On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework

Cláudio Sant'Anna[1], Alessandro Garcia[1], Christina Chavez[2], Carlos Lucena[1], Arndt von Staa[1]

[1]PUC-Rio, Computer Science Department, TecComm, SoC+Agents Group
{afgarcia,claudios,lucena,arndt}@inf.puc-rio.br
http://www.teccomm.les.inf.puc-rio.br/socagents

[2]UFBA, Computer Science Department
flach@ufba.br

Abstract

Aspect-oriented software development (AOSD) is gaining wide attention both in research environments and in industry. Aspect-oriented systems encompass new software engineering abstractions and different complexity dimensions. As a consequence, AOSD poses new problems to empirical software engineering. It requires new assessment frameworks specifically tailored to measure the reusability and maintainability degrees of aspect-oriented systems. This paper presents an assessment framework for AOSD, which is composed of two components: a suite of metrics and a quality model. These components are based on well-known principles and existing metrics in order to avoid the reinvention of well-tested solutions. The proposed framework has been evaluated in the context of two different empirical studies with different characteristics, diverse domains, varying control levels and different complexity degrees. Based on empirical and quantitative analysis, the advantages and drawbacks of the framework components are discussed.

Keywords: Aspect-oriented software development, software metrics, quality model, empirical software engineering.

1. Introduction

Object-oriented abstractions currently are recognized as being unable to capture all concerns of interest in a software system [23, 32]. Many important concerns often crosscut several objects and classes of object-oriented systems. Aspect-oriented software development (AOSD) is a promising paradigm to promote improved separation of concerns, leading to the production of software systems that are easier to maintain and reuse. AOSD is centered on the aspect notion as an abstraction aimed to modularize such crosscutting concerns and improve the system maintainability and reusability. However, since the aspect-oriented paradigm is still in its infancy, it is very difficult to determine what are good design and implementation decisions for AOSD. There is only a small consensus that classical and obvious crosscutting concerns should be modularized within aspects, such as logging and exception handling. There is no rationale to assist the design of other important and more domain-dependent crosscutting concerns. It is difficult to understand when to use aspects such as architectural and design solutions. As a consequence, aspects currently are being applied in an ad hoc manner.

The usefulness of new development paradigms and associated design practices can be evaluated through empirical studies. Software metrics are used in empirical studies as indicators of the strengths and weaknesses of the studied approach. Many software metrics have been proposed [12, 15], used and, sometimes, empirically validated [3, 25], e.g. number of lines of code [15], McCabe complexity metric [15], CK metrics [12], etc. Many companies have built their own quality models based on product metrics [4, 27]. Moreover, a number of

development environments have incorporated support for metrics, such as Together [6]. However, the available metrics are not dedicated to AOSD. As a result, most empirical studies involving the application of aspect technology have been based on qualitative assessment [17, 20, 22]. These studies do not rely on a structured quality model and well-accepted software engineering principles. Experimenters often use poorly understood concepts to investigate the quality of aspect-oriented solutions, such as pluggability and composability. Although the goal of these concepts is to capture several facets of software reuse and maintenance, their definitions are largely vague and widely founded on the intuition of their proponents. Our viewpoint is that the maintainability and reusability degrees of aspect-oriented systems should be assessed in terms of well-established software engineering principles and well-tested metrics.

In this context, this paper presents a framework for assessing reusability and maintainability of aspect-oriented software. The definition of the framework is centered on the separation of concerns principle and other software attributes that are well known and explored in empirical software engineering, such as coupling, cohesion and size. Our assessment framework encompasses two main components: a metrics suite and a quality model. The quality model defines precisely how to measure reusability and maintainability based on a set of proposed metrics. The model also assists software engineers in the interpretation of the data gathered from the measurement process. Software engineers can use the proposed framework both to assess design decisions in AOSD, and compare aspect-oriented solutions and object-oriented solutions. The metrics and the model have been evaluated in the context of two different empirical studies with different characteristics, diverse domains, varying control levels and different complexity degrees. Based on empirical and quantitative analysis, the advantages and drawbacks of these metrics are discussed.

The remainder of this paper is organized as follows. Section 2 introduces basic concepts for AOSD and the requirements for the proposed suite of metrics. Section 3 presents the proposed metrics and Section 4 shows how these metrics are related to other components of our assessment framework. Section 5 presents the empirical evaluation of our assessment framework and an analysis of its usefulness. Section 6 discusses our proposal and related work in terms of usability issues. Section 7 presents some concluding remarks and directions for future work.

2. AOSD: Background and Measurement Requirements

2.1. Basic Concepts

Separation of concerns is a well-established principle in software engineering. A *concern* is some part of the problem that we want to treat as a single conceptual unit [32]. Concerns are modularized throughout software development using different abstractions provided by languages, methods and tools. The basic abstractions of object-oriented software development (OOSD) are classes, objects, methods and attributes. However, these abstractions may not be sufficient for separating some special concerns found in most complex systems. These concerns have been called *crosscutting concerns* since they naturally cut across the modularity of other concerns. Without proper means for separation and modularization, crosscutting concerns tend to be scattered and tangled up with other concerns. The natural consequences are reduced comprehensibility, ease of evolution and reusability of software artifacts.

Aspect-oriented software development (AOSD) [23, 32] has been proposed as a technique for improving separation of concerns in the construction of OO software and supporting improved reusability and ease of evolution. AOSD supports the modularization of crosscutting concerns by providing abstractions that make it possible to separate and compose

them to produce the overall system. AOSD uses *aspects* as a new abstraction and provides a new mechanism for composing aspects and *components* (classes, methods, etc.) at specific *join points*.

AspectJ [24] is a practical aspect-oriented extension to the Java programming language. *Aspects* are AspectJ's unit of modularity for crosscutting concerns. They are defined in terms of pointcuts, advice and introduction. In AspectJ, *join points* are well-defined points in the program flow, such as method calls, field sets, etc. *Pointcuts* describe join points and values at those points. *Advice* is a method-like abstraction that defines code to be executed when a join point is reached; pointcuts are used in the definition of advice. *Introduction* defines how AspectJ modifies a program's static structure — namely, the members of its classes and the relationship between classes. Pointcuts and advice dynamically affect program flow, and introduction statically affects a program's class hierarchy.

In this work, we are interested in the following abstractions related to AOSD: two basic types of *components*[1], classes and aspects, and two basic types of *operations*, methods and advices (crosscutting operations).

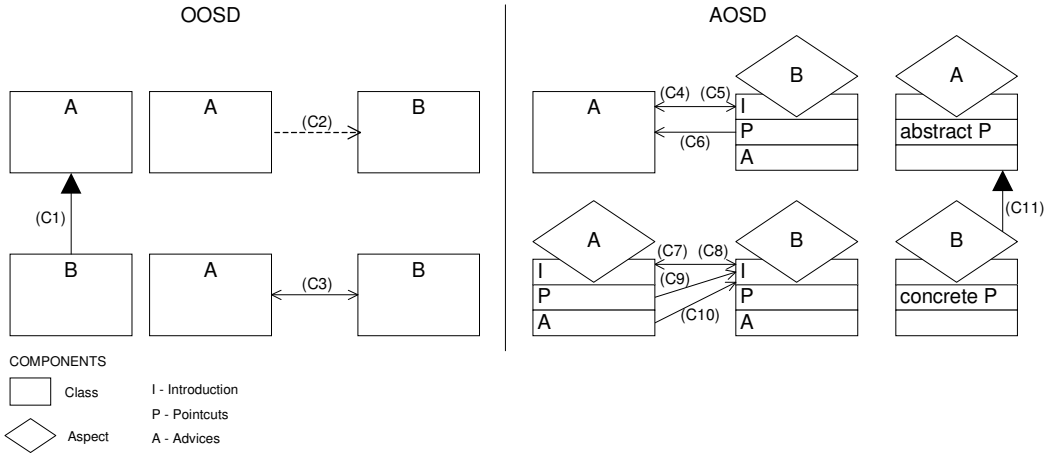

Figure 1. Coupling Dimensions on AOSD.

2.2. Measurement Requirements

The central purpose of using aspects is to achieve improved separation of concerns, but it can affect other software attributes, such as coupling, cohesion and size. Aspects are effective to modularize crosscutting concerns, minimize replication of code and, as a consequence, reduce the size of the system. However, the inappropriate use of aspects can affect negatively these software attributes and increase the complexity of the system. Software metrics are the most effective way to supply empirical evidence that may improve our understanding of the different dimensions of the software complexity [7]. Metrics evaluate the use of abstractions during software development in terms of software attributes, such as coupling and cohesion. The metrics are more effective when they are associated with some assessment framework so that software engineers can understand and interpret the meanings of the collected data.

The literature contains several sets of traditional metrics and others for OO systems. Most existing metrics cannot be applied straightforwardly to aspect-oriented software [32, 33, 34], since AOSD introduces new abstractions to software engineering (Section 2.1). The system components (classes and aspects) are composed in different ways and, as a consequence, aspect-oriented abstractions encompass different dimensions of coupling and

[1] In general, the AOSD community adopts a clear distinction between components (or *base components*) and aspects; however, in this work, the term *component* is used to denote classes *and* aspects.

cohesion [33, 34]. In addition, as stated previously, AOSD has direct impact on the system size and on the separation of the system concerns. For instance, Figure 1 illustrates the different ways of combining classes and aspects, which are the potential sources of coupling in an aspect-oriented system. In this way, the definition of adequate metrics for aspect-oriented software should satisfy the following requirements:

Requirement #1 – measure well-known software attributes such as separation of concerns, coupling, cohesion and size.

Requirement #2 – rely as much as possible on traditional metrics and on the extension of OO metrics to AOSD, since aspect-oriented abstractions extend the set of OO abstractions.

Requirement #3 – capture different dimensions of coupling and cohesion of aspect-oriented software.

Requirement #4 - support the identification of benefits and drawbacks in the use of aspects into a software project when compared with an object-oriented solution for the same problem.

3. The Metrics Suite

The proposed suite of metrics captures information about the design and code in terms of fundamental software attributes such as separation of concerns, coupling, cohesion and size. This suite reuses and refines classical metrics - e.g. LOC - and OO metrics - e.g. Chidamber and Kemerer (CK) metrics [12] - for coupling, cohesion and size [12, 15]. We have tailored the definition of these metrics to reflect the new abstractions introduced by aspects in terms of these software attributes. The criteria for the selection of these metrics were based on theoretical and practical demands. For example, the CK metrics are based on a sound measurement theory and have been widely used and empirically validated [3]. Furthermore, we proposed some metrics for separation of concerns that refine Lopes' metrics [26].

Our suite is composed of five design metrics and five code metrics. In the following subsections, these metrics are grouped according to the attributes they measure: (i) separation of concerns (SoC), (ii) coupling, (iii) cohesion and (iv) size. The description of each metric emphasizes how it satisfies our measurement requirements (Section 2.2). The relevance of these metrics for reuse and maintenance is described in our assessment framework (Section 4). The metrics suite is presented apart from the framework description because it can be reused by others assessment frameworks (e.g. frameworks intended to measure other quality attributes, such as reliability and testability).

3.1. SoC Metrics

Separation of concerns refers to the ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern [32]. We defined the following metrics of SoC:

Concern Diffusion over Components (CDC). CDC is a design metric that counts the number of primary components whose main purpose is to contribute to the implementation of a concern. Furthermore, it counts the number of components that access the primary components by using them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods.

Concern Diffusion over Operations (CDO). CDO counts the number of primary operations whose main purpose is to contribute to the implementation of a concern. In addition, it counts the number of methods and advices that access any primary component by calling their methods or using them in formal parameters, return types, throws declarations and local variables. Constructors also are counted as operations.

Concern Diffusion over LOC (CDLOC). CDLOC counts the number of transition points for each concern through the lines of code. The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas [18]. The shadowed areas are lines of code that implement a given concern. Transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. The intuition behind it is that they are points in the program text where there is a "concern switch." For each concern, the program text is analyzed line by line in order to count transition points. The higher the CDLOC, the more intermingled is the concern code within the implementation of the components; the lower the CDLOC, the more localized is the concern code. An extensive set of guidelines to assist the shadowing process is reported elsewhere [18].

3.2. Coupling Metrics

Coupling is an indication of the strength of interconnections between the components in a system. Highly coupled systems have strong interconnections, with program units dependent on each other [31]. CBC and DIT are the coupling metrics of our suite.

Coupling between Components (CBC). CBC is defined for a component (class or aspect) as a tally of the number of other components to which it is coupled. It counts the number of classes that are used in attribute declarations; i.e., it captures the couplings C2 and C3 depicted in Figure 1. It also counts the number of components declared in formal parameters, return types, throws declarations and local variables, and classes and aspects from which attribute and method selections are made. If a component A is coupled to a component B in an arbitrary number of forms, CBC counts only once. This metric is an extension of the CK metric for coupling between objects (CBO). In order to define CBC, we change the definition of CBO to deal with new coupling dimensions in AOSD: accesses to aspect methods and attributes defined by introduction (couplings C4, C5, C7, C8, C10), and the relationships between aspects and classes or other aspects defined in the pointcuts (couplings C6, C9). This metric encompasses nine coupling dimensions described in Figure 1 (from C2 to C10).

Depth of Inheritance Tree (DIT). DIT is defined as the maximum length from a node to the root of the tree. It counts how far down the inheritance hierarchy a class or aspect is declared. DIT is an extension of a CK metric with the same name that considers the inheritance between aspects. This metric encompasses the coupling dimensions C1 and C11 illustrated in Figure 1.

3.3. Cohesion Metric

The cohesion of a component is a measure of the closeness of the relationship between its internal components [31]. In the following, we describe the cohesion metric of our suite.

Lack of Cohesion in Operations (LCOO). This metric measures the lack of cohesion of a component. If a component C1 has *n* operations (methods and advices) $O_1,\ldots, O_n$ then $\{I_j\}$ is the set of instance variables used by operation O_j. Let |P| be the number of null intersections between instance variables sets. Let |Q| be the number of non-empty intersections between instance variables sets. Then: LCOO = |P| – |Q|, if |P| > |Q|, LCOO = 0 otherwise. LCOO measures the amount of method/advice pairs that do not access the same instance variable. As such, it is a measure of lack of cohesion. This metric extends the CK metric LCOM. We regard advices and methods of aspects in the same way that CK regards methods of classes.

3.4. Size Metrics

The software size physically measures the length of a software system's design and code [15]. Our metric suite encompasses the following size metrics.

Vocabulary Size (VS). VS counts the number of system components, i.e. the number of classes and aspects into the system. This metric measures the system vocabulary size. Each component name is counted as part of the system vocabulary. The component instances are not counted.

Lines of Code (LOC). It counts the number of code lines. This is the traditional measure of size. Documentation and implementation comments as well as blank lines are not interpreted as code. Different programming styles usually bias the results of this metric application. In our empirical studies (Section 5), we have overcome this problem by ensuring the same programming style was used in both projects.

Number of Attributes (NOA). This metric counts the internal vocabulary of each component, i.e. the number of attributes of each class or aspect. Inherited attributes are not included in the count.

Weighted Operations per Component (WOC). This metric measures the complexity of a component in terms of its operations. Consider a component C_1 with operations (methods or advices) O_1 ,..., O_n. Let c_1 ,…, c_n be the complexity of the operations. Then: $WOC = c_1 + ... + c_n$. This metric originally does not specify the operation complexity measure, which should be tailored to the specific contexts. The operation complexity measure is obtained by counting the number of parameters of the operation, assuming that an operation with more parameters than another is likely to be more complex. This metric extends the CK's WMC metric. We treat advices and methods of aspects in the same way that CK treats methods of classes.

4. The Assessment Framework

The measure of a particular internal attribute, such as coupling, is useful if it is related to a measure of some external attribute of the object of study (e.g. reusability). In software engineering measures of internal product attributes are artificial concepts and, in themselves, have no meaning [7]. In this context, we developed a framework to capture the understanding of the SoC, coupling, cohesion and size attributes in terms of their usefulness as predictors of the maintainability and reusability qualities. In fact, the goal of the assessment framework is to provide support for assessment of reusability and maintainability of aspect-oriented systems based on the proposed metrics (Section 3).

The framework components help organize the assessment process and assist in data collection and interpretation (Figure 2). The basic components of the framework are: (i) the suite of metrics (Section 3), and (ii) the quality model. The quality model establishes the relationships between the external attributes, internal attributes and the metrics. The framework requires some artifacts as inputs to the measurement process. First, it requires the design documents and the system code for the use of the metrics. In addition, the assessment framework requires a description of the system concerns to guide the identification of the concerns when using the metrics of separation of concerns (shadowing process described in Section 3.1).

4.1. The Quality Model

Our quality model defines a terminology and clarifies the relationships between the reusability, maintainability and the metrics suite. It is a useful tool for guiding software engineers in data interpretation. It was defined based on a set of assumptions. The definition of our quality model is based on: (i) an extensive review of a set of existing quality models [15, 18], (ii) classical definitions of quality attributes [28, 31] and traditional design theories, such as Parnas' theory [29], which are commonly accepted among researchers and practitioners and (iii) the software attributes impacted by the aspect-oriented abstractions

(Section 2). The quality model has been built and refined using Basili's GQM methodology [2] (Section 4.2).

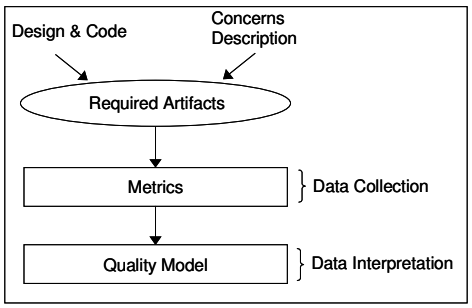

Figure 2. The Assessment Framework

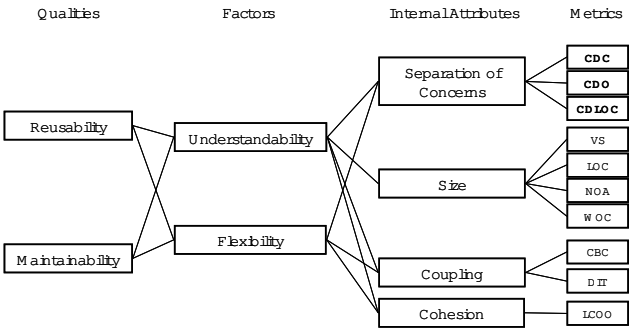

Figure 3. The Quality Model

Quality models are constructed in a tree-like fashion since quality actually is a composite of many other qualities [15]. The notion of software quality is usually captured in a model that depicts other intermediary qualities, which we have called factors. Our quality model is composed of three different elements: (i) qualities, (ii) factors and (iii) internal attributes. In addition, the quality model connects the internal attributes to our suite of metrics. The qualities are the attributes that we want to primarily observe in the software system (reusability and maintainability). The factors are the secondary quality attributes that influence the defined primary qualities. The attributes are related to internal properties of software systems. These attributes are related to well-established software engineering principles, which in turn are essential to the achievement of the qualities and their respective factors [15]. Figure 3 presents the elements of our quality model. The upper branches contain important high-level qualities and factors that we wish to quantify. The internal attributes are easier to measure than the qualities and factors and, thus, actual metrics are connected to these attributes [15]. The following subsections describe the elements of our quality model.

4.1.1. Qualities and Factors

As stated previously, maintainability and reusability are the quality focus of our assessment framework. Reusability is the ability of software elements to serve for construction of different elements in the same software system or across different ones [28]. In our model, we are interested in evaluating the reusability of elements of design and code of aspect-oriented systems. Maintenance is the activity of modifying a software system after initial delivery. Software maintainability is the ease with which software components can be modified. Maintenance activities are classified into four categories [15, 31]: corrective maintenance, perfective maintenance, adaptive maintenance and evolution. Since the main goal of AOSD is to improve the evolution of OO systems (Section 2.1), our focus is on the evolution aspect of aspect-oriented systems.

The quality model emphasizes that similar factors are useful for the promotion of maintainability as well as reusability. This similarity is related to the fact the reuse and maintenance activities encompass common cognitive tasks. Flexibility and understandability are the central factors for promoting reuse and maintainability [15, 28, 29, 31]. Both kinds of activities require software abstractions to support understandability and flexibility. Understandability indicates the level of difficulty for studying and understanding a system's design and code [29]. Flexibility indicates the level of difficulty for making drastic changes to one component in a system without a need to change others [29]. An understandable system enhances its own maintainability and reusability; this is so because most maintenance and

reuse activities require that software engineers first try to understand the system's components before making any subsequent system modifications or extensions. Furthermore, a software system must be flexible enough to support the addition and removal of functionalities and the reuse of its components with a minimum amount of effort.

In our model, the understandability factor is related to the following internal attributes: (i) size, (ii) coupling, (iii) cohesion and (iv) separation of concerns. Coupling and cohesion affect understandability because a component of the system cannot be understood without reference to the other components to which it is related. The size of design and code may indicate the amount of effort needed for understanding the software components. The separation of concerns criterion is a predictor of understandability because the more localized are the concerns of the system, the easier it is to understand them. The flexibility factor is influenced by the following internal attributes: (i) coupling, (ii) cohesion and (iii) separation of concerns. High cohesion, low coupling and separation of concerns are desired characteristics because they mean that a component represents a single part of the system and the system components are independent or almost independent. Furthermore, the system's concerns are not scattered and tangled. If it becomes necessary to add, remove or reuse functionality, it is localized in a single component and the maintenance and reuse activities are flexibly restricted to this isolated component. Note that the flexibility factor is not influenced by the principle of size.

4.1.2. Internal Attributes and Metrics

Each internal attribute is related to a set of the proposed metrics. In the following paragraphs, we state the relationships between the internal attributes, the metrics and the qualities and factors in terms of each internal attribute.

Separation of Concerns. The metrics CDC, CDO and CDLOC measure the degree to which a single concern in the system maps to software elements in the software design and code. The more directly a concern maps to these elements, the easier it is for software engineers to understand it. The more directly a concern maps to the elements, the fewer number of components will be changed during maintainability activities, or a fewer number of components should be needed to understand and extend during reuse activities.

Coupling. The metrics CBC and DIT measure coupling from different viewpoints. The component understanding involves the understanding of the components to which it is coupled. So the larger the number of couples of a component, the more difficult it is to understand the system. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and, therefore, maintenance is more difficult. Excessive coupling between components is detrimental to modular design and prevents reuse. The more independent is a component, the easier it is to reuse it in another application.

Cohesion. The LCOO metric detects the degree to which a component implements a single logical function. The higher the degree to which different actions performed by a component contribute towards distinct functions, the harder it is to reuse and maintain the component or one of its functionalities.

Size. Size metrics are concerned with the different aspects of the system size. In general, the higher the size, the more difficult it is to understand the system. For example, the metric LOC measures the system size in terms of lines of code. The greater the number of code lines, the more difficult it is to understand the system. The more lines of code, the harder it is to find the lines that must be changed during evolution activities or understand the implementation of the required functionalities during reuse activities.

4.2. GQM Goal and Questions

In order to facilitate data interpretation, we believe that a measurement process must take place according to an explicit goal and a set of questions that represent an operational definition of it. So we use the Goal/Question/Metric approach [2] to define the goal of our measuring framework and derive from it the questions that must be answered to determine if the goal has been met. The GQM approach provides a three-step framework: (1) list the major **goals** of the empirical study; (2) derive from each goal the **questions** that must be answered to determine if the goals have been met; (3) decide what must be measured in order to be able to answer the questions adequately (definition of the **metrics**). The definition of these questions has helped us in the definition of the quality model (Section 4.1) and the proposed set of metrics (Section 3). The questions also associate the quality model with a more precise semantics for the qualities, factors and internal attributes. Moreover, such questions assist software engineers in the interpretation of the gathered data during the measurement process. Figure 4 presents the goal and questions generated. Questions 1 and 2 are derived directly from the stated goal and, therefore, refer to ease of evolution and reusability. These two questions are refined into questions about understandability and flexibility that are further refined into questions about separation of concerns, coupling and cohesion. Questions about understandability are also refined into questions related to the system size. Questions 1.1.2, 1.2.1, 2.1.2, 2.2.1 can be refined into one or more questions, depending on the number of concerns (N) identified in the system to be assessed.

Goal

Assess aspect-oriented systems for the purpose of prediction with respect to maintainability and reusability from the viewpoint of the developer.

Questions

1. How easy is it to evolve the system?

1.1. How easy is it to understand the system?
- 1.1.1. How concise is the system?
  - 1.1.1.1. How many components are there?
  - 1.1.1.2. How many lines of code are there?
  - 1.1.1.3. How many attributes are there?
  - 1.1.1.4. How many methods and advices are there?
- 1.1.2. How well are the concerns localized?
  - 1.1.2.1. How scattered and tangled is the <concern1 name> definition?
  - 1.1.2.N How scattered and tangled is the <concernN name> definition?
- 1.1.3. How high is the coupling of the system?
  - 1.1.3.1. How high is the coupling between components?
- 1.1.4. How high is the cohesion of the system?
  - 1.1.4.1. How high is the cohesion of the system components?

1.2. How flexible is the system?
- 1.2.1. How well are the concerns localized?
  - 1.2.1.1. How scattered and tangled is the <concern1 name> definition?
  - 1.2.1.N. How scattered and tangled is the <concernN name> definition?
- 1.2.2. How high is the coupling of the system?
  - 1.2.2.1. How high is the coupling between components?
- 1.2.3. How high is the cohesion of the system?
  - 1.2.3.1. How high is the cohesion of the system components?

2. How easy is it to reuse the system elements?

2.1. How easy is it to understand the system?
- 2.1.1. How concise is the system?
  - 2.1.1.1. How many components are there?
  - 2.1.1.2. How many lines of code are there?
  - 2.1.1.3. How many attributes are there?
  - 2.1.1.4. How many methods and advices are there?
- 2.1.2. How well are the concerns localized?
  - 2.1.2.1. How scattered and tangled is the <concern1 name> definition?
  - 2.1.2.N. How scattered and tangled is the <concernN name> definition?
- 2.1.3. How high is the coupling of the system?
  - 2.1.3.1. How high is the coupling between components?
- 2.1.4. How high is the cohesion of the system?
  - 2.1.4.1. How high is the cohesion of the system components?

2.2. How flexible is the system?
- 2.2.1. How well are the concerns localized?
  - 2.2.1.1. How scattered and tangled is the <concern1 name> definition?
  - 2.2.1.N. How scattered and tangled is the <concernN name> definition?
- 2.2.2. How high is the coupling of the system?
  - 2.2.2.1. How high is the coupling between components?
- 2.2.3. How high is the cohesion of the system?
  - 2.2.3.1. How high is the cohesion of the system components?

Figure 4. GQM Goal and Questions

5. Empirical Evaluation

The metrics and the model have been evaluated in the context of two different empirical studies with different characteristics, diverse domains, varying control levels and different degrees of complexity. The first study was a semi-controlled experiment [19] to compare the use of an object-oriented approach (based on design patterns [16]) and an aspect-oriented approach to design and implement Portalware, a multi-agent system (MAS). The second study involved the application of the proposed framework to evaluate Hannemann's Java implementations and AspectJ implementations of the GoF design patterns [20].

This section reports the partial results of the first study; the goal is to present a substantive evaluation of our proposed framework. This experiment was designed to demonstrate the usefulness of the metrics suite and the quality model in order to predict the maintainability and reusability of software systems. We collected data on the development of two versions of the Portalware system: an aspect-oriented version and an object-oriented version. The selection of this study and the choice of the agent domain were based on the fact that it is not obvious which problem entities should be designed as classes and which should be designed as aspects. Furthermore, this case study was chosen for a number of other reasons: (i) it involves both domain-specific and application-dependent concerns; (ii) it is not focused only on traditional and trivial crosscutting concerns (such as logging and tracing); and (iii) it also encompasses concerns that have not been investigated in the AOSD community. We used our assessment framework to determine if we made good design choices.

5.1. The Experimental Setting

The project upon which the MAS is based has been derived from a case study undertaken in the SoC+Agents/TecComm Group at PUC-Rio in Brazil. The Portalware system is a Web-based environment that supports the development and management of Internet portals. As the needs of the Internet Portals market change ever more rapidly, the weaknesses in the software engineering techniques that are used become increasingly apparent. UML notations [5] and the Java language, respectively, were used to generate the object-oriented designs and implementation. A UML extension for aspect-oriented design [11] and the AspectJ programming language [24] were used to generate the aspect-oriented designs and implementation. The MAS concerns handled in this project are real-world reactive MASs, including agent types, roles, collaboration, adaptation, autonomy, and so on.

The experiment subjects developed two versions of the Portalware system based on both OOSD and AOSD (Section 2.1). Since these two approaches were not developed with MAS concerns (or *agency concerns*) in mind, we used two supporting methods [17] specially tailored to the MAS development. Each method is associated with each investigated approach and used by the experiment subjects to apply the respective approach and associated abstractions. Figure 5 represents, respectively, slices of the object-oriented and aspect-oriented designs for the Portalware system. The left side shows a combination of different design patterns to address the MAS concerns. Each pattern is surrounded by a dotted line. On the right side, a diamond shape is used to express aspects. Each diamond may be related to one or more rectangles used to describe classes. This relationship is expressed as a line from the aspect to a class. These figures also illustrate some changes required in the maintenance and reuse scenarios for further clarification in the Section 5.3.

The subjects have participated both in the development of the aspect-oriented (AO) system and in the development of the object-oriented (OO) system. Three of the subjects were PhD candidates and one was a Master's degree student at PUC-Rio. All subjects had widespread experience in OO software analysis, design and programming and some experience in MAS development. The PhD candidates had already implemented large (

thousand lines of code) Java programs. Among them, two had considerable experience in aspect-oriented programming. The study was divided into two major phases: (1) the Construction phase (Section 5.2), and (2) the Reuse and Evolution phase (Section 5.3). In the Construction phase, the individuals were asked to develop the selected MAS using OOSD and AOSD. The OO and AO designs were based on the same requirements specification and the satisfying of the same set of functionalities.

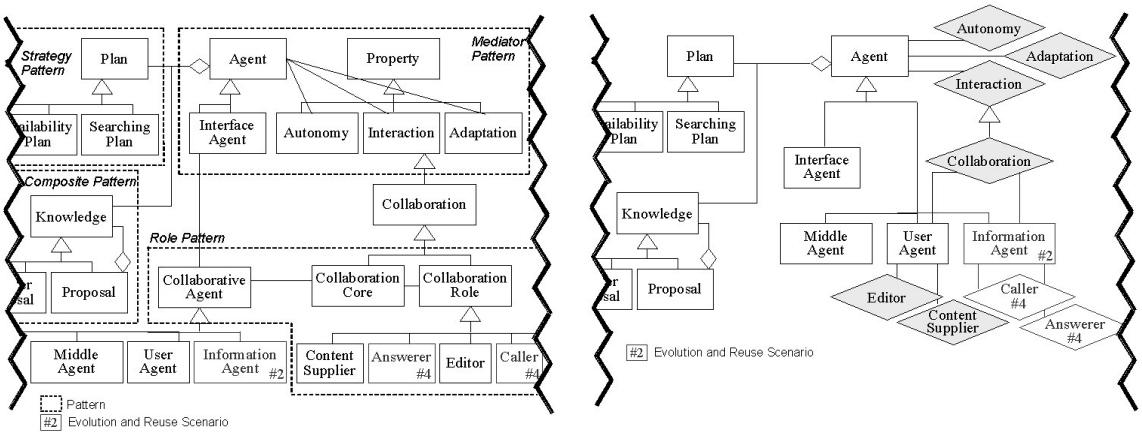

Figure 5. Slices of the OO Design and AO Design

5.2 Data Collection and Interpretation: Predicting Reusability and Maintainability

In the construction phase, the data was partially gathered by the CASE tool Together 6.0. This tool supports some metrics[2]: LOC, NOA, WOC, CBC, LCOO, and DIT. We also processed the shadowing of each system concern. Figure 6 presents an example of code shadowing for the Interaction concern in the PAgent class of the Portalware system. In general, the data collected demonstrated that the aspect-oriented system is easier to maintain and reuse than the OO system, as summarized below. The complete description of the data and a more detailed discussion of the results of this empirical study are beyond the scope of this paper and can be found in [19].

```
public class PAgent {

    private String agentName;
   ...
   ...
    protected Interaction theInteraction;
    protected Autonomy    theAutonomy;
    protected Adaptation  theAdaptation;

    public PAgent(String aName, Vector pl) {
        init();
         agentName = aName;
        theInteraction = new Interaction(this);
        theAutonomy = new Autonomy(this);
        theAdaptation = new Adaptation(this);
        planList = pl;
        System.out.println(" Name == " + agentName);
   }
   ...
   ...
/* Interface  for  Interaction */
   public void receiveMsg(Message msg)
   {
       theAutonomy.makeDecision(msg);
       theAdaptation.adaptBeliefs(msg);
   }

   public void outcomingMsg(Message msg)
   {
       theInteraction.outcomingMsg(msg);
   }
}
```

Figure 6. An example of code shadowing

Size. The aspect-oriented project produced a more concise system according the number of lines of code. The LOC was 1445 in the OO implementation and 1271 in the AO implementation. The OO system was also more complex in terms of the number of components (VS metric) and the number of component attributes

[2] These metrics have different acronyms in the Together environment: WOC is termed WMPC2, CBC is CBO, LCOO is LOCOM1, and DIT is DOIH.

(NOA metric). For example, the amount of design and implementation components in the OO solution (VS = 60) was higher than in the AO solution (VS = 56). The main reason for this result is that the Role and Mediator patterns (Fig. 5) required additional classes to address the decomposition and composition of multiple agent roles and behavior properties, respectively. However, the use of aspects produced more complex operations, i.e. advices, than the use of the OO patterns (WOC metric).

Coupling and Cohesion. The AO system incorporated components with higher coupling (CBC metric). The OO project has led to the abuse of the inheritance mechanism, which was fundamental for establishing high inheritance coupling (DIT metric). The LCOO metric detected some components of the OO system and produced better results in terms of cohesion than the components of the AO system.

Separation of Concerns. The use of aspects clearly provided better support for separation of MAS concerns. This result is supported by all SoC metrics. The CDC measures detected that every MAS concern required more components for their implementation in the OO solution than in the AO solution. For example, all agent roles required more than five classes for their definition, while one single aspect is able to encapsulate each system role. In addition, all concerns required more operations (methods/advices) in the OO system than in the AO system (CDO metric). Finally, the CDLOC measures also pointed out that the AO solution was more effective in terms of modularizing the MAS concerns across the lines of code.

5.3 Reuse and Maintenance Scenarios: Evaluating the Predicted Data

The Reuse and Evolution phase involved the same subjects. The goal of this phase was to confirm the results of the application of the proposed metrics as a useful mechanism to predict reusability and maintainability. We simulated simple and complex changes involving agency concerns to both the OO and AO solutions in order to measure how easy it was to evolve and reuse their components. We selected seven maintenance and reuse scenarios that are recurrent in large-scale MAS: change of the agent roles (S1), creation of an agent type (S2), reuse of the agenthood concern (S3), inclusion of collaboration in an agent type (S4), reuse of roles (S5), creation of a new agent instance (S6) and change of the agenthood definition (S7). For each scenario, the difficulty of maintainability and reusability was defined in terms of structural changes to the artifacts in the AO and OO systems, such as number of components (aspects/classes) added, number of components changed, number of relationships included, and so forth. Table 1 presents the complete list of the measures. Figure 5 illustrates some design changes required in the S2 and S4 scenarios.

The results presented in Table 1 have confirmed the data predicted using our software metrics, which provides substantial evidence of the usefulness of our metrics. For instance, the inclusion of new roles (S1) required some additional lines in the change of the OO system. The reason is that the AO technique supports improved separation of concerns, as indicated in the SoC measures (Section 5.2). The introduction of collaboration capabilities to a specific agent type (S4) was the scenario that resulted in more substantial differences between the changes in the OO solution and the AO solution: (1) the OO code required 20 lines more than the AO code, (2) more relationships were added in the OO design, and (3) eight lines were removed from the AO code while no line was changed in the OO code. These results confirmed that coupling (DIT and CBC metrics), number of components and attributes (VS and NOA metrics) and separation of concerns (SoC metrics) impact directly on the maintenance and reuse activities. This finding is confirmed in the scenarios S5 and S7. However, it was not possible to understand the interplay between cohesion (LCOO metric) and the reusability and maintainability of the produced systems.

	EVOLUTION																			
	REUSE																			
	Changed Entities		Changed Operations		Added Entities		Added Operations		Changed Relations.		Added Relations.		Added LOCs		Changed LOCs		Copied Entities		Copied LOCs	
	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
S1	1	1	3	3	5	5	2	3	0	0	15	15	101	98	1	1	-	-	-	-
S2	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	-	-	-	-
S3	0	0	2	2	4	4	0	0	0	0	10	10	84	86	0	0	0	0	0	0
S4	0	0	2	3	8	8	0	0	0	0	29	25	188	167	0	8	0	0	0	0
S5	1	1	2	1	0	0	1	1	0	0	4	2	16	14	0	0	0	0	6	6
S6	0	0	0	0	0	0	0	0	0	0	0	0	15	15	0	0	-	-	-	-
S7	5	1	0	0	0	0	0	0	5	2	1	1	0	0	5	1	0	0	40	0

Table 1. The Results of the Reuse and Maintenance Scenarios

5.4 Threats to Validity

The goal of this study was to evaluate the metrics as predictors of maintainability and reusability and serves as a first step of an empirical validation. In this context, this section discusses the constraints on the validity of this evaluation.

Construct Validity. Maintainability, reusability, understandability and flexibility are difficult concepts to measure. The dependent variables used here (Section 5.3) are based on a previous study performed by Li et al. [25]. In their work, the concept "maintenance effort" was reified as the number of lines of code changed. In future work, we are planning to use other more representative measures, such as "time to understand, develop and implement modifications" [1]. The achievement of the construct validity for our independent variables (the metrics in Section 3) would be desirable but is beyond the scope of this paper. However, some of our metrics are extensions of CK metrics that have been theoretically validated in [8, 9, 10].

Internal Validity. Our empirical study cannot be considered a controlled experiment, since all subjects took part in the development of the two systems. However, we tried to minimize the bias, selecting two subjects who defended (before the study) the pattern-oriented approach and two others who defended the aspect-oriented approach. Despite the effect caused by the subjects learning as the study proceeded, the AO system, which was developed first, showed better results.

External Validity. The limited size and complexity of the system and the use of student subjects may restrict the extrapolation of our results. However, while the results may not be directly generalized to professional developers and real-world systems, the academic setting allows us to make useful initial assessments of whether these metrics would be worth studying further. In spite of its limitations, the study constitutes an important initial empirical work on the AO metrics proposed and is complementary to qualitative work [17] that we performed previously.

6. Discussion and Related Work

Up to now, most empirical studies in the AOSD context rest on subjective criteria and qualitative investigation [13, 17, 20, 22]. For example, Hannemann and Kiczales compare Java implementations and AspectJ implementations of the GoF design patterns [20] in terms of weakly defined measurement criteria, such as composability and pluggability. Only a few papers propose software metrics for AOSD, such as Lopes' work [26]. She has defined a set of different metrics for separation of concerns. In fact, our metrics CDC, CDO and CDLOC (Section 3) are somewhat inspired by her set of metrics. However, the Lopes' metrics only capture different dimensions of separation of concerns. In addition, the definition of her metrics is quite strongly coupled to her empirical study and tailored to the distribution concern in Java code. Our suite of metrics generalizes her metrics to apply to different

concerns of design and code. Moreover, our metrics are early prediction mechanisms for other stringent principles in the design of aspect-oriented software, such as coupling and cohesion.

Zhao has proposed a metrics suite for aspect-oriented software, which is specifically designed to quantify the information flows in an aspect-oriented program [34]. His metrics are based on a dependence model for aspect-oriented software that consists of a group of dependence graphs; each can be used to explicitly represent various dependence relations at different levels of an aspect-oriented program. Although Zhao's metrics can be viewed as complementary to our set of proposed metrics, their application is cumbersome and time-consuming due to different reasons. The use of such metrics requires that software engineers construct a number of dependence graphs for different levels of modularity, such as the method dependence graph (MDG), the advice dependence graph (ADG), the introduction dependence graph (IDG), and so on. As a consequence, such metrics are very complex to understand and use, and requires the implementation of a dependence analysis tool that is likely to differ from one language to another. In addition, Zhao's metrics are not derived from well-tested metrics, and the associated dependence model is not based on any well-known software engineering model.

Concerning the application of our assessment framework in different contexts and case studies (Section 5), there is one general type of criticism that could be applied to our software metrics. This refers to theoretical arguments leveled at the use of conventional size metrics, such as LOC and VS (Section 3), as they are applied to traditional (non-AO software) design and development. However, in spite of the well-known limitations of these metrics we have learned that their application cannot be analyzed in isolation and they have shown themselves to be extremely useful when analyzed in conjunction with the other metrics of the proposed suite (Section 5). Furthermore, our assessment framework provides some guidelines to interpret effectively the data generated by these metrics in the context of reusability and maintainability. In addition, some researchers (such as Henderson-Sellers [21]) have criticized the LCOM metric as being without solid theoretical bases and lacking empirical validation [3]. However, we understand this issue as a general research problem in terms of cohesion metrics. In the future, we intend to investigate another emerging cohesion metrics based on program dynamics to include them in our assessment framework.

Up to now, we have not implemented a tool to support our metrics. However, our metrics are easy to use because most of them are based on metrics that already are well known in the software engineering community. As a result, a number of SDEs support most of them. For example, we have used Together [6] in our empirical studies (Section 5) for the data-gathering process of the size, coupling and cohesion metrics. Since our SoC metrics are innovative, there is no tool that directly supports their use. As a consequence, the identification of the concerns is currently imposed on the user of the SoC metrics since most concerns are clearly application-dependent. However, there are some tools, recently proposed in the AOSD community, which can assist software engineers in the identification of concerns. For instance, the FEAT tool [30] supports the location, description, and analysis of code implementing one or more concerns in a Java system.

7. Conclusions and Ongoing Work

Building quality systems has been the driving goal of all software engineering efforts over the last two decades. The lack of design and implementation guidance can lead to the misuse of the aspect-oriented abstractions, worsening the overall quality of the system. Important quality requirements, such as reusability and maintainability, are likely to be affected negatively due to the inadequate use of the aspect-oriented languages and respective abstractions. In this way, as AOSD moves forward, a significant research effort is required to

define quality measures. Measuring the structural design properties of software artifacts, such as coupling, cohesion, and separation of concerns, is a promising approach towards early quality assessments. To use such metrics effectively, quality models are needed for quantitatively describing how these internal properties relate to relevant external qualities. However, AOSD research has focused mainly on implementation and design language constructs. Some empirical studies have been undertaken in the context of AOSD. However, the assessment in these studies is qualitative and not generally applicable to other contexts.

We have presented a framework, which is based on a suite of metrics and a quality model, to assist the assessment of aspect-oriented software in terms of reusability and maintainability. The proposed metrics satisfy important requirements in order to achieve successful measurements in the AOSD context (Section 2.2). Our metrics suite is founded on well-known attributes of software design and implementation. Most of them rely on traditional metrics and extensions of OO metrics. The expectations regarding the extended and new metrics are related to the need to deal with new abstractions and new dimensions of coupling and cohesion introduced by AOSD. We also have discussed the usefulness and usability of our assessment framework and associated metrics, thus facilitating the development of future empirical studies as well as the framework refinement. We are planning to implement an Eclipse [14] plug-in to support the use of our assessment framework.

Acknowledgements. We would like to thank Gail Murphy for her valuable contributions to this work. This work has been partially supported by CNPq under grant No. 141457/2000-7 for Alessandro, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. Cláudio is supported by CAPES under grant No. 31005012004M-9. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

References

[1] Bandi, R., Vaishnavi, V., Turk, D. "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics". IEEE Transactions on Software Engineering, 29(1), January 2003, pp. 77-87.

[2] Basili, V., Caldiera, G., Rombach, H. "The Goal Question Metric Approach". Encyclopedia of Soft. Eng., vol. 2, September 1994, pp. 528-532, John Wiley & Sons, Inc.

[3] Basili, V., Briand, L., Melo, W. "A Validation of Object-Oriented Design Metrics as Quality Indicators". IEEE Trans. on Software Eng., 22 (10), October 1996, pp. 751-761.

[4] Boehm, B. "Software Eng. Economics", Prentice-Hall, 1981.

[5] Booch, G., Rumbaugh, J., Jacobson, I. "The Unified Modeling Language User Guide". Addison Wesley, 1999.

[6] Borland TogetherSoft website. URL: http://www.togethersoft.com.

[7] Briand, L., El Emam, K., Morasca, S. "Theoretical and Empirical Validation of Software Product Measures". Technical Report ISERN-95-03, Fraunhofer Institute for Experimental Software Engineering, Germany, 1995.

[8] Briand, L., Morasca, S., Basili, V. "Property-Based Software Engineering Measurement". IEEE Transactions of Software Engineering, 22 (1), 1996, pp. 68-86.

[9] Briand, L., Daly, J., Wüst, J. "A Unified Framework for Cohesion Measurement in Object-Oriented Systems". Empirical Software Eng. Journal, 3 (1), 1998, pp 65-117.

[10] Briand, L., Daly, J., Wüst, J. "A Unified Framework for Coupling Measurement in Object-Oriented Systems". IEEE Trans. on Software Engineering, 25 (1), 1999, pp 91-121.

[11] Chavez, C., Lucena, C. "Design Support for Aspect-oriented Software Development". Doctoral Symposium at OOPSLA' 2001, Tampa Bay, USA, October 2001, pp 1418.
[12] Chidamber, S., Kemerer, C. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 20 (6), June 1994, pp. 476-493.
[13] Driver, C. "Evaluation of Aspect-Oriented Software Development for Distributed Systems", Masters Thesis, University of Doublin, September 2002.
[14] The Eclipse website. URL: http://www.eclipse.org.
[15] Fenton, N., Pfleeger, S. "Software Metrics: A Rigorous and Practical Approach". 2.ed. London: PWS, 1997.
[16] Gamma, E. et al. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, Reading, 1995.
[17] Garcia, A., Silva, V., Chavez, C., Lucena, C. "Engineering Multi-Agent Systems with Aspects and Patterns". J. of the Brazilian Computer Society, July 2002, 1 (8), pp 57-72.
[18] Garcia, A. et al. "Agents and Objects: An Empirical Study on Software Engineering". Technical Report 06-03, Computer Science Department, PUC-Rio, February 2003. Available at ftp://ftp.inf.puc-rio.br/pub/docs/techreports/ (file 03_06_garcia.pdf).
[19] Garcia, A. et al. "Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems". Proc. of the SELMAS'03 Workshop at ICSE'03, Portland, USA, May 2003, pp. 11-22.
[20] Hannemann, J., Kiczales, G. "Design Pattern Implementation in Java and AspectJ", Proceedings of OOPSLA'02, November 2002, pp. 161-173.
[21] Henderson-Sellers, B. "Object-Oriented Metrics: Measures of Complexity". Prentice Hall, 1996.
[22] Kersten, A., Murphy, G. "Atlas: A Case Study in Building a Web-based learning environment using aspect-oriented programming". Proceedings of OOPSLA, November 1999.
[23] Kiczales, G. et al. "Aspect-Oriented Programming". European Conference on Object-Oriented Programming (ECOOP), LNCS (1241), Springer-Verlag, Finland, June 1997.
[24] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. "Getting Started with AspectJ". Communication of the ACM, October 2001, pp. 59-65.
[25] Li, W., Henry, S. "Object-Oriented Metrics that Predict Maintainability". Journal of Systems and Software, 23 (2), February 1993, pp. 111-122.
[26] Lopes, C. "D: A Language Framework for Distributed Programming". PhD Thesis, College of Computer Science, Northeastern University, 1997.
[27] McGarry, F., Pajersk, R., Page, G., Waligora, S., Basili, V., Zelkowitz, M. "Software Process Improvement in the NASA Software Eng. Laboratory", Carnegie Mellon Univ., Software Eng. Inst., Technical Report CMU/SEI-95-TR-22, December 1994.
[28] Meyer, B. "Object-Oriented Software Construction". 2.ed. Prentice Hall, 1997.
[29] Parnas, D. "On the Criteria to Be Used in Decomposing Systems into Modules". Communications of the ACM, 15 (12), December 1972, pp. 1053-1058.
[30] Robillard, M., Murphy, G. "FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code". Proc. of ICSE'03, Portland, May 2003, pp. 822-824.
[31] Sommerville, I. "Software Engineering", 6.ed. Harlow, England, Addison-Wesley, 2001.
[32] Tarr, P. et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". Proceedings of the 21st International Conference on Software Engineering, May 1999.
[33] Zacaria, A., Hosny, H. "Metrics for Aspect-Oriented Software Design". Proc. Third International Workshop on Aspect-Oriented Modeling, AOSD' 03, 2003.
[34] Zhao, J. "Towards a Metrics Suite for Aspect-Oriented Software". Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.