Reengineering of Object-Oriented Software into Aspect-Oriented Ones Supported by Class Models

Paulo Afonso Parreira Júnior[1(✉)], Rosângela Dellosso Penteado[1(✉)], Matheus Carvalho Viana[1(✉)], Rafael Serapilha Durelli[2(✉)], Valter Vieira de Camargo[1(✉)], and Heitor Augustus Xavier Costa[3(✉)]

[1] Departament of Computer Science, Federal University of São Carlos, São Carlos, Brazil
{paulo_junior, rosangela, matheus_viana, valter}@dc.ufscar.br

[2] Computer Systems Department, University of São Paulo, São Carlos, Brazil
rdurelli@icmc.usp.br

[3] Departament of Computer Science, Federal University of Lavras, Lavras, Brazil
heitor@dcc.ufla.br

Abstract. Object-Oriented Software Reengineering (OO) into Aspect-Oriented Software (AO) is a challenging task, mainly when it is done by means of refactorings in the code-level. The reason is that direct transformation from OO code to AO one needs of several design decisions due to differences of both paradigms. To make this transformation more controlled and systematic, we propose the use of concern-based refactorings, supported by class models. It allows design decisions to be made during the reengineering process, improving the quality of the final models. An example is presented to assess the applicability of the proposed refactorings. Moreover, we also present a case study, in which AO class models created based on the refactorings are compared with another obtained without the aid of them. The data obtained indicated that the use of the proposed refactorings improved the efficacy and productivity of maintenance groups during the process of software reengineering.

Keywords: Concern-based refactorings · Class models · Aspect-Orientation · Reengineering

1 Introduction

Aspect-Orientation (AO) can be used in the revitalization of Object-Oriented (OO) legacy software. AO allows encapsulating the so-called "crosscutting concerns" (CCC) - software requirements whose implementation is tangled and scattered by functional modules - in new abstractions such as pointcuts, aspects, advices and inter-type declarations [12].

S. Hammoudi et al. (Eds.): ICEIS 2013, LNBIP 190, pp. 296–313, 2014.
DOI: 10.1007/978-3-319-09492-2_18

Reengineering from OO to AO in code-level is not an easy task due to existing differences between concepts related to both approaches. However, if the reengineering process was supported by models, it could facilitate future maintenance. In this paper we propose the use of concern-based refactorings on OO class models annotated with information of CCC to obtain AO models. In the context of this paper, annotated OO class models are UML OO class models whose elements (classes, interfaces, attributes and methods) are annotated with stereotypes corresponding to the CCC that exist in the software. The main idea is that concern-based refactorings can be applied to transform these models into AO models.

There are many studies in the literature that present code-based refactorings [8, 10, 13, 14, 19]. Our main reasons to create and apply **concern-based refactorings supported by models** ("model-based refactorings" in the rest of this paper) are:

(i) code-level refactorings can be applied to transform OO software in AO ones. However, this transformation is usually done in one step, which has as input an OO code and as output an AO one. It makes the reengineering process less flexible, because the responsibility to generate a code that follows good design practices of AO is on the refactorings. The transformation supported by model-based refactorings introduces at least one more step in the process before generating the final code. Thus, to ease the inflexibility of the process, in this step the outcome AO model can be modified by the software engineer according to the environment and stakeholder requirements;

(ii) generally, the source code is the only available artifact of the legacy software. Applying model-based refactorings, both the legacy software and the generated software will have a new type of artifact (i.e., UML class models), improving their documentation; and

(iii) unlike the code-based refactorings, model-based ones are platform independent. Thus, models can be transformed and good designs can be produced regardless of programming language.

A set of nine model-based refactorings was developed [1]. It is subdivided into: (i) three generic refactorings, which are concern-independent refactorings; and (ii) six specific refactorings to the following concerns: persistence (subdivided into connection, transaction and synchronization management), logging and Singleton and Observer design patterns [6]. Due to the limitation of space, only five of them are presented in more details in this paper. The AO class models presented in this paper are based on AOM (Aspect-Oriented Modeling) approach proposed by Evermann [4].

The remainder of this paper is structured as follows. Some concepts related to the AOM approach proposed by Evermann, the annotated OO class models and the computational support DMAsp [3], used to generate automatically annotated OO class models, are discussed in Sect. 2. The generic and specific refactorings are presented in Sect. 3. An example that illustrates the use of one generic refactoring is shown in Sect. 4 and an evaluation of whole set of refactorings is presented in Sect. 5. Some related works are summarized in Sect. 6. Finally, conclusions and suggestions for future work are presented in Sect. 7.

2 Background

ProAJ/UML (UML Profile for AspectJ) is one of the most used approaches to model AO software [4]. This approach consists of a set of stereotypes that can be applied on UML class models, such as:

- <<CrossCuttingConcern>>: it is an extension of the *Package* meta-class, in the UML meta-model. Its aim is to encapsulate aspects related to the same cross-cutting concern;
- <<Aspect>>: it extends the UML *Class* meta-class. Its goal is to cluster *pointcuts* and *advices* in an aspect and to allow aspects to extend classes or aspects and implement interfaces;
- <<Advice>>: it is a *BehavioralFeature* meta-class extension. Its aim is to associate advices with aspects; and
- <<PointCut>>: it is a *StructuralFeature* meta-class extension, whose goal is to specify a static behavior. Its modelling is performed by concrete subclasses of *PointCut*, such as *CallPointCut* and *ExecutionPointCut*.

These stereotypes are used in the AO class models generated with the application of the refactorings proposed in this work. Furthermore, OO class models annotated with information of CCC are used in the proposed refactorings. These annotations are represented using stereotypes on the left side of the classes, interfaces, attributes and methods identifiers. Figure 1 illustrates a class annotated with indications of persistence CCC. The DMAsp (Design Model to Aspect) tool [3], developed in a previous work, is used to generate automatically the annotated OO class models.

Based on the concept of annotated OO class models, the following concepts, proposed by Figueiredo [5], were adapted to the context of this work and are commented in the refactoring descriptions.

- **Components Affected by a Concern** are software elements such as classes, interfaces, attributes and methods which have indications of this concern. These elements are annotated with stereotypes of the concern that affect them;
- **Primary Concern** is the main concern of a component and it is related to the reason by which it was created. For example, the `openConnection` method (Fig. 1) was created to open database connections. Then "Persistence" is the primary concern of this method. The primary concerns are identified by the prefix "Pri_" in the stereotypes;
- **Secondary Concern** of a component corresponds to functions that this component plays. However, these functions are not directly related to the reason for which it was created. The `Account` class and its method `withdraw`, Fig. 1, were created to perform the business rules of a hypothetical banking system. Thus "Persistence"

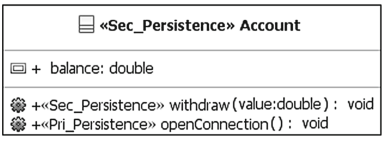

Fig. 1. An UML class annotated with information about persistence CCC.

is a secondary concern in these components. The secondary concerns are identified by the prefix "Sec_" in the stereotypes; and

- **Well-modularized Components** are software elements composed only by the primary concern for which they were created. For example, the `openConnection` method (Fig. 1) is considered well-modularized, because the only type of stereotype of this method is a primary concern related to "Persistence" concern.

3 Model-Based Refactorings

Hannemann [9] proposed the following classification of AO software refactorings:

(i) **conventional OO refactorings adapted for AO software**. These refactorings only involve OO elements. The difference between these refactorings and the well-known OO refactorings is they are aware of the existence of AO elements;
(ii) **specific refactorings for AO software**. These refactorings involve OO and AO elements and they are specific to lead to the AO abstractions, such as aspects, pointcuts, etc.; and
(iii) **crosscutting concerns refactorings**. Also called concern-based refactorings, they should take all the elements (classes, aspects, interfaces, etc.) that participate in a crosscutting concern and their relationships into consideration. This happens, because concerns usually are manifested in several components.

A set of nine concern-based refactorings is shown in Table 1. Only five of them are described in this paper with more details. The remaining refactorings were omitted for reasons of limitation of space and can be found in [16].

The refactorings are presented with: (i) Acronym and Name of the Refactoring; (ii) Application Scenario, which defines the situations the refactoring can be applied; (iii) Motivation, which presents some problems caused by tangling and scattering of CCC; and (iv) ProAJ/UML Mechanism, which is a set of steps to obtain an AO class model from an OO one, according to the ProAJ/UML profile.

3.1 Generic Refactorings

The generic refactorings are responsible for transforming an annotated OO class model to a partial AO class model. The generated model is named "partial", because

Table 1. Model-based refactorings.

Generic Refactorings	
Name	**Description**
R-1	Encapsulating a Secondary Concern with Association Relationships.
R-2	Encapsulating a Secondary Concern with Generalization/Specialization Relationships.
R-3	Extracting a Primary Concern.
Specific Refactorings	
Name	**Description**
R-Connection	Encapsulating the CCC responsible for managing database connections.
R-Transaction	Encapsulating the CCC responsible for managing database transactions.
R-Sync	Encapsulating the CCC responsible for managing database synchronization.
R-Logging	Encapsulating the CCC responsible for controlling the application of logging record.
R-Singleton	Encapsulating the CCC corresponding to the Singleton design pattern.
R-Observer	Encapsulating the CCC corresponding to the Observer design pattern.

the existing CCC may not be well-modularized yet. In this case, there still can exist classes/interfaces, methods and/or attributes affected by crosscutting concerns. All three generic refactorings are presented as follows.

R-1. Encapsulating a Secondary Concern with Association Relationships.

Application Scenario: when there are classes with Primary Concerns (Crosscutting Concerns) which are Secondary Concerns in other classes and these classes are related through association/aggregation relationships.

Motivation: the invocation of methods of classes which Primary Concern is a Crosscutting Concern can improve the tangling/scattering of this concern, hurting the software maintenance.

ProAJ/UML Mechanism: 1) Create a *CrossCuttingConcern* element called "CCC", in which "CCC" represents the concern name that is being modularized; **2)** Inside the element created previously, add an *Aspect* element called "CCCAspect"; **3)** Move each well-modularized attribute and method from the classes affected by the concern to the "CCCAspect" element; and **4)** Move all classes that have the concern in analysis as a Primary Concern to the "CCC" element.

R-2. Encapsulating a Secondary Concern with Generalization/Specialization Relationships.

Application Scenario: when there are classes with Primary Concerns (Crosscutting Concerns) which are Secondary Concerns in other classes and these classes are related through generalization/specialization relationships.

Motivation: the override of methods of classes which Primary Concern is a Crosscutting Concern can improve the tangling/scattering of this concern, hurting the software maintenance.

ProAJ/UML Mechanism: 1) Create a *CrossCuttingConcern* element called "CCC", in which "CCC" represents the concern name that is being modularized; **2)** Inside the element created previously, add an *Aspect* element called "CCCAspect"; **3)** Move each well-modularized attribute from the classes affected by the concern to the "CCCAspect" element; **4)** For each well-modularized method from the classes affected by the concern, create a "IntroductionMethod" element to add this method to the aspect "CCCAspect"; and **5)** Move the inherence and interface realization to the aspect "CCCAspect", using "DeclareParents" elements.

R-3. Extracting a Primary Concern.

Application Scenario: when there are classes with Secondary Concerns, which are Crosscutting Concerns and these ones are not Primary Concerns in any classes.

Motivation: some crosscutting concerns can be scattered in several classes and there are not specific classes that implement them. One concern of this type is not a Primary Concern in any class of the application. This scenario represents a high level of concern tangling and a low level of software modularization.

ProAJ/UML Mechanism: 1) Create a *CrossCuttingConcern* element called "CCC", in which "CCC" represents the concern name that is being modularized; **2)** Inside the element created previously, add an *Aspect* element called "CCCAspect"; and **3)** Move each well-modularized attribute and method from the classes affected by the concern to the "CCCAspect" element.

Looking at the application scenarios of these refactoring we understand that: "no matter what concern we are dealing, the scenario described above represent a low level of modularization".

***R-Singleton*. Encapsulating the CCC corresponding to the Singleton Design Pattern.**

Application Scenario: when there are classes dedicated to implementation of Singleton pattern.

Motivation: the Singleton pattern can cause problems of tangling and scattering of concerns in OO application. The modularization using AO is one alternative to solve these problems [7].

ProAJ/UML Mechanism: 1) Identify the "CCCAspect" aspect related to implementation of the singleton concern and verify if this aspect is abstract. If not, transform it into abstract one; **3)** Create, inside the element "CCC" related to the singleton concern, an empty interface called `Singleton`; **4)** Define an execution pointcut called `instance` that intercepts the calls to constructor of the classes that realize the `Singleton` interface and add it to the "CCCAspect"; **5)** Identify the set of classes, "S", that implement the Singleton pattern, *i. e*., the classes whose instance must be unique in the application. If the constructor of these classes be *private*, transform it into *public*; **6)** For each class "N" ϵ "S", create an aspect "CCCAspectN", where "N" corresponds to the class name and create inheritance relationships from the aspects "CCCAspectN" to the aspect "CCCAspect"; **7)** Each aspect "CCCAspectN" created previously must declare an interface realization relationship between the class "N", represented by this aspect, and the interface `Singleton`; and **8)** Create an around advice that returns a `Singleton` object and associate it to the `instance` pointcut. This advice implements the logic of the Singleton pattern: if there exists an instance, return it; otherwise, create one instance and return it.

***R-Transaction:* Encapsulating the CCC corresponding to the Database Transaction Management.**

Application Scenario: when there are classes dedicated to implementation of Transaction CCC as a Secondary concern.

Motivation: applications generally have transactional methods that cause a high rate of concern tangling/scattering in the software. The modularization of these concerns can facilitate the software understanding, maintenance and reusability.

ProAJ/UML Mechanism: 1) Identify the "CCCAspect" aspect that corresponds to concern of transaction control and verify if this aspect is abstract. If not, transform it into abstract one; **2)** Define an abstract pointcut, called `transactionalMethods()`, and one around advice related to this pointcut; **3)** For each class "N" affected by this concern, create an aspect "CCCAspectN", where "N" corresponds to the class name and create inheritance relationships from the aspects "CCCAspectN" to the aspect "CCCAspect"; **4)** Identify the transactional methods of these classes and create one concrete appropriate pointcut for each aspect, called `transactionalMethods()`.

For example, based on the *R-3* refactoring, we already can apply a modularization strategy to this concern, whatever it is, putting all the well-modularized elements (attributes and methods) related to this concern in a specific module, in this case, an aspect. In all refactorings presented above, only well-modularized elements are moved to the aspect, avoiding problems related to the dependence of other concerns.

3.2 Specific Refactorings

The specific refactorings are responsible for transforming partial AO class models in final ones. These refactorings are named "specific", because they only can be applied to a specific type of concern. For example, there is a specific refactoring to the transaction management concern that generates an AO class model with the modularization of this concern using aspects. Six specific refactorings were developed, as presented in Table 1.

These refactorings were created based on the most common strategies for implementing these types of crosscutting concerns. For example, the database connection concern is usually implemented with a class responsible for creating connections and each persistent method must open the connection at the beginning of its execution and close it at the end. In another example, the singleton pattern is generally implemented as follows [6]: (i) create an attribute of the same type of the Singleton class; (ii) become private the constructor of the Singleton class; and (iii) create a method responsible for keeping only one instance of the Singleton class. Therefore, it is possible to define some steps for modularization of this type of concern, based on the most common strategies for implementing them.

The specific refactorings are applied on the models generated by the generic refactorings. Thus, in ProAJ/UML Mechanism description, aspects created previously are mentioned. To illustrate this case the refactorings *R-Singleton* and *R-Transaction* are presented.

Unlike the generic refactorings, in this case, we can use some more specific steps to modularize the CCC, because they are well-known concerns. Thus, for example, in the case of the Singleton concern, the aspect created by a generic refactoring has been transformed into an abstract one and for each class affected by this concern one aspect has been created. This strategy follows a good practice for AO design suggested by Piveta [18]. Furthermore, it is similar to Hannemann and Kiczales' solution [7] and was adapted to the context of annotated OO class models.

3.3 Considerations About the Refactorings

Some of the main reasons to apply generic refactorings are: **(i) the application of generic refactorings can facilitate the achievement of a better AO model**: wrong decisions made by software engineers, due to their inexperience, can prejudice the AO model quality. Thus, an initial modularization strategy offered by these refactorings can minimize this problem; and **(ii) generic refactorings can be applied to any type of concern, even to those concerns that are not widely known as crosscutting**

concerns: it is not easy to identify whether a particular concern is or not a crosscutting concern. Thus, with the help of generic refactorings, we can identify scenarios that demonstrate or provide evidence of the existence of crosscutting concerns in software. For example, the application scenario for the refactoring *R-3* states a configuration that can evidence the existence of a crosscutting concern (many classes of software related to a secondary concern in these classes).

There is not a specific sequence to apply generic refactorings proposed in this work. The steps created for refactoring are applied when a specific element is well-modularized, *i.e.*, when there is no interference of other concerns in this element. Moreover, some modularization strategies described in the steps of the refactoring were considered to avoid interference in the order of execution of the refactoring. Similarly to what happens with the generic refactorings, the order in which the specific refactorings are applied does not interfere in the final AO class model. It happens because each refactoring acts only on a particular concern at a time, not compromising elements related to other concerns.

The manual execution of the steps described in the refactoring presented on class models of software for medium and large scale can be hard and error-prone. Thus, an Eclipse plug-in called MoBRe (Model-Based Refactorings) was developed to perform tasks related to refactoring of crosscutting concerns in a semi-automatic way. MoBRe [17] allows transforming an annotated class model into a partial AO class model, when the generic and specific refactorings are applied. The AO class models generated can be visualized within the Eclipse.

4 Example of Use

To present the applicability of the proposed refactorings, an example, using the Health Watcher software [20], is presented. This software registers complaints in the health area and it was chosen because it: (i) has an OO and an AO version; and (ii) was modularized by expert software engineers by using best practices of AO design.

The crosscutting concern partially modularized in this example is the Singleton pattern, represented by the "Singleton" stereotype. Other crosscutting concerns affect this application, such as connection and transaction management, represented by the "Conn" and "Trans" stereotypes, but the modularization of them is not performed in this paper because of limitations of space.

One part of Health Watcher OO class model, responsible for the maintenance of the patient complaints, is presented in Fig. 2. This model is annotated by using stereotypes of the concerns that affect the software classes, according to the information provided by Soares et al., [20].

The `HealthWatcherFacade` class provides methods necessary for execution of the business logic of the application, as complaints registration, diseases, and symptoms. The `singletonHW` and `singletonPS` attributes and the `getInstanceHW` and `getInstancePS` methods have "Singleton" as Primary Concern, because they were created specifically for implementing the Singleton pattern. The same way, "Conn" and "Trans" are Primary Concerns of the `IPersistenceMechanism` interface and the `PersistenceMechanism` class, because they were

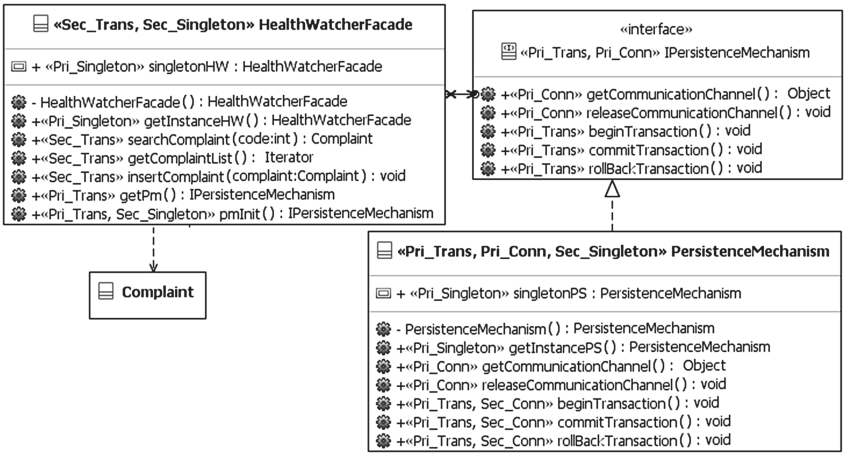

Fig. 2. A UML class stereotyped with CCC indications.

created for implementing these concerns. The `HealthWatcherFacade` class has "Trans" and "Singleton" as Secondary Concerns, because this class was not created for implementing these concerns, but it is affected by them. This information about what concerns are primary or secondary one was provided by the Health Watcher developers.

According to the scenario of tangling/scattering of the model presented in Fig. 2, the singleton concern can be initially refactored by the *R-3* refactoring. This happens because "Singleton" is a Secondary Concern in some classes of this model and it is not a Primary Concern in none other classes. After applying the *R-3* refactoring to the "Singleton" concern, the partial AO class model presented in Fig. 3 was obtained.

The changes made were: (i) the `SingletonAspect` aspect was created; and (ii) the `singletonHW` and `singletonPS` attributes and the `getInstanceHW` and

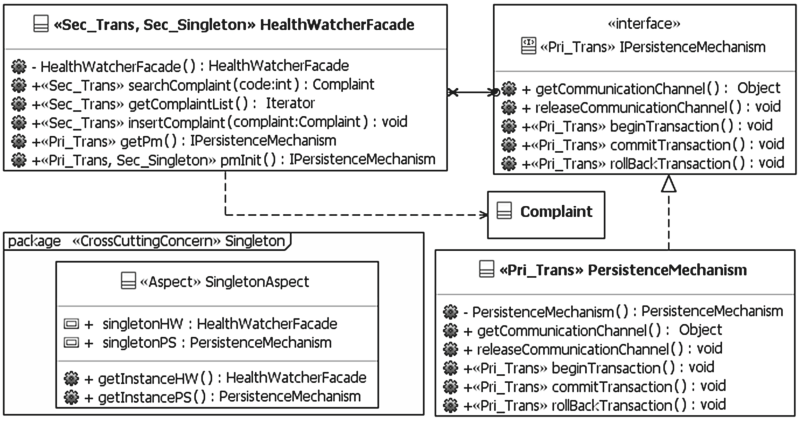

Fig. 3. Health watcher AO class model obtained through R-3 refactoring.

`getInstancePS` methods were moved to the `SingletonAspect` aspect. It is because these elements are well-modularized in the `HealthWatcherFacade` and `PersistentMechanism` classes.

The application of the *R-Singleton* refactoring was omitted for reasons of limitation of space and can be found in [1].

5 Evaluation

The crosscutting concern modularization may be performed with or without the assistance of refactorings. In the second case, the process of modularization is extremely dependent on the expertise of the software engineer. He/She must have knowledge about the crosscutting concerns to be modularized and best practices and strategies for the modularization of these concerns. Refactorings minimize this dependence, making the final product (modularized software) more standardized and improving its quality.

The question we want to answer with this case study is: *how much can the refactorings affect the efficacy of the modularization process and the productivity of the maintenance group?* In this context, productivity is defined as the time that a group takes to modularize the crosscutting concerns of a software product. Besides, efficacy consists in verifying whether all crosscutting concerns were suitably modularized or not. Thus, the case study was carried out and it is shown in the next subsections.

5.1 Case Study Definition

The efficacy and productivity evaluation of the refactorings was performed in two ways:

(i) comparing the generated AO class models with another version of them, obtained from a reverse engineering using the AO code found in the literature (in this study, we use the JSpider AO code available in [11]). To do this, a set of seven **Metrics for Modularization** were used to compare both versions of the application AO class model (Table 2). All of them, except *MQ* and *AVG(MQ)*, accept the following values: 1.0 – Completely Compliant; 0.5 - Partially Compliant; and

Table 2. Metrics for modularization.

Metric	Metric Description
CC_As	Correctly Created Aspects: specifies if all needed aspects were correctly created.
CM_AM	Correctly Modularized Attributes and Methods: specifies if all attributes and methods affected by a concern were correctly modularized.
CC_PA	Correctly Created Pointcut and Advices specifies if all needed pointcuts and advices were correctly created.
CC_GSR	Correctly Created Generalization and Specialization Relationships: specifies if all needed relationships were correctly created.
CS_PC	Correctly Specified Profile Concepts: specifies if all ProAJ/UML concepts were correctly used.
MQ	Modularization Quality: CC_As + CM_AM + CC_PA + CC_GSR + CS_PC.
AVG(MQ)	Average of the Metric *MQ*.

0.0 – Not Compliant. These values are assigned to the metrics for modularization by specialists after comparing the models created by the participants of this experiment to the models obtained from the literature. The metrics *MQ* and *AVG(MQ)* accepts values between [0.0; 5.0] and the higher the value of them, the better is the modularization of a concern; and

(ii) comparing the time spent by each participant to complete the modularization of a given OO class model. For this, we used the metric *Productivity* (*Pr*), given to the Formula (1). The higher the value of *Pr*, the better is the productivity of a participant.

$$\mathrm{Pr} = \mathrm{AVG(MQ)} \,/\, \mathrm{T}, \tag{1}$$

where *AVG(MQ)* is the average of the metric *Modularization Quality* and *T* is the time (in hours) spent by a participant to modularize the crosscutting concerns.

5.2 Case Study Planning

(a) Selection of Context and Formulation of Hypothesis. The study was carried out with graduate students at the Federal University of São Carlos. The system used as object of study was JSpider [11], a highly configurable and customizable Web Spider engine. The participants had to modularize the Logging and Singleton crosscutting concerns and generate an AO class model from the OO model classes of the JSpider application.

Four hypotheses were elaborated (Table 3), two of which refer to the efficacy and two ones refer to the productivity. Besides, the metrics *MQ* and *Pr* were used for formulating the hypotheses.

(b) Selection of Variables and Participants. Independent variables are those manipulated and controlled during the experiment. In this context, they are the way how the participants performed the modularization: with or without the use of the refactorings. Dependent variables are those under analysis, whose variations must be

Table 3. Hypotheses of the case study.

Hypotheses for Efficacy	
H_{0Ef}	There is no difference of using refactorings or not using them, regarding the efficacy. H_{0Ef}: $MQ_{WR} = MQ_{WOR}$
H_{1Ef}	There is difference of using refactorings or not using them, regarding the efficacy. H_{1Ef}: $MQ_{WR} \neq MQ_{WOR}$
Hypotheses for Productivity	
H_{0Pr}	There is no difference of using refactorings or not using them, regarding the productivity. H_0: $Pr_{WR} = Pr_{WOR}$
H_{1Pr}	There is no difference of using refactorings or not using them, regarding the productivity. H_0: $Pr_{WR} \neq Pr_{WOR}$
Legends: • $\mathbf{X_{WR}}$, where X is a metric, means: the value of X obtained by a specific participant using the refactorings proposed in this paper (WR = With Refactorings). • $\mathbf{X_{WOR}}$, where X is a metric, means: the value of X obtained by a specific participant that did not use the refactorings proposed in this paper (WOR = Without Refactorings).	

observed. In this experiment, they are the efficacy and productivity. The participants were selected through a convenient non-probabilistic sampling.

(c) Design of the Experiment. The distribution of the participants in groups was done by using a profile characterization questionnaire.

The questions were about their level of experience in OO and AO, modularization and UML profile to modelling AO software. All questions had the possible answers: 1 - None; 2 - Basic; 3 - Medium; 4 - Advanced; 5 - Expert. The obtained values are plotted in the graph shown in Fig. 4.

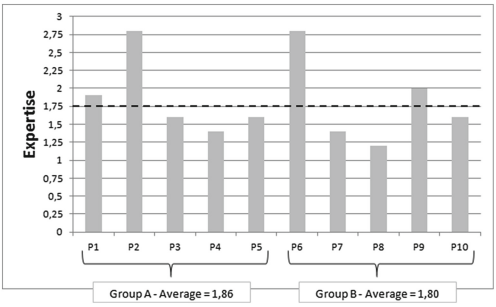

Fig. 4. Expertise of the participants.

The groups were created as follows: Group A - participants P1 to P5; Group B - participants P6 to P10. The average of expertise of Group A is approximately 1.86 and Group B, 1.80, representing that the groups were balanced. To separate the experts and novices we have defined the value 1.75 (horizontal line in Fig. 4). This value was defined according to our experience with the required knowledge to perform the modularization of CCCs. Above this value the participants were considered experts (P1, P2, P6 and P9) and below novices (P3, P4, P5, P7, P8 and P10). It is important to notice that both groups have the same number of expert and novice participants.

The documents used in this experiment were: (i) a registry form to be filled out with information related to the execution of the study; (ii) a script of execution with the steps to be followed to perform the experiment; and (iii) the description of the proposed refactorings. The registry form contained the participant name, the application to be modularized, the starting time and the observations and/or problems noticed by the participant. The script of execution contained a list of tasks that the participants should carry out and had the goal of assisting them and minimizing the possibility of failures during the execution. The description of the proposed refactorings presents the refactoring according the template used in Sect. 3.

The experiment was divided in three phases. In the first phase (*Training*), we conducted a training aimed at homogenizing the knowledge of the participants on the modularization of crosscutting concerns using hypothetical applications. In the second phase (*Pilot*) all participants had to discover how to modularize the persistence concern that crosscuts pieces of the HealthWatcher application manually and using the proposed refactorings. The goal of the pilot was to minimize the difficulties of

following the steps described in the refactorings. Besides, the pilot also was intended to avoid that problems related to the filling out of the forms could interfere in the results of the experiment. In the third phase (*Execution*) the goal was to modularize the Logging and Singleton concerns in the JSpider application. Different types of concern between *Execution* and *Pilot* phases were used to avoid that the knowledge on the persistence concern obtained in the previous phase (*pilot*) to influence the results.

(d) Collected Data. Tables 4 and 5 show the data obtained in third phase of the experiment (*Execution*) by the Groups A and B, respectively (AoE means *Average of the Experts* and AoN means *Average of the Novices*).

Table 4. Execution of the case study – Group A (with refactorings

Data	P1	P2	P3	P4	P5	AVG	AoE	AoN
Time (m)	35	47	60	60	60	52	41	60
Time (h)	0,58	0,78	1,0	1,0	1,0	0,87	0,68	1,0
Singleton Pattern								
CC_As	1,0	1,0	1,0	1,0	1,0			
CM_AM	1,0	1,0	1,0	1,0	1,0			
CC_PA	1,0	1,0	0,5	1,0	0,5			
CC_GSR	1,0	1,0	1,0	1,0	1,0			
CS_PC	1,0	1,0	1,0	1,0	1,0			
MQ	**5,0**	**5,0**	**4,5**	**5,0**	**4,5**	**4,8**	**5,0**	**4,7**
Logging								
CC_As	1,0	1,0	1,0	1,0	1,0			
CM_AM	1,0	1,0	1,0	1,0	1,0			
CC_PA	1,0	1,0	0,5	0,5	0,5			
CC_GSR	1,0	1,0	1,0	1,0	1,0			
CS_PC	1,0	1,0	1,0	1,0	1,0			
MQ	**5,0**	**5,0**	**4,5**	**4,5**	**4,5**	**4,7**	**5,0**	**4,5**
Results								
AVG(MQ)	**5,0**	**5,0**	**4,5**	**4,8**	**4,5**	**4,8**	**5,0**	**4,6**
Pr	**8,57**	**6,38**	**4,5**	**4,8**	**4,5**	**5,75**	**7,48**	**4,6**

Table 5. Execution of the case study - Group B (without refactorings).

Data	P6	P7	P8	P9	P10	AVG	AoE	AoN
Time (m)	38	45	60	60	30	46	49	45
Time (h)	0,63	0,75	1,0	1,0	0,5	0,78	0,69	0,83
Singleton Pattern								
CC_As	1,0	1,0	0,0	0,0	1,0			
CM_AM	1,0	1,0	0,0	0,0	1,0			
CC_PA	0,0	0,0	0,0	0,0	0,0			
CC_GSR	0,0	0,0	0,0	0,0	0,0			
CS_PC	0,0	0,0	0,0	0,0	1,0			
MQ	**2,0**	**2,0**	**0,0**	**0,0**	**3,0**	**1,4**	**1,0**	**1,7**
Logging								
CC_As	1,0	1,0	0,5	0,0	1,0			
CM_AM	1,0	1,0	0,0	0,0	0,5			
CC_PA	0,5	1,0	0,0	0,0	0,0			
CC_GSR	0,0	0,0	0,0	0,0	0,0			
CS_PC	0,0	0,0	0,0	0,0	1,0			
MQ	**2,5**	**3,0**	**0,5**	**0,0**	**2,5**	**1,7**	**1,3**	**2,0**
Results								
AVG(MQ)	**2,3**	**2,5**	**0,3**	**0,0**	**2,8**	**1,6**	**1,1**	**1,8**
Pr	**3,63**	**3,33**	**0,3**	**0,0**	**5,6**	**2,57**	**3,48**	**1,97**

The participants, assigned by "P#", and the titles of the table columns are presented in first line. The time used by the participants for performing the modularization is presented in lines from 2 to 3. The concern names are presented in lines 3 and 10 and the values of the metrics described in Table 2 are presented in lines from 5 to 10 (for the Singleton concern) and from 12 to 17 (for the Logging concern). The average of the metric *MQ* and the value of the metric *Pr* are presented in lines 19 and 20. The columns that contain the data of participants classified as experts were highlighted in gray color.

(e) Data Analysis. Regarding the *Efficacy* and *Productivity*, Tables 4 and 5 show that most of participants that used the proposed refactorings got better results than those ones that do not used them.

Regarding the *Productivity*, just one of the participants that used the proposed refactorings was less productive. Although there was an extra task to be carried out (to follow the steps described in the refactorings), the developer will need to modify the models during the process of refactoring less times, thus minimizing the final time to perform the activity.

As it can be observed in Fig. 5, the average of metric *MQ*, when the participants had the aid of the refactorings was higher than the one when they did not have this aid. According to this chart, the value of the metric *MQ* was 206 % higher, in average, for all participants, and 344 % higher, in average, for participants classified as experts and 150 % higher, in average, for participants classified as novices).

Analogously, in Fig. 6, the average of metric *Pr* also was better when the participants used the refactorings. This chart presents productivity 124 % higher, in average, for all participants, and 115 % higher, in average, for participants classified as experts and 134 % higher, in average, for participants classified as novices).

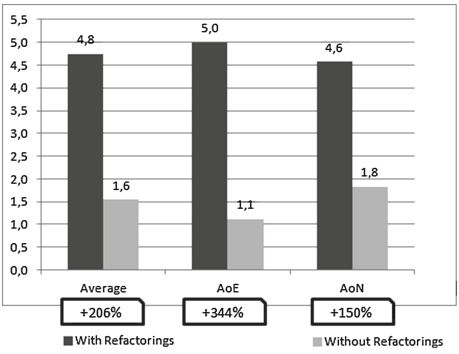

Fig. 5. Average of the metric MQ.

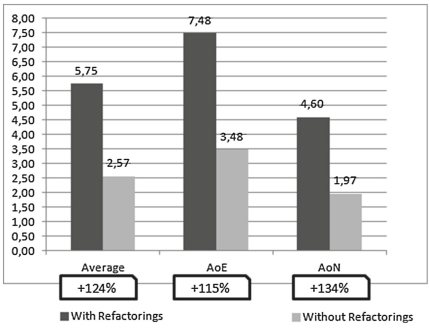

Fig. 6. Average of the metric Pr.

It is also possible to notice in Figs. 5 and 6 that the refactorings helped more expert participants than non-expert ones. It happened maybe because the description of the proposed refactorings was not well detailed enough to guide non-expert participants to modularize the concerns correctly.

(f) Hypothesis Testing. After outlier analysis, it was noticed that none outlier was identified and the hypotheses tests were performed. The verification of the normality of the distribution sample data was made using the non-parametric test called Shapiro-Wilk [15].

The aim of the hypothesis test is to verify if the null hypothesis (H_{0Ef} and H_{0EPr}) can be rejected, with some significance degree, in favor of an alternative hypotheses (H_{1Ef} or H_{1Pr}) based in the set of data obtained.

The *t-test* test was applied to the set of sample data in two stages, because of the existence of two dependent variables, *Efficacy* and *Productivity* were observed. In first stage, the sample relative to the values of the metric *Pr* was compared. In second one, the comparison was made using samples referring to the values of *MQ* metric. For the purpose of this study, the minor degree of significance α was used in both stages to reject the null hypothesis and the maximum degree of significance equal to 5 % was considered.

First Stage. Based in two independent samples (Pr_{WR} and Pr_{WOR}) with averages equals to 5.75 and 2.57, respectively in Tables 4 and 5, the null hypothesis (H_{0Pr}) could be rejected with 0.0151 % of significance. In others words, it is possible to

assure with 99.9 % of accuracy that the average of the values of the productivity obtained by the participants that used the refactorings is different.

Second Stage. Based in two independent samples (MQ_{WR} and MQ_{WOR}) with averages equals to 4.8 and 1.6, respectively, the null hypothesis (H_{0Ef}) could be rejected with 0.0007 % of significance. In others words, it is possible to assurance with 99.9 % of accuracy that the average of the values of the efficacy obtained using the refactorings is different as compared to not using the refactorings.

With the rejection of H_0, it can be stated that the observed differences in the average of *efficacy* and *productivity* of the participants who used the refactorings and participants who have not use them, have statistical significance. Thus, the change in efficacy and productivity of the groups was due to the strategies for software modularization adopted in the experiment, *i.e.*, with or without refactorings.

As presented in Figs. 5 and 6, the average value of the metric *MQ* of the participants who used the refactorings was higher than that of the participants who have not used ($MQ_{WR} > MQ_{WOR}$). These data show that the use of refactorings for modularization of crosscutting concerns is generally more effective than when such refactoring are not used.

Analogously, with respect to productivity, it was expected that the systematic description of the steps of refactorings becomes more agile the execution of the participants' tasks. Based on the data and hypothesis test, there are evidences that the use of refactorings can increase the productivity of a group.

(g) Threats to the Validity of the Study. *Concluding Validity*: the *t-test* was adopted because our study was a project with one factor with two treatments. This is the most suitable test for projects with this configuration, which the aim is to compare the obtained averages from two distinct treatments. The *t-test* usually requires normally distributed data. So, the Shapiro-Wilk test was applied and the result was positive for our study.

Internal Validity: a point that may have influenced the results of the experiment is the use of graduate students as participants. However, they were not influenced by the conductors of this study and we did not show any expectation in favor or against the refactorings proposed in this paper. Besides, the students were properly grouped according to their experience levels in order to have homogeneous groups. This was done to avoid that a group could finish the tasks much earlier than other group. The participants did not receive any grade for the participation.

External Validity: an important bias is the choice of the concerns to be modularized in the experiment. Different types of concern were used to avoid that the knowledge on a specific concern obtained in the training phase to influence the results in other phases of the experiment. Another bias in this case study is that the proposed refactorings have been applied in software of fairly small size that cannot reflect the real scenario of a company that develops/maintains software. It is intended to replicate such experiment with different participants, concerns and applications, in order to isolate the obtained results from these possible influences.

6 Related Works

Many works have been proposed for refactoring of OO software to AO ones and the refactorings are only applied at source-code level, from OO to AO [8, 10, 13, 14, 19]. Moreover, it was noted a lack of related works related to model-based refactorings.

Boger [2] developed a plug-in for the CASE tool ArgoUML that support UML model-based refactorings. The refactoring of class, states and activities is possible, allowing the user to apply refactorings that are not simple to apply at source-code level. Van Gorp [21] proposed a UML profile to express pre and post-conditions of source-code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: (i) verify pre and post-conditions for the composition of sequences of refactorings; and (ii) use the OCL consulting mechanism to detect bad smells.

The differential of this work in relation to others is the proposal to construct an AO model considering OO class models annotated with stereotypes representing crosscutting concerns.

From the conducted case study was performed an evaluation of the obtained results with the support of AO metrics. It was realized that the use of proposed refactorings allows to obtain high quality AO models because: (i) it provided a step by step guide to modularization of certain CCC; and (ii) the proposed refactorings were elaborated considering good design AO practices. Therefore, the use of these refactorings can lead to build high quality AO models, because it prevents software engineers to choose inappropriate strategies for modularization of crosscutting concerns. The limitations of this study is considered: (i) lack of a more quantitative evaluation of the computational support and the proposed refactorings; (ii) the need for new metrics to improve the evaluation process of the refactorings; (iii) lack of studies about the security semantics of legacy software after the application of refactorings; and (iv) a little amount of refactorings for CCC.

7 Final Considerations and Future Works

The idea of using annotated OO class models to build AO models was adopted because they can bring the following benefits: (i) it helps to visualizing possibilities for modularization without using AO; (ii) provides higher level of abstraction by helping the software understanding; (iii) the generated models serves as documentation for the AO software and legacy ones and are independent of programming language.

As future works we intend: (i) to determine if, by means of a controlled experiment, the AO project model generated with the use of refactorings has better benefits than an AO project only obtained with code refactorings; (ii) to develop new specific refactorings for other types of concerns such as security, exception handling, among others; (iii) to create a module for detecting the impacts that can cause a refactoring on a particular software before being applied; and (iv) to proposed strategies for guarantee the behaviour-preservation of OO and AO models after using the refactorings.

Acknowledgements. The authors would like to thank CNPq for the financial support (Proc. No. 133140/2009-1 and 560241/2010-0).

References

1. Parreira Júnior, P.A., et al.: Concern-based refactorings supported by class models to reengineer object-oriented software into aspect-oriented ones. In: International Conference on Enterprise Information Systems (ICEIS), 2013, Angers/FR (2013)
2. Boger, M., Sturm, T.: Tools-support for model-driven software engineering. In: Proceedings of Practical UML-Based Rigorous Development Methods (2001)
3. Costa, H.A.X., Parreira Júnior, P.A., Camargo, V.V., Penteado, R.A.D.: Recovering class models stereotyped with crosscutting concerns. In: Session Tool of XVI Working Conference on Reverse Engineering, Lille, France (2009)
4. Evermann, J.: A metalevel specification and profile for aspectj in UML. In: AOSD. Victoria University Wellington, Wellington (2007)
5. Figueiredo, E., Sant'Anna, C., Garcia, A., Lucena, C.: Applying and evaluating concern-sensitive design heuristics. In: Brazilian Symposium on Software Engineering, Fortaleza (2009)
6. Gamma, E., Helm, R., Johnsn, R., Vlisside, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
7. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: Conference on Object-Oriented Programming Systems, Languages and Applications. SIGPLAN Notices, Vol. 37(11), pp. 161–173. ACM (2002)
8. Hannemann, J., Murphy, G.C., Kiczales, G.: Role-based refactoring of crosscutting concerns. In: AOSD, New York, pp. 135–146, (2005)
9. Hannemann, J.: Aspect-oriented refactoring: classification and challenges. In: International Workshop On Linking Aspect Technology and Evolution, Bonn (2006)
10. Iwamoto, M., Zhao, J.: Refactoring aspect-oriented programs. In: International Workshop On Aspect-Oriented Modeling With UML, Boston, pp. 1–7 (2003)
11. JSpider. j-spider.sourceforge.net/. Accessed January 2013
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-C., Irwin, J.: Aspect-oriented programming. In: Akşit, Mehmet, Matsuoka, Satoshi (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Marin, M., Moonen, L., Van Deursen, A.: An approach to aspect refactoring based on crosscutting concern types. Sigsoft Softw. Eng. Notes **30**(4), 1–5 (2005)
14. Monteiro, M.P., Fernandes, J.M.: Towards a catalogue of refactorings and code smells for aspectj. In: Rashid, A., Akşit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 214–258. Springer, Heidelberg (2006)
15. Montgomery, D.C.: Design and Analysis of Experiments, 5th edn. Wiley, New York (2000)
16. Parreira Júnior, P.A.: Recovering aspect-oriented class models from object-oriented systems by model-based refactorings. Master Dissertation. UFSCar, São Carlos. Brazil (2011) (in Portuguese)
17. Parreira Júnior, P.A., Penteado, R.A.D., Camargo, V.V., Costa, H.A.X.: Mobre: refactoring from annotated OO class models to AO class models. In: CBSoft Tools Session, São Paulo/SP (2011) (in Portuguese)
18. Piveta, E., Moreira, A., Pimenta, M., Araújo, J., Guerreiro, P., Price, T.: Avoiding bad smells in aspect-oriented software. In: International Conference on Software Engineering and Knowledge Engineering, Boston, pp. 81–87 (2007)

19. Da Silva, B.C., Figueiredo, E., Garcia, A., Nunes, D.: Refactoring of crosscutting concerns with metaphor-based heuristics. Electron. Notes Theor. Comput. Sci. (Entcs) **233**, 105–125 (2009)
20. Soares, S., Laureano, E., Borba, P.: Implementing distribution and persistence aspects with AspectJ. In: ACM Conference OOPSLA'02, pp. 174–190 (2002)
21. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 144–158. Springer, Heidelberg (2003)