql.jdbc.Util);
7      ...
8  }
```

**Listing 10.** `Tracer` aspect

The `QueueStateAspect` (Listing 11) defines a behaviour according to class initialisation. The after advice with the `initialization(new()) && target(q)` pointcut expression affects 34 classes (line 2). If new classes are added to the system, they will be automatically affected by the aspect.

```
1  public aspect QueueStateAspect {
2    after(Queue q): initialization(new()) && target(q) {
3      q.setState(empty);
4    }
5     ...
6  }
```

**Listing 11.** `QueueStateAspect` aspect

The `ErrorHandling` aspect, in the Glassbox project, affects 33 classes using a composite pointcut, defining the affected join points using nine separated predicates (one for each set of points).

Aspects with low *cda* values can be inspected to evaluate if they can be converted to classes or merged with other aspects. Sometimes the aspects with low *cda* values extend other aspects in a similar way that the *Template Method* design pattern [13] is implemented in object-oriented systems.

Consider, for example, the `ExampleProjectCalls` aspect (Listing 12) in the Surrogate project. It extends the `SurrogateCalls` aspect (line 5), that defines an abstract pointcut named `mockPointcut` (line 6) and an advice that implements a certain behaviour each time the `mockPointcut` join points are reached (line 7).

```
1   aspect ExampleProjectCalls extends SurrogateCalls {
2     protected pointcut mockPointcut() : (call (java.io.*Reader.new(..)) ||
3       call(* java.lang.System.currentTimeMillis())) ;
4   }
5   public abstract aspect SurrogateCalls  {
6     protected abstract pointcut mockPointcut();
7     Object around() : mockPointcut(){
8        ...
9     }
10  }
```

**Listing 12.** `ExampleProjectCalls` aspect

Other examples of aspects that implement the same pattern and have a low value for *cda* include the four `InfraREDBaseAspect` children in the Infra Red project (`EntityBeanAspect`, `SessionBeanAspect`, `StrutsAspect`, and `WebAspect`) and the `ColorObserver` and `RequestCounting` aspects in the AspectJ Design Patterns project.

Another situation occurs when the classes are *Lazy Aspects*. In the AspectJ Design Patterns project, for example, lazy aspects appear in four aspects: `StrategyProtocol`, `MementoProtocol`, `FlyweightProtocol`, and `CompositeProtocol`. These aspects do not have any crosscutting members and can be converted to classes. Whenever an aspect does not have members implementing crosscutting concerns a class can (and should, if possible) be used instead. One *Lazy Aspect* was detected in Glassbox. The `AbstractResourceMonitor` aspect does not have crosscutting members, but it cannot be converted to a class because it extends the `AbstractRequestMonitor` aspect (in AspectJ, classes cannot extend aspects). Such situations can be automatically detected in AspectJ using a bad smell detector [22].

### 5.6. Coupling on advice execution

High values of *cae* can be an indicative of aspect interaction. The developer should focus the search for aspects interaction on the modules with the highest values for this metric. Table 9 shows the number of modules with $cae > 1$ in the selected projects. Table 10 shows the same information considering a $cae > 2$. Classes with a *cae* value higher than one can have interaction issues. The probability of having interaction problems is higher in modules with high values for the *cae* metric.

The aTrack and the AspectJ Examples projects have several classes with values of *cae* higher than two. Note that this fact is not a problem itself, but the classes should be inspected to detect aspect interaction issues.

**Table 9**
Number of modules with cae > 1.

| Project | # of modules |
|---|---|
| aTrack | 51 |
| ajExamples | 22 |
| glassbox | 9 |
| ajDesignPatterns | 8 |
| GTalkWAP | 2 |
| ajHotDraw | 2 |

**Table 10**
Number of modules with cae > 2.

| Project | # of modules |
|---|---|
| aTrack | 47 |
| ajExamples | 11 |
| glassbox | 3 |

Consider, for example, the `LoginAction` class (Listing 13) in the aTrack project. This class has a *cae* value equals to ten. It means that ten different aspects affect this class. The class has only two methods and its behaviour is heavily influenced by the aspects. It is difficult to see if the interaction of all the aspects affecting this class are correct, in the right order or even how they affect the behaviour of the class.

```
1  public class LoginAction extends Action {
2    public ActionForward execute (ActionMapping mapping,
3      ActionForm form, HttpServletRequest request, HttpServletResponse response)
4      throws Exception{
5        LoginForm loginForm = (LoginForm)form;
6        ...
7        return mapping.findForward
8          (Consts.SUCCESS_REDIRECT);
9    }
10   private Subject authenticate(String username,
11     String password){
12     ...
13   }
14 }
```

**Listing 13.** `LoginAction` class

The class is advised by three pieces of advice defined in the `ExecutionTracer` and the `ExceptionHandling` aspects, there are three parent declarations from the `PersistenceControl` and the `AtrackLogManager` aspects. Thirteen methods are added by inter-type method declarations from the `LogManager` aspect. The `execute` method (line 2) is advised by twelve different advices, defined in eight different aspects, and one exception is softened by the `ErrorHandling` aspect. The `authenticate` method (line 10) is advised by three pieces of advice defined in two different aspect. It is difficult to reason about the resulting behaviour of the aspects that affect this class. The development environment can help to show these interactions. However, it is currently an open issue how this is modelled and implemented in a way to ensure that the behaviour is correct.

Low values of *cae* are common and do not represent any issues in terms of complexity, reusability, or maintainability. A *cae* value of zero denotes that the class or aspect is not affected by any aspects.

### 5.7. Margin of error and analytical results

The accuracy of samples is usually measured using margin of error. According to Snedecor and Cochran [28], the amount by which the proportion obtained from the sample will differ from the true population proportion rarely exceeds one divided by the square root of the number in the sample $(1/\sqrt{n})$, in which $n$ represents the number of elements in the sample. In this paper, the margin of error for the analysis related to values for aspects is 8% and for the values for classes is 3%.

Regarding the analytical results, the metrics described here satisfy most of the properties analysed, except for Property 6, which states that when two modules are combined, the metric value can increase. However, this failure implies that the metric values can increase if an aspect or class is divided in more aspects or classes. Chidamber and Kemerer [9] claim that this property may not be an essential feature for object-oriented software design complexity metrics and not satisfying can be seen as beneficial in object-oriented software. We agree with their interpretation and corroborate it for aspect-oriented software. As happens with the metrics for object-oriented software [9], the *dit* and *noc* for aspects fails to satisfy Property

**Table 11**
Correlation coefficients between values for aspects.

| Correlation between values for aspects | | | | | | |
|---|---|---|---|---|---|---|
| | dit | cae | cda | locc | noc | wom |
| dit | 1.00 | 0.18 | −0.25 | −0.19 | −0.06 | −0.21 |
| cae | | 1.00 | 0.17 | 0.26 | 0.06 | 0.26 |
| cda | | | 1.00 | 0.51 | 0.02 | 0.40 |
| locc | | | | 1.00 | 0.22 | 0.87 |
| noc | | | | | 1.00 | 0.26 |
| wom | | | | | | 1.00 |

**Table 12**
Correlation coefficients between values for classes.

| Correlation between values for classes | | | | | |
|---|---|---|---|---|---|
| | dit | cae | locc | noc | wom |
| dit | 1.00 | −0.07 | 0.02 | −0.02 | 0.02 |
| cae | | 1.00 | 0.00 | 0.00 | 0.01 |
| locc | | | 1.00 | 0.05 | 0.81 |
| noc | | | | 1.00 | 0.13 |
| wom | | | | | 1.00 |

4 (which states that the value of the metric for the composition of two modules can never be less than the metric values of each individual module) only when two aspects are in a parent-descendent relationship. Not following this property does not invalidate the use of *dit* and *noc* as metrics to assess the use of inheritance mechanisms (the same occurs to the object-oriented version of these metrics). The same occurs with the *cae* metric, in which Property 4 does not hold only in very specific situations and with a difference of one in the metric value.

### 5.8. Data correlation

The Pearson product–moment correlation coefficient (PMCC) measures the correlation (linear dependence) between two variables $\mathcal{A}$ and $\mathcal{B}$ and is defined as the covariance of these variables divided by the product of their standard deviations. It is usually denoted by $r$. Correlation indicates the strength and direction of a linear relationship between two random variables [28]. We measured the correlation between the metrics using the *gretl*[11] tool. Please, refer to Rodgers and Nicewander [24] for more details on how to compute PMCC.

Tables 11 and 12 show the correlation values ($r$) for aspects and classes (rounded to two digits using unbiased rounding).

We use correlation squared ($r^2$) [28] to help in the data interpretation. Correlation squared describes the proportion of variance in common between the two variables. High values of correlation squared appear only between the *locc* and the *wom* metrics. This correlation squared is 0.76, which means that, across all the aspects in the sample projects, 76% of their variance on these two metric values is in common. Fig. 8 shows the correlation between these two metrics in a scatter plot. There is also a small squared correlation between *locc* and *cda* (0.26) and between *wom* and *cda* (0.16). These squared correlation values lead to the notion that there is a relationship between the size of aspects (in terms of the values of *locc* and *wom*) and the crosscutting degree of an aspect (*cda*) values. The remaining correlation values indicate very low correlation or no correlation at all between *dit* and the other metrics, *cae* and the other metrics, and *noc* and the other metrics.

Table 12 also shows that the metrics for classes are not correlated, except for *wom* and *locc*, with a correlation of 0.81 (with a correlation squared of 0.66). Usually there is a balance between the number of lines of code per method in both aspects and classes. Note that correlation shows that a couple of values change together, but does not necessarily imply causation, as the causes underlying the correlation may be indirect and unknown. Further investigation is needed to correlate these metrics with additional ones and to study the causes behind this correlation.

### 5.9. Lessons learned

Metrics adapted from Chidamber and Kemerer [9] and metrics specifically tailored for aspect-oriented software can be used to evaluate software applications in the presence of aspects in several ways. Metrics can be used as indicators of quality, used to measure quality attributes, such as reusability or modularity, for example. Also, they can be used to detect problems that can appear in the software. In this section we provide a summary of how the usage guidelines and the examples discussed can provide insights on how each metric can be used to spot shortcomings in aspect-oriented software.

11 http://gretl.sourceforge.net/.

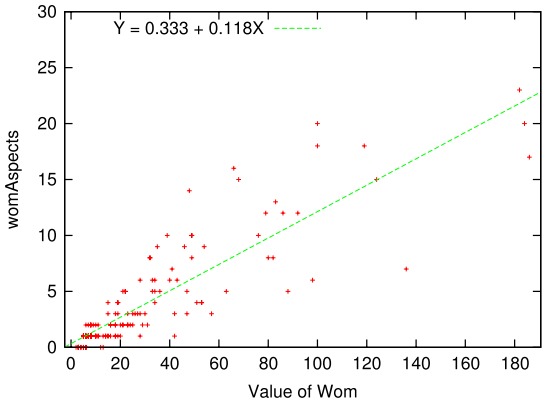


**Fig. 8.** Wom Versus Locc.

The *locc* metric can be used in combination with other metrics as an indicator of effort, complexity, productivity and cost. In the selected projects, the majority of classes and aspects have low values of *locc*, showing probable design efforts attempting to improve the comprehensibility and to reduce the number of defects per module. More specifically the *locc* metric can be used to spot:

- *Large operations:* The *locc* metric in combination of $wom$ is used to evaluate the size of operations. High *locc* values with low values of $wom$ can show large methods, large pieces of advice or large inter-type method declarations.
- *Unnecessary use of aspects:* Classes with high *locc* values and with low *cda* values can denote that the aspect has state or behaviour that is not dealing with crosscutting concerns.
- *Aspect interactions:* Aspects or classes with low *locc* values and high *cae* values can suffer from aspect interactions, where more than one aspect affect the same join points at the same time.
- *Large modules:* High values of *locc* in classes and aspects can denote large classes or large aspects. Those modules should also be inspected for code duplication, for unused operations and attributes, and for the encapsulation of more than one concern per module.
- *Lazy modules:* Classes or aspects with very low values of *locc* should be inspected to evaluate if they have enough responsibilities to exist at all.

The $wom$ metric can indicate how much effort is needed to develop an aspect or a class. Modules with high values for $wom$ are likely to be more application specific, with a lower reusability. The metric can also be used to detect a set of situations:

- *Lazy modules:* Modules with low values of $wom$ can be evaluated to see if they have enough responsibilities to be first class entities of a system or if it is better to merge them with other modules;
- *Large modules:* Modules with high values of $wom$ can be large modules, with several concerns being encapsulated or having a large number of inner classes or unrelated operations, for example;
- *Refused bequest:* Modules with high values of $wom$ are more likely to suffer from this shortcoming, in which sub-classes or sub-aspects inherit methods that are not used.
- *Influence of aspects:* The $cda/wom$ metric can show how much influence (in terms of affected modules) an aspect has.
- *How affected a module is:* The $cae/wom$ metric can be used to see how much the aspects influence the overall behaviour of the module.

The values for the $wom$ metric are highly correlated to those of *locc*. As happens with the values of *locc*, the $wom$ of classes is higher than in the aspects. Around 80% of modules have a maximum of ten operations, but several modules with high values of $wom$ can be found in practice.

The values of *dit* in classes are usually higher than in aspects, as in AspectJ the aspects have a limited inheritance mechanism. The *dit* metric can be used to:

- *Measure complexity:* Modules with high values of *dit* are usually more complex and more project specific, limiting reuse.
- *Misuse of inheritance:* Modules with high *dit* values should be inspected to search for misuse of inheritance, i.e. should be analysed to evaluate if too much emphasis is given to inheritance.
- *Project specific aspects:* A zero value of *dit* for concrete aspects indicates that pointcuts and pieces of advice are defined in the same aspect. This kind of aspect can be inspected to see if its behaviour can be encapsulated in an abstract aspect, decoupling it from the concrete affected points and enabling the reuse of the aspect.

The *noc* metric can indicate how many modules use inheritance as a reuse mechanism. In the selected projects, around 85% of the modules do not have children. The *noc* of an aspect or class is usually used to spot:

- *Indicatives of reuse:* Modules with several children are likely to be extensively reused. The developer should look for sub-classes or sub-aspects that do not use effectively the composing elements of their super-classes or super-aspects;

- *Important modules:* Sometimes, modules with several children are important modules of a project. These modules can be tracked and analysed for any inheritance misuses.

The *cda* metric is used to measure the influence of an aspect in other modules and can be used to:

- *Evaluate usefulness of aspects:* Aspects with high values of *cda* are usually more valuable, as the equivalent object-oriented modularisation would be scattered over several modules.
- *Find lazy aspects:* Aspects with a low *cda* can be inspected to see if the behaviour and state encapsulated by the aspect can be moved to or merged with other modules.

Aspects that deal with global policies, such as logging, tracing or authentication are more likely to have high values of *cda* than other aspects. In the selected projects, the majority of aspects do have low values for *cda* and some of them can be seen as lazy aspects.

The *cae* metric represents the number of aspects that affects a given module and is mainly used to detect:

- *Aspects interaction:* The higher the value of *cae* of a module, the higher the probability of having more than one aspect affecting the same join points of this module;
- *Affected modules:* The modules that have a *cae* different from zero are those that are affected by some aspect. These modules have to be treated more carefully, as modifications in the affected module can potentially affect the aspect behaviour.

In the selected projects, classes are more affected by aspects than the aspects themselves. Furthermore, the values of this metric are quite low, except for a few modules (there are modules, for example with a *cae* of ten). Low values of *cae* are common and do not represent any issues in terms of quality attributes.

## 6. Related work

Previous works on metrics applicable to aspect-oriented software are typically extensions of the Chidamber and Kemerer [9] object-oriented metrics. In fact, some of these metrics were revisited to take into account the specific features of aspect-oriented software. Castor Filho et al. [7] propose a suite of metrics, including metrics for separation of concerns, coupling, cohesion and size. The metrics included in the suite of Castor Filho et al. [7] can be briefly summarized as follows:

- *Lines of Class Code (locc):* Counts the lines of code.
- *Number of Attributes (noa):* Counts the number of fields of each class or aspect.
- *Number of Operations (noo):* Counts the number of methods and advices of each class or aspect.
- *Concern Diffusion over Components (cdc):* Counts the number of components that contribute to the implementation of a concern and other components which access them.
- *Concern Diffusion over Operations (cdo):* Counts the number of methods and pieces of advice that contribute to the implementation of a concern plus the number of other methods and pieces of advice accessing them.
- *Concern Diffusion over locc (cdl):* Counts the number of transition points (points in the code where there is a *concern switch*) for each concern through the lines of code.
- *Coupling Between Components (cbc):* Counts the number of components declaring methods or fields that may be called or accessed by other components.
- *Depth of Inheritance Tree (dit):* Counts how far down in the inheritance hierarchy a class or aspect is declared.
- *Lack of Cohesion in Operations (lco):* Measures the lack of cohesion of a class or aspect in terms of the amount of method and advice pairs that do not access the same field.

Ceccato and Tonella [8] also discuss metrics to count the number of operations, the depth of inheritance tree and the lack of cohesion in operations. Other metrics include the following:

- *Number of Children (noc):* Number of immediate sub-classes or sub-aspects of a given module, indicating the proportion of modules potentially dependent on inherited properties.
- *Coupling on Advice Execution (cae):* Number of aspects containing pieces of advice possibly triggered by the execution of operations in a given module.
- *Crosscutting Degree of an Aspect (cda):* Number of modules affected by the pointcuts and by the inter-type declarations of a given aspect.
- *Coupling on Intercepted Modules (cim):* Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.
- *Coupling on Method Call (cmc):* Number of modules or interfaces declaring methods that are possibly called by another given module.
- *Coupling on Field Access (cfa):* Number of modules or interfaces declaring fields that are accessed by another given module.
- *Response for a Module (rfm):* Methods and pieces of advice potentially executed in response to a message received by a given module, measuring the potential communication between this module and the other ones.

Analysis of empirical data is a straightforward way to investigate the benefits and the disadvantages of software properties. This is also the goal of some related work in the literature. In the following paragraphs, we summarize their main characteristics and briefly compare them with our work.

Baxter et al. [3] analysed a corpus of Java programs to provide information about the typical values of metrics in Java programs, aiming to understand the relationship among Java classes and objects. Our work differs from theirs in the sense that we discuss the shape of small-scale open source aspect-oriented programs and that we focus on the rigorous definition, evaluation of the metrics and empirical data rather than verifying if the distribution function of the metrics obeys power laws (as in their work).

Zhao [33] proposes a set of metrics to aspect-oriented software to quantify the information flow in aspect-oriented programs. He also discusses a set of metrics to measure coupling in aspect-oriented software [34] and a set of metrics to compute the cohesion of aspect-oriented software [35]. His metrics are also rigorously defined and evaluated according to a set of well-defined criteria. The main difference is that we deal with a different set of metrics, regarding size, inheritance, and aspect-specific coupling metrics. Also, we show typical values for open source aspect-oriented software available to the research community and provide data interpretation and correlation of the metrics.

Santanna et al. [26], Zakaria and Hosny [32], and Tonella and Ceccato [29] propose metrics for aspect-oriented software. They defined the metrics informally and do not conduct analytical evaluation of the proposed metrics, neither empirical data to describe the common characteristics of aspect-oriented software. We complement their work by providing a rigorous definition for the metrics, empirical data and analytically evaluated the metrics.

Bartsch and Harrison [1] deal with the empirical validation of aspect-oriented coupling measures as indicators of maintainability of aspect-oriented software and with the validity of those metrics in terms of a set of theoretical principles [2]. They state that there is a weak correlation between a set of coupling and size metrics with the maintenance effort between different versions of an application. Although there is the need for further research to validate coupling metrics for aspect-oriented software as indicators of maintainability, their work is a first step on the validity of coupling metrics for aspect-oriented software. Both of their works are in an initial stage and are grounded on an informal basis. Our work complements their works by providing rigorous definitions, analytical evaluation, usage scenarios, empirical data and interpretation to two of the five coupling metrics that the authors discuss (*cae* and *cda*).

## 7. Conclusions

In this paper, we provide a set of contributions to the use of metrics for aspect-oriented software. More specifically, we provide for a set of six metrics: (i) a rigorous definition and a set of usage scenarios, (ii) an interpretation of collected empirical data, the correlation between metrics, and (iii) an analytical evaluation of the metrics against established criteria for validity.

The use of rigorous definitions helps to understand the metrics more clearly and unambiguously, making it easier to ensure that the computation of the metric values can be done in a repeatable fashion. It can also facilitate the automation of the metric collection process.

The set of usage scenarios can show the developers how the metrics can be used in practice to detect potential shortcomings in existing software artefacts. For example, the metrics evaluated in this paper can be used to show the occurrence of several situations, as follows.

The *locc* metric can be used to spot large operations, unnecessary use of aspects, aspects interactions, and large modules. The $wom$ metric can be used to detect lazy and large modules, refused bequests, influence of aspects, and how affected is a given module. The *dit* metric is used primarily to measure complexity, misuse of inheritance and the occurrence of project specific aspects. The *noc* metric can indicate the degree of reuse of a given module and to spot key modules of a project. The *cda* metric is used to evaluate the usefulness of an aspect and to find lazy aspects. The *cae* metric is mainly used to show if the module is affected by aspects and to show the possibility of aspect interaction scenarios.

The data interpretation shows typical values of aspects and classes in small-sized, open-source systems available to the research community. The provided histograms can be used to compare the metric values for an aspect-oriented project with the set of open source projects used in this paper. The examples discussed show how the metrics can be used as indicators of shortcomings or indicators of good design.

Furthermore, the correlation between the metrics explains how certain metrics change together and how they can be combined and used to evaluate aspect-oriented software. We show that the metrics both for aspects and classes are not correlated, except for *locc* and $wom$ (high correlation) and between *locc* and *cda* and $wom$ and *cda* (small correlation), but can nevertheless be combined to show situations in which the software application can be improved.

The analytical evaluation of the selected metrics against established criteria for validity showed that the aspect-oriented adapted metrics, in general, satisfy the criteria originally satisfied by the object-oriented metrics, which means that they can be used to assess aspect-oriented software and provide comparable results.

Future work will focus on the use of these metrics in the context of a query language for aspect-oriented software, in order to enable the search for metric-based refactoring opportunities. Furthermore, the interaction between the metrics and with quality attributes will be investigated to help in the definition of heuristic rules for the evaluation of software artefacts. Other metrics for aspect-oriented software are also the subject of planned future work.

## Acknowledgements

This work was partially supported by CNPq under grant no. 140046/06-2 for Eduardo K. Piveta and Project CNPQ-PROSUL grant no. 490478/06-9 - *Latin-America Research Network on Aspect-Oriented Software Development (Latin-AOSD)*. It is also

supported by Capes-Grices project grant no. 2051-05-2 - *Identification of Crosscutting Concerns and Refactoring in Aspect-Oriented Systems*, FAPERGS grant no. 10/0470-1, and FCT MCTES.

## References

[1] M. Bartsch, R. Harrison, Towards an Empirical Validation of Aspect-Oriented Coupling Metrics, in: Workshop on the Assessment of Aspect-Oriented Technologies (ASAT) - In Proceedings of the 6th Aspect-Oriented Software Development Conference (AOSD), Vancouver, Canada, 2007.
[2] M. Bartsch, R. Harrison, An evaluation of coupling measures for AspectJ - revised, in: Workshop on Linking Aspect Technology and Evolution - In Proceedings of the 5th Aspect-Oriented Software Development Conference (AOSD), Bonn, Germany, 2006.
[3] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, E. Tempero, Understanding the shape of Java software, in: Proceedings of the 21st ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications, OOPSLA'06, ACM Press, 2006, pp. 397–412.
[4] L. Briand, S. Morasca, V. Basili, Property-based software engineering measurement, IEEE Transactions on Software Engineering 22 (1) (1996).
[5] N. Cacho, C. Santanna, E. Figueiredo, A. Garcia, T. Batista, C. Lucena, Composing design patterns: a scalability study of AOP, in: Proceedings of the 5th Aspect-Oriented Software Development Conference, AOSD'06, Bonn, Germany, 2006.
[6] F. Castor Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, C. Rubira, Exceptions and aspects: the devil is in the details, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, USA, 2006, 2006, pp. 152–162.
[7] F. Castor Filho, A. Garcia, C. Rubira, A quantitative study on the aspectization of exception handling, in: Proceedings of ECOOP 2005 Workshop on Exception Handling in Object-Oriented Systems, 2005.
[8] M. Ceccato, P. Tonella, Measuring the effects of software aspectization, in: Proceedings of the 1st Workshop on Aspect Reverse Engineering, WARE 2004, Delft, The Netherlands, 2004.
[9] S. Chidamber, C. Kemerer, A metric suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) pp. 476–49.
[10] T. Elrad, R. Filman, A. Bader, Aspect-oriented programming, Communications of ACM 44 (10) (2001) 29–32.
[11] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.
[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the design of existing code, in: Object Technology Series, Addison-Wesley, 2000.
[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns — Elements of Reusable Object-Oriented Software, in: Addison Wesley Professional Computing Series, 1995.
[14] A. Garcia, C. Santanna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, in: Transactions on Aspect-Oriented Software Development, Springer Verlag, 2006, pp. 36–74.
[15] P. Greenwood, L. Blair, A framework for policy-driven auto-adaptive systems using dynamic framed aspects, in: Transactions on Aspect-Oriented Software Development, Springer Verlag, 2006.
[16] J. Hannemann, G. Kiczales, Design pattern implementation in Java and AspectJ, in: Proceedings of the 17th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2002, pp. 161–173.
[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Mehmet Aksit, Satoshi Matsuoka (Eds.), 11th European Conference on Object-Oriented Programming, in: LNCS, vol. 1241, Springer Verlag, 1997, pp. 220–242.
[18] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems and Software 23 (2) (1993) 111–122.
[19] R. Martin, OO design quality metrics: an analysis of dependencies, in: Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics at OOPSLA, 1994.
[20] M. Monteiro, J. Fernandes, Towards a catalog of aspect-oriented refactorings, in: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD-2005, ACM Press, 2005.
[21] E. Piveta, M. Hecht, M. Pimenta, R. Price, Bad smells in Aspect-Oriented Systems, in: Brazilian Symposium on Software Engineering, SBES 2005, Uberlandia - Brasil, 2005 (in Portuguese).
[22] E. Piveta, M. Hecht, M. Pimenta, R. Price, Detecting bad smells in AspectJ, Journal of Universal Computer Science 12 (7) (2006) 811–827.
[23] E. Piveta, M. Hecht, A. Moreira, M. Pimenta, J. Araújo, P. Guerreiro, R. Price, Avoiding bad smells in aspect-oriented software, in: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering, SEKE, Boston, 2007.
[24] J. Rodgers, A. Nicewander, Thirteen ways to look at the correlation coefficient, The American Statistician 42 (1) (1988) 59–66.
[25] F. Rubbo, R. Machado, A. Moreira, L. Ribeiro, D. Nunes, On the interaction of advices and raw types in AspectJ, Journal of Universal Computer Science 14 (21) (2008) 3534–3555.
[26] C. Santanna, A. Garcia, C. Chavez, C. Lucena, A. von Staa, On the reuse and maintenance of AO software: an assessment framework, in: XVII Brazilian Symposium on Software Engineering, October 2003.
[27] H. Shimazaki, Recipes for selecting the bin size of a histogram, Ph.D. Thesis, Kyoto University, 2006.
[28] G. Snedecor, W. Cochran, ISU Statistics Dept. Staff, and D.F. Cox, Statistical Methods, 8th ed., Blackwell Publishing Limited, 1989.
[29] P. Tonella, M. Ceccato, Aspect mining through the formal concept analysis of execution traces, in: Proceedings — 11th Working Conference on Reverse Engineering, 2004.
[30] K. van den Berg, J. Conejero, R. Chitchyan, AOSD Ontology 1.0. Technical Report AOSD-Europe-UT-01 D9, AOSD-Europe, May 2005.
[31] E. Weyuker, Evaluating software complexity measures, IEEE Transactions on Software Engineering 14 (9) (1988) 1357–1365.
[32] A. Zakaria, H. Hosny, Metrics for aspect-oriented software design, in: 3rd International Workshop on Aspect-Oriented Modelling. Boston, USA, 2003.
[33] J. Zhao, Towards a metrics suite for aspect-oriented software, Technical Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.
[34] J. Zhao, Measuring coupling in aspect-oriented systems, in: 10th International Software Metrics Symposium, METRICS'04, Chicago, USA, 2004.
[35] J. Zhao, B. Xu, Measuring aspect cohesion, in: 7th International Conference on Fundamental Approaches to Software Engineering, FASE'04, 2004.