Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method

Eduardo Figueiredo[1], Alessandro Garcia[2], Cláudio Sant'Anna[1],
Uirá Kulesza[1], Carlos Lucena[1]

[1]PUC-Rio, Computer Science Department, LES
Rio de Janeiro - Brazil
{emagno, claudio, uira, lucena}@les.inf.puc-rio.br

[2]Lancaster University, Computing Department, InfoLab 21
Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk

Abstract. Aspect-oriented (AO) software development is an emerging paradigm that provides new abstractions and mechanisms to support the modularization of crosscutting concerns through the software development. However, the achievement of high-quality AO software is not trivial. The inappropriate use of aspect-oriented abstractions and mechanisms potentially leads to the violation of important design principles, such as low coupling, high cohesion, incomplete modularization of crosscutting concerns into aspects, and so forth. These problems are not easily detectable and an *ad hoc* analysis of large designs and implementations is often expensive and time-consuming. Hence there is a need for an assessment method that assists software engineers in the analysis of their AO designs and implementations. This paper reports our efforts in the ongoing development of a systematic approach to support the quantitative assessment of aspect-oriented artifacts generated through the system design and implementation. The approach is organized in a stepwise fashion and is founded on a metrics suite and a comprehensive set of complementary rules. Our proposal is supported by a prototype measurement tool and has been applied to four medium-sized software systems in different domains and with distinct degrees of complexity.

1 Introduction

Several concerns in software development cannot be represented in a modular fashion using existing software engineering abstractions. They inherently crosscut system modules and their crosscutting structure can manifest not only at the implementation level but also at earlier development stages. A concern is crosscutting if it is *tangled* to other concerns in a single module or it is *scattered* over multiple system modules. Aspect-oriented software development (AOSD) is an emerging paradigm that provides new abstractions and mechanisms to support the modularization of crosscutting concerns through the software development. An aspect is a new modular unit to capture both scattered and tangled concerns. The expected benefits of AOSD are superior separation of concerns, minimized code replication, improved module cohesion, reduced systemic coupling, and, as a consequence, increased potential for reuse and ease of evolution in the development of complex software systems.

However, the aspects themselves may be easily the locus of further complexity and also reduce the quality of the classes affected by them [7, 8]. In fact, the achievement of high-

quality AO artifacts is challenging for five main overlapping reasons. First, software developers are empowered with additional decomposition means, which can easily lead to the misuse of this new paradigm [19]. Second, the separation of concerns achieved with AO techniques does not come always for free – sometimes it impacts negatively other important software attributes [7]. The inappropriate application of aspect-oriented abstractions potentially leads to the violation of important design principles, like low coupling, high cohesion, and lack of information hiding. Third, the early identification of domain-specific aspects requires some systematic reasoning about the design elements. Fourth, even when all aspects are successfully identified, the complete modularization of crosscutting structures is not always straightforward as it is not trivial to capture all the pieces of crosscutting behaviors. Finally, the internal design of the aspects themselves can also entail crosscutting-related problems [15].

While AOSD has become an increasingly important research topic in recent years, insufficient attention has been paid to methods for evaluation of AO designs and implementations. In general, the literature only includes some isolate case studies that assess the quality of AspectJ implementations, which are mainly focused on issues related to separation of concerns. The main reason for this problem is that it is very difficult to assess multiple factors without a systematic analysis approach and supporting tools. As a result, software engineers have assumed that the most impacted property of an aspect-oriented system is separation of concerns. However, some recent studies (e.g. [7, 8]) have shown that other fundamental software engineering principles, such as low coupling and high cohesion, need to be assessed in conjunction with separation of concerns issues.

This paper addresses these shortcomings by proposing a quantitative assessment method for aspect-oriented software development. We conjecture that properly assessing the relevant attributes of aspect-oriented design and implementation is a prerequisite for achieving high-quality AO software, and that exploiting those attributes will open up a broader design evaluation, which is essential to allow the AO software engineers reason about and make a proper trade-off analysis between different solution alternatives. The proposed method is structured in a stepwise fashion, and is supplemented by: (i) a set of design and implementation rules, and (ii) a metrics suite for separation of concerns, coupling, cohesion, and size [19]. The metrics introduce quantitative flavors to our method, while the rules complement it with some qualitative rationale. A prototype supporting tool is also presented to facilitate the analysis of complex designs. We also report our experience in applying the proposed method to four different case studies.

The remainder of this paper is organized as follows. Section 2 presents basic concepts on AOSD. Section 3 presents our assessment approach. Section 4 introduces the current state of the architecture and implementation of our tool. Section 5 illustrates the use of our approach using an example, and discusses the evaluation of our method in the context of four case studies. Section 6 briefly describes comparisons with related work. Section 7 includes some concluding remarks and description of ongoing work.

2 Background on AOSD

Aspect-oriented software development [14, 22] has been proposed as a technique for improving separation of concerns (SoC) in software construction and support improved reusability and maintainability. The central idea is that while pure abstractions of the object paradigm are extremely useful, they are inherently unable to modularize tangled and

scattered concerns. *Aspects* are modular units of crosscutting concerns that are associated with a set of classes or objects. Central to the process of composing aspects and classes is the concept of *join points*, the elements that specify how classes and aspects are related. Join points are well-defined points in the structure and dynamic execution of a system. Examples of join points are method calls, method executions, and field sets and reads. *Advice* is a special method-like construct attached to join points. An aspect may also define attributes and methods to be introduced into the classes to which the aspect is attached. Weaver is the mechanism responsible for composing the classes and aspects. AspectJ [1] is a practical aspect-oriented extension to the Java programming language. AspectJ supports the definition of aspects, advices, join points, and introduction of structures by inter-type declarations [1].

3   The Assessment Method

This section presents our method for supporting the assessment of aspect-oriented artifacts. The method is structured according to two major phases: design and implementation. At the design phase, the method encompasses four main steps, which are organized in four internal activities. The method is grounded on design metrics and rules that are governed by more generic issues because they consider the higher-level information available at the design level; they are applied to aspect-oriented design artifacts, including both structural and behavioural UML-based diagrams. These steps and resources are programming language agnostic.

The implementation phase consists of metrics and rules that are tailored to specific features of the AspectJ programming language [1]; most of the design metrics and rules are redefined to cope with specific AspectJ constructs (e.g. inter-type declarations) [1]. The design metrics and rules are also reapplied to the code since the design artifacts are refined and some desirable properties may have been violated. Due to space limitations, this paper will focus on the description of the implementation phase.
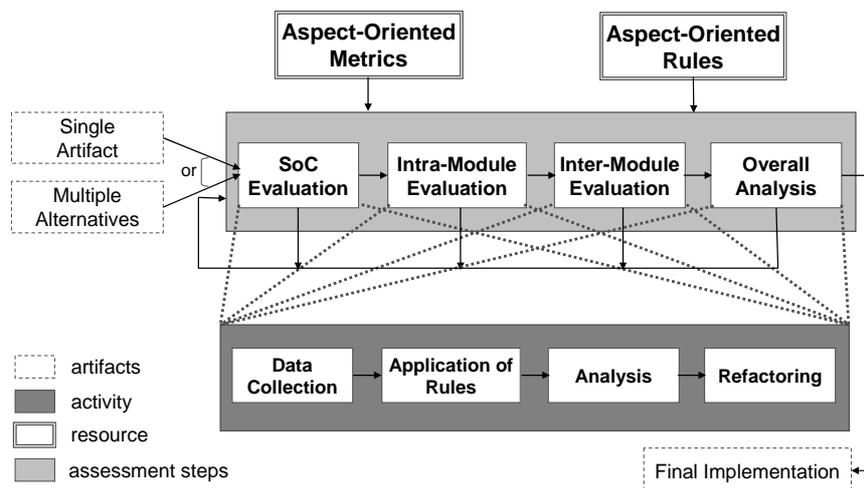

Figure 1. The proposed assessment method.

Our assessment method is based on the principle that not only separation of concerns should be the driving factor of analysis, but also other essential software engineering attributes, such as coupling, cohesion, and size. In this context, the implementation is first assessed in terms of to what extent the concerns of interest in an application are modularized through the implementation units (Figure 1), such as classes, aspects, methods, and advices. Afterwards, one or more alternative design solutions are assessed in terms of the internal complexity of the modules, and then they are assessed from a systemic perspective, i.e. with respect of the module relationships. Finally, the system is assessed in terms of more general attributes, such as size.

The evaluation of separation of concerns and the subsequent phases are composed of four activities: (i) *data collection* – application of the metrics to collect data with respect to a specific software attribute, (ii) *application of rules* – evaluation of the measures in the light of well-defined design and implementation rules, (iii) *analysis* – reasoning and judgment of the outcomes generated by the rules application, and (iv) *refactoring* – a set of changes may be required to be undertaken as a result of the analysis activity.

3.1 Metrics Overview

Our method relies on a suite of aspect-oriented metrics for separation of concerns, coupling, cohesion and size. These metrics have already been used in three different studies [7, 8, 19]. This metrics suite was defined based on the reuse and refinement of some classical and OO metrics [3, 4]. The metrics suite also encompasses new metrics for measuring separation of concerns. Table 1 presents a brief definition of some metrics of the suite, and associates them with the attributes measured by each one. Refer to [19] for further details about the metrics.

Table 1. The Metrics Suite

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	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.
	Concern Diffusion over Operations (CDO)	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.
	Concern Diffusions over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
	Depth Inheritance Tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.

3.2 Assessment Steps

This subsection presents the four steps of our assessment method: (i) SoC evaluation, (ii) intra-module evaluation, (iii) inter-module evaluation, and (iv) overall analysis. Due to space limitation and simplicity purposes, we will describe the SoC evaluation step in more

detail since the other ones follow similar inner activities. We will only emphasize the main issues that distinguish the other ones from the SoC evaluation step.

Step 1. Evaluation of Separation of Concerns. Evaluation of separation of concerns is the first step of our method since the main goal of using aspects is to improve the separation of crosscutting concerns. This step aims at helping the designer to determine which aspect-oriented design decisions effectively contribute to improve the modularization of concerns tangled and scattered in the components of the software system being developed or maintained. The first activity in this step, data collection, consists of the application of the metrics of separation of concerns (Table 1). In order to apply these metrics, the designer must define the concerns of interest and identify which implementation elements are affected by each of concern. These elements can be classes, methods, attributes, aspects, inter-type declarations, and pieces of code. The identification and annotation of these elements are supported by measurement tool described in Section 4. Besides, the definition of the concerns of interest can be based on predefined concern models [21].

The second activity involves the application of rules for assessing the design in terms of separation of concerns; the rules (Fig. 2) are applied over the results gathered from the measures of separation of concerns. The application of them will result in a classification of the concerns into *scattered concerns* and *tangled concerns*. A tangled concern is a concern which is interleaved with other concerns within a single component (i.e. a class or an aspect). A scattered concern is a concern spread over the implementation of multiple components. If a concern is scattered, it is also tangled as a consequence. The goal of this classification is to highlight potential design problems related to crosscutting concerns that are not trivial to detect, such as, (i) crosscutting concerns that are not easily identified as such, (ii) parts of a crosscutting concern which remains not modularized by aspects in which they should be localized, or (iii) crosscutting concerns spread over the aspect structure, e.g. methods replicated in subaspects of the same aspect or the repetition of the use of inter-type declarations which could be avoided by using the *Container Introduction* idiom [9].

A number of rules for SoC evaluation are shown in Figure 2. Figure 2 also presents the relationship between the rules and the aspect-oriented metrics (Section 3.1) they use in their computation. The rules RSoC1 and RSoC2 provide evidence of existing scattered concerns by using respectively the Concern Diffusion over Components metric (CBC) and the Concern Diffusion over Operation metric (CDO). In addition, the rule RSoC3 uses the Concern Diffusion over LOC (CDLOC) for advising the designer of the probable existence of tangled concerns.

The application of the rules warns the designer of potential problems caused by crosscutting concerns. However, it does not guarantee that the problems really exist, and the implementation should be changed. Therefore, if the application of the rules

---

RSoC1:
IF CONCERN is spread across several components (CDC metric)
THEN CONCERN is a SCATTERED CONCERN

RSoC2:
IF CONCERN is spread across several operations (CDO metric)
THEN CONCERN is a SCATTERED CONCERN

RSoC3:
IF the lines of code of a CONCERN is tangled with the lines of code of other concerns (CDLOC metric)
THEN CONCERN is a TANGLED CONCERN

Figure 2. Examples of SoC Rules

produces warnings, the designer should analyse the design and code in order to find out what are the reasons for the warnings. This task is represented by the analysis activity of the method (Figure 1). Actually, the warnings encompass information that helps the designer to concentrate on certain parts of the design and code which are possibly problematic. After the analysis, if the designers detected that there are real problems caused by crosscutting concerns, they should re-engineer the design and/or code in order to well localize the crosscutting concerns using an appropriate aspect-oriented refactoring, such as *Extract Feature into Aspect* [16]. Finally, the designer should re-apply the metrics to the re-engineering components and compare the results to the data gathered before the changes in the design. This comparison aims at detecting design improvements or degenerations.

Step 2. Evaluation of Intra-Module Issues. Since the SoC evaluation has been concluded, an internal evaluation of each component' complexity needs to be followed. The evaluation process encompasses all the components, i.e. not only the aspects, but also the classes. One or more implementation alternatives may have to be analyzed. New implementation solutions may have also been generated as a result of the refactoring activities undertaken in the SoC evaluation process (Step 1). The complexity of each module is assessed according to three dimensions: (i) cohesion, (ii) operation complexity, and (iii) number of attributes. A number of rules and metrics are associated with each of these dimensions in order to categorize each system module similarly to the SoC evaluation step.

For example, the components are classified according to their internal cohesion into five categories: *highly cohesive component*, *cohesive component*, *average*, *non-cohesive component*, and *highly non-cohesive component*. The cohesion analysis is important because the separation of concerns previously achieved in each implementation alternative can affect positively or negatively the cohesivity of the system modules [7, 8]. When a given modular decomposition is leading to several non-cohesive and highly non-cohesive components, the developers probably will decide for discarding that specific AO implementation alternative.

Additional cohesion-related problems can be identified here such as the lack of cohesivity in one or more aspects, and different refatoring-based decisions may need to be taken. For instance, the internal implementation of a given aspect may be aggregating non-related chunks of information and behavior and, as consequence, such an aspect needs to be decomposed into two or more separate aspects, each dealing with a specific chunk. This problem may also be the motivation for decomposing a single aspect into an abstract generic aspect and one or more concrete specialized aspects. The result of applying the intra-module rules over the measurement data and classify the system modules also serve as signs or warnings of problems to the software engineers.

Figure 3 illustrates one of the cohesion rules, called *RCohesion4*, used to identify the non-cohesive components. A "component" in this rule means a class or

RCohesion4:
IF internal ATTRIBUTES and OPERATIONS are loosely coupled to each other (LCOO metric)
THEN COMPONENT is a NON-COHESIVE COMPONENT

RCoupling5:
IF COMPONENT is dependent on several other components (CBC metric)
THEN COMPONENT is a HIGHLY COUPLED COMPONENT

RInheritance6:
IF COMPONENT is very far down in an inheritance tree (DIT metric)
THEN COMPONENT is a DEEP COMPONENT

Figure 3. Examples of Cohesion, Coupling, and Inheritance Rules

an aspect, while an "operation" is a method, an advice, or a method defined as part of an inter-type declaration (Section 2). This rule is applied to the data gathered with the LCOO metric (see Section 3.1 – Table 1), which is the mechanism to quantify the cohesion of a given component. It is worth to highlight that these rules need to be specialized in our tool implementation (Section 4) by associating multiple ranges and threshold values with them. As the tool implementation progresses, these thresholds values are going to be abstracted from our experimental experience and from the long-term application of our method to new empirical studies. The tool will also allow the application engineers to set up these values according to their specialized knowledge and preferences.

Step 3. Evaluation of Inter-Module Issues. The third step guides the programmers of the AO system in analyzing the inter-module complexity of their AspectJ code. The current version of our method includes the analysis of two inter-module issues: coupling and inheritance. These attributes are measured on the basis of the CBC and DIT metrics (Table 1). As in the previous steps, multiple solution alternatives may have to be analyzed, including the original ones and those ones generated as a result of refactorings realized in the two previous steps.

Rules are also used here to classify each component with respect to its coupling to the rest of the system and its level in the inheritance tree. For example, with regard to coupling, the modules are categorized into five distinct types: *totally non-coupled component, highly coupled component*, *coupled component*, *average*, *low coupled component*. A similar classification is defined for the inheritance attribute. Figure 3 presents examples of coupling and inheritance rules. Again, the application of the metrics and rules can identify some symptoms (or *bad smells* [16]) that signalize potential problems in the aspect-oriented solutions. For example, an aspect can be identified as highly coupled to the classes that it is affecting. If in the previous step, this aspect and those classes were ranked as non-cohesive components, it probably consists in an evidence that this specific aspect-oriented solution should be discarded by the software developers.

Step 4. Overall Analysis. In this phase, the software developers analyze general software attributes, such as the tally of components in each implementation alternative, and LOC-related measures. The latter can help to identify for example some remaining replicated code over the implementation of different advices, methods, classes, and aspects. They can also apply more sophisticated rules that combine different software attributes. At the end, they decide for the solution that seems to mostly satisfy the performed evaluations and their specific quality requirements.

4 A Measurement Tool

This section presents the architecture and implementation of the tool that supports the measurement activities of our assessment method. The goals of the tool are: (i) to compute all the metrics presented in Section 3.1; and (ii) to support the rules application step. The current tool implementation just provides support to the metric collection in systems implemented using Java and AspectJ. Figure 4 shows the tool architecture focusing on the modules required to compute the measures according to our metrics suite. The tool defines four main modules, as follows, AspectJ Model Extractor, Concern Manager, Metric Collector and Rule Analyzer. Figure 4 also illustrates the steps when using the tool.
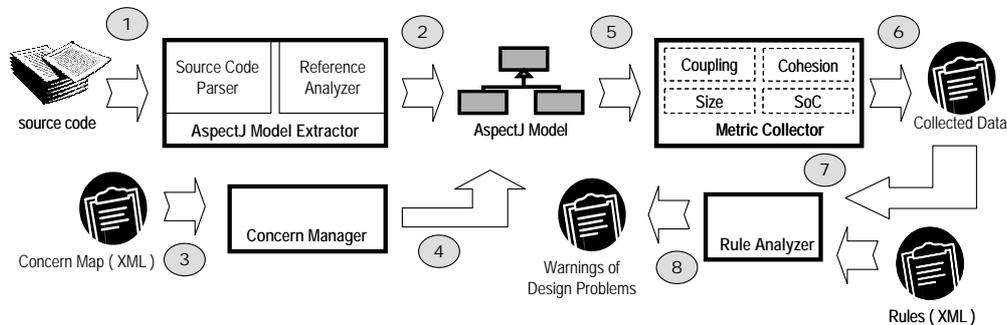
Figure 4. Metric tool architecture.

The AspectJ Model Extractor module takes as input all the source files (classes, interfaces and aspects) and detects the structure of the analyzed files, in terms of their components, attributes, pointcuts, operations and statements (steps 1 and 2). The Extractor module is composed of two sub-modules: (i) Source Code Parser (SCP); and (ii) References Analyzer. The SCP parsers the AspectJ code and partially build a representative model of the system, called AspectJ Model. This model is a suitable representation of the source code which makes the metrics-based data collection easier. The SCP sub-module uses MetaJ environment [17] to manipulate source code. MetaJ is a meta-programming environment build as a Java extension which works with code written in different programming languages. The Reference Analyzer sub-module is responsible to capture information related to reference types that exist between code elements. Examples of reference types are: import, inheritance and association. After collect the reference types, the Reference Analyzer updates the initial model generated by the SCP using this information.

The application of the SoC metrics requires a mapping from the abstract crosscutting concern to program elements in the code that implements that specific concern. Therefore, after the AspectJ Model Extractor module has captured information about the program syntactic elements, it is necessary to map them to specific concerns which the developer would like to analyze. The Concern Manager module implements the mapping from syntactic elements in AspectJ model to concerns defined by the developer (steps 3 and 4).

The Metric Collector module is responsible for computing the metrics (step 6). It takes as input the AspectJ model (step 5). Finally, the purpose of Rule Analyzer module is to use the collected metric
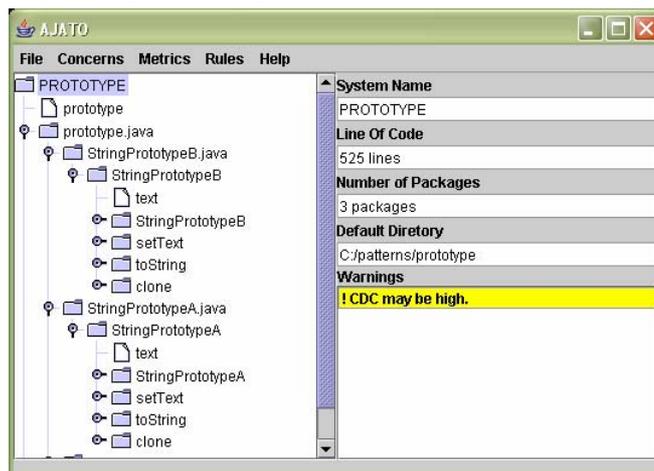
Figure 5. Tool interface with a CDC warning.

data and design/implementation rules (step 7) to generate specific warnings of design problems (step 8). As mentioned before, this module is still under development.

The current user interface of the tool presents two main views (as shown in figure 5): tree view and data view. The tree view (left side of the Figure 5) presents the system model as a tree. The data view (right side) shows information about the selected node on the tree. The tool also allows the configuration of metrics and rules to be applied to analyze a multiple implementation alternatives.

5 Evaluation

5.1 Case Studies

The empirical evaluation of our assessment method is very important to understand the usefulness and effectiveness of the set of steps, rules, and metrics. Our approach has been used in the context of four different empirical studies with different characteristics, diverse domains, varying control levels and different complexity degrees. The chosen studies are medium-sized software systems that have been designed and implemented by AOSD experts, which argue that the resulting systems are reusable and maintainable AO solutions.

The first study encompassed the investigation of the reusability and the maintainability of OO and AO designs and implementations of a multi-agent system to manage and automate the reviewing process of research conferences [8]. The second study [7] involved the application of our approach to compare Java and AspectJ implementations [10] of the 23 GoF design patterns [6]. The third study [20] was a comparison of the AO and OO versions of a web-based information system in which aspects were used to improve the modularization of distribution and persistence concerns. Finally, our method was also used to analyze different AO design and implementation alternatives to isolate exception handling concerns in a software system called Telestrada [5].

The specific results obtained as well as the respective discussions for each of the studies mentioned above can be found in the cited references. As a general conclusion, we have noticed that the achievement of high-quality AO design is very difficult even for AOSD experts. Several problems happen because there are various concerns of interest to be modularized in a software development and, at the same time, many fundamental engineering principles need to be observed and satisfied in the system modules. The next subsection illustrates the application of the SoC evaluation step of our approach to implementations of the Prototype pattern as part of one of our performed studies [7].

5.2 An Example: the Prototype Pattern

We have decided to use the aspectization of the Prototype pattern [10] as an example because it is included in one of our case studies [7] mentioned above. The intent of the Prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [6]. The Prototype pattern defines a single concern called "Prototype". This concern involves the definition of the behavior for cloning objects. This Prototype's instance is used to create a less limited `String` object that can be cloned, as described in [10]. System classes interested to be cloned need to implement a `Prototype` interface.

The left side of Figure 6 depicts the class diagram of the OO implementation of the Prototype pattern. It shows the `Cloneable` interface which is part of the Java standard

platform. This interface indicates to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of classes that implement it. The `StringPrototypeA` and `StringPrototypeB` classes implement the `Cloneable` interface. Each of these classes defines a `String` object capable to be cloned.

The application of the SoC evaluation steps, as defined in our approach, suggests that there is a problem in the OO version while addressing the Prototype concern. Figure 6 illustrates how the OO implementation of the Prototype pattern is scattered over the code of the application classes. The shadowed attributes and methods represent code necessary to implement the Prototype concern in the application context. The application of the SoC metrics and rules using our tool generate a warning, as presented in the screen of Figure 5.
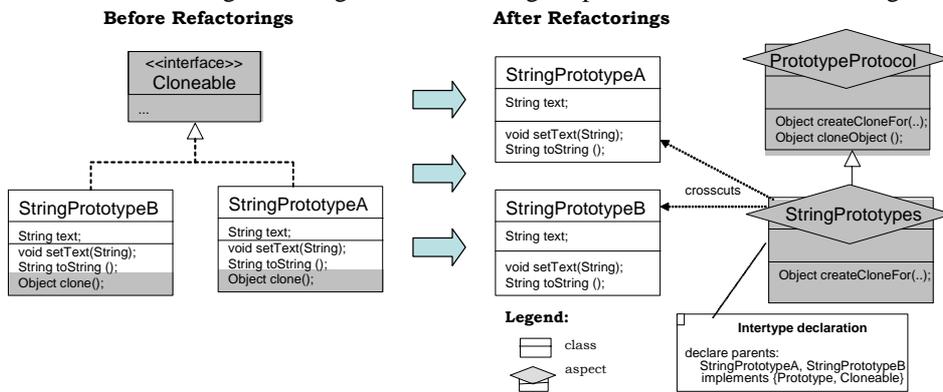
Figure 6. The **a**pplication of SoC Evaluation steps to Prototype implementations

As a result, the system developers can decide to use an aspect-oriented design to isolate the Prototype concern, as defined in [10]. In this AspectJ solution of the Prototype pattern, the code for implementing the pattern is textually localized in two categories of aspects (right side of Figure 6): (i) the `PrototypeProtocol` abstract aspect that encapsulates the common part to all potential instantiations of the pattern; and (ii) concrete extensions of the abstract aspect that instantiate the pattern for specific contexts. The two aspects in Figure 6 are entirely shadowed since all the code within them is related to the Prototype pattern. The Prototype concern is realized as a protected inner interface named `Prototype`. The concrete `StringPrototype` extension of the `PrototypeProtocol` aspect assign the Prototype concern to the application classes. Besides, it also defines a different implementation of the `createCloneFor()` method to allow the clone of string prototypes objects in case of failure during the execution of the `clone()` method.

The AspectJ implementation of the Prototype pattern, as prescribed in [10], separates almost all the code pertaining to the Prototype concern into aspects. However, when the SoC evaluation step is re-applied for this new implementation alternative (as defined in our method – Figure 1) a new problem is detected in the AO version defined by [10]. The `StringPrototypeA` and `StringPrototypeB` classes still implement the `Cloneable` interface. Thus, the software developers can also transfer this interface implementation to the concrete aspects using inter-type declaration (see the right side of Figure 6) in order to achieve a complete modularization of the Prototype concern. In fact, sometimes it is difficult to detect all the pieces of behavior that should be modularized into a specific aspect. Our approach was very useful to detect this problem in the AspectJ implementation

of the Prototype pattern and in other pattern implementations as provided by [10]. The paper [7] summarizes our findings using our assessment approach.

6 Related Work

Tekinerdogan [23] propose an aspect-oriented analysis method, but it is targeted at the evaluation of software architectures in terms of aspect identifications. In addition, it analyses the architectural design with respect to the scenario coverage [23], but it does not assess the architectural design in terms of important software attributes. Our approach, on the other hand, propose a quantitative assessment method for aspect-oriented design and implementation based on fundamental software engineering principles, such as coupling, cohesion, size and separation of concerns.

Aspect-oriented refactorings [12, 16] is already defined in the literature but they are not connected with metrics values for cohesion, coupling, size, and SoC. In [16], a few aspect-oriented bad smells are used to identify refactoring opportunities and [12] presents an analysis of many Fowlers' refactorings and conclude that a few of them can be used in aspect-oriented software without modifications. In [11] is introduced a role-based refactoring, but this approach is not related on software metrics. Catalogue of refactorings can be considered complementary to our method and used in a step of the proposed method.

A number of tools [18, 13] have been developed to specifically address the problem of finding concerns in source code. These approaches can help alleviate the problem of capturing segments of program code related to a concern. Text-based and type-based analyses are used to mining aspects in these approaches. These tools are viewed as complementary to our tool (Section 4) since they can help the designers to find the concerns of interest to be assessed using our method and tool. An aspect-oriented metric tool [2] is proposed to automate the measure process. His work are to a large degree in common with our tool (Section 4), however, [2] does not presents a method for assessment aspect-oriented design and implementation. In addition, our tool supports separation of concerns metrics as well as coupling, cohesion and size.

7 Conclusion and Ongoing Work

AOSD is a topic of growing interest within both the academic and industrial communities. Despite the popularity of this topic, little attention focuses on methods for analyzing software designs and implementations to show that it satisfies certain desirable properties. AOSD is primarily motivated by stringent quality factors, such as maintainability, evolvability, and reusability. This paper has presented method of analysis with respect to fundamental software engineering attributes, such as separation of concerns, coupling, cohesion, and size. This method was derived from our extensive experience on performing systematic assessments of AOSD approaches [5, 7, 8, 20]. Our approach was demonstrated in this paper by means of a case study involving aspect-oriented implementations of design patterns (Section 5). We have also performed some case studies to support our arguments in favor of the usefulness of our assessment method. Up to now, our proposed tool (Section 4) only supports the measurement activities. We are now working on the implementation of additional modules that support the other steps of our method.

References

1. AspectJ Team: The AspectJ Programming Guide. http://eclipse.org/aspectj/.
2. Ceccato, M., Tonella P.: Measuring the Effects of Software Aspectization. In Proc. of the 1st Workshop on Aspect Reverse Engineering (CD-ROM), The Netherlands, (2004).
3. Chidamber, S., Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, (1994), pp. 476-493.
4. Fenton, N., Pfleeger, S.: Software Metrics: A Rigorous Practical Approach. London: PWS, (1997).
5. Filho, F., Rubira, C., Garcia, A.: Assessing Aspect-Oriented Programming for Modularizing Exception Handling. Submitted to ECOOP Work. on Exception Handling in OO Systems, (2005).
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995).
7. Garcia, A. et al.: Modularizing Design Patterns with Aspects: A Quantitative Study. In Proc. of the AOSD'05, Chicago, USA, (2005), pp. 3-14.
8. Garcia, A. et al.: Separation of Concerns in Multi-Agent Systems: An Empirical Study. In Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940, (2004).
9. Hanenberg, S., Schmidmeier, A.: AspectJ Idioms for Aspect-Oriented Software Con-struction, Proc. of EuroPLoP'03, Germany, (2003).
10. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In Proc. of the OOPSLA'02, (2002), pp. 161-173.
11. Hannemann, J., Murphy, G., Kiczales, G.: Role-Based Refactoring of Crosscutting Concerns. In Proc. of the AOSD'05, Chicago, USA, (2005), pp. 135-146.
12. Iwamoto, M., Zhao, J.: Refactoring Aspect-Oriented Programs. In Proc. of the 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, USA, (2003).
13. Janzen, D., Volder, K.: Navigating and querying code without getting lost. In Proc. of the AOSD'03, Boston, Massachusetts, (2003), pp. 178 - 187.
14. Kiczales, G. et al.: Aspect-Oriented Programming. In Proc. of ECOOP'97, LNCS 1241, Springer, Finland, (1997), pp. 220-242.
15. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In Proc. of the AOSD'05, Chicago, USA, (2005), pp. 90-99.
16. Monteiro, M., Fernandes, J.: Towards a Catalog of Aspect-Oriented Refactorings. In Proc. of the AOSD'05, Chicago, USA, (2005), pp. 111-122.
17. Oliveira, A. et al: MetaJ: An Extensible Environment for Metaprogramming in Java. Journal of Universal Computer Science, vol. 10, no. 7, (2004), p. 872-891.
18. Robillard, M., Murphy, G.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In Proc. of the ICSE'02, USA, (2002), pp. 406-416.
19. Sant'Anna, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. of Brazilian Symposium on Software Engineering, Brazil, (2003), pp. 19-34.
20. Soares, S.: An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, (2004).
21. Sutton Jr, S., Rouvellou, I.: Concern Modeling for Aspect-Oriented Software Development, In Aspect Oriented Software Development, Addison-Wesley, (2004), pp 479-505.
22. Tarr, P. et al.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Proc. of the ICSE'99, Los Angeles, USA, (1999), pp. 107-119.
23. Tekinerdogan, B.: ASAAM: Aspectual Software Architecture Analysis Method. In Proc. of the IEEE/IFIP Conference on Software Architecture, Norway, (2004), pp. 5-14.