An Approach Based on Maintainability Criteria for Building Aspect-Oriented Software Design Model

Rodrigo Pereira dos Santos [1]
Heitor Augustus Xavier Costa [2]
Cláudia Maria Lima Werner [1]
André Fonseca Amâncio [2,3]
Paulo Afonso Parreira Júnior [4]
Antônio Maria Pereira de Resende [2]
Fábio Fagundes Silveira [5]

Abstract: Software modeling is an important activity for maintenance since it can facilitate the software comprehension as well as the understanding of its activities towards evolution, correction and adaptation. In this sense, maintainability and its sub-characteristics as presented in the ISO/IEC 9126 standard should be incorporated to the artifacts produced in the modeling activity aiming at designing software with characteristics that render its maintenance less costly. Especially in non-trivial software, such as those that are aspect-oriented, the research on maintenance process considering it during the software development is remarkable. These categories of software aim at maintainability and reusability since they provide the separation of concerns. Thus, seeking to reduce the transition effort to the artifacts generated during the Aspect-Oriented Software Development among the different abstraction levels, this paper presents a proposal of maintainability criteria for building aspect-oriented software design models based on the Maintainability Criteria for Implementation Models, on the aSideML language modeling conventions and on the ISO/IEC 9126 standard.

[1] PESC/COPPE/UFRJ, Federal University of Rio de Janeiro
Post Box 68511, Zip Code 21945-970 – Rio de Janeiro, RJ, Brazil
{rps, werner@cos.ufrj.br}
[2] PqES/DCC/UFLA, Federal University of Lavras
Post Box 3037, Zip Code 37200-000 – Lavras, MG, Brazil
{heitor@ufla.br, tonio@dcc.ufla.br}
[3] DCC/UFMG, Federal University of Minas Gerais
Zip Code 31270-010 – Belo Horizonte, MG, Brazil
{afancio@dcc.ufmg.br}
[4] DC/UFSCar, Federal University of São Carlos
Post Box 676, Zip Code 13565-905 – São Carlos, SP, Brazil
{paulo_junior@dc.ufscar.br}
[5] DCT/UNIFESP, Federal University of São Paulo
Zip Code 04020-041 – São José dos Campos, SP, Brazil
{fsilveira@unifesp.br}

1 Introduction

Software modeling consists on building models to assist the system analysis and/or development, and may expose opportunities for evolution and reuse [1]. In Software Engineering, modeling is known as an important activity for software maintenance because it can facilitate its comprehension, evolution, correction and adaptation, especially when considering the role of maintenance in the life cycle [2]. This scenario highlights the importance of research on the maintenance considering it during the development process, notably in the organization of non-trivial software, such as aspect-oriented ones, since this category of software aims at maintainability and reusability when allowing the separation of concerns [3]. A strategy consists on incorporating maintainability into the artifacts produced during the modeling activity, in order to build software with characteristics that render its maintenance less costly. In this sense, the maintainability sub-characteristics from the ISO/IEC 9126 standard, *analyzability*, *stability*, *changeability* and *testability* can improve attributes that evidence the effort to carry out modifications in software [4] [5], since this standard relates to software quality evaluation.

In this context, this paper presents a proposal of maintainability criteria for building aspect-oriented software design models. The proposal is based on the maintainability criteria for building aspect-oriented implementation models developed by Santos [6], on the aSideML language modeling conventions developed by Chavez [7], and on the definition of the maintainability as described in the ISO/IEC 9126 standard [4] [5]. This paper is an extended version from the best paper previously published on the Workshop on Modern Software Maintenance 2009 [8], and it is structured as follows: Section 2 describes the background; Section 3 presents the proposal of maintainability criteria; Section 4 shows the criteria in an example; and Section 5 provides the conclusion and future work.

2 Background

From the complexity of the “non-trivial” software structure, new technologies are studied and applied to the development process, such as aspect orientation (AO). AO aims at supplementing the existing development paradigms, seeking to provide the separation of crosscutting concerns and contribute for the comprehension, development and maintenance [3]. In spite of the improvements, some points may hinder the maintenance activity (e.g., dependency inversion and obliviousness) [9], which require studies that relate it to the Aspect-Oriented Software Development (AOSD), given the growth of research in this area. Amongst the AOSD technologies, it is possible to highlight AspectJ, an extension of Java language for coding aspects. Table 1 shows the main elements of AspectJ.

In software quality, the ISO/IEC 9126 standard [4] [5] – Information Technology / Software Quality Characteristics and Metrics – proposes a reference model for software products evaluation, providing a general methodology, and presents a Brazilian version by ABNT (Brazilian Technical Standards Association) [11]. This standard defines six quality

characteristics: reliability, efficiency, functionality, maintainability, portability, and usability. Each one of them can be understood by its sub-characteristics. The ISO/IEC 9126 standard defines software maintainability as the set of attributes that evidences the effort to carry out the changes. Table 2 shows its sub-characteristics.

Table 1. Main Elements of AspectJ [10]

Element	Description
join points	Code sections in the execution of a software where aspects are applied. Abstract concepts in AspectJ.
pointcuts	Declarations responsible for selecting join points, that is, detecting which join points the aspect should intercept.
advices	Code used to implement crosscutting concerns, that is, code sections executed at each join point occurrence as described in the pointcut.
introductions (inter-type declarations)	Code that structurally modifies a class, adding new members and relationships to it through a declare parents clause.
aspects	Entity that encapsulates pointcuts, advices, and introductions in a modular code unit, defined similarly to classes.

Table 2. ISO/IEC 9126 maintainability sub-characteristics [10]

Sub-characteristic	Description
Analyzability	Software attributes that evidence the effort needed to diagnose deficiencies or causes for failure, or to identify parts to be modified.
Changeability	Software attributes that evidence the effort needed to modify it, remove its defects or adapt it to the environmental changes.
Stability	Software attributes that evidence the risk from unexpected effects as caused by the modifications.
Testability	Software attributes that evidence the effort needed to validate the modified software.

2.1 aSideML: AO Software Modeling Language

In order to specify and communicate AO projects, Chavez [7] developed the aSideML modeling language, which offers semantic notations and rules to treat the system conceptual modeling in terms of aspects and crosscutting elements[6]. The aSideML modeling elements defined by the software engineer can be structural or behavioral. aSideML also has compositional modeling elements to describe elements in combination process (e.g., woven collaborations) and defines new (and also enriches) UML (Unified Modeling Language) diagrams to present the crosscutting elements and their relationships with the base elements[7]:

- **Aspect Diagram**. This diagram describes an aspect which incorporates the crosscutting interfaces[8], the local characteristics (i.e., attributes/methods) and the inheritance. Each behavioral crosscutting feature[9] can be visualized in a Aspect Collaboration Diagram[10];

[6] Crosscutting element denotes a mechanism to compose aspects and components at the designated joint points.
[7] The terminology used in this work is in agreement with Chavez [7].
[8] Crosscutting interfaces are sets of crosscutting features with associated name that mark crosscutting behavior.
[9] Crosscutting features are operations that describe improvements for system components behavior.

- **Extended Class Diagram**. This diagram supports the graphical notation for the static software design view. Apart from that, it allows the visualization of each aspect (in detail and in separate) in the corresponding Aspect Diagram, and represents a collection of structural modeling elements, such as aspects, classes, interfaces and their relationships, connected among themselves as a graph;
- **Aspectual Sequence Diagram**. This diagram provides a graphical notation for the messages organized in a temporal sequence – messages denote aspect operations –, and support to the interaction view;
- **Combination Process Diagram**. This diagram provides a graphical notation for combined elements (i.e., base elements with some information to emphasize the improvements provided by crosscutting elements). These elements can be specialized for each available implementation model;
- **Aspectual Collaboration Diagram**. This diagram provides a graphical notation for an aspectual collaboration with support to the interaction view, which involves instances of aspects and base elements. An aspectual collaboration has a static part (i.e., describing roles that objects and aspect instances play) and a dynamic part (i.e., showing message flows to carry out crosscutting behavior).

Although there is no consensus over which AO modeling language to use, aSideML has shown to be adequate with the proposal of an aspect high-level model (independent from the programming language), contemplating the main concepts, properties and architectures introduced by the AO design [12]. The Theme/UML [13] notation, in spite of being independent from an implementation approach, does not offer as many resources (i.e., diagrams and modeling elements) for structural debugging and testing as aSideML. The AODM (Aspect-Oriented Design Model) [14] was considered too specific for AspectJ (i.e., little use to describe high-level solutions that can be implemented in other languages) [12]. The new version of aSideML updated and added functions to attend the demand, relying on two IDE Eclipse [15] plug-in tools to generate aSideML models from AspectJ programs [16] and to represent and edit these models [17].

2.2 Related Work

Some related works point that a gap between the design and implementation models hinders maintenance since the former does not portray the latter. Fiutem & Antoniol [18] present an approach to verify the characteristics of object-oriented (OO) software, which consists of: i) obtaining a design from code; ii) comparing the design models (original and obtained); and iii) dealing with inconsistencies, locating regions that are not common between the two design models. Harrison *et al.* [19] present a coding method using Java and UML to change modeling elements in concepts and properties of the OO paradigm, from high abstraction level diagrams. Murphy *et al.* [20] present an approach that defines a high

[10] Aspect Collaboration Diagram describes the general structure of objects and aspect instances, which interact in a context to implement the crosscutting behavior of a behavioral crosscutting characteristic.

abstraction level model and specify how the model is mapped into code. However, these works focus on OO development and/or on maintainability, considering only the implementation model.

Costa [21] proposes criteria and guidelines to verify the analysis model and build OO software design and implementation models, based on Singleton and State design patterns, on the conventions of Eiffel, Java and Smalltalk languages, and on the ISO/IEC 9126 standard. However, the approach considers only OO software and makes no provision for extension or application in AOSD. Santos *et al.* [22] extended this work, setting standards or requirements to guide the AO software coding process through the incorporation of maintainability characteristic, named Maintainability Criteria for Implementation Model (MC_IM). These criteria were separated into categories to facilitate their applicability and were prepared from characteristics of Java and AspectJ languages and research on the programming conventions, as extracted from a sample of works in the literature, based on maintainability (ISO/IEC 9126). The description of MC_IMs can be obtained in [6]. These researches provided the basis for the execution of this work, whose differential lies in the treatment of maintainability in AO design models.

3 Maintainability Criteria for the AO Design Model

The context of this research involves an approach related to the incorporation of maintainability to the AOSD models through the criteria that guide the building and/or the evaluation/adaptation of software artifacts, apart from guidelines and facilitators that assist the application of the criteria to higher abstraction level artifacts [8]. A reduction in the transition effort of artifacts among the abstraction levels is sought. By extending [21] and [6] to deal with maintainability on the AOSD model level, this approach presents a set of maintainability standards that aims at guiding the building of design model artifacts (i.e., creation of models and diagrams) for maintainable AO software. These criteria are named Maintainability Criteria for Design Model (MC_DM) [23] and were prepared from MC_IMs, the conventions and characteristics of the aSideML modeling language, and research on good style and modeling guidelines found in [24] based on maintainability as defined in the ISO/IEC 9126 standard.

An effort was made to keep a set of criteria that covered important issues on the building of AO design model in order to favor the applicability and calibration (i.e., adjustments following future case studies). The criteria present the following structure: i) **criterion**: identifies the MC_DM, with number and description; ii) **justification**: justifies the use of the MC_DM; iii) **sub-characteristic(s)**: lists the maintainability sub-characteristic(s) as defined in the ISO/IEC 9126 standard which the MC_DM aims to improve. The set of MC_DMs can be used in two situations: i) building the design model (i.e., guiding design decisions); and ii) evaluation/adaptation of an existing design model (i.e., verifying the design to adapt it to criteria standards). MC_DMs are shown below and their details can be obtained from [25].

MC_DM 1 – The design model consists of Aspect Diagram, Extended Class Diagram, Extended Sequence Diagram, Aspectual Collaboration Diagram, Sequence Diagram, Combined Classes Diagram, Combined Collaboration Diagram, and Combined Sequence Diagram.

Justification: Provides a minimum set of AO design model artifacts to allow the building of implementation model, aiming at producing maintainable AO software.
Sub-characteristic(s): Analyzability, Stability, Changeability and Testability.

MC_DM 2 – It is easy to find aspect implementation.

Justification: Facilitates to find aspect implementation with the Components Diagram.
Sub-characteristic(s): Analyzability.

MC_DM 3 – Messages shown to the users are allocated in a specific aspect.

Justification: The messages can be considered as a crosscutting concern and their allocation in an aspect facilitates their reuse, standardization (as it is possible to track the composition of other similar messages), maintenance and elimination of redundancy related to messages with similar semantics. These messages are prepared during the building of analysis and design models.
Sub-characteristic(s): Analyzability and Changeability.

MC_DM 4 – Aspects, pointcuts and introductions identifiers allow their rapid recognition.

Justification: Facilitates the identification of the elements that form the AO software by the maintenance team through a pattern for the building of their identifiers. When discerning aspects, pointcuts and introductions identifiers, the work of the software maintainer is facilitated. A suggestion for standardization is: i) the aspect identifier starts with the letter "A", the first capital letter and the words that follow start with capital letters and the remaining letters are small; ii) pointcut identifier is made with the first letter being small, and the words that follow the first start with capital letters and the remaining letters are small (with no space between them); and iii) the identifier of an aspect that contains an introduction starts with the sequence "AI", where the first capital letter and the words that follow the first start with capital letters and the remaining letters are small (with no space between them).
Sub-characteristic(s): Analyzability.

MC_DM 5 – Only the objects of an aspect have access to their attributes in the sense of setting or getting its value.

Justification: Implements the encapsulation, one of the main properties of the OO paradigm that, in its turn, extends to the AO paradigm.
Sub-characteristic(s): Analyzability and Stability.

MC_DM 6 – The aspect keeps the target class attributes updated.

Justification: Keeps the consistency of the software features for the attributes and the methods as included and/or modified during the aspect activities (via advices or introductions).
Sub-characteristic(s): Analyzability and Testability.

MC_DM 7 – Relationships among aspects are organized to minimize the number of inheritance hierarchy trees, keeping only those needed.

Justification: Minimizes the effort to carry out the testing task in order to verify if the inherited properties, the local characteristics (i.e., attributes and methods) and the crosscutting interfaces do not compromise the old functionality and/or if the new functionality is adequate. In this way, the smaller the number of inheritance hierarchy trees, the smaller the effort required to test is.
Sub-characteristic(s): Testability.

MC_DM 8 – Aspects attributes, methods and crosscutting interfaces are located to reduce the height of the inheritance hierarchy trees.

Justification: Improves the comprehension and renders the maintenance less difficult, given that the less height in the inheritance hierarchy tree, the smaller the number of methods inherited by the aspects on the lower level. The execution of tests to assess the behavior of these aspects becomes less complex.
Sub-characteristic(s): Analyzability, Stability, Changeability and Testability.

MC_DM 9 – The sub-aspects define a small number of new local characteristics or overlap few existing local characteristics.

Justification: Renders maintenance and testing tasks less complex in relation to an inheritance hierarchy, given that there are few re-written sub-aspect methods (i.e., overriding). Otherwise, a design problem may occur, with the violation of the super-aspect abstraction that will affect the implementation, resulting in a weak inheritance hierarchy. Apart from that, it increases the number of tests to verify the software correctness.
Sub-characteristic(s): Analyzability, Changeability and Testability.

MC_DM 10 – Sub-aspects define a small number of new crosscutting characteristics, organized in new crosscutting interfaces or as extensions of existing crosscutting interfaces.

Justification: Renders maintenance and testing tasks less complex in relation to an inheritance hierarchy, given that there are few sub-aspect methods with additional crosscutting features. Otherwise, a design problem may occur, with the violation of the super-aspect abstraction that will affect the implementation, resulting in a weak inheritance hierarchy. Apart from that, it increases the number of tests to verify the software correctness.
Sub-characteristic(s): Analyzability, Changeability and Testability.

MC_DM 11 – The relationships among aspects are organized to minimize their coupling.

Justification: Causes the less effort in the aspect comprehension and tests, because the fact that an aspect has low coupling indicates that its interaction is low with the other aspects or classes. On the other hand, when the coupling level is high, a higher effort is required to keep them.
Sub-characteristic(s): Analyzability, Stability, Changeability and Testability.

MC_DM 12 – Aspects should have only one crosscutting concern.

Justification: Improves the concerns encapsulation, contributing for software maintainability.
Sub-characteristic(s): Analyzability, Changeability and Testability.

Moreover, aiming at maintaining the traceability between MC_IMs (Table 3 and Table 4) and MC_DMs, and the support for consistent AO software by valid analysis, design and implementation artifacts, a matrix was elaborated to relate these criteria (Table 5). Maintainability with an abstraction focus is sought, in order to minimize the effort to maintain the artifacts updated and real, in different abstraction levels (i.e., documentation, models and code). For example, MC_DM 9 states that the sub-aspects define a small number of new local characteristics or overlap few local existing characteristics. There is a relation between MC_DM 9 and MC_IM 12 (Figure 1) and MC_IM 31 (Figure 2), given that the structure of the sub-aspects should separate logically cohesive command blocks, facilitating the visualization and comprehension of the local characteristics, and its logic should not implement abstract methods via introductions, avoiding undesired or non-externalized dependencies between super-aspects and sub-aspects that can harm the local characteristics.

Table 3. Categorized MC_IMs (I) [10]

Category	Maintainability Criteria for AO Implementation Model
Storage	MC_IM 1 – The object of a persistent class, when instanced, has its state stored in a persistent medium through aspects.
	MC_IM 2 – The object of a persistent class, when modified, has its state updated in a persistent medium through aspects.
	MC_IM 3 – The object of a persistent class, when excluded, has its state excluded from the persistent medium through aspects.
Comments	MC_IM 4 – Files containing aspects have an introductory comment that provides information on the file name, as well as its contents, author, cohesive aspects, affected classes, version, location, and other relevant information for the characterization of the aspect.
	MC_IM 5 – Descriptive comments precede complex advice and methods.
Structure	MC_IM 6 – The aspects' structural component elements are organized in a standardized way in order to find them easily.
	MC_IM 7 – The advices' structural component elements are organized in a standardized way in order to find them easily.
	MC_IM 8 – Advices in an aspect are grouped according to their nature (i.e., type).
	MC_IM 9 – Advice commands are organized on a one per line basis.
	MC_IM 10 – The declaration of aspect attributes or advice variables is positioned on a one per line basis.
	MC_IM 11 – The declaration of aspect pointcuts is positioned on a one per line basis, so that long pointcuts are divided into simple pointcuts which are grouped in a new general pointcut formed from simple pointcuts.
	MC_IM 12 – A blank line separates logically cohesive command blocks.
	MC_IM 13 – Advices have few formal parameters
Format	MC_IM 14 – Aspects, pointcuts and introductions identifiers allow their rapid recognition.
Location	MC_IM 15 – The tests applied to the software for its release are available in an aspect.
	MC_IM 16 – The return of an advice is in its last line of code.
	MC_IM 17 – An aspect is responsible for verifying the pre-conditions of a use case.
	MC_IM 18 – An aspect is responsible for verifying the post-conditions of a use case.
	MC_IM 19 – The association analysis through cardinality checking is carried out by an aspect.
	MC_IM 20 – An aspect with an introduction does not contain anything apart from this statement.
Logics	MC_IM 21 – Files have one, and only one aspect definition.
	MC_IM 22 – The attributes of an aspect are characterized as private to it.
	MC_IM 23 – The constructor method of an aspect has two, and only two functions. One of them is to instantiate the aspect (depending on the type set by the software engineer) and the other is to attribute initial values to the attributes of the instanced aspect (i.e., defining the initial state).
	MC_IM 24 – It is easy to find advices, and they have lines of code sufficient to implement only one function/task (i.e., cohesion).
	MC_IM 25 – The precedence between aspects is declared when more than one aspect presents pointcuts which intercept the same join points.
	MC_IM 26 – Duplicate pointcuts or advices are refactored.
	MC_IM 27 – Each aspect should deal with a specific concern.
	MC_IM 28 – Aspects with few responsibilities or speculative generality should not be coded.
	MC_IM 29 – Pointcuts are named.
	MC_IM 30 – Pointcuts declared in classes should be easy to migrate to aspects.

Table 4. Categorized MC_IMs (II) [10]

Category	Maintainability Criteria for AO Implementation Model
Logics	MC_IM 31 – Abstract methods should not be implemented through introductions.
Requirements	MC_IM 32 – Non-functional requirements such as distribution, concurrency control, exception handling, debugging, synchronization of concurrent objects, multiple objects coordination, serialization, atomicity, replication, security, visualization, logging, tracing, and fault tolerance are implemented as aspects.
	MC_IM 33 – Functional requirements related to the business rules scattered and/or tangled with the basic functionality of the software are implemented as aspects.

MC_IM 12 – A blank line separates logically cohesive command blocks

Justification: The code segmentation (i.e., using blank lines and grouping code lines logically related and close) increases the code legibility. This way, separations should be considered, for example, amongst the advices, the local variables of the same abstract data type in the methods, the local variables and the first line of code in the advices, prior to the comments, and the logically connected sections.
Sub-characteristic(s): Analyzability.
Category: Structure.

Figure 1. Description of MC_IM 12

MC_IM 31 – Abstract methods should not be implemented through introductions

Justification: Avoids problems stemming from hierarchy. Aspects can be used to add state and behavior to existing classes through introductions. However, their use can cause damages to the software, if abstract methods are inserted to the classes. This insertion forces the developer to provide concrete implementations for the methods introduced in the affected classes and/or sub-classes. This dependency increases the coupling between the affected aspect and classes, and the complexity of the solution implies that a sub-class of an affected class be created. Implementations for abstract methods inserted by the aspect should be provided.
Sub-characteristic(s): Analyzability, Stability, Changeability and Testability.
Category: Logics.

Figure 2. Description of MC_IM 31

4 Using MC_DMs in an Example

To verify the MC_DMs applicability, the Banking Domain System (BDS) design model was modeled. BDS manages the banking transactions of an agency: login and logoff operations, clients' registering and removal, password change, bank statement access, transfers, deposits and withdrawals. Users are classified as administrator or client. The following aspects were modeled: `APersistence` (persistence), `ATransactions` and `ABankingTransactions` (transaction control), `AIUser` (declare parents), `ALogging` (history), `ATracing` (execution flow control) and `APreAndPostConditions` (pre and post-conditions verification support). Some of the uses for MC_DMs are discussed below.

To model the `APreAndPostConditions` aspect and compose it with other modeling elements, the following steps were taken: i) describe the `APreAndPostConditions` aspect in an Aspect Diagram; ii) describe aspectual collaborations for each crosscutting characteristic in Aspectual Collaboration Diagrams; and

iii) describe the behavior of each crosscutting concern using Sequence Diagrams. Table 6 briefly describes the BDS aspects apart from representing the crosscutting behavior between each one of them and other aspects/classes.

Table 5. Traceability Matrix between MC_DMs and MC_IMs

Maintainability Criteria for Implementation Model (MC_IM)	Maintainability Criteria for Design Model (MC_DM)											
	1	2	3	4	5	6	7	8	9	10	11	12
1	✔		✔			✔						
2	✔		✔			✔						
3	✔		✔			✔						
4	✔	✔		✔								
5	✔											
6	✔			✔	✔	✔						
7	✔			✔		✔						
8	✔											
9	✔											
10	✔											
11	✔											
12	✔						✔	✔	✔			✔
13	✔											
14	✔			✔		✔						
15	✔			✔		✔						
16	✔			✔	✔	✔						

Maintainability Criteria for Implementation Model (MC_IM)	Maintainability Criteria for Design Model (MC_DM)											
	1	2	3	4	5	6	7	8	9	10	11	12
17	✔			✔	✔	✔						
18	✔										✔	✔
19	✔											✔
20												✔
21	✔				✔	✔						
22	✔			✔		✔						
23	✔			✔		✔						
24	✔											
25	✔						✔	✔				
26	✔											✔
27	✔			✔								
28	✔											
29	✔			✔								
30	✔										✔	
31	✔								✔	✔		
32	✔		✔									
33	✔											

Table 6. Description of BDS Aspects [25]

Aspect	Classes Affected	Related Aspects	Description
`APersistence`	*Bank*, *Account* and *User*	None	Stores the objects of the persistent classes in a persistent medium.
`ATransactions`	*AccountRepository* and *DBConnection*	None	Executes the transaction control.
`ABankingTransactions`	*AccountRepository* and *DBConnection*	`ATransactions`	Inherits `ATransactions` features and provides implementation to its abstract methods.
`AIUser`	*Administrator* and *Client*	None	States the inheritance between the *Administrator* and *Client* classes, and the *User* base class.
`ALogging`	*Bank*	None	Controls system history.
`ATracing`	*Bank*	None	Controls application monitoring.
`APreAndPostConditions`	*Administrator* and *Client*	None	Checks pre and post-conditions of a use case.

Diagrams were built, as required by MC_DM 1, which highlights the set of AO design model artifacts required for the process of building the AO Implementation Model [25], and some of these diagrams are shown in the next examples. Figure 3 illustrates the `APreAndPostConditions` aspect, where the rhombus symbol aims at making its identification and location easy, impacting the code structure (use of MC_DM 2), as well as the nomenclature adopted (use of MC_DM 4). MC_DM 11 and MC_DM 12 also are applied: the aspect has only one crosscutting interface (i.e., supporting pre and post-conditions verification), and presents only one crosscutting concern with a low interaction with other aspects and classes, indicating that it has a low coupling level (as the box in the right side).

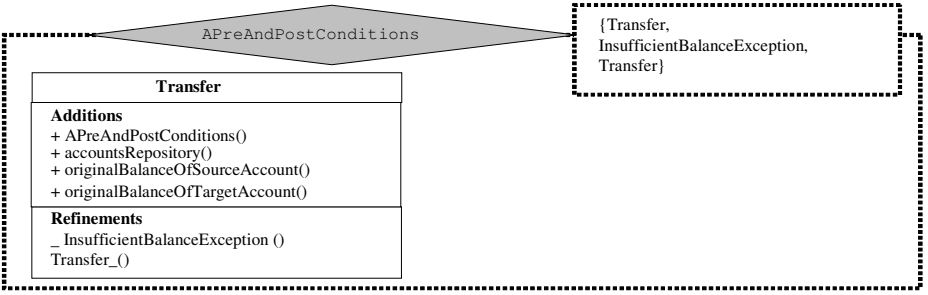

Figure 3. Aspect Diagram for `APreAndPostConditions`

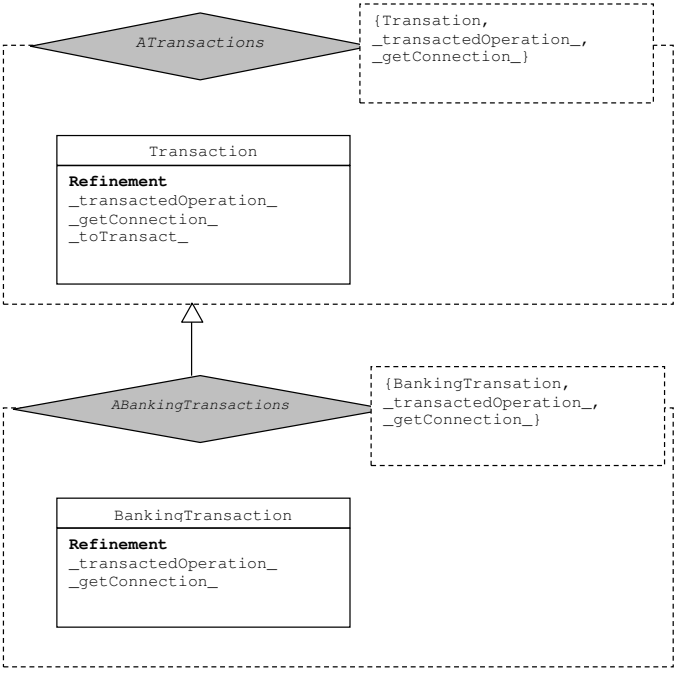

Figure 4. Aspect Diagrams of `ATransactions` and `ABankingTransactions`

In the `ABankingTransactions` aspect (Figure 4), MC_DM 9 and MC_DM 10 were observed, as this aspect defines a small number of local characteristics or overlaps few local characteristics (smaller doted rectangle), apart from a small number of new crosscutting characteristics in new crosscutting interfaces or as extensions of crosscutting interfaces (internal rectangle). MC_DM 7 and MC_DM 8 contribute to make the maintenance and testing tasks less difficult in relation to an inheritance hierarchy, as they enjoy few re-written sub-aspect methods and low height in the inheritance hierarchy tree.

Figure 5 and Figure 6 present, respectively, the Aspectual Collaboration Diagram and the Aspectual Sequence Diagram, which describe the crosscutting behavior and show the `APreAndPostConditions` aspect instance interacting with affected instances of classes or base aspects. MC_DM 5 and MC_DM 6 were applied during the process of building the design models, since only the BDS aspects have access to their attributes (i.e., value setting or getting) and are responsible for maintaining the consistencies of the classes attributes with which they interact (i.e., modifying their behaviors), through the relationships described in the Aspect, Aspectual Collaboration and Aspectual Sequence Diagrams.

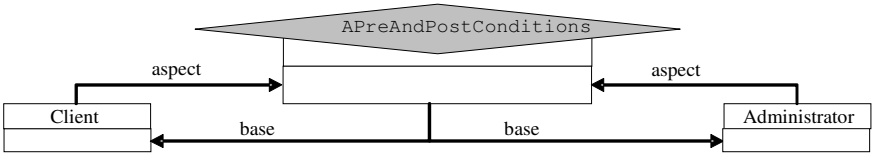

Figure 5. Aspectual Collaboration Diagram for `APreAndPostConditions`

MC_DM 3 was not presented in this paper, since the messages issued by the BDS were conceived during the building of the analysis model (and refined during the building of the design model, by another incorporated aspect), which corresponds to the third part of this research, according Santos *et al.* [10]. However, future uses are planned in new applications. Thus, 11 MC_DMs were applied and presented in this paper, spanning 91.67% of the criteria. Analyzing the BDS context, it is possible to verify that a set of used criteria added important information to the models, useful by the maintenance team.

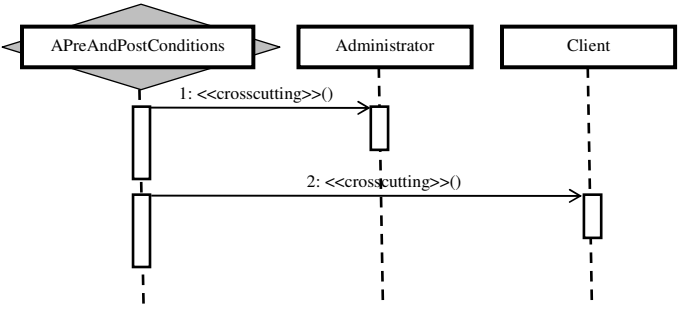

Figure 6. Aspectual Sequence Diagram for `APreAndPostConditions`

5 Conclusion and Future Work

Maintenance becomes inevitable in software development [26]. Aiming at improving the software comprehension, evolution, correction, and adaptation, it is important to consider maintainability during the development process. More specifically, when dealing with design model artifacts, the incorporation of the maintenance ideal in its generation can contribute for improving the stakeholders' profile, influenced by the culture of higher-quality software development [27]. Thus, a set of maintainability criteria was proposed to guide the building of AO design models, part of an approach related to the definition of maintainability criteria, guidelines and facilitators for each model generated in AOSD [10].

MC_DMs seek to explore and clarify important information in order to understand AO models and facilitate their maintenance, as exemplified through the building of the banking domain system design models. Future works are related to apply MC_DMs through case studies in larger scale applications, including maintenance activities (e.g., perfective, corrective etc.) and definition and use of parameters for measurement or qualitative analysis. In this way, we intend to check if MC_DMs improve maintainability. Finally, it is interesting that studies on maintainability in the analysis model are carried out.

Acknowledgements: **The authors would like to thank National Council for Scientific and Technological Development (CNPq/Brazil) for financial support.**

References

[1] LARMAN, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 3. ed., Prentice Hall, Upper Saddle River, NJ, USA, 2004.

[2] SCHACH, S.R., TOMER, A., A Maintenance-Oriented Approach to Software Construction. *Journal of Software Maintenance: Research and Practice*, v. 12, n. 1 (January-February), pp. 25-45, 2000.

[3] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C.V., LOINGTIER, J.M., IRWIN, J., Aspect-Oriented Programming, In: 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, June. *Lecture Notes in Computer Science*, v. 1241, pp. 220-242, 1997.

[4] ISO Std. 9126, *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use*. International Organization for Standardization, 1991.

[5] ISO Std. 9126, *Software Engineering – Product Quality Part 1: Quality Model*. International Organization for Standardization, 2001.

[6] SANTOS, R.P., *Maintainability Criteria for Building and Evaluating Aspect-Oriented Software Products*. Bachelor Monograph, Computer Science Department, Federal University of Lavras, Lavras, MG, Brazil, 2007. (in Portuguese)

[7] CHAVEZ, C.F.G., *A Model-Based Focus for Aspect-Oriented Design*. PhD Thesis, Informatics Department, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2004. (in Portuguese)

[8] AMÂNCIO, A.F., SANTOS, R.P., PARREIRA JÚNIOR, P.A., COSTA, H.A.X., WERNER, C.M.L., RESENDE, A.M.P., SILVEIRA, F.F., A Proposal of Maintainability Criteria for Aspect-Oriented Software Design Models, In: *Proceedings of the VI Workshop on Modern Software Maintenance*, VIII Brazilian Symposium on Software Quality, Ouro Preto, MG, Brazil, June, pp. 1-9, 2009. (in Portuguese)

[9] SANTOS, R.P., SILVA, M.C.O., WERNER, C.M.L., MURTA, L.G.P., Aspect Orientation Issues and Challenges in the Software Reuse Context, In: *Proceedings of the I Latin-American Workshop on Aspect-Oriented Software Development*, XXI Brazilian Symposium on Software Engineering, João Pessoa, PB, Brazil, October, p. 185, 2007. (in Portuguese)

[10] SANTOS, R.P., COSTA, H.A.X., PARREIRA JÚNIOR, P.A., AMÂNCIO, A.F., RESENDE, A.M.P., WERNER, C.M.L., An Approach Based on Maintainability Criteria for Building Aspect-Oriented Software Implementation Model. *INFOCOMP Journal of Computer Science*, Special Edition, pp. 11-20, 2008. (in Portuguese)

[11] ABNT NBR13596, *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for its Use*. Brazilian Technical Standards Association, 1996. (in Portuguese)

[12] GARCIA, A., CHAVEZ, C., SOARES, S., PIVETA, E., PENTEADO, R., CAMARGO, V.V., FERNANDES, F., Special Report of First Brazilian Workshop on Aspect-Oriented Software Development, In: *Proceedings of the I Workshop on Aspect-Oriented Software Development*, XVIII Brazilian Symposium on Software Engineering, Brasília, DF, Brazil, October, 2004. (in Portuguese)

[13] CLARKE, S., *Composition of Object-Oriented Software Design Models*. PhD Thesis, Dublin City University, Dublin, Ireland, 2001.

[14] STEIN, D., *An Aspect-Oriented Design Model Based on AspectJ and UML*. Master Thesis, University of Duisburg-Essen, Essen, Germany, 2002.

[15] ECLIPSE, *Eclipse Project*, In: <http://www.eclipse.org/>, December, 2009.

[16] SOUZA, R.R.G., *REAJ: A Reverse Engineering Tool for AspectJ Code*. Bachelor Monograph, Computer Science Department, Federal University of Bahia, Salvador, BA, Brazil, 2007. (in Portuguese)

[17] LIMA, L.M., *A Graphic Editor for Build aSideML Models*. Technical Report, Computer Science Department, Federal University of Bahia, Salvador, BA, Brazil, 2007. (in Portuguese)

[18] FIUTEM, R., ANTONIOL G., Identifying Design-Code Inconsistencies in Object-Oriented Software, In: *Proceedings of the 14th International Conference on Software Maintenance*, Bethesda, Maryland, November, pp. 94-102, 1998.

[19] HARRISON, R., COUNSELL, S., NITHI, R., Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. *Journal of Systems and Software*, v. 52, n. 2-3 (June), pp. 173-179, 2000.

[20] MURPHY, G.C., NOTKIN, D., SULLIVAN, K.J., Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, v. 27, n. 4 (April), pp. 364-380, 2001.

[21] COSTA, H.A.X., *Maintainability Criteria and Guidelines for Building Object-Oriented Design Model*. PhD Thesis, São Paulo University, São Paulo, SP, Brazil, 2005. (in Portuguese)

[22] SANTOS, R.P., WERNER, C.M.L., COSTA, H.A.X., PARREIRA JÚNIOR, P.A., AMÂNCIO, A.F., RESENDE, A.M.P., Maintainability Criteria for Building Aspect-Oriented Software Products, In: *Proceedings of the V Workshop on Modern Software Maintenance*, VII Brazilian Symposium on Software Quality, Florianópolis, SC, Brazil, June, pp. 1-9, 2008. (in Portuguese)

[23] AMÂNCIO, A.F., SANTOS, R.P., COSTA, H.A.X., RESENDE, A.M.P., SILVEIRA, F.F., Maintainability Criteria for Aspect-Oriented Software Design Model, In: *Proceedings of the II Latin-American Workshop on Aspect-Oriented Software Development*, XXII Brazilian Symposium on Software Engineering, Campinas, SP, Brazil, October, pp. 90-91, 2008.

[24] FILMAN, R.E., ELRAD, T., CLARKE, S., AKSIT, M., *Aspect-Oriented Software Development*, 1. ed., Addison-Wesley, 2004.

[25] AMÂNCIO, A.F., *Maintainability Criteria for Building Aspect-Oriented Software Design Model*. Technical Report, Computer Science Department, Federal University of Lavras, Lavras, MG, Brazil, 2008. (in Portuguese)

[26] ZVEGINTZOV, N., PARIKH, G., 60 years of Software Maintenance: Lessons Learned, In: *Proceedings of the 21st International Conference on Software Maintenance*, Budapest, Hungary, September, pp. 726-727, 2005.

[27] SNEED, H.M., BROSSLER, P., Critical Success Factors in Software Maintenance: A Case Study, In: *Proceedings of the 19th International Conference on Software Maintenance*, Amsterdam, The Netherlands, September, pp. 190-198, 2003.