J Braz Comput Soc (2010) 16: 133–146
DOI 10.1007/s13173-010-0008-0

ORIGINAL PAPER

On the benefits of quantification in AspectJ systems

Marco Tulio Valente · Cesar Couto · Jaqueline Faria · Sérgio Soares

Received: 13 November 2009 / Accepted: 13 April 2010 / Published online: 7 May 2010

Abstract In this paper, we argue that the most favorable uses of aspects happen when their code relies extensively on quantified statements, i.e., statements that may affect many parts of a system. When this happens, aspects better contribute to separation of concerns, since the otherwise duplicated and tangled code related to the implementation of a crosscutting concern is confined in a single block of code. We provide in the paper both qualitative and quantitative arguments in favor of quantification. We also propose two new metrics to capture in a simple way the amount of quantification employed in the aspects of a given system. Finally, we describe an Eclipse plugin, called ConcernMetrics that estimates the proposed metrics directly from the object-oriented code of an existing system, i.e., before crosscutting concerns are extracted to aspects. Our main motivation is to help developers and maintainers to decide in a cost-effective way if it is worthwhile to use aspects in their systems.

Keywords Aspect-oriented programming · AspectJ · Quantification · Separation of Concerns · Metrics · Refactoring

M.T. Valente (✉) · C. Couto
Department of Computer Science, UFMG, Belo Horizonte, Brazil
e-mail: mtov@dcc.ufmg.br

C. Couto
e-mail: cesarfmc@dcc.ufmg.br

J. Faria
Institute of Informatics, PUC Minas, Belo Horizonte, Brazil
e-mail: jaquefari@gmail.com

S. Soares
Informatics Center, UFPE, Recife, Brazil
e-mail: scbs@cin.ufpe.br

1 Introduction

Aspect-oriented programming (AOP) targets a relevant problem in software engineering: the modularization of crosscutting concerns [26]. In order to tackle this problem, AOP extends traditional programming paradigms, such as procedural and object-oriented, with powerful abstractions. For example, AspectJ—the most mature aspect-oriented language—extends Java with new modularization abstractions, including join points, pointcuts, advices, and aspects [27]. In AspectJ, an aspect defines a set of pointcut descriptors that matches well-defined points in the program execution (called join points). Advices are anonymous methods implicitly invoked before, after, or around join points specified by pointcuts.

Despite the increasing number of qualitative and quantitative assessments of AOP [2, 15, 17, 20, 24], there is still no consensus on the impact and the real benefits of using aspects—as proposed by AspectJ—to modularize crosscutting concerns [21, 43, 44]. In this paper, we contribute to such assessment efforts by emphasizing the benefits of a well-known characteristic of AOP: the notion of quantification [18]. This notion is used to designate statements that have effect at several parts of the code. We argue that one of the most favorable uses of aspects happen when their code relies extensively on quantified statements. When this happens, aspects better contribute to separation of concerns, since the otherwise duplicated and tangled code related to the implementation of a crosscutting concern is confined in a single block of code.

We start by providing qualitative arguments about the benefits of quantification in terms of comprehensibility, changeability, and independent development. Next, in order to strengthen such arguments, we evaluate the aspect-oriented versions of two medium-sized systems using a suite of separation of concerns metrics, including concern diffusion over components (CDC) and concern diffusion over operations (CDO) [20, 22, 41]. The results of this quantitative comparison have shown that quantification leads to better separation of concerns.

Based on the lessons learned in the mentioned case studies, we propose two new metrics, called Quantification Degree (QD) and Scattering Reduction (SR) that express in a simple way the amount of quantification employed in the aspects of a given system. We describe in details the concern model and the rationale behind these metrics. We also analyze their application in the aspect-oriented versions of three systems.

Finally, we describe an Eclipse plugin, called ConcernMetrics that estimates QD and SR metrics directly from the object-oriented code of an existing system, i.e., before eventual crosscutting concerns have been extracted to aspects. The goal is to provide developers and maintainers with earlier feedback about the gains that would be achieved if they decide to use aspects in their systems. This feature distinguishes ConcernMetrics from other tools already proposed to calculate AOP-related metrics. In general, such tools operate directly over the aspect-oriented code, i.e., they require developers to implement the aspects to then provide quantitative information about the eventual benefits achieved in this implementation.

The remainder of this paper is organized as follows. Section 2 illustrates the use of quantified statements in two medium-sized systems (JAccounting and JSpider). It also includes a quantitative evaluation about the benefits achieved by aspects in such systems. Section 3 describes the QD and SR metrics and provides examples about their application. In Sect. 4, we present the ConcernMetrics tool. Section 5 covers related work. Section 6 concludes the paper and outlines future research lines.

2 Motivating systems

In order to support our claims, we will first rely on two medium-sized AspectJ systems: JAccounting[1] and JSpider.[2] JAccounting is a Web-based business accounting system that automates invoicing, bills, and accounts handling. JSpider is a Web robot engine that supports downloading and validation of web pages. The aspect-oriented version of both systems have been independently developed by Binkley et al., in order to illustrate the application of aspect-oriented refactorings, i.e., step-by-step transformations that prescribe how to modularize crosscutting concerns [6, 7]. Furthermore, the aspects implemented in the mentioned systems make opposite uses of quantification. In JAccounting, quantified statements are widely used in order to modularize transaction handling concerns. On the other hand, the aspectization of the logging concern in JSpider makes a minimal use of quantification.

First, we illustrate the use of quantified statements in the JAccounting and JSpider systems. Next, we assess the benefits of quantification in such systems, using a suite of separation of concerns metrics. Such metrics are closely related to modularity, which influences comprehensibility, changeability, and independent development [39].

2.1 JAccounting

Figure 1 presents the transaction handling idiom implemented in the original version of JAccounting.[3] This idiom is used in several parts of the system, tangled with its core logic. It prescribes that developers need to start a transaction context before performing any database operation. After such operations, they must issue a commit (in case of success) or a rollback (in case of failures).

In the aspect-oriented version of JAccounting, the presented transaction handling code has been removed from the system's classes and moved to the aspect described in Fig. 2 (for the sake of clarity, we have not shown the pointcut descriptors in this figure). This aspect illustrates the benefits of quantification. With a small number of quantified statements (or advices in AspectJ's terminology), developers have been able to modularize the described transaction handling idiom.

In Fig. 2, the transaction concern has been implemented by a "general aspect," i.e., an aspect that implements such concerns in the whole system. Basically, the abstraction provided by this aspect was possible because the transaction concern has a homogeneous implementation, i.e., the same statements are used to provide transactional behavior throughout the system [13]. It is also worth to mention that an equivalent form of abstraction is the key benefit provided by traditional modularization abstractions. For example, procedures and functions abstract out computations that are needed in many parts of a system. Also, in object-oriented programming, inheritance allows developers to implement in superclasses identical methods required in subclasses.

2.2 JSpider

As in several other systems, logging in JSpider is a crosscutting concern, requiring developers to call methods from

[1] http://jaccounting.dev.java.net.

[2] http://j-spider.sourceforge.net.

[3] By transaction handling idiom, we mean a pattern expected to be presented at any transaction handling implementation [42].

```
tx= sess.beginTransaction();      // starts a transaction
try {
  ...                             // database operations
}
catch (...) {                     // handles database exceptions
  if (tx != null)  {
     tx.rollback();               // performs a rollback
     tx= null;
  }
}
finally {
  if (tx != null) tx.commit();    // commits
}
```

Fig. 1 Transaction handling in the original JAccounting implementation

```
aspect TransactionManagement {
  ...
  // pointcut p0 captures when database sessions must be opened
  after(): p0() {
    tx = sess.beginTransaction();
  }

  // pointcut p1 captures when database exceptions must be handled
  before(): p1() {
    if (tx != null)  {
       tx.rollback(); tx= null;
    }
  }

  // pointcut p2 captures when database sessions must be closed
  before(): p2() {
    if (tx != null) tx.commit();
  }
}
```

Fig. 2 Aspect that modularizes transaction handling in JAccounting

the logging API in several parts of the system. For example, Fig. 3 describes the logging calls performed in one of the classes of the system. Although invoking the same method, the presented calls have different strings as arguments, which is sufficient to force the logging concern to have a heterogeneous implementation, i.e., different statements (one for each string argument) are needed to provide logging behavior throughout the system [13]. As a consequence, developers cannot extract and merge the logging calls in a single or in a small number of advices. In fact, when using AspectJ to modularize logging in JSpider, the 190 logging calls that are used in the system have been refactored to 176 advices. Thus, the extracted advices have a reduced degree of quantification, i.e., they affect a small number of locations of the base program (in most cases, just one location).

It can be argued that the AO implementation has modularized the logging calls, in the sense that they have been removed from the underlying system and moved to a single aspect. However, the refactored (aspectized) code does not present clear advantages, in terms of comprehensibility, changeability, and independent development [39], when compared with the original code. Comprehensibility is impacted by the fact that the extracted advices are tightly coupled to their associated join points in the base program. For example, in order to understand the motivation behind each of the messages presented in Fig. 3, developers should refer to the OO code. On the other hand, by not being tangled with the extracted concern, the base code is clearer, therefore, more comprehensible. Changeability is also not considerably increased. For example, a change in the name of the method that handles the logging message impacts 176 points of the code in the AO version and 190 locations in the OO version. Finally, independent development is also hampered, since the logging messages are inherently dependent from the structure and the evolution of the OO code.

```
log.info("Loading " + pluginCount + " plugins.");
...
log.info("Loading plugin configuration '" + pluginInstance + "'...");
...
log.info("Plugin class ... not found");
.....
log.info("Plugin uses local event filtering");
...
log.info("Plugin not configured for local event filtering");
...
log.info("Plugin Name : " + plugin.getName());
```

Fig. 3 Logging calls in the original JSpider implementation

Table 1 Information about the AO versions

	JAccounting	JSpider
# Aspects	1	1
# JPS	44	190
# Advices	3	176

Table 2 Metrics

Metrics	JAccounting			JSpider		
	OO	AO	%	OO	AO	%
CLC	105	72	−32%	244	2400	+884%
CDO	11	7	−36%	108	243	+125%
CDC	10	3	−70%	39	3	−92%

2.3 Quantitative evaluation

In order to strengthen our previous arguments in favor of quantification, this section compares the OO and AO versions of both JAccounting and JSpider from a quantitative perspective. Table 1 starts out by presenting relevant information about the AO versions of both systems, including information such as number of aspects, advice, and join point shadows (JPS). JPS are the program elements to which a given concern is mapped.

The following metrics have been considered in the study [20, 22, 41]:

Concern Lines of Code (CLC) This metric counts the number of lines of code whose main purpose is to contribute to the implementation of a concern (excluding comments, blank lines, and `import` statements) [14]. In the OO versions, CLC counts basically the method calls that support the considered crosscutting concerns. On the other hand, in the AO versions, the metric counts the size of the extracted aspects, in terms of lines of code (including pointcut descriptors, advice signatures, and advice implementations). Table 2 presents the results of applying this metric to JAccounting and JSpider. The results are aligned with our central argument in this paper. In JAccounting, quantification has contributed to reduce in 32% the CLC value (from 105 LOC to 72 LOC). On the other hand, the minimal amount of quantification employed in JSpider explains the tremendous increase of 884% in the CLC metric result (from 244 LOC to 2400 LOC). Basically, this difference is explained by the significant number of extra lines dedicated to pointcut descriptors and advice signatures, as illustrated in Fig. 4. In order to modularize a single logging call (line 6), six extra lines of code have been required: four lines to declare the pointcut (lines 1–4) and two lines to delimit the advice body (line 5 and line 7).[4]

Concern Diffusion over Operations (CDO) This metric counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them [41]. As shown in Table 2, CDO results reflect the benefits engendered by quantification. Particularly, the CDO value in JAccounting has decreased 36% (from 11 in the OO version to 7 in the AO version). On the other hand, JSpider's CDO has had a significant increase of 125% (from 108 in the OO version to 243 in the AO version). This increase was expected due to the reduced amount of quantification observed in JSpider's aspects, where every logging call have been just moved from one location to another in the system (i.e., from a method to an advice), as described in Sect. 2.2. Moreover, in the OO version, CDO is increased by only one in case of methods whose body contains multiple logging calls. On the other hand, in the AO version such calls are implemented in different advices. For this reason, CDO increases by the number of such advices.

[4]This figure is a verbatim copy of the original code. One can argue that the pointcut could have been specified in a single line of code, thus reducing the CLC value. However, we believe this line break layout has been adopted to increase readability. It is also important to clarify that to measure CLC we have not made any editing in the source code.

```
pointcut p_27(DBUtil _this, SQLException e ):
  this(_this)
  && execution(void DBUtil.sqlException(SQLException))
  && args(e);
  before(DBUtil _this, SQLException e): p_27(_this, e) {
     _this.log.error("SQL Exception during JDBC Connect", e);
 }
```

Fig. 4 Example of pointcut descriptor and advice implementation in JSpider

Concern Diffusion over Components (CDC) This metric counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them [41]. Table 2 presents the results of applying CDC to JAccounting and JSpider. The results show that aspects have contributed to decrease CDC in both systems. The reason is clear, since the crosscutting code has been confined in a small number of aspects.

From the presented quantitative assessment, we can expect that the higher the quantification, the lower the lines of code related to the implementation of crosscutting concerns (CLC) and the diffusion of concerns over operations (CDO). However, quantification does not have an impact in the diffusion of concerns over components (CDC). In other words, quantification does not contribute to reduce the number of aspects required in the aspectization of a given system, but it contributes to reduce the number of advices and consequently the size of such aspects.

3 Quantification metrics

Based on the experience with the mentioned case studies, we describe in this section two metrics specifically designed to measure the benefits of quantification in AspectJ systems. First, we describe the concern model our quantification metrics are based on (Sect. 3.1). Next, we define the proposed metrics (Sect. 3.2) and give some examples of their application (Sect. 3.3).

3.1 Concern model

We have reused the concern model proposed by Eaddy et al. to correlate crosscutting concerns to defects [14]. This model assumes that concerns are logically organized into a tree hierarchy. For example, in the JAccounting case study, transaction handling is a concern that includes the following child concerns: starting, committing, and rollbacking a transaction (Fig. 5). In the JSpider's concern model, logging has the following child nodes: generating debug, error, fatal, info, or warning messages (Fig. 6).

The quantification metrics target only concerns that represent leafs on this tree, i.e., concerns that do not have child (as starting a transaction or generating a debug message). Since our main intention is to measure the physical (i.e., source code based) amount of quantification achieved when using aspects, it is not reasonable to consider concerns that have conceptual differences. In other words, if two concerns are not atomic (i.e., if they correspond to internal nodes in the concern model), we should not expect their implementation to occur in a single quantified statement. For example, we should not expect all transaction handling operations to be confined in a single advice body. However, it is perfectly desirable to have rollback related concerns implemented in a single block of code.

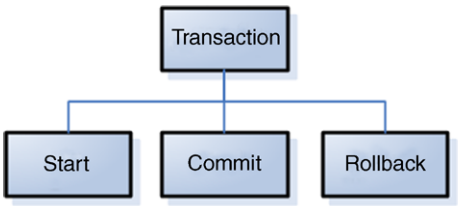


Fig. 5 Concern model for JAccounting's transaction concern

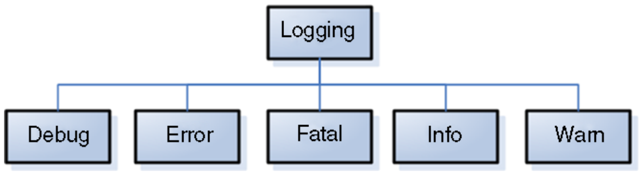


Fig. 6 Concern model for JSpider's logging concern

Moreover, we consider that concerns can be mapped to static program elements (i.e., nodes of the abstract syntax tree of the target program). In other words, we assume that there is a relation that maps nodes from the concern model to nodes of the AST of the target program. For example, a rollback concern can be mapped to the AST nodes where a rollback operation must be triggered. According to this definition, a crosscutting concern is a concern that is mapped to multiple program elements [14]. In order to follow AspectJ terminology, we call join point shadows the program elements that a given concern is mapped to [23].

3.2 QD and SR metrics

Suppose that the implementation of a crosscutting concern C represented as a leaf node in our concern model requires the instrumentation of *jps* join point shadows of the base program. Also, suppose that *adv* advices have been used to implement C in an AspectJ based system.[5] Then we first calculate the following ratio:

$$\frac{\text{Number of advices to implement } C}{\text{Number of join point shadows instrumented by such advices}} = \frac{adv}{jps}$$

Regarding this ratio, the best scenario happens when the concern is modularized in a single advice body that affects the *jps* required join point shadows. On the other hand, the worst case happens when *jps* advices are required to instrument the *jps* required join point shadows. Therefore, the described ratio is in the interval $[1/jps, 1]$. In order to have a zero value for the worst case, we must subtract the previous ratio from 1:

$$1 - \frac{adv}{jps}$$

This new ratio is in the interval $[0, 1 - (1/jps)]$. Finally, in order to transpose the ratio to the interval [0, 1], we must normalize the result considering its maximum value, which is equal to $1 - (1/jps)$. The result is the following formula for the proposed quantification degree (QD) metric:

$$\mathrm{QD}(C) = \frac{1 - \frac{adv}{jps}}{1 - \frac{1}{jps}} = \frac{jps - adv}{jps - 1}, \quad jps > 1$$

In this formula, C is a concern that has been implemented using *adv* advices affecting a total of *jps* join point shadows of the object-oriented code.

As we can observe, $\mathrm{QD}(C)$ ranges from 0 (worst case) to 1 (best case). The worst case happens when $jps = adv$, i.e., each implemented advice affects a single locus of the base program. In the best case, we have the maximal benefit of quantification, i.e., a single advice ($adv = 1$) instruments all the static locations of the base program where the concern C is required.

In the QD formula, the numerator ($jps - adv$) measures the benefits of quantification in terms of *scattering reduction*. In OO implementations, we will need to place code related to the concern C in *jps* join point shadows. In AO implementations, it is possible to confine this code to *adv* advices (where $jps > 1$ and $adv \geq 1$).[6] In other words, using Java the concern will be scattered along *jps* locations of the base program; using AspectJ, the scattering is reduced to *adv* advices. Therefore, we can define the Scattering Reduction (SR) metric in the following way:

$$\mathrm{SR}(C) = jps - adv$$

and based on SR(C) the formula for QD(C) can be rewritten to

$$\mathrm{QD}(C) = \frac{\mathrm{SR}(C)}{jps - 1}, \quad jps > 1$$

It is important to state that we cannot affirm the concern is considered homogeneous if it has QD higher than a predefined value or heterogeneous otherwise. There is no such value. QD defines a homogeneity spectrum, i.e., if $\mathrm{QD}(C) = 1$, the concern is homogeneous, and if $\mathrm{QD}(C) = 0$, the concern is heterogeneous. The values between 0 and 1 will be an indicative, supporting developers' decisions.

3.3 Examples

In this section, we discuss the application of our metrics in five examples (including three real AspectJ systems).

Example 1 Supposing a concern C mapped to 20 joint point shadows, Fig. 7 shows the possible values that QD can assume, when the number of advices used to implement C ranges from 1 to 20. As we can observe in this figure, QD has the following properties: (a) it is a ratio-scale value; (b) its value is normalized between 0 and 1; and (c) it is unitless.

Example 2 QD just indicates the degree of quantification employed in the aspectization of a given concern (considering the maximal possible degree). For example, suppose the hypothetical systems S1 and S2—and associated concerns C1 and C2—described in Table 3. In both cases, QD = 0.5. This value means that the number of extracted advices is exactly the average between the minimal and the maximal number of advices that could be employed in these systems. For example, in system S1, the best scenario would include only one advice and the worst scenario would require 101 advices to instrument all program locations requiring the execution of code related to concern C1. Since 51 advices have been effectively implemented in this case, $\mathrm{QD}(C1) = (101 - 51)/100 = 0.5$. Similarly, in system S2, $\mathrm{QD}(C2) = (11 - 6)/10 = 0.5$.

[5] Since it is based on the number of advices, this first definition aims to evaluate a particular aspect-oriented implementation. Thus, it is possible to have another implementation for the same system with different values for the proposed metrics.

[6] In this formula, *jps* should be greater than one, since with only one *jps* the concern is well modularized and, therefore, it is not a crosscutting concern.

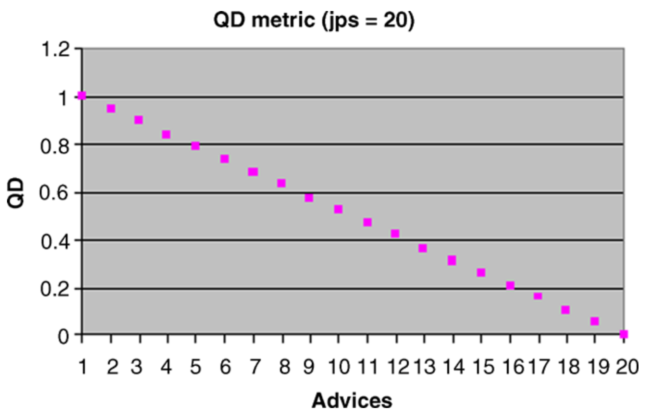


Fig. 7 QD values (assuming a concern mapped to 20 join point shadows)

Table 3 Comparing QD and SR values

System	Concern	*jps*	*adv*	SR	QD
S1	C1	101	51	50	0.5
S2	C2	11	6	5	0.5

However, in system S1 quantification has contributed to reduce from 101 to 51 the number of program locations requiring the presence of code related to concern C1, i.e., $\mathrm{SR}(C1) = 101 - 51 = 50$. On the other hand, in system S2 quantification has contributed to reduce from 11 to 6 the number of program locations requiring the presence of C2 related code, i.e. $\mathrm{SR}(C2) = 11 - 6 = 5$.

Therefore, despite having the same QD values, aspects provide more benefits in the S1–C1 case than in S2–C2. In the former case, aspects have saved the insertion of code in 50 program locations; in the later case, they have precluded the implementation of code in just 5 program locations. In summary, when assessing the benefits of quantification, it is important to consider both QD and SR metrics. QD gives a ratio-based value regarding the best possible use of aspects in a particular system/concern scenario; SR informs the absolute number of program locations where maintainers will not need to insert crosscutting code in case they decide to rely on aspects.

Example 3 (JAccounting) As described in Sect. 2.1, transaction handling in JAccounting can be decomposed in three atomic crosscutting concerns: starting a transaction context, committing a transaction, and rollbacking a transaction. Table 4 describes the QD and SR values for these concerns.

In line with the results from the case study described in Sect. 2, the calculated QD values show that JAccounting's AO version has been largely based on quantified statements. The three implemented advices have achieved the maximum possible amount of quantification (QD = 1). Furthermore,

Table 4 JAccounting's QD and SR

Concern	*jps*	*adv*	SR	QD
begin transaction	15	1	14	1
commit	15	1	14	1
rollback	14	1	13	1

Table 5 JSpider's QD and SR

Concern	*jps*	*adv*	SR	QD
debug(Object)	45	44	1	0.02
debug(Object, Throwable)	12	12	0	0.00
error(Object)	12	11	1	0.09
error(Object, Throwable)	68	68	0	0.00
fatal(Object, Throwable)	1	1	0	0.00
info(Object)	44	38	6	0.14
warn(Object)	2	2	0	0.00

aspects have removed the need of implementing code related to starting a transaction from 14 static locations of the base program (SR = 14). The same value was obtained for committing a transaction. Finally, for rollbacking, SR = 13.

Example 4 (JSpider) Logging in JSpider is implemented using Log4J[7], a popular logging package for Java. In Log4J, logging messages are categorized according to some developer-chosen criteria and logging requests are made by invoking printing methods from a `Logger` instance. The provided methods are `debug`, `info`, `warn`, `error`, and `fatal`. Table 5 presents the QD and SR values for this concern.

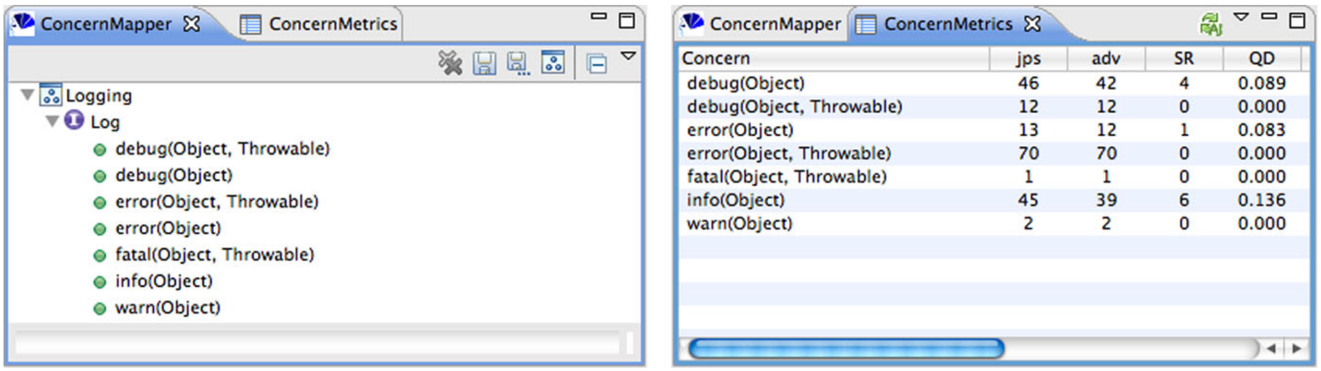
As can be observed, the calculated QD values have confirmed that the AO version uses very few quantified statements, since the values are zero or very close to zero. In other words, such values express the way that advices are used in JSpider's aspect-oriented version. In most cases, logging calls have just been moved from methods to classes, in an one-to-one basis.

Example 5 (JHotDraw) JHotDraw is an object-oriented framework for 2D graphics.[8] The system has been initially proposed as a "design exercise" to demonstrate the application of design patterns in a real application. Later, it has also been used as a case study in several papers about AOSD [1, 6, 34]. Finally, some concerns of the system have been refactored to aspects by Marin et al. [35], leading to a version called AJHotDraw.[9] In this example, we have calcu-

[7] http://logging.apache.org/log4j.

[8] http://www.jhotdraw.org, version v.54b1.

[9] http://sourceforge.net/projects/ajhotdraw.



- Logging
  - Log
    - debug(Object, Throwable)
    - debug(Object)
    - error(Object, Throwable)
    - error(Object)
    - fatal(Object, Throwable)
    - info(Object)
    - warn(Object)

Concern	jps	adv	SR	QD
debug(Object)	46	42	4	0.089
debug(Object, Throwable)	12	12	0	0.000
error(Object)	13	12	1	0.083
error(Object, Throwable)	70	70	0	0.000
fatal(Object, Throwable)	1	1	0	0.000
info(Object)	45	39	6	0.136
warn(Object)	2	2	0	0.000

Fig. 8 Concern map and the metrics estimated for JSpider's logging concern

Table 6 JHotDraw's QD and SR

Concern	*jps*	*adv*	SR	QD
fireSelectionChanged()	4	1	3	1
setUndoActivity(Undoable)	13	2	11	0.92
checkDamaged()	28	15	13	0.48

lated the proposed QD and SR metrics for three methods related to different crosscutting concerns: `fireSelectionChanged` (that is part of the figure selection concern), `setUndoActivity` (that is part of the undo concern), and `checkDamaged` (that is part of the command concern). We have chosen such methods by comparing the OO and AO versions of the system. In the AO version, they are called by code that resides in advices.

Table 6 presents JHotDraw's QD and SR values. As can be observed, for two methods we have found high values for QD: `fireSelectionChanged` (QD = 1) and `setUndoActivity` (QD = 0.92). On the other hand, 15 advices have been implemented to modularize the existent 28 calls to `checkDamaged`. Therefore, QD = 0.48, which shows that intermediary QD values can be observed in practice (since in the other examples the values are usually in the extremes of the scale).

4 ConcernMetrics tool

We have implemented a prototype tool, called ConcernMetrics that estimates both conventional separation of concern metrics and the quantification metrics proposed in this paper without requiring developers to implement aspects. Our main motivation is to help developers and maintainers to decide in a cost-effective way if it is worthwhile to use aspects in their systems.

The implemented tool reuses the interface provided by the ConcernMapper Eclipse-based plugin, proposed by Robillard et al. to logically reorganize the modularity of a software system [40]. In fact, the ConcernMapper allows developers to build a logical model of the concerns of a software system that is similar to our concern model.[10] Basically, they just need to drag-and-drop methods and fields associated to crosscutting concerns to a tree-like data structure called concern map. This structure can then be used to reason about concerns without requiring changes in the source code.

Our ConcernMetrics tool leverages the ConcernMapper plugin by supporting the estimation of CDC, CDO, QR, and SR metrics directly from the information available in the concern map. In order to start using the plugin developers must first create a concern map and drag-and-drop to it methods previously classified as presenting a crosscutting behavior (possibly, using an aspect mining tool [9, 25]). For example, in the case of the logging concern in JSpider, developers must drag-and-drop to the concern map methods such as `debug(Object,Throwable)`, `debug(Object)`, `error(Object)`, etc. (as shown in Fig. 8).

4.1 Implementation

In order to calculate the separation of concerns and quantification metrics from the information available in the concern map, the ConcernMetrics tool relies on the ASM framework for Java bytecode manipulation and analysis.[11] Using this framework, the bytecode of the system is transversed looking for calls to the methods included in the concern map. In order to estimate CDC, CDO, QR, and SR in case aspects

[10]The main difference is that ConcernMapper assumes a flat concern model, i.e., it is not possible to define sub-concerns, as proposed in Sect. 3.

[11]http://asm.objectweb.org.

are used to modularize the mapped concerns, the crucial decision is to identify the method calls that can be extracted to the same advice. For this purpose, the following decision algorithm is used:

> Suppose the following calls to method m: $t_1.m(arg_1)$ and $t_2.m(arg_2)$, where t_i denotes the target and arg_i denotes the argument of the calls ($i = 1$ or $i = 2$). These calls can be moved to a single advice when the following condition holds:
>
> 1. In case of static methods, t_1 and t_2 must denote the same class (or classes having a common superclass). In case of dynamic methods, t_1 and t_2 must denote fields having the same type (or that are derived from a common type).
> 2. arg_1 and arg_2 are the same constant value or fields having the same type (or that are derived from a common type).

Suppose the calls to `start` and `log` in the following classes:

```
class A {                          class B {
  Transaction tx;                    Transaction tx;
  void foo(){                        void bar(){
    tx.start(1);                       tx.start(1);
    ....                               ....
    Logger.log("finished");            Logger.log("panic");
    ....                               ....
  }                                  }
}                                  }
```

In case aspects are used in this system, the `start` calls can be moved to a single advice because: (i) their target are fields having the same type (`Transaction`); (ii) their arguments are the same integer constant (`1`). On the other hand, the `log` calls should be extracted to different advices, because although they rely on a static method of the same class (`Logger`), their arguments are distinct strings.

4.2 Examples

In order to evaluate the proposed decision algorithm, we have applied the ConcernMetrics tool to our three case studies: JAccounting, JSpider, and JHotDraw. In general, the results suggested by the ConcernMetrics have matched the real values calculated from the AO versions of these systems. In case the results have not matched, they were due to new requirements implemented by the AO code or to missing calls in the AO versions. The details are discussed in the following examples.

Example 6 (JAccounting) Table 7 presents the SR and QD values calculated from the JAccounting's AO version and the values estimated by the ConcernMetrics directly from the object-oriented version of the system.

Since the ConcernMetrics has provided precisely the same values measured in the aspect-oriented version of the system, we can conclude that it has been able to infer the same design decisions made by the developers of JAccounting-AOP.

Table 7 SR and QD for JAccounting's transaction handling concern (AO = value measured for the AO version; CM = value estimated by the ConcernMetrics)

Concern	AO		CM	
	SR	QD	SR	QD
beginTransaction	14	1	14	1
commit	14	1	14	1
rollback	13	1	13	1

Table 8 SR and QD for JSpider's logging concern (AO = value measured for the AO version; CM = value estimated by the ConcernMetrics)

	Concern	AO		CM	
		SR	QD	SR	QD
1	debug(Object)	1	0.02	4	0.09
2	debug(Object,Throwable)	0	0.00	0	0.00
3	error(Object)	1	0.09	1	0.08
4	error(Object,Throwable)	0	0.00	0	0.00
5	fatal(Object,Throwable)	0	0.00	0	0.00
6	info(Object)	6	0.14	6	0.14
7	warn(Object)	0	0.00	0	0.00

Example 7 (JSpider) Table 8 presents the SR and QD values calculated from the JSpider's AO version and the values estimated by the ConcernMetrics from the object-oriented version of the system.

As we can observe in this table, the values indicated by the ConcernMetrics have matched the real AO values in five out of seven methods considered in our evaluation (more especifically, methods 2, 4, 5, 6, and 7). The differences observed in the remainder methods are explained in the following way:

- The AO version misses some logging messages, i.e., there are logging messages that exist in the OO version, but are not implemented in any advice from the AO version. For example, there are 46 calls to method `debug(Object)` in the OO version analyzed by the ConcernMetrics tool. However, in the AO version, only 45 from such calls are reinserted in the OO code by the implemented advices. Due to the high number of logging messages in JSpider, we believe the developers of the AO version have forgot to extract some messages to the implemented aspects.
- Probably to enhance the quality of the logging implementation, the AO version includes new logging messages. In other words, there are logging messages in the AO version that do not exist in the original OO code. For exam-

```
class EventDispatcherImpl {
   public EventDispatcherImpl (...) {
     ...
     log.debug("EventFilter for engine events = " + ...);
     log.debug("EventFilter for monitor events = " + ...);
     log.debug("EventFilter for spider events = " + ...);
     ...
}
```

Fig. 9 Consecutive calls to `debug` in JSpider

Table 9 SR and QD for JHotDraw (AO = value measured for the AO version; CM = value estimated by the ConcernMetrics)

	Concern	AO		CM	
		SR	QD	SR	QD
1	fireSelectionChanged()	2	0.67	3	1
2	setUndoActivity(Undoable)	1	0.13	11	0.92
3	checkDamaged()	12	1	13	0.48

ple, two new calls to `error(Object)` have been implemented in the AO version.

- In some situations, there are consecutive calls to the same logging method in the original OO code. For example, Fig. 9 shows three consecutive calls to `debug(Object)` in the constructor from class `EventDispatcherImpl`. The metrics estimation algorithm employed by ConcernMetrics consider that consecutive calls to methods included in the concern map can be extracted to the same advice. However, in the particular case from Fig. 9, JSpider's developers have decided to implement such calls in different advices.

Example 8 (JHotDraw) Table 9 presents the SR and QD values calculated from the AJHotDraw—JHotDraw's AO version—and the values estimated by the ConcernMetrics directly from the OO version of the system.

The differences observed in the results are explained as follows. The calls to `fireSelection` could have been confined in a single advice (instead AJHotDraw's designers have chosen to implement them in two advices). In the case of `setUndoActivity`, AJHotDraw's designers have left four calls scattered in the OO code (i.e., such calls have not been extracted to aspects). Finally, from the 28 calls to `checkDamaged` that exist in the OO code, only 12 calls have been moved to aspects.

4.3 ConcernMetrics limitations

In order to evaluate the benefits of quantification, we just need to consider dynamic crosscutting concerns, i.e., crosscutting concerns that can be modularized by means of advices [27]. Moreover, the ConcernMetrics tool assumes that (dynamic) crosscutting concerns always correspond to single method calls. Therefore, concerns associated to other statements (assignments, loops, etc.) or concerns associated to multiple statements must be first extracted to a method, using the Extract Method refactoring [19]. However, we believe that requiring the previous application of Extract Method in such cases would not represent a fruitless effort. Even if developers decide to not move to aspects, extracting methods in most cases contribute to eliminate cloned code and to increase comprehensibility.

AspectJ only allow aspects to advise well-defined points in the execution of OO systems. When crosscutting concerns do not happen in advisable join points, developers usually need to transform the base program in order to associate statements implementing crosscutting concerns to static locations that can be captured by AspectJ's pointcuts. Usually, such transformations can be classified as statement reordering or method extraction [7, 37]. However, the ConcernMetrics tool just evaluate if it is possible to extract calls associated to crosscutting concerns to the same advice. It does not consider if a previous transformation in the base program is necessary to enable such extraction. In fact, we have designed another ConcernMapper extension, called TransformationMapper [38] that provides exactly this information. In the future, we have plans to integrate both extensions in a single tool.

5 Related work

Related work can be arranged in five groups: AOP metrics, AOP assessments, ConcernMapper Extensions, AOP refactorings, and AOP languages and systems.

AOP metrics The separation of concerns metrics used in Sect. 2 have been proposed by Sant'Anna et al. to evaluate the benefits of aspects in the implementation of design patterns [20, 41]. They have also shown how to adapt to AOP classical coupling and cohesion metrics proposed by Chidamber and Kemerer (CK) [10]. Ceccato and Tonella have also refined CK's metrics and applied them to a small example; an AspectJ implementation of the Observer design pattern [8]. They have proposed a metric called Crosscutting Degree of an Aspect (CDA) that counts the number of modules affected by the pointcuts and by the introductions in a given aspect. However, CDA only provides a coarse-grained notion about quantification. It neither presents a ratio-based behavior nor it is associated to a previously defined concern model, as in the case of our QD metric. Zhao and Xu have proposed a set of metrics to assess cohesion in aspect-oriented systems [45]. However, the proposed metrics have not been applied to realistic AspectJ systems.

In order to investigate whether crosscutting concerns cause defects, Eaddy et al. have proposed two new separation of concern metrics, called Degree Of Scattering across Classes (DOSC) and Degree Of Scattering across Methods (DOSM) [14]. Unlike Garcia's absolute CDC and CDO metrics, DOSC and DOSM consider how the concern's code is distributed among the program elements. Particularly, DOSC and DOSM present a ratio-scale measure, where zero means "no scattering."

Herrejon and Apel have proposed a set of metrics to categorize crosscutting concerns within a spectrum that goes from homogeneous to heterogeneous [32]. They rely on a metric called Homogeneity Quotient (HQ) that classifies a concern as homogeneous when the number of classes that are crosscut by the pieces of advice in the aspect-oriented implementation of the concern is equal to the number of classes that are crosscut by the concern's advices and intertype declarations. According to this metric, transactions in JAccouting and logging in JSpider are both homogeneous concerns, since in these systems the defined advice crosscut all the classes crosscut by the concern. Therefore, different from our approach the HQ metric does not consider quantification in order to classify a concern as homogeneous or heterogeneous.

Code Replication Reduction (CRR) is another metric proposed to evaluate AspectJ systems [2]. Basically, CRR measures the number of lines of code that could be reduced by moving scattered and tangled code to aspects. CRR's value is calculated by multiplying the number of lines of each advice and intertype declaration by the number of join points it affects (minus one). There are two main differences between CRR and our SR metric. First, CRR denotes the number of lines of code saved by relying on aspects. On the other hand, SR counts the number of static program locations affected by each advice (minus one). Second, CRR gives a unique and global number for the whole system, while our SR metric is calculated for individual concerns previously organized in a hierarchical concern model.

AOP assessments Using their separation of concerns and adapted CK metrics, Garcia et al. have investigated the use of aspects in a wide range of domains and systems, including design patterns [20, 41], exception handling [17], web-based information systems [29], and software product lines [15]. In general, in each of such empirical studies they have identified positive and negative effects of using aspects. Interestingly, in most studies the situations where they do not recommend the use of aspects can be linked to a reduced degree of quantification. For example, in the empirical study about exceptions, they mention that when the exception handling code is nonuniform aspects can bring more harm than good. As another example, in their work about design patterns, they mention that separation of concerns metrics cannot be considered as the only factor to conclude for the use of aspects, since it can generate more complicated designs (as we have observed for example in the JSpider case study).

Kästner, Apel, and Batory have reported their experience on refactoring features from the Oracle Berkeley DB into aspects [24]. They have observed that the extracted aspects in general present a small degree of quantification, i.e., reduced number of advices that can affect more than one execution point (join point). For example, they mention that from the 482 extracted advices only 7 advise more than one joint point. They also mention that most of the defined pointcuts are tightly coupled to the base program and, therefore, are particularly fragile to modifications in this program. We believe that by analyzing the QD and SR metrics reported by the ConcernMetrics tool, it would be possible to reach the same conclusions without extracting any piece of advice from the original object-oriented code.

Apel has analyzed the use of AspectJ in eleven systems, using metrics such as fraction of classes, interfaces, and aspects (CIA), code replication reduction (CRR), code fraction associated with static and dynamic crosscuts (SDC) etc. [2] The study has shown that 86% of the considered code is object-oriented, 12% uses basic crosscutting mechanisms (intertype declarations or single method extensions), and only 2% uses advanced crosscutting mechanisms (including advices that affect whole sets of join points). Since in most of the evaluated systems AspectJ has been used to extract heterogeneous features from the source code, the results reinforce the conclusions derived from the Oracle Berkeley DB aspectization.

ConcernMapper extensions ConcernTagger is a ConcernMapper extension that computes scattering metrics including CDC, CDO, and the previous mentioned DOSC and DOSM metrics [14]. However, at least in its current version, the metrics are measured only for the OO code. On the other hand, a distinguishing feature of our ConcernMetrics tool is its ability to infer QD and SR's values regarding an eventual aspectization of the code. ConcernMorph is another ConcernMapper extension designed to detect crosscutting patterns based on metrics collected directly from the OO code [16].

AOP refactorings Several researches have investigated aspect-oriented refactorings, i.e., refactorings that prescribe how to modularize crosscutting concerns using aspects [6, 7, 12, 30, 36]. However, they in general do not address the issue about how to decide when the proposed refactorings are worthwhile.

AOP languages and systems Systems such as AHEAD [5, 31] and FeatureC++ [4] represent another alternative to the implementation of heterogeneous concerns, mainly in the context of feature-oriented programming and collaboration languages [3, 33]. In such systems, heterogeneous concerns are normally associated to increments in program functionality. Therefore, they do not present the massive crosscutting behavior that is observed in concerns such as logging and transactions. For this reason, AHEAD and FeatureC++ do not support quantification. On the other hand, in this paper, we restricted our focus to dynamic crosscutting concerns implemented using AspectJ's advice and pointcut abstractions, which are exactly the abstractions that distinguish AspectJ from the mentioned systems.

6 Conclusions

In this section, we conclude describing the contributions and the limitations of our work. We also outline future research lines.

Contributions We have argued that quantification is the key mechanism to abstract out computations associated to crosscutting concerns. Quantification allows developers to implement in a single advice code required in many static locations of object-oriented systems. Therefore, quantification directly tackles the code scattering and tangling problems that typically characterize crosscutting concerns. In order to better assess the benefits of quantification, we have proposed two metrics: quantification degree (QD) and scattering reduction (SR). As another contribution of the paper, we have implemented the ConcernMetrics Eclipse-based plugin that calculates such metrics without requiring the physical extraction of crosscutting concerns to aspects. The ConcernMetrics tool has been used to evaluate in advance the benefits of using aspects in three small-to-medium Java systems. From the best of our knowledge, ConcernMetrics is the first tool that estimates AOP-related metrics directly from the object-oriented code. The system is still a research prototype. However, its current version is available upon request from the authors.

Limitations We have decided to do not provide precise guidelines about when aspects should be employed in a given system, based on the estimated QD and SR values. We believe that the final decision should be made by the developers of the candidate system. The reason is that this decision usually involves other variables, as the crosscutting concern nature (including its importance, implementation difficulty, frequency of maintenance, etc.) and the experience of the development team in aspect-oriented technologies. Despite this fact, we consider that it is crucial that developers correlate QD an SR values in order to make this decision. Since QD is a ratio-based metric, it provides information about the gains that would be achieved by using aspects with respect to the best possible scenario. On the other hand, SR provides absolute information about the code reduction achieved with aspects.

We have concentrated solely in the aspectization of dynamic crosscutting concerns. However, it is important to mention that AspectJ also provides support to static crosscutting mechanisms (a.k.a intertype declarations), which allow developers to instrument the static structure of an object-oriented system without changing its source code. Inter-type declarations can be used to support other programming approaches, such as software product lines [11], feature-oriented programming [5, 33], and collaboration-based designs [3]. However, such programming approaches are not the central focus of this paper. Finally, we have concentrated on the dynamic crosscutting mechanisms provided by AspectJ, since it is the most widely used AOP language nowadays.

A possible side effect of high QD is the pointcut fragility problem [28]. According to this issue, high degrees of quantification might lead to less stable systems for evolution, since new code might be inadvertently affected by high QD pointcuts. On the other hand, AO development tools can mitigate this problem, by detecting which parts of the code are affected by advices and their pointcuts.

Further research We intend to apply the proposed QD and SR metrics to other studies. We have plans to extend the ConcernMetrics with support to other metrics. We also have plans to integrate ConcernMetrics with our previous TransformationMapper tool, which provides information about OO transformations required to enable the extraction of aspects. Finally, in another study, we will aim to address the (possible) correlation between pointcut fragility and high QD values.

Acknowledgements This work has been supported by grants from FAPEMIG and CNPq. We would like to thank David Binkley and Mariano Ceccato for providing us the source code of the aspect-oriented version of the JSpider and JAccounting systems. The authors also thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Anbalagan P, Xie T (2007) Automated inference of pointcuts in aspect-oriented refactoring. In: 29th international conference on software engineering (ICSE), May 2007
2. Apel S (2010) How AspectJ is used: an analysis of eleven AspectJ programs. J Object Technol 9(1):117–142
3. Apel S, Batory D (2006) When to use features and aspects: a case study. In: 5th international conference on generative programming and component engineering (GPCE), pp 59–68
4. Apel S, Leich T, Rosenmüller M, Saake G (2005) FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In: 4th international conference on generative programming and component engineering (GPCE). Lecture notes in computer science, vol 3676. Springer, Berlin, pp 125–140
5. Batory D (2004) Feature-oriented programming and the AHEAD tool suite. In: 26th international conference on software engineering (ICSE), pp 702–703
6. Binkley D, Ceccato M, Harman M, Ricca F, Tonella P (2005) Automated refactoring of object oriented code into aspects. In: 21st IEEE international conference on software maintenance (ICSM), pp 27–36
7. Binkley D, Ceccato M, Harman M, Ricca F, Tonella P (2006) Tool-supported refactoring of existing object-oriented code into aspects. IEEE Trans Softw Eng 32(9):698–717
8. Ceccato M, Tonella P (2004) Measuring the effects of software aspectization. In: 1st workshop on aspect reverse engineering (WARE 2004)
9. Ceccato M, Marin M, Mens K, Moonen L, Tonella P, Tourwé T (2005) A qualitative comparison of three aspect mining techniques. In: 13th international workshop on program comprehension (IWPC), pp 13–22
10. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
11. Clements P, Northrop LM (2002) Software product lines: practices and patterns. Addison-Wesley, Reading
12. Cole L, Borba P (2005) Deriving refactorings for AspectJ. In: 4th international conference on aspect-oriented software development (AOSD), pp 123–134
13. Colyer A, Clement A (2004) Large-scale AOSD for middleware. In: 3rd international conference on aspect-oriented software development. ACM, New York, pp 56–65
14. Eaddy M, Zimmermann T, Sherwood KD, Garg V, Murphy Gail C, Nagappan N, Aho AV (2008) Do crosscutting concerns cause defects? IEEE Trans Softw Eng 34(4):497–515
15. Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari FC, Khan SS, Filho FC, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: 30th international conference on software engineering (ICSE), pp 261–270
16. Figueiredo E, Whittle J, Garcia AF (2009) ConcernMorph: metrics-based detection of crosscutting patterns. In: 7th international symposium on foundations of software engineering (FSE), pp 299–300
17. Filho FC, Cacho N, Figueiredo E, Maranhao R, Garcia A, Rubira C (2006) Exceptions and aspects: the devil is in the details. In: 14th international symposium on foundations of software engineering (FSE), pp 152–162
18. Filman RE, Friedman DP (2000) Aspect-oriented programming is quantification and obliviousness. In: OOSPLA workshop on advanced separation of concerns, October 2000
19. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley, Reading
20. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, de Lucena CJP, von Staa A (2005) Modularizing design patterns with aspects: a quantitative study. In: 4th international conference on aspect-oriented software development (AOSD), pp 3–14
21. Garcia A, Greenwood P, Heineman G, Walker R, Cai Y, Yang HY, Baniassad E, Lopes CV, Schwanninger C, Zhao J (2007) Assessment of contemporary modularization techniques (ACoM) 2007: workshop report. SIGSOFT Softw Eng Notes 32(5):31–37
22. Greenwood P, Bartolomei TT, Figueiredo E, Dósea M, Garcia AF, Cacho N, Sant'Anna C, Soares S, Borba P, Kulesza U, Rashid A (2007) On the impact of aspectual decompositions on design stability: an empirical study. In: 21st European conference on object-oriented programming (ECOOP), pp 176–200
23. Hilsdale E, Hugunin J (2004) Advice weaving in AspectJ. In: 3rd international conference on aspect-oriented software development (AOSD), pp 26–35
24. Kastner C, Apel S, Batory D (2007) A case study implementing features using AspectJ. In: 11th international software product line conference (SPLC), pp 223–232
25. Kellens A, Mens K, Tonella P (2007) A survey of automated code-level aspect mining techniques. Trans Aspect-Oriented Softw Dev 4:145–164
26. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: 11th European conference on object-oriented programming (ECOOP). LNCS, vol 1241. Springer, Berlin, pp 220–242
27. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: 15th European conference on object-oriented programming (ECOOP). LNCS, vol 2072. Springer, Berlin, pp 327–355
28. Koppen C, Störzer M (2004) PCDiff: attacking the fragile pointcut problem. In: European interactive workshop on aspects in software (EIWAS), September 2004
29. Kulesza U, Sant'Anna C, Garcia A, Coelho R, von Staa A, Lucena C (2006) Quantifying the effects of aspect-oriented programming: a maintenance study. In: 22nd IEEE international conference on software maintenance, pp 223–233
30. Laddad R (2003) Aspect-oriented refactoring. TheServerSide.com
31. Liu J, Batory D, Lengauer C (2006) Feature oriented refactoring of legacy applications. In: 28th international conference on software engineering (ICSE), pp 112–121
32. Lopez-Herrejon R, Apel S (2007) Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies. In: 10th international conference on fundamental approaches to software engineering (FASE), March 2007
33. Lopez-Herrejon R, Batory D, Cook WR (2005) Evaluating support for features in advanced modularization technologies. In: 19th European conference on object-oriented programming (ECOOP). LNCS, vol 3586. Springer, Berlin, pp 169–194
34. Marin M, van Deursen A, Moonen L (2007) Identifying crosscutting concerns using fan-in analysis. ACM Trans Softw Eng Methodol 17(1)
35. Marin M, van Deursen A, Moonen L, van der Rijst R (2009) An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw. Autom Softw Eng 16(2):323–356
36. Monteiro MP, Fernandes JM (2006) Towards a catalogue of refactorings and code smells for AspectJ. Trans Aspect-Oriented Softw Dev 3880:214–258
37. Nassau M, Valente MT (2009) Object-oriented transformations for extracting aspects. Inf Softw Technol 51(1):138–149
38. Nassau M, Oliveira S, Valente MT (2009) Guidelines for enabling the extraction of aspects from existing object-oriented code. J Object Technol 8(3):101–119
39. Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

40. Robillard MP, Weigand-Warr F (2005) ConcernMapper: simple view-based separation of scattered concerns. In: OOPSLA eclipse technology exchange workshop (ETX), pp 65–69
41. Sant'Anna C, Garcia A, Chavez C, Lucena C, von Staa A (2003) On the reuse and maintenance of aspect-oriented software: an assessment framework. In: 17th Brazilian symposium on software engineering (SBES), pp 19–34
42. Soares S, Laureano E, Borba P (2002) Implementing distribution and persistence aspects with AspectJ. In: Proceedings of the 17th ACM conference on object-oriented programming, systems, languages, and applications, OOPSLA'02, Seattle, WA, USA, November 2002. ACM, New York, pp 174–190
43. Steimann F (2006) The paradoxical success of aspect-oriented programming. In: 21st conference on object-oriented programming systems, languages, and applications (OOPSLA), pp 481–497
44. Wand M (2003) Understanding aspects: extended abstract. In: 8th international conference on functional programming (ICFP). ACM, New York, pp 299–300
45. Zhao J, Xu B (2004) Measuring aspect cohesion. In: 7th fundamental approaches to software engineering (FASE). Lecture notes in computer science, vol 2984. Springer, Berlin, pp 54–68