Journal of Universal Computer Science, vol. 15, no. 11 (2009), 2254-2283
submitted: 18/1/09, accepted: 29/5/09, appeared: 1/6/09 © J.UCS

Assessment of the Design Modularity and Stability of Multi-Agent System Product Lines

Camila Nunes
(Pontifical Catholic University of Rio de Janeiro, Brazil
cnunes@inf.puc-rio.br)

Uirá Kulesza
(Federal University of Rio Grande do Norte – UFRN, Natal, Brazil
uira@dimap.ufrn.br)

Cláudio Sant'Anna
(Federal University of Bahia – UFBA, Salvador, Brazil
santanna@dcc.ufba.br)

Ingrid Nunes
(Pontifical Catholic University of Rio de Janeiro, Brazil
ioliveira@inf.puc-rio.br)

Alessandro Garcia
(Pontifical Catholic University of Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br)

Carlos Lucena
(Pontifical Catholic University of Rio de Janeiro, Brazil
lucena@inf.puc-rio.br)

Abstract: A multi-agent system product line (MAS-PL) defines an architecture whose design and implementation is accomplished using software agents to address its common and variable features. MAS-PL promotes the large-scale reuse of common and variable agency features across multiple MAS applications. The development of MAS-PLs can be achieved through MAS-specific platforms and implementation techniques, such as conditional compilation and aspect-oriented programming (AOP). However, there is not much evidence on how these techniques provide better modularity, allowing the conception of stable MAS-PL designs. This paper presents a quantitative study on the design modularity and stability of an evolving MAS-PL. The MAS-PL was built following the reactive product line adoption approach. The product line was developed and evolved based on several versions of a conference management web-based system, named Expert Committee (EC). Our evaluation is made through a series of change scenarios related to new agency features, which are agent characteristics that enhance the system with autonomous behavior. The quantitative study consists of a systematic comparison between two different versions of the EC MAS-PL based on a MAS-specific platform, called JADE. One version was implemented with object-oriented and conditional compilation techniques. The other one relied on AOP. Our analysis was driven by well-known modularity and change impact metrics.

Keywords: Software Product Lines, Multi-agent Systems, Software Metrics, Empirical Software Engineering
Categories: D.1.5, D.2.8, D.2.11, D.2.10

1 Introduction

One of the latest trends in software engineering is to produce techniques and tools that allow the development of families of similar products, instead of individual products. With the aim to address this need, many approaches have been proposed for software product line development over the last years [Clements and Northrop, 2001] [Czarnecki and Eisenecker, 2000] [Pohl et al., 2005]. Software product lines (SPLs) comprise engineering techniques for systematically creating similar software systems from a shared set of software assets. Most of the existing SPL approaches motivate the development of a flexible and reusable architecture to enable large-scale reuse. A SPL architecture addresses a set of common and variable features of a family of products. A feature [Czarnecki and Eisenecker, 2000] is a system property or functionality that is relevant to some SPL stakeholder and is used to capture commonalities or discriminate among systems.

Similar to development of single-purpose systems, SPL approaches also need to address change scenarios in a disciplined manner. The evolution of SPLs needs to be conducted with as minimum impact as possible due to their frequent change demands. Examples of usual changes in SPLs are: introduction, modification or removal of optional and alternative features. Thus, SPL architectures need to be stable and flexible to support such frequent changes in order to allow the reduction of the modularity degeneration due to evolution scenarios. During the evolution of SPLs, it is necessary to consider adequate mechanisms to implement a determined variability. The inappropriate choice can increase the SPL complexity and bring difficulties to its maintenance. Therefore, it is important to apply and analyze different variability techniques to promote the stability of the architecture during the SPL evolution. Examples of such techniques are: object-oriented frameworks [Fayad et al., 1999], conditional compilation [Antenna, 2008], and Aspect-Oriented Programming (AOP) [Kiczales et al., 1997].

Recent research presents some studies and benefits of using AOP to improve the modularization of features in SPLs [Alves et al., 2006] [Alves et al., 2005] [Figueiredo et al., 2008] [Griss, 2000] [Lee et al., 2006], object-oriented frameworks [Kulesza et al., 2006a] or multi-agent systems (MAS) [Garcia et al., 2003] [Sant'Anna et al., 2003]. The increasing complexity of modern applications motivates the use of AOP [Kiczales et al., 1997] because it is aimed at modularizing crosscutting features. Crosscutting features produce tangled, scattered and replicated code, and tend to occur often in the context of SPLs in general [Alves et al., 2005] [Figueiredo et al., 2008]. All these problems can cause difficulties regarding the management, maintenance and reuse of common and variable features in SPLs.

On the other hand, over the past few years, the agent technology has emerged as a new software engineering paradigm to allow the development of distributed complex systems [Jennings, 2001] [Wooldridge and Ciancarini, 2000]. Some recent research has investigated the synergy of MASs and SPLs technologies, characterizing the development of Multi-Agent Systems Product Lines (MAS-PLs) [Pena et al., 2006a] [Pena et al., 2006b]. A MAS-PL defines a SPL that uses software agents to model, design and implement its common and variable features in a family of MAS products. There are some specific platforms to design and implement MASs such as JADE [JADE, 2008] and Jadex [Jadex, 2008]. These platforms can be used as the base to implement MAS-PLs, but they must be combined with other modularization techniques in order to improve the modularization of the agency features.

However, recent research only explores the use of AOP to modularize SPL crosscutting features in general. No work analyzes the impact of adding agency features to an existing system. Besides, there is no work that assesses the (dis)advantages and complementarities of using different implementation strategies for improving MAS-PL design longevity using a specific development platform of MAS. Nevertheless, it is important to analyze the circumstances in which a variability technique is more appropriate. This knowledge is essential to support the development of stable MAS-PL which is resilient to different types of changes.

In this context, this paper presents an empirical study of development and evolution of a MAS-PL with the aim to compare the modularity and stability of object-oriented (OO) and aspect-oriented (AO) different implementations of the MAS-PL. Our MAS-PL has been developed from the evolution of a conference management web-based system, called Expert Committee (EC) [Nunes et al., 2008a]. In this MAS-PL, we developed seven releases, focusing on several change scenarios. Each release was implemented separately using two sets of technologies based on JADE platform: (i) an implementation in Java language with conditional compilation support; and (ii) the other one using AO techniques using the AspectJ language [Kiczales et al., 2001]. These techniques were used because they offer mechanisms to implement and modularize core and varying features. Conditional compilation is based on preprocessor directives indicating which piece of code should be compiled. AOP is used to support variability and encapsulation of features though modular units called aspects. Both techniques provide support to fine-grained variability implementation. Additionally, conditional compilation is a common technique adopted in industry to modularize features and AOP is an emergent technique to improve the modularization of crosscutting features. Most of the new features are related to the introduction of typical agency features in the original system using MASs abstractions such as, agents, roles and their associate behaviors [JADE, 2008]. Our objective is to compare the AO techniques and conditional compilation in the decomposition of agency features, guided by the JADE platform.

The design modularity and stability evaluation of the MAS-PL versions is based on existing metrics suites for modularity analysis [Sant'Anna et al., 2007] [Sant'Anna et al., 2003] and change impact metrics [Yau and Collofello, 1985]. Through that assessment of the design modularity and stability of MAS-PL, it is possible to compare and analyze specific techniques to implement variabilities in SPL. The main contributions of this paper are: (i) the assessment of the design modularity and stability of different implementation techniques using a MAS platform for implementing and evolving a MAS-PL; and (ii) discussions about which techniques are more appropriate to allow superior stability in the implementation of agency features and support the construction of reusable and maintainable MAS. The results of the design modularity showed the AO solution presented a high number of components and operations for some features when compared to the OO solution. However, the AO solution presented better values in terms of tangling of features/concerns by enabling their modularization using AOP. In terms of design stability, the AO solution exhibited less change in its components, operations, and lines of code. As general conclusions, we can say that the AO solution was more appropriate to implement optional features and maintain a more modularized SPL.

The remainder of this paper is organized as follows. Section 2 presents the study settings. Section 3 presents the empirical study phases. Section 4 describes the results of the modularity analysis. Section 5 describes the results of the stability analysis. Section 6 presents the threats to validity of the study. Section 7 discusses some lessons learned. Related work is presented in Section 8. Finally, final remarks are presented in Section 9.

2 Study Settings

This section describes the MAS-PL used in the context of our study. Initially, the feature model of the MAS-PL is described (Section 2.1). After that, we describe the development and evolution process of the MAS-PL (Section 2.2). The MAS-PL architecture is then presented in terms of the components and agents that compose the system (Section 2.3). Finally, the AO and OO designs of the MAS-PL are presented in Section 2.4.

2.1 The Expert Committee System

The Expert Committee (EC) [Nunes et al., 2008a] is a MAS-PL for web applications that aims at managing the paper submission and reviewing processes of conferences and workshops. The EC system provides functionalities to support the complete process of conference management. such as: (i) create conferences; (ii) define conference basic data, program committee, areas of interest and deadlines; (iii) choose areas of interest; (iv) submit paper; (v) assign papers to be reviewed; (vi) accept/reject to review a paper; (vii) review paper; (viii) accept/reject paper; (ix) notify authors about the paper review; and (x) submit camera ready. Each of these functionalities can be executed by an appropriate user type, such as, chair, coordinator, program committee members and authors. Figure 1 presents a partial view of the EC feature model according to the FODA notation [Czarnecki and Eisenecker, 2000] with the mandatory, optional and alternative features.

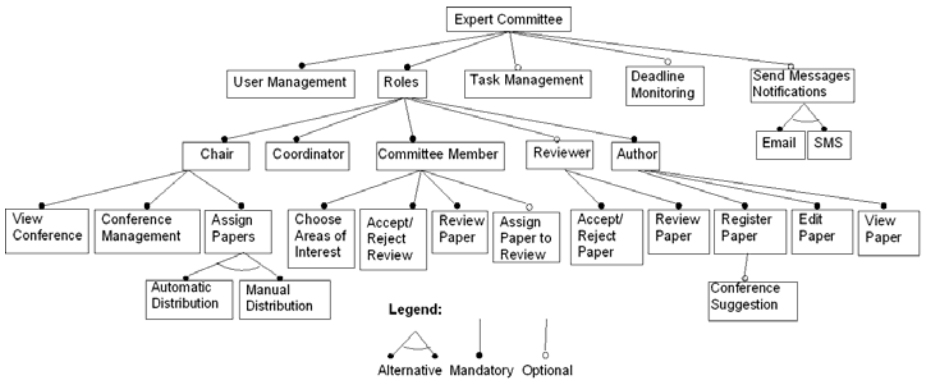


Figure 1: Simplified EC feature model

2.2 Development and Evolution of the EC MAS-PL

According to [Krueger, 2002], there are three strategies to implement SPL, which are: proactive, extractive and reactive. The proactive approach motivates the development of all the artifacts of the SPL. The extractive approach the SPL is developed starting from existing software systems. Finally, the reactive approach adopts the incremental development of SPLs.

The EC MAS-PL was developed following the reactive approach. During the development and evolution of our MAS-PL, we first implemented the SPL base architecture of the EC. After that, we applied a series of change scenarios, adding optional and alternative features to the SPL architecture. Seven new releases of the EC MAS-PL were generated. Each release of our MAS-PL was always implemented in two different versions: (i) one codified in Java with conditional compilation [Antenna, 2008]; and (ii) the other one codified in Java and AspectJ. Each new release was also implemented based on the previous one. For example, the OO release 2 represents the evolution of the OO release 1. Most of the change scenarios are related to the addition of new agency features. In order to implement these features, new software agents and roles have to be added. Roles represent collaborative activities of the agents in a specific context [Bäumer et al., 1997]. Each role also involves the specification of its knowledge. Table 1 summarizes the changes undertaken to implement the releases.

Releases	**Description**	**Change Type**
R1	ExpertCommittee core	
R2	Addition of the Reviewer role.	Inclusion of optional feature.
R3	New feature to add user agents including the author and chair roles. New feature to allow the suggestion of conferences to the authors.	Inclusion of optional features.
R4	Addition of a Notifier agent to send messages to the system users through email and SMS.	Inclusion of optional and alternative feature.
R5	Addition of the Deadline agent. This agent is responsible for monitoring the conference deadlines.	Inclusion of optional feature.
R6	Addition of a feature that allows the chair to automatically assign papers to reviewers. Extension of the deadline agent to allow reminder deadlines.	Inclusion of alternative feature and extension of optional feature.
R7	Addition of a Task agent.	Inclusion of optional feature.

Table 1: Scenarios of change in MAS-PL

During the evolution of the EC MAS-PL, we basically added three types of optional/alternative features:

- *New conference management features* – these features represent new functionalities related directly to the conference management process, such as the addition of support to program committee members assign papers to reviewers (release R2);
- *New autonomous behavior* – several software agents were introduced in the EC MAS-PL architecture (releases R3, R4, R5 and R7). These agents were implemented in the system with the purpose of implementing autonomous behavior related to recommendations to researchers (paper authors), deadline monitoring and pending tasks monitoring. The *Task Management* feature (release R7), for example, implied in the addition of a new agent to the system with a set of associated behaviors, which can be present or not, depending on the product being derived;
- *New behavior for an agent or role* – added internal variabilities to the agents, such as new agent roles or behavior. These types of features were modularized as: (i) specific plans to be executed by the agent under specific conditions (releases R5 and R6) or (ii) specific roles to be played by the agent in a specific context (release R3). The conference suggestion feature (release R3) is an example of such autonomous optional feature. The user agent, or more specifically the author role, can perform it. When a paper is registered in a conference, the author agent role perceives it and sends suggestions of related conferences for the author who has registered his/her paper.

2.3 The MAS-PL Architecture

The EC MAS-PL was structured according to the Layer architectural pattern [Buschmann et al., 1996]. It is composed of the following components/layers: (i) ***GUI*** **–** this layer is responsible for processing the web requests submitted by the system users; (ii) ***Business*** **–** is responsible for structuring and organizing the business services provided by the EC system; and (iii) ***Data*** **–** aggregates the classes of database access, and it was implemented using the Data Access Object (DAO) design pattern [Alur et al., 2001]. The agents are in a separated module, named Agents Module.

Figure 2 illustrates the architecture of the EC web-based system and highlights the core architecture. In our implementation, the JADE platform [JADE, 2008] was used as base platform to implement the software agents. These agents are responsible for monitoring the execution of different functionalities of the EC core system in order to provide the new agency features. The integration between the web architecture and the agents was accomplished by means of the adoption of the Observer pattern [Gamma et al., 1995]. Next, a brief detail about the agents of the EC MAS-PL is presented:

- *Environment Agent* - this agent monitors the EC system by observing the execution of specific business operations and its aim is to notify the other agents of the MAS-PL about the system changes. Each user agent is specified to perceive changes in the environment and take actions according to them. The environment agent was implemented using the Observer design pattern [Gamma et al., 1995];

- *User Data Agent* – this agent receives notifications when new users are created in the database. When it happens, it creates a new user agent that will be the representation of the user in the system. The initial execution of the user data agent demands the creation of a user agent for each user already stored in the database;
- *User Agent* – each user stored in the system has an agent that represents him/her in the system. This is the autonomous behavior, agents performing actions that the users should do. The user agent was designed in such a way that it can dynamically incorporate new roles. An example of autonomous behavior is when the paper submission deadline expires and the user agent in the chair role will automatically distribute the papers to the committee members. Besides this example, most of the user agents are responsible: (i) for analyzing and discovering pending tasks for user agents based on the roles the users play in the system; and (ii) for asking the notifier agent to send email or SMS notifications;
- *Deadline Agent* – this agent is responsible for monitoring the conference deadlines. This monitoring serves two purposes: (i) to notify the user agents when a deadline is nearly expiring; and (ii) to notify the user agents when a deadline has already expired;
- *Notifier Agent* – this agent receives requests from other agents to send messages to the system users. In the current implementation, it sends these messages through email and SMS;
- *Task Agent* - this agent is responsible for managing the user tasks. It receives requests for creating, removing and setting the execution of tasks. The requests are made by the user agents.

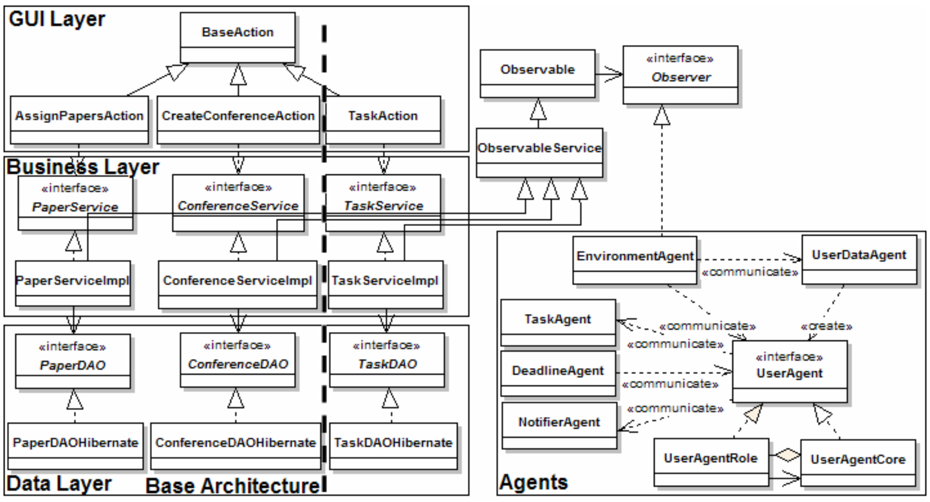


Figure 2: Expert Committee MAS-PL Architecture

2.4 MAS-PL Object and Aspect-Oriented Design

Figure 3 presents a partial class diagram of the OO implementation of the MAS-PL, illustrating the main components that were affected during the evolution of the system architecture. The OO releases were implemented using the Java programming language. The AO implementation was also structured following the Layer architectural pattern. Figure 4 shows a partial diagram of the AO implementation of the MAS-PL, illustrating a subset of its aspects. The `<<aspect>>` stereotype represents the aspects of the system. The dependency arrows represent that an aspect "crosscuts" the structure of system classes. The classes and aspects were marked in Figures 3 and 4 with a sequence of *Rs* above them. This indicates whether a class or aspect was added (+Rx) or changed (~Rx) during the implementation of the release X. In the AO implementation (Figure 4), we cannot observe any changes in its classes and aspects, it happens because only new aspects were added. The reasons will be explained in the next sections.

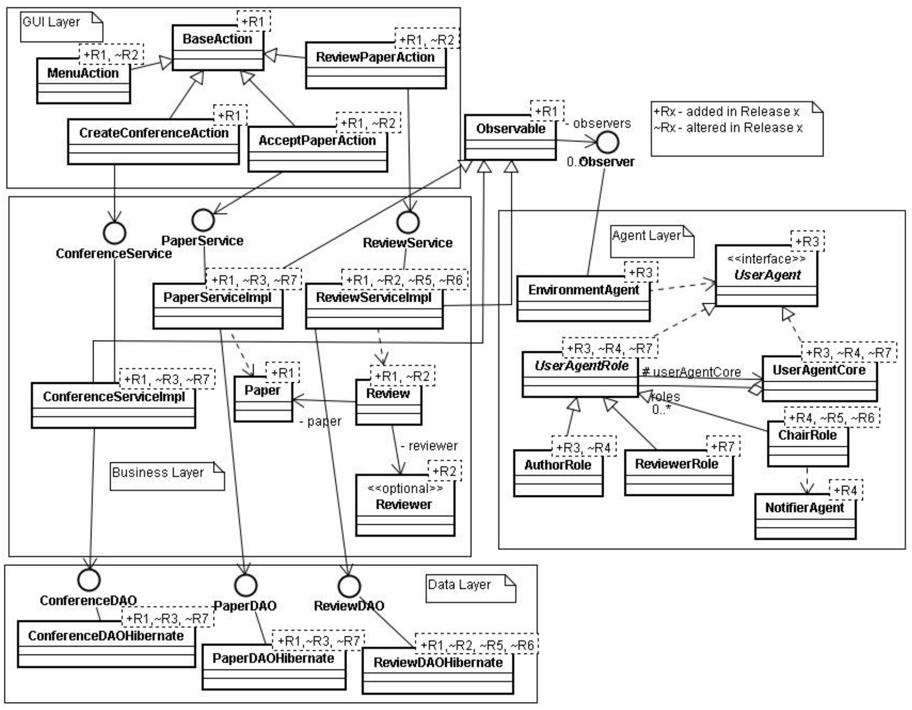

GUI Layer
BaseAction +R1
MenuAction +R1, ~R2
ReviewPaperAction +R1, ~R2
CreateConferenceAction +R1
AcceptPaperAction +R1, ~R2
Observable +R1
- observers
0..* Observer
+Rx - added in Release x
~Rx - altered in Release x
ConferenceService
PaperService
ReviewService
Agent Layer
PaperServiceImpl +R1, ~R3, ~R7
ReviewServiceImpl +R1, ~R2, ~R5, ~R6
EnvironmentAgent +R3
<<interface>> UserAgent +R3
UserAgentRole +R3, ~R4, ~R7
# userAgentCore
UserAgentCore +R3, ~R4, ~R7
roles 0..*
ConferenceServiceImpl +R1, ~R3, ~R7
Paper +R1
Review +R1, ~R2
- paper
- reviewer
ChairRole +R4, ~R5, ~R6
AuthorRole +R3, ~R4
ReviewerRole +R7
Business Layer
<<optional>> Reviewer +R2
NotifierAgent +R4
ConferenceDAO
PaperDAO
ReviewDAO
Data Layer
ConferenceDAOHibernate +R1, ~R3, ~R7
PaperDAOHibernate +R1, ~R3, ~R7
ReviewDAOHibernate +R1, ~R2, ~R5, ~R6

Figure 3: OO EC MAS-PL Simplified Architecture

During the MAS-PL development, the changes were performed following the best practices of design, using several patterns and current technologies that provided an important support in the development [Buschmann et al., 1996] [Gamma et al., 1995] [Bäumer et al., 1997]. In R2 (Figure 3), some classes were added and other modified, which are identified by the symbols *+R2* and *~R2* respectively:

`ReviewServiceImpl`, `Review`, `ReviewPaperAction` and `Reviewer`. The changes in these classes were done through conditional compilation in the OO EC MAS-PL. While in the AO EC MAS-PL (Figure 4), aspects were included to affect the core classes in order to preserve the base architecture. Thereby, we did not need to change the MAS-PL core classes, making the changes less invasive than using conditional compilation. From R3, we included the agents. In order to connect the agents and the existing architecture we used the Observer pattern. In the inclusion of this pattern in the OO EC MAS-PL, the original services implementation was changed to codify it, for example: `ConferenceServiceImpl` and `PaperServiceImpl`. While in the AO EC MAS-PL, aspects were added to affect these classes without modifying them directly, and include the code related to the Observer pattern such as: `ServicesInterceptAspect` and `ConferenceDAOAndServiceAspect`. In R4, agents and roles are included, such as: `NotifierAgent` and `ChairRole`. In the OO implementation, we used the role pattern [Bäumer et al., 1997] to separate better the roles. In order to include roles in the user agent it was necessary to modify the `UserAgentRole` and `UserAgentCore` classes (Figure 3). In the AO implementation, aspects were included to affect the `UserAgentRole` and `UserAgentCore` classes every time a role is included, for example: `UserAgentAspect` and `ChairRoleAspect` (Figure 4). This implementation was based on the AO role pattern [Kendall, 1999].

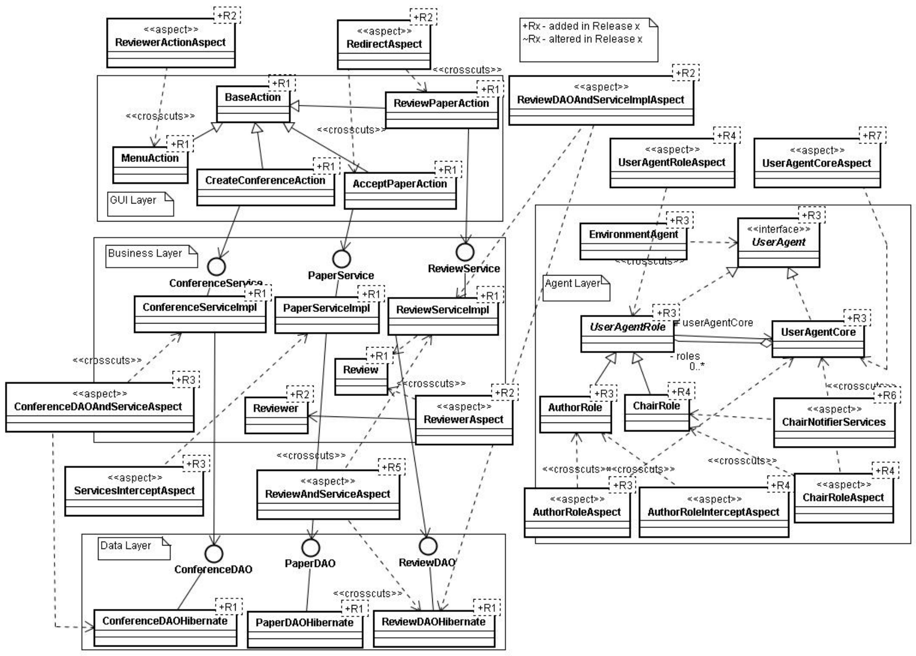

+Rx - added in Release x
~Rx - altered in Release x
<<aspect>> ReviewerActionAspect +R2
<<aspect>> RedirectAspect +R2
<<crosscuts>>
BaseAction +R1
ReviewPaperAction +R1
MenuAction +R1
CreateConferenceAction +R1
AcceptPaperAction +R1
GUI Layer
<<aspect>> ReviewDAOAndServiceImplAspect +R2
<<aspect>> UserAgentRoleAspect +R4
<<aspect>> UserAgentCoreAspect +R7
Business Layer
ConferenceService
PaperService
ReviewService
ConferenceServiceImpl +R1
PaperServiceImpl +R1
ReviewServiceImpl +R1
Review +R1
Reviewer +R2
<<aspect>> ReviewerAspect +R2
<<aspect>> ConferenceDAOAndServiceAspect +R3
<<aspect>> ServicesInterceptAspect +R3
<<aspect>> ReviewAndServiceAspect +R5
Data Layer
ConferenceDAO
PaperDAO
ReviewDAO
ConferenceDAOHibernate +R1
PaperDAOHibernate +R1
ReviewDAOHibernate +R1
EnvironmentAgent +R3
<<interface>> UserAgent +R3
Agent Layer
UserAgentRole +R3
userAgentCore
UserAgentCore +R3
roles 0..*
AuthorRole +R3
ChairRole +R4
<<aspect>> ChairNotifierServices +R6
<<aspect>> AuthorRoleAspect +R3
<<aspect>> AuthorRoleInterceptAspect +R4
<<aspect>> ChairRoleAspect +R4

Figure 4: AO EC MAS-PL Simplified Architecture

3 Empirical Study Phases

This section describes how the empirical study was organized. The study was structured following the guidelines and principles of Experimental Software Engineering defined by [Wohlin et al., 2000]. It focuses on the definition (Section 3.1), planning (Section 3.2) and operation (Section 3.3) of the experiment.

3.1 Definition of the Experiment

Initially, following the experiment process [Wohlin et al., 2000] the first step is to elaborate its definition. The purpose of the definition phase is to define the goals of the experiment. This definition is based on the GQM template [Basili and Rombach, 1988]. Following this template, our experiment goals were:

> ***Analyze*** the two different versions of the EC MAS-PL
> ***for the purpose of*** evaluating programming techniques
> ***with respect to their*** modularity and stability
> ***from the point of view of*** the developer
> ***in the context of*** MAS-PL.

3.2 Planning the Experiment

The planning phase of the experiment is divided in the following way: (i) context selection and selection of subjects; and (ii) variables selection.

Context Selection and Selection of Subjects. The experiment was run off-line, not in an industrial software development environment. The subjects were two MS.c students and four senior researchers. The EC MAS-PL has been developed by the MS.c students. The MS.c students have good experience in the MAS design and development. All the senior researchers have good experience in the software engineering area. All the participants already had experience in the elaboration of empirical studies of MAS [Garcia et al., 2003], product lines [Figueiredo et al., 2008], architectures of typical web-based information system [Kulesza et al., 2006b] [Greenwood et al., 2007] and more recently MAS-PL [Nunes et al., 2008b] [Nunes et al., 2008c].

Variables Selection. The variables selection comprises the independent and dependent variables. The independent variables are the two techniques used together with the JADE platform, which are: conditional compilation and AOP. For all these implementations, the main aim was to provide good modularizations of the new features introduced in the EC MAS-PL. The dependent variables in the experiment are modularity and stability. In our study, we use a suite of metrics for quantifying the modularity, which are: separation of concern metrics, interaction between concerns, size, cohesion and coupling [Sant'Anna et al., 2003] [Sant'Anna et al., 2007]. Table 2 briefly presents the modularity metrics used in this work. Our study also comprised typical change impact measures [Yau and Collofello, 1985] (Section 5). We chose these metrics for several reasons. First, they are conventional measures (previously used to assess single systems and program families), which were already validated in terms of software stability [Greenwood et al., 2007] [Figueiredo et al., 2008]. [Eaddy et al., 2008] conducted a study with the concern metrics (e.g. CDO and CDC), which demonstrates a statistic correlation between them and a relevant stability factor, namely error proneness. We also noticed from other studies that these metrics have a correlation with instability factors in different artifacts of abstraction levels, including use cases [Conejero et al., 2009], architectural descriptions [Sant'Anna et al., 2008], and source code [Greenwood et al., 2007] [Figueiredo et al., 2008]. Moreover, a combined analysis of the metrics that we used is similar to the isolate use of some Eaddy's metrics, for example, with size metrics (e.g. number of components). Because some Eaddy's metrics are normalized, measuring scattering/tangling in terms of the total number of components of a system.

Attributes	Metrics	Definition
Separation of Concerns	Concern Diffusion over Components (CDC)	It counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern Diffusion over Operations (CDO)	It counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusion over LOC (CDLOC)	It counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Interaction Between Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which a given concern shares at least a component.
Size	Lines of Code (LOC)	It counts the lines of code.
	Number of Components (NOC)	It counts the number of components (classes and aspects).
	Number of Operations (NOO)	It counts the number of operations of a given component.
Coupling	Coupling Between Components (CBC)	It counts the number of other classes and aspects to which a class or an aspect is coupled.
Cohesion	Lack of Cohesion in Operations (LCOO)	It measures the lack of cohesion of a class or an aspect in terms of the amount of methods and advice pairs that do not access the same instance variable.

Table 2: The Metrics suite

We have carefully avoided proposing new metrics in our study because it does not know the empirical correlation with stability. Second, despite not proposing new SPL-specific metrics, the choice of the metrics was performed taking characteristics of SPL into consideration. For example, we used the concern-sensitive metrics that allowed us to evaluate the modularity properties from each feature point of view. Additionally, coupling and cohesion metrics were used because they allow us to evaluate the dependencies of the core/variable modules. Finally, from our experience it is possible to use conventional metrics to SPL domain provided that the measurement analyses focus on SPL specificities. However, as said previously, the analysis and application are different, focusing mainly on the features modularity analysis and thus relevant to the SPL evolution.

3.3 Operation

Preparation. Initially, the EC MAS-PL was developed by two subjects (Section 3.2). We have initially implemented the version with the JADE platform and conditional compilation technique. After that, we implemented the other version with JADE platform and AO techniques. During the development of both EC MAS-PL versions, we first implemented the MAS-PL core. After that, we applied a set of changes (Table 1). The changes applied to the OO and AO versions have been done by two MS.c students and one MS.c student, respectively. All the changes were supervised and validated by the senior researchers to guarantee the best design practices of MAS, OO and AO development were adopted. These changes were originally predicted by all involved subjects. Both students and researchers have good knowledge in the design and development of OO and AO systems. Besides, they have good knowledge in software metrics. During the development, we have used the same best design practices throughout all versions of the EC MAS-PL releases, such as, to adopt the layer architectural style and common OO and AO design patterns [Buschmann et al., 1996] [Gamma et al., 1995] [Bäumer et al., 1997] [Hannemann and Kiczales, 2002] that refine each layer. These good practices and validation activities were accomplished by the subjects, evaluating the MAS-PL design. These designs assure the comparison was equitable and fair.

Execution. The execution has been applied in the following way: (i) application of change impact metrics; (ii) application of the modularity metrics; and (iii) data analysis. The counting process was done by two MS.c students. The other four senior researchers monitored the counting process and helped to interpret the collected data. This monitoring was important to guarantee that the data were being collected in the right way. Thereby, during the couting process there was much communication among the subjects. The students dedicated one hour per day to count the metrics in order to avoid the tiredness.

4 Modularity Analysis

In this section, we discuss the study results for the modularity metrics, which are related to the following software attributes: separation of concerns, interaction between concerns, cohesion, coupling and size. Our goal was to observe the stability of each modularity attribute in the EC MAS-PL implementations.

4.1 Separation of Roles and Agent Concerns

In our study, we have analyzed three optional features included in the releases 2, 3, and 4 of the EC MAS-PL using the separation of concerns metrics (Table 2). These selected features represent multi-agent abstractions (roles or agents) that modularize relevant agency features of the MAS-PL. The chosen features also represent fine- and coarse-grained implementations in MAS-PLs. In addition, these features were also selected because they were added to the MAS-PL during the first three evolution scenarios. This allowed us to analyze the behavior of these features throughout the last three releases (Section 2.2). Figure 5 presents the results of the CDC, CDO and CDLOC metrics for the Reviewer role, which is an optional feature, added in the R2 (Table 1). The results show that the Reviewer role is scattered over fewer components and operations (CDC and CDO metrics) and tangled with fewer features in the AO implementation (CDLOC metric). This indicates that the AO implementation was more effective to modularize this feature when compared to the OO implementation. This occurred because in the AO solution, the pieces of code in charge of realizing the optional roles are transferred from classes to a set of dedicated classes and one or more glue aspects. In the AO implementation of SPLs, aspects usually play an excellent role as the glue between the core and optional features [Alves et al., 2005] [Kulesza et al., 2006b]. The conditional compilation technique, adopted in the OO solution, lacks this ability because it has a somewhat intrusive effect on the code, due to the need to add the *#ifdef* and *#endif* clauses locally at the places where features intersect.

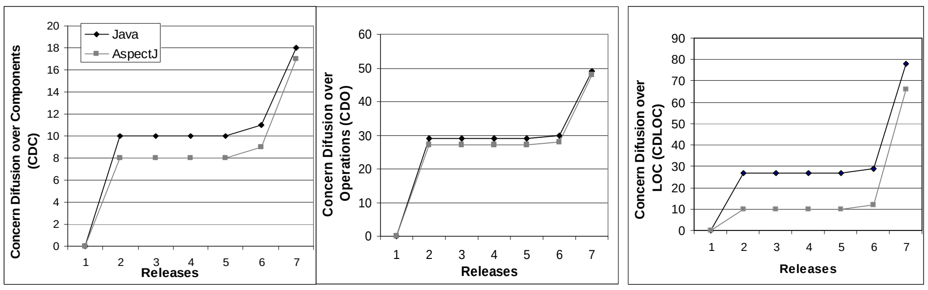
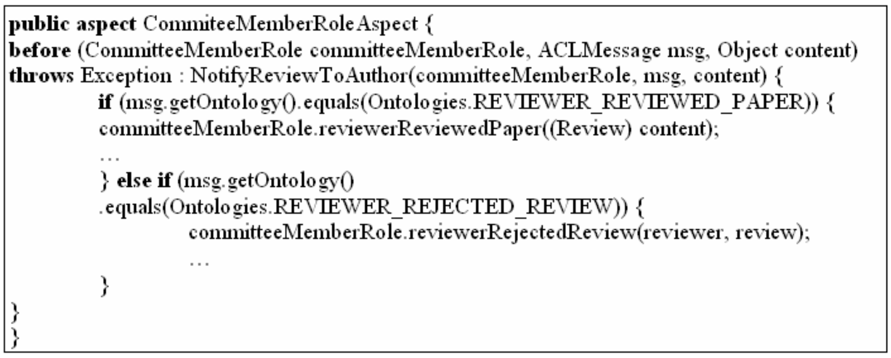


Figure 5: Concern Metrics for Reviewer feature (R2)

In the OO implementation, the Reviewer feature is spread over a number of classes, such as: `Reviewer`, `Review`, `ReviewPaperAction`, and `ReviewDAOHibernate`. In Figure 3 these classes underwent changes in the R2 (symbol ~R2). Such changes were carried out in order to introduce code related to Reviewer role in the mentioned classes. Also note that the `Review` class has a direct association with the `Reviewer` class. The `Reviewer` class was introduced in the R2 and is totally dedicated to implement the `Reviewer` role. In the AO implementation (Figure 4), part of the Reviewer role is implemented by the Reviewer class and three aspects: `ReviewerAspect`, `RedirectAspect` and `ReviewDAOAndServiceImplAspect`. These aspects introduce the Reviewer role behavior in the `Review`, `ReviewPaperAction`, and `ReviewDAOHibernate` classes, which are therefore free from code related to this specific role. The `ReviewerAspect` aspect is responsible for adding the Reviewer attribute and some methods using inter-type declarations from AspectJ. This aspect works as glue code between the `Review` and `Reviewer` classes. The

`ReviewDAOAndServiceImplAspect` is responsible to affect business and database access methods that manipulate the review object. This aspect affects methods from `ReviewServiceImpl` and `ReviewDAOHibernate` classes to treat by different ways the reviews done by a reviewer or a program committee member. This is the main reason for the decreasing of the degree of scattering and tangling in the AO solution, reflected in the concern metrics. Note that the four classes affected by the aspects were not changed in release 2 of the AO implementation (Figure 4). In Figure 5, we can see that the tangling of the Reviewer feature with other features is largely higher in the OO implementation. On the other hand, the scattering of the Reviewer feature over operations and components is almost the same in both implementations. This occurred because the Reviewer feature is not much scattered in the OO version. Thus, the benefits of using AO in this case were not so significant for this feature.

In the OO and AO implementations of the R7, there is a significant increase in all SoC metrics because the addition of the Task Agent feature includes several event classes that communicate with the Reviewer feature. Besides, there are changes in other classes that implement the roles and they also communicate with the reviewer feature, such as: `ChairRole` and `CommitteeMemberRole`. For example, Figure 6 depicts the communications of the committee member with the reviewer defined in the `CommiteeMemberRoleAspect` aspect. Some types of communications are: reviewer reviewed the paper and reviewer rejected the review. If the review is rejected by the reviewer, the committee member is notified and the `reviewerRejectedReview` method is called.



```
public aspect CommiteeMemberRoleAspect {
before (CommitteeMemberRole committeeMemberRole, ACLMessage msg, Object content)
throws Exception : NotifyReviewToAuthor(committeeMemberRole, msg, content) {
            if (msg.getOntology().equals(Ontologies.REVIEWER_REVIEWED_PAPER)) {
            committeeMemberRole.reviewerReviewedPaper((Review) content);
            ...
            } else if (msg.getOntology()
            .equals(Ontologies.REVIEWER_REJECTED_REVIEW)) {
                        committeeMemberRole.reviewerRejectedReview(reviewer, review);
                        ...
            }
}
}
```

Figure 6: Communication between committee member and reviewer with AOP

Figure 7 shows the results of the CDC, CDO and CDLOC metrics for the User Agent feature, which is also an optional feature, in terms of concern metrics. For this feature, the collected values for the AO solution did not present better results compared to the OO solution in terms of CDO and CDC metrics. The number of operations of the User Agents feature increased through the evolution of the MAS-PL because new operations were added in the MAS-PL core using inter-type statements to implement this feature. Figure 3 shows that in the OO implementation, the User Agent feature is spread over fewer components (classes or aspects). This happens because with conditional compilation, it is only necessary to add the *#if/#endif* clauses locally in a few classes. Thus, the degree of scattering along classes and operations presents low values in the OO solution. The `UserAgent`, `UserAgentCore` and `UserAgentRole` classes were introduced in the R3 and are totally dedicated for implementing the User Agent feature in the OO and AO implementations. Note that Figure 3 shows that the classes added in R3 were modified during the evolution of the MAS-PL. The changes in these classes increase the tangling as can be seen in Figure 8. In the AO implementation (Figure 4), a significant part of the User Agent feature is implemented by: (i) the `UserAgent`, `UserAgentCore` and `UserAgentRole` classes; and (ii) a set of aspects that affect the specified roles, such as `AuthorRoleAspect` and `ChairRoleAspect`. Due to the use of the role pattern, the number of aspects to manage the roles separately that belong to User Agent feature are higher**.** But on the other hand, this AO implementation of the agent roles is less tangled with other agent features. This way, the degree of scattering is higher in the AO solution (CDC and CDO), but less tangled than the OO solution. This high number of aspects can be seen as a negative characteristic of the AO solution, because it can harm the understanding of the feature, since there are more components to deal with.

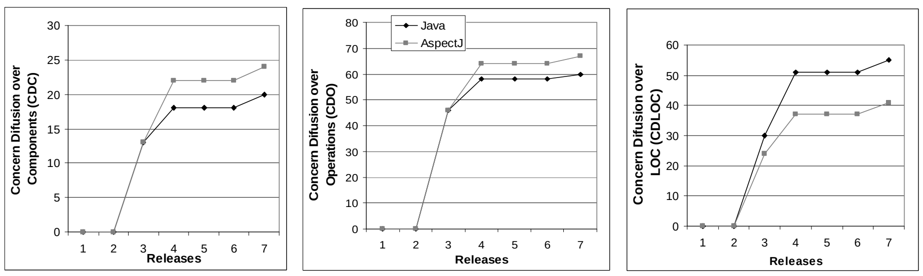


Figure 7: Concern Metrics for User Agent feature (R3)

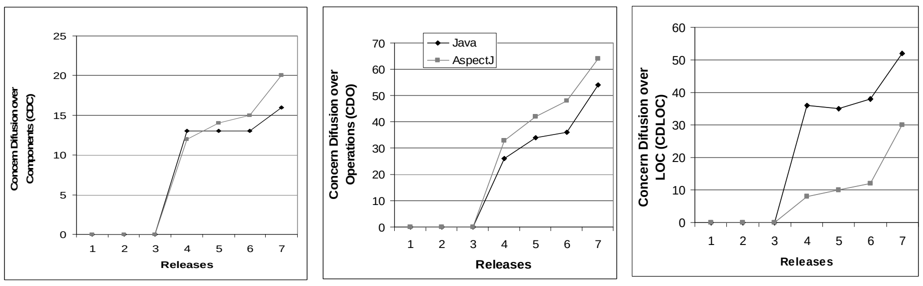


Figure 8: Concern Metrics for Notifier Agent feature (R4)

Figure 8 shows the results of the CDC, CDO and CDLOC metrics for the Notifier Agent feature. During the evolution of the MAS-PL, there was a significant increase in the number of components for the AO implementation from R5. This occurred because in the OO implementation, the notification code from specific system events was codified directly in the classes using *#ifdef/#endif* clauses. On the other hand, in the AO implementation, different aspects were created to affect these existing classes. They were created because each of them is related to one specific role of user agent which needs to be managed separately in order to guarantee an easy inclusion/removal of the optional feature that it represents. Thus, in the AO implementation the code was modularized in separated aspects, such as: `AuthorRoleInterceptAspect and ChairNotifierServices` (Figure 4). As a consequence of the number of added components, in the AO solution we have more operations related to the feature implementation. However, this high number of components in AO solution gets to reduce the tangling, as can be seen in the results of the CDLOC metric.

4.2 Coupling and Cohesion

This section presents all the coupling and cohesion measurement of the EC MAS-PL. Figure 9 presents the coupling average per component (classes and aspects), CBC metric (Table 2). The coupling average in the AO solution presented a better result than the OO solution. Observing the chart, the AO implementation presented more stable values for the coupling. Despite many aspects reducing the coupling among the classes along of the MAS-PL evolution, some of them still maintain references to some classes. This occurs due to the use of inter-type declarations to allow the modular introduction of optional or alternative features. However, the AO solution presented more components (classes and aspects) during the EC MAS-PL change scenarios, but the final components are more decoupled between them than the OO solution. One example of this coupling in OO solution is the role pattern implementation [Bäumer et al., 1997]. In the OO implementation, each role class accesses methods of several classes, while in the AO solution the roles were modularized in aspects, which contributed to reduce the coupling.

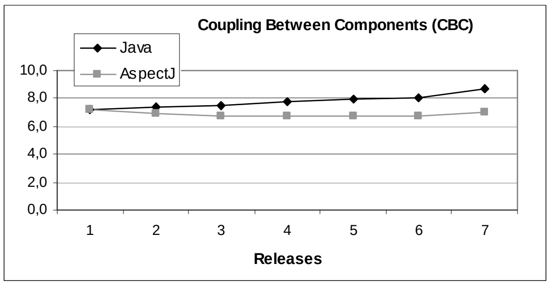


Figure 9: Coupling Average per Component

Figure 10 presents the results of the cohesion average per component during the MAS-PL change scenarios, LCOO metric (Table 2). When the LCOO metric presents low values, this indicates a high cohesion, while superior values indicate a lack of cohesion. In the presented chart, the value 0.2 for the R2, for example, means that the sum of the cohesion metric for all the components in R2 divided by the total number of components is 0.2. According to Figure 10, the LCOO metric also presents better results in the AO solution. One reason that contributed for this improved cohesion was the implementation of the Observer pattern [Gamma et al., 1995] and the inclusion of the task agent feature with AOP. Thus, AOP contributed to modularize attributes and operations more strongly related thus contributing to improve the cohesiveness of the MAS-PL. For example, when a new user is stored in the system, the Environment Agent discloses that information, and the User Data Agent perceives it and creates a new agent (*User Agent*) for representing the new user in the system. That notification is done by the `ServiceInterceptAspect` aspect, which affects the `store()` method of the `UserServiceImpl` class and creates a user creation event to notify the Environment Agent. In Figure 11, it is possible to see that the `ServiceInterceptAspect` aspect affects some services implementation such as: `PaperServiceImpl` and `UserServiceImpl`. According to this diagram, the code of the Observer pattern is included in a transparent way by the `ServiceInterceptAspect` aspect. One reason for the better results in the AO solution in terms of LCOO metric is that in the OO implementation, the classes contain methods and attributes of its original implementation as related to the Observer pattern. Besides, the OO implementation is more coupled with the classes of the pattern (CBC metric).

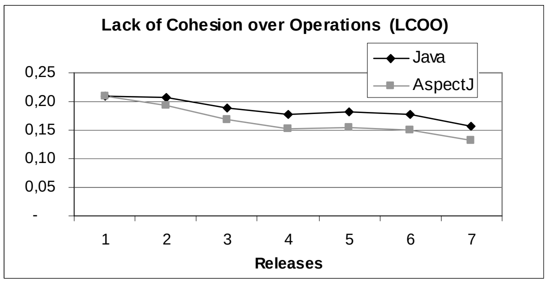


Figure 10: Cohesion Average per Component

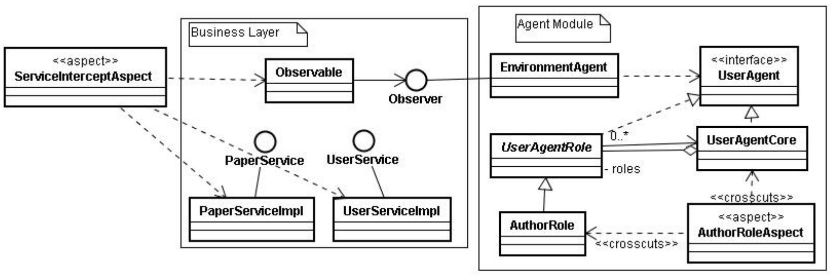
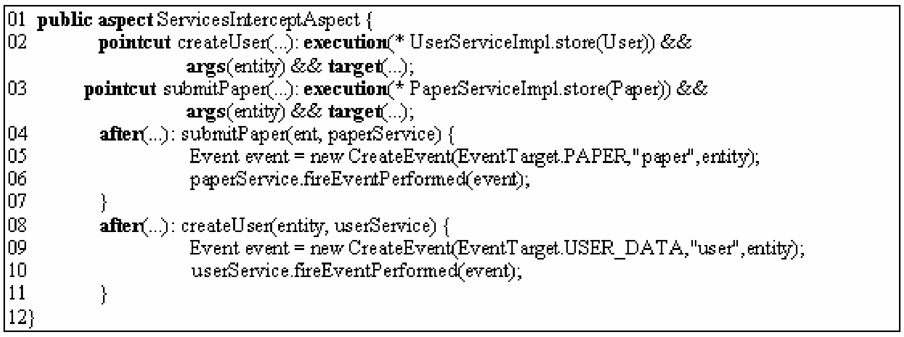


Figure 11: Simplified Diagram of the Observer Design Pattern

Figure 12 shows the `ServiceInterceptAspect` aspect, which implements the observing relationship for the *Environment Agent* (Observer). This aspect is responsible for intercepting some methods of the `PaperServiceImpl` and `UserServiceImpl` service classes and notifying the *Environment Agent*. First, the advice associated to the `createUser` pointcut is called to store the user. After the user creation, roles can be dynamically added to this agent. For example, the author role is assigned to the agent when the user submits some paper to the conference. Thus, after the paper submission, the *Environment Agent* divulges that event and the *UserAgentCore* agent perceives it and assigns the author role to the agent. That notification is also done for the Observer pattern. This can also be viewed in Figure 12 through the `submitPaper` pointcut. When the notification arrives in the *UserAgentCore* agent, the `AuthorRoleAspect` aspect is responsible for intercepting the `addRole()` method of the `UserAgentCore` class and creates the author role and includes in the user agent. From this code (Figure 12), it is possible to see how the AOP helps to reduce the values of the LCOO metric.



```
public aspect ServicesInterceptAspect {
        pointcut createUser(...): execution(* UserServiceImpl.store(User)) &&
                    args(entity) && target(...);
      pointcut submitPaper(...): execution(* PaperServiceImpl.store(Paper)) &&
                    args(entity) && target(...);
        after(...): submitPaper(ent, paperService) {
                    Event event = new CreateEvent(EventTarget.PAPER,"paper",entity);
                    paperService.fireEventPerformed(event);
        }
        after(...): createUser(entity, userService) {
                    Event event = new CreateEvent(EventTarget.USER_DATA,"user",entity);
                    userService.fireEventPerformed(event);
        }
}
```

Figure 12: Propagation of events using the Observer pattern

4.3 Feature Dependency Analysis

Figure 13 shows the results of the Component-level Interlacing between Concerns (CIBC). This metric aims at quantifying the interaction between concerns. It shows the degree of interlacing between concerns/features along the different classes and aspects in the investigated system or product line. Figure 13 shows, for example, the interaction of the Reviewer and User Agents features with the other MAS-PL concerns (Roles: Author, Chair, CommitteeMember, Coordinator; ACLMessage, Persistence, Review and MessageFactory). According to Figure 13, the Reviewer feature is encountered tangled with fewer concerns in the AO implementation.

This occurred because the AO implementation transferred almost all the elements in charge of realizing this feature from `Reviewer`, `Review`, `ReviewPaperAction`, and `ReviewDAOHibernate` classes (Figure 3) to aspects: `ReviewerAspect`, `ReviewDAOAndServiceImplAspect` and `RedirectAspect` (Figure 4). This contributed to separating this feature from the other concerns. Figure 13 also shows the CIBC metric for the User Agent feature. Note that the degree of interaction between the User Agent feature and other concerns also presented lower values in the AO solution along the MAS-PL evolution. This was due to the same reasons noted for the Reviewer feature.

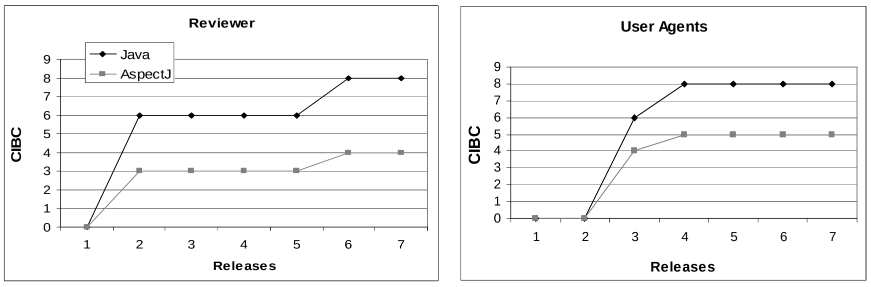


Figure 13: CIBC metric

4.4 Size

Figure 14 presents the results of the following size metrics: Lines of Code (LOC), Number of Components (NOC) and Number of Operations (NOO). It shows that the collected values for the AO implementation were higher when compared to the OO implementation. This happened mainly because we decided to include different aspects to maintain and improve separation of concerns and feature management. An example of this modularization strategy is the inclusion of the Role pattern. This causes the creation of many aspects (increasing the values collected for the NOC metric), each of them with different advices and pointcuts (LOC and NOO metrics are higher in the AO implementation) affecting the system classes. On the other hand, in the OO implementation, the use of conditional compilation with the addition of AND and OR operators in the existing classes were sufficient to support the combination of determined features, such as: User Agents and Notifier Agent. The AO implementation required the creation of new aspects to represent those combinations of features, such as: `AuthorRoleAspect` and `AuthorRoleInterceptAspect` (Figure 4). The results obtained for the size metrics showed that although the AO implementation improved the separation and interlacing of concerns, coupling and cohesion of agency features (Sections 4.1 to 4.3), it required to create and manage new aspects with their respective pointcuts and advices (high values obtained for the NOC, LOC and NOO metrics).

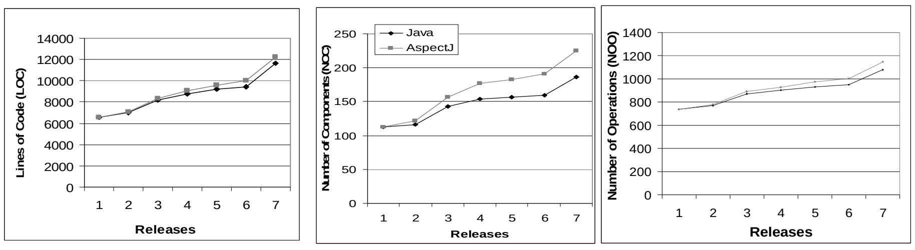


Figure 14: Size Metrics of the Expert Committee

5 Stability Analysis

Our study comprised the following typical change impact measures [Yau and Collofello, 1985]: number of added or changed components (aspects and classes), number of added or changed operations and number of added or changed lines. The purpose of using these metrics is to assess the propagation effects in terms of components, lines of code and operations during the introduction of agency features in this EC MAS-PL. Table 3 presents the change propagation in the EC MAS-PL implementation considered the mentioned metrics.

		R2	**R3**	**R4**	**R5**	**R6**	**R7**
Added Components	**OO**	3	27	11	3	3	26
	OA	9	35	20	6	8	34
Changed Components	**OO**	9	6	8	8	7	13
	OA	0	0	0	0	0	0
Added Operations	**OO**	32	103	31	29	20	128
	OA	43	112	35	49	27	145
Changed Operations	**OO**	4	2	15	2	2	31
	OA	0	0	0	0	0	0
Added Pointcuts	**OA**	5	7	9	1	1	19
Changed Pointcuts	**OA**	0	0	0	0	0	0
Added LOC	**OO**	418	1134	639	391	249	2203
	OA	511	1202	784	470	496	2166
Changed LOC	**OO**	0	0	0	0	0	0
	OA	0	0	0	0	0	0

Table 3: Change Propagation in the EC MAS-PL releases

According to Table 3, the AO solution presented a high number of added components to implement the agency features for all the releases when compared to the OO solution. During the development of the releases R3 and R7, there was a significant increase in the number of added components for both OO and AO implementations. In the AO solution, this higher number of components occurred to: (i) allow the separation of concerns in the roles level; and (ii) because the addition of the task agent brings impact to several components. Thus, several aspects needed to be created to allow the easy addition/removal of these features, thus providing a better variability management. In the OO solution, this high number of added components occurred because of: (i) the addition of the Task Agent feature includes several event classes that communicate with all the roles; and (ii) the introduction of a series of associated classes for handling the events of specific users in order to create, remove tasks, and setting their execution date.

Note that in the AO solution in all releases, there were no changes in its components (classes and aspects). This is due to the fact that only new aspects were added to implement new features. This was a positive factor to preserve the design of the MAS-PL architecture during its evolution. On the other hand, in the OO solution several components were changed during the MAS-PL evolution along the releases. This occurred because the use of conditional compilation in the OO solution demanded the addition of the *AND* and *OR* operators in the existing classes to support the combination of the features. Note that in R7 (Table 3), there were 13 changed components in the OO solution, while the AO solution did not have any change, thus providing a better modularization of the agency feature over the components**.** As a result, this reflected in the number of added operations: the AO solution presented higher values for this metric than the OO solution in all releases. This happens because in the AO solution, the changes are accomplished in existing components of the core MAS-PL using inter-type declarations and using pointcuts that affect specific join points of the MAS-PL. However, there were no changes in operations and pointcuts in the AO solution. In the OO solution, some operations were modified to allow the combination of the features. As a consequence of the added operations and components in the AO solution, the number of lines of code was almost always superior for all the releases.

An interesting observation in the collected data is the absence of changed LOC in both versions OO and AO. Basically, this occurred because the proposed architecture through the adoption of design patterns for both OO and AO versions facilitated the inclusion of new features with minimum impact. This characteristic made the EC MAS-PL versions design and implementation easy to evolve. Thus, the lines of code already implemented previously did not need to be changed, only new lines of code were included to implement a particular optional or alternative feature. Thus, the results collected for the Changed LOC stability metric has shown that the use of good OO and AO practices and design patterns can benefit the design stability and facilitate the development and evolution of MAS-PL by reducing the change of source code when incorporating new features. In fact, in a recent study [Nunes et al., 2008c], we have used the same architectural pattern to incorporate autonomous behaviour [Nunes et al., 2008d] in another web-based SPL, and the results for many stability metrics were quite similar.

The absence of changes in the components, operations, lines of code and pointcuts for the AO releases confirmed a superior adherence to the Open-Closed Principle [Meyer, 1998], which states that "software should be open for extension, but closed for modification". The AO solution behaved following this principle, showing that it was more appropriate to implement the agency features in our case study.

6 Threats to Validity

In this section, we discuss the threats to the study validity. Threat to conclusion is concerned with the relation between treatments and the outcome. The main threat to conclusion is related to the size of the MAS-PL. Although only one experiment was presented in this paper, it involved a representative and non-trivial web-based system, which it was implemented using several mainstream technologies and current best practices. Additionally, we tried to perform real change scenarios that could be applied in other web-based modern systems, and related to the introduction of relevant autonomous behaviour for the investigated domain.

The goal of our study was to compare two different implementation technologies (OO with conditional compilation and AO techniques) in the development and evolution of a MAS-PL. We have decided to focus on analyzing optional features in this study in order to fill the gap of previous work. In our recent study [Nunes et al., 2008c], we have analyzed alternative and mandatory features using different modularization technologies. On the basis of contrasting our previous experience with this study, we observed that the maintenance of optional features tend to have more impact in the MAS-PL core architecture and, therefore, need special attention on SPL development**.** The AO implementation was developed using the AspectJ language. We used the AspectJ due to its stability and because it is widely used and the most consolidated AO language. Moreover, other works cited below also used AspectJ to implement different SPLs.

Threats to construct validity are related to the design of the experiment. The threat to construct validity includes the suite of metrics used for quantifying concern-sensitive modularity properties and change impact metrics. The metrics used in this work have already been used and validated in several recent empirical studies [Figueiredo et al., 2008] [Greenwood et al., 2007] [Kulesza et al., 2006b] [Sant'Anna et al., 2003] [Eaddy et al., 2008]. The concern metrics had to be calculated manually. In order to count the concern metrics, it is necessary to do the "shadowing" of the code to verify the piece of code that implements a determined feature in MAS-PL. This process started only after all releases (Java and AspectJ) were developed and aligned. In fact, it may be a threat to validity of the study since it involves direct code inspection. However, the outcomes were always validated by different subjects of the study. On the other hand, for gathering the values of the coupling, cohesion, size and change impact metrics we used automated tools, which were: Eclipse Metrics plugin [Eclipse, 2008] and the Together tool [Together, 2008]. Therefore, the threats to construct validity are reduced, because we have used automated tools to count most of the metrics.

Threats to internal validity are factors that can affect the independent variables. The threats to internal validity are related the MAS-PL alignment rules, in other words, to maintain the same design practices throughout all the EC MAS-PL releases. During the development we have used the same design practices throughout all OO and AO EC MAS-PL releases, such as, the layer architectural pattern and classical design patterns in order to minimize this threat. This alignment was necessary to ensure the quality of design in all versions and to do the comparison fairer and more equitable.

Threats to external validity are conditions that allow to generalize the results of the experiment. The main threat to external validity is the nature of the chosen experiment. In order to minimize this threat, we tried to involve a number of experienced people in the empirical software engineering area. The EC MAS-PL different versions were implemented by two experienced MS.c students from Computer Science Department at PUC-Rio. Both students have good knowledge on Java and AspectJ languages. However, the design decisions taken during the implementation of the EC MAS-PL architectures were always validated by senior researchers with good knowledge and experience in the conduction of other empirical studies. It is important to do more experiments involving other MAS-PLs and subjects with different experiences in order to be able to generalize the study findings related to the modularization of MAS-PL agency features with different implementation techniques. Recently, we have conducted a new and different empirical study with this aim in mind [Nunes et al., 2008c].

7 Discussions and Lessons Learned

This section discusses and analyzes the collected results with this study. Basically, we emphasize the advantages and drawbacks of using AOP or conditional compilation. Besides, we cite some challenges addressed to extend the benefits of such techniques.

Construction of maintainable MAS. Analyzing the design stability (Section 5) of the EC MAS-PL, we can say that AOP was more effective to allow the superior stability in the implementation of agency features, demanding less intrusive modifications in the existing components, operations and pointcuts (Table 3), and presenting superior adherence to the Open-Closed principle [Meyer, 1998]**.** In the AO solution, most of the evolution scenarios were developed with the codification of new aspects. This choice was made to allow the modularization and (un)plug of the new features being included. While the OO solution required more extensive and invasive changes inside its existing and already convoluted classes, as presented in the addition of the reviewer feature (Section 5). Conditional compilation technology is largely used in the industry, especially in the development of embedded systems and mobile games. In our study, we used conditional compilation in a different domain, which is the web system domain. Nevertheless, this mechanism is not so appropriate, because it has poor legibility and leads to lower maintainability. Thus, according to change impact, the AO solution provides a better management of the MAS-PL features, bringing facilities to their maintenance.

Modularity of the MAS-PLs Implementations. This paper provides empirical evidences that AOP in determined situations is better than conditional compilation**.** In general, the use of conditional compilation presented less components and operations directly related to feature implementations (CDC and CDO metrics) as can be seen in the User Agents and Notifier Agent features (Section 4.1). As mentioned previously, this occurs because the changes are performed locally through the operators using conditional compilation. However, the impact during the MAS-PL evolution in the AO solution was less invasive in terms of concerns diffusion over lines of code (CDLOC) and over components (CDC). In our study, AOP provided a better modularization by decreasing the tangling and coupling between components, and providing a higher cohesion. On the other hand, the use of AOP resulted in more components and lines of code to manage. Although there are more components and lines of code, they are well modularized in separate aspects dedicated to implement specific optional and alternative features. The implementation of variabilities with AOP also brings another benefit: the capacity of plugging/unplugging aspects from the SPL core implementation. With AOP it is possible to extract crosscutting features to aspects to provide a better modularization [Alves et al., 2005] [Kulesza et al., 2006a] and to allow the integration among the features in SPL. During the evolution of the MAS-PL, several aspects worked as a "glue" code between the OO core structure and the different optional and alternative agency features added to this core. This design decision was very useful because it allows injecting new properties and behaviors (agency features) in a transparent way into the base OO structure. Also, the use of AO technologies makes it easy to remove specific agency features or replacing them with other implementation (alternative features). The main example in our study of the use of the aspects as "glue" code was the implementation of the Reviewer feature (R2). The complete isolation of crosscutting optional and alternative features in SPLs and application frameworks using AOP also brings benefits to the process of automatic product derivation as emphasized by some recent works [Voelter and Groher, 2007] [Cirilo et al., 2008].

Incremental Implementation of the MAS-PL. During the development and evolution of the MAS-PL, several aspects helped in the integration between the MAS-PL core architecture and the different optional and alternative agency features. The AOP mechanisms allowed the non-invasive introduction of code related to agency features in the MAS-PL core (original web-based system). The integration of the software agents with the MAS-PL core architecture was accomplished using the Observer pattern (Figure 12). The AO implementation adopted a variant implementation of this pattern, which brings minimum impact to the system being observed. In summary, in the AO version of the MAS-PL, the aspects contributed to modularize the variabilities related to the agents, agent roles and the integration of the agents with the system core. The inclusion of the agents and its roles was possible because the aspects affected some classes adding specific functionalities.

Feature Management. During the MAS-PL evolution the addition of some agency features (such as User Agents and Notifier Agent) caused a high number of new components (aspects) in the AO solution. Although the use of aspects increases the number of components in these cases, it was also useful to reduce the tangling and coupling between the concerns/features. Thus, the AO solution was more effective to modularize these features (Sections 4.1 to 4.3). In our study, it was observed that the AO solution exhibits higher values for all the size metrics (LOC, NOC and NOO). This was a negative finding regarding the AO implementation because it can demand the understanding of additional code in the new aspects added to the system, and thus harming the evolution of the MAS-PL. Therefore, a trade-off analysis is required to determine if the benefits in terms of separation of concerns, demonstrated for the MAS-PL features (Section 4.1), can overcome the occasional difficulties to deal with the additional aspects, operations and lines of code brought by an AO implementation.

8 Related Work

Recent research presents some studies with the use of AOP in SPL development [Alves et al., 2005] [Colyer et al., 2004] [Griss, 2000] [Hunleth and Cytron, 2002]. There are also some empirical studies comparing OO and AO implementations of systems and product lines. However, most of these studies focus on the modularization of conventional crosscutting concerns such as: persistence [Kulesza et al., 2006b] [Soares et al., 2006], exception handling [Filho et al., 2006] and design patterns [Garcia et al., 2005] [Hannemann and Kiczales, 2002]. None of the cited works analyzes the impact of adding agency features in evolution scenarios of a MAS-PL. We considered a different approach of other works, that is the MAS-PL and the several change scenarios applied to the core architecture focusing on the quantitative assessment of AO and OO solutions. Next, we give an overview of the different AOP studies conducted for different application domains, emphasizing the main differences and findings between our study and those ones.

[Figueiredo et al., 2008] present an empirical study focusing on requirements evolution of two product-lines for the mobile application domain, called MobileMedia and BestLap. This work analyzes the evolution of product lines in terms of metrics for modularity, change propagation and feature interaction. Similar to our study, two variabilities implementation techniques were considered: conditional compilation and AOP. They concluded that AOP promoted more stable designs, mainly in alternative and optional features. They also observed that AOP presents stable values for all the kinds of interlacing interactions. In our study, we have also found that AOP was more appropriate to modularize optional agency features, like agents and its roles. Additionally, our study also presented better results in terms of design stability for the AO implementation, requiring less change in its components, operations, and lines of code.

[Apel and Batory, 2006] present a study comparing the feature-oriented programming (FOP) and AOP mechanisms to implement features of a product line. The SPL implementation used AML (Aspectual Mixin Layers), which is an approach that integrates FOP and AOP. The metrics used in their study were only: lines of code and number of components (classes, mixins and aspects). They showed the utility of FOP to address the inclusion of new classes and members in existing classes; and the use of aspects to modularize some crosscutting characteristics. In our study, we used a more extensive set of metrics and focused explicitly on the evaluation of the MAS-PL modularity and stability. Besides, our study discussed some relevant characteristics of AOP to implement agency features.

[Kastner et al., 2007] present a case study on refactoring a legacy application into a SPL using aspects to implement features. Their case study was the Berkeley DB database system. The goal of their work was to implement features using AspectJ in order to show the suitability of this language for this purpose. As a general result, they observed a strong coupling between classes and aspects that makes the maintenance and evolution of the SPL difficult. In our study, the AO modularization of features contributed to reduce the coupling average per component of the SPL compared to the OO implementation, but on the other hand it has created a set of new coupling dependencies between the aspects and the SPL core (classes). These new coupling relationships between aspects and classes were not affected by the subsequent releases of the AO MAS-PLs, thus guaranteeing its stability.

[Garcia et al., 2003] present an experiment that makes use of two different OO techniques for MAS development. The techniques used were: aspect and pattern-based implementations. Their study was based on a suite of modularity attributes in order to evaluate the reuse and maintainability of some crosscutting concerns of MAS, such as: mobility, learning, autonomy. They concluded that AOP is appropriate to improve separation of MAS concerns, resulting in less components and lines of code, and lower cohesion and coupling. Different from that experiment, our study concentrated mainly on the modularization of agent features and their respective roles, instead of particular agent internal properties (mobility, learning, autonomy). Similar to that study, our AO MAS-PL implementation have presented better results for the separation of the features (SoC metrics).

[Lobato et al., 2008] assess four evolution requirements of a code mobility agent framework, called MobiGrid. Their work assesses quantitatively and qualitatively the positive and negative impacts of AOP on a number of widely-scoped framework modifications. Their study showed that AO improved the modularity and stability of crosscutting mobility concerns in the MobiGrid compared to OO techniques. In our study, we did not analyze mobility concerns, but we also found that AO was more appropriate to implement modularized agents in a MAS-PL.

9 Conclusions

Assessing the quality of software has been one the main concerns of software engineers. The assessment of the software internal quality through suites of modularity and change impact metrics is an important existing mechanism to improve the quality of software by making possible to discover modularity problems related to the inadequate implementation of features. In this paper, we presented a quantitative and qualitative study of the design modularity and stability of the incremental development of a multi-agent system product line (MAS-PL). We compared two different versions of the MAS-PL based on JADE platform, implemented using the following technologies: (i) one implementation in Java language with conditional compilation support; and (ii) the other one implemented with the AspectJ language. The MAS-PL was originated from a traditional web-based system that was extended to incorporate autonomous or pro-active behavior. We initially developed a traditional web-based system to support the conference management process, named Expert Committee. Subsequently, we evolved this system to incorporate a series of change scenarios (agency features) in the EC MAS-PL.

Our empirical study consisted on applying a suite of modularity and change impact metrics to the different implementations of the EC MAS-PLs. The collected results for these different metrics along the different releases have shown the following general conclusions: (i) the MAS-PL features tended to be more scattered and less tangled in the AO implementation compared to the OO conditional compilation-based solution. It means that several aspects were required to implement the agent features, but on the other hand, they have successfully modularized thus avoiding the feature interlacing; (ii) the AO implementation also presented better results in terms of stability, because the codification of new aspects demanded less changes to the existing components (classes and aspects) and operations from the MAS-PL; (iii) the modularization of agent features in different aspects brought as consequence the increase in the number of components, operations and lines of code, bringing complexity to manage these new aspects; and, finally, (iv) regarding the average coupling and cohesion metrics per component, both AO and OO solutions presented relative stable values, with a slight advantage for the AO implementation.

References

[Alur et al., 2001] Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In Proceedings of the 5th international conference on Generative programming and component engineering (GPCE '06). ACM Press, 201-210, New York, USA. October 2006.

[Alves et al., 2005] Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In Proceedings of the 9th International Conference of Software Product Lines (SPLC'05), LNCS 3714, Springer-Verlag, 70-81, September 2005.

[Antenna, 2008] Antenna Preprocessor**.** http://antenna.sourceforge.net/wtkpreprocess.php

[Apel and Batory, 2006] Apel, S., Batory, D.: When to use features and aspects?: a case study. In Proceedings of the 5th international Conference on Generative Programming and Component Engineering (GPCE '06). ACM, New York, NY, 59-68. October, 2006.

[Basili and Rombach, 1988] Basili, V., Rombach, H.: The TAME project: towards improvement-oriented software environments. IEEE Transactions on Software Engineering, 14(6), 1988, 728-738.

[Bäumer et al., 19997] Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The Role Object Pattern. Washington University Dept. of Computer Science, 1997.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.

[Cirilo et al., 2008] Cirilo, E., Kulesza, U., Lucena, C.: A Product Derivation Tool Based on Model-Driven Techniques and Annotations. Special Issue on "Software Components, Architectures and Reuse". JUCS (Online), v. 14, n. 8, 1344-1367, 2008.

[Clements and Northrop, 2001] Clements, P. and Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, USA, 2001.

[Colyer et al., 2004] Colyer, A., Rashid, A., Blair, G.: On the separation of concerns in program families. Technical report, Computing Department, Lancaster University, 2004.

[Conejero et al., 2009] Conejero, J., Figueiredo, E., Garcia, A., Hernandez, J., Jurado, E.: Early Crosscutting Metrics as Predictors of Software Instability. In 47th International Conference Objects, Models, Components, Patterns (TOOLS), 2009. (to appear).

[Czarnecki and Eisenecker, 2000] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[Eclipse, 2008] Eclipse Metrics Plugin. http://eclipse-metrics.sourceforge.net/.

[Eaddy et al., 2008] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., Aho, A. V.: Do Crosscutting Concerns Cause Defects?. IEEE Transactions on Software Engineering. 34, 4, 497-515. July, 2008.

[Fayad et al., 1999] Fayad, M., Schmidt, D., and Johnson, R. (1999). Building application frameworks: object-oriented foundations of framework design. John Wiley & Sons, Inc., New York, NY, USA.

[Figueiredo et al., 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F. C., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In Proceedings of the 30th International Conference on Software Engineering (ICSE '08), 261-270, New York, NY, USA. ACM. May, 2008.

[Filho et al., 2006] Filho, F. C., Cacho, N., Figueiredo, E., Raquel Maranhão., Garcia, A., Rubira, C. M. F.: Exceptions and aspects: the devil is in the details. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14), 152-162, New York, NY, USA. ACM. November, 2006.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.

[Garcia et al., 2003] Garcia, A., Sant'Anna, C., Chavez, C., da Silva, V. T., de Lucena, C. J. P., von Staa, A.: Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems. In ACM International Conference on Software Engineering, Proceedings on 2nd International Workshop on of Software Engineering for Large-scale Multi-Agent Systems (SELMAS'03), 19-34, Portland,Oregon,USA, 2003.

[Garcia et al., 2005] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing design patterns with aspects: a quantitative study. In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05), 3-14, New York, NY, USA. ACM Press. March, 2005.

[Greenwood et al., 2007] Greenwood, P., Bartolomei, T., Figueiredo, E., Garcia, A., Cacho, N., Sant'Anna, C., Borba, P., Kulesza, U., Rashid, A.: On the impact of aspectual decompositions on design stability: An empirical study. In Proceedings of European Conference on Object-Oriented Programming (ECOOP'07), LNCS, 176-200. Springer-Verlag. August, 2007.

[Griss, 2000] Griss, M. L.: Implementing product-line features by composing aspects. In Proceedings of the first conference on Software Product Lines: Experience and Research Directions, 271-288, Norwell, MA, USA. Kluwer Academic Publishers, 2000.

[Hannemann and Kiczales, 2002] Hannemann, J. Kiczales, G.: Design pattern implementation in Java and aspectJ. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, Systems, Languages, and Applications (OOPSLA '02), 161-173, New York, NY, USA. ACM. November, 2002.

[Hunleth and Cytron, 2002] Hunleth, F. Cytron, R. K.: Footprint and feature management using aspect-oriented programming techniques. In Proceedings of the Joint Conference on Languages, Compilers and Tools For Embedded Systems: Software and Compilers for Embedded Systems. ACM, New York, NY, 38-45. June, 2002.

[JADE, 2008] JAVA Agent Develoment Framework. http://jade.tilab.com/.

[Jadex, 2008] Jadex BDI Agent System.
http://vsis-www.informatik.uni-hamburg.de/projects/jadex/.

[Jennings, 2001] Jennings, N. R.: An agent-based approach for building complex software systems. Commun. ACM, 44(4):35-41. 2001.

[Kastner et al., 2007] Kastner, C., Apel, S., Batory, D.: A case study implementing features using aspectj. In Proceedings of the 11th International Software Product Line Conference (SPLC '07), 223-232, Washington, DC, USA. IEEE Computer Society. September, 2007.

[Kendall, 1999] Kendall, E.: Role Model Designs and Implementations with Aspect-oriented Programming. In Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented programming, Systems, Languages, and Applications (OOPSLA '02), New York, NY, 353-369. November, 1999.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J.: Aspect-Oriented Programming. In Proceedings European Conference on Object-Oriented Programming (ECOOP'97), V.1241, 220-242, Berlin, Heidelberg, and New York. Springer-Verlag, 1997.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting Started with AspectJ, Communication ACM, 44, 59-65, 2001.

[Kulesza et al., 2006a] Kulesza, U., Alves, V., Garcia, A. F., de Lucena, C. J. P., Borba, P.: Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In Proceedings of the 9th International Conference on Software Reuse (ICSR'06), 231-245, Torino, 2006.

[Kulesza et al., 2006b] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., Lucena, C.: Quantifying the effects of aspect-oriented programming: A maintenance study. In Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06), 223-233, Washington, DC, USA. IEEE Computer Society. September, 2006.

[Krueger, 2002] Krueger, C. W.: Easing the Transition to Software Mass Customization. In 4th International Workshop on Software Product-Family Engineering. F. v. Linden, Ed. Lecture Notes in Computer Science, vol. 2290. Springer-Verlag, London, 282-293. October, 2003.

[Lee et al., 2006] Lee, K., Kang, K. C., Kim, M., Park, S.: Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In Proceedings of the 10th international on Software Product Line Conference (SPLC'06). IEEE Computer Society, Washington, DC, 103-112. August, 2006.

[Lobato et al., 2008] Lobato, C., Garcia, A., Kulesza, U., Staa, A. v., and Lucena, C.: Evolving and Composing Frameworks with Aspects: The MobiGrid Case. In Proceedings of the Seventh international Conference on Composition-Based Software Systems (ICCBSS 2008) – V (00). IEEE Computer Society, Washington, DC, 53-62. February, 2008.

[Meyer, 1998] Meyer, B.: Object Oriented Software Construction, 1st ed. Prentice-Hall, Englewoood Cliffs, 1998.

[Nunes et al., 2008a] Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Developing and evolving a multi-agent system product line: An exploratory study. In Luck, M. and Gomez-Sanz, J. J., editors, Agent-Oriented Software Engineering IX – LNCS, V. 5386, 228-242.Spring-Verlag, 2008.

[Nunes et al., 2008b] Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I., Lucena, C.: On the Modularity Assessment of Aspect-Oriented Multi-agent Systems Product Lines: A Quantitative Study. In Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08). Porto Alegre, Brazil, 122-135. August, 2008.

[Nunes et al., 2008c] Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I., Garcia, A, Lucena, C.: Comparing Stability of Implementation Techniques for Multi-Agent System Product Lines. In 3th European Conference on Software Maintenance and Reengineering (CSMR'09). Kaiserslautern, Germany, 229-232. March, 2009.

[Nunes et al., 2008d] Nunes, I., Kulesza, U., Nunes, C., Cirilo, E, Lucena, C.: Extending Web-Based Applications to Incorporate Autonomous Behaviour. In: Brazilian Symposium on Multimedia and the Web Systems (WebMedia'08). Vilha Velha, Brazil, 115-122. October, 2008.

[Pena et al., 2006a] Pena, J., Hinchey, M. G., Resinas, M., Sterritt, R., Rash, J. L.: Managing the Evolution of an Enterprise Architecture Using a MAS-Product-Line Approach. In Arabnia, H. R. and Reza, H., editors, Software Engineering Research and Practice, 995-1001. CSREA Press, 2006.

[Pena et al., 2006b] Pena, J., Hinchey, M. G., Ruiz-Cortés, A., Trinidad, P.: Building the core architecture of a multiagent system product line: with an example from a future nasa mission. In 7th International Workshop on Agent Oriented Software Engineering (AOSE'06). LNCS, 2006.

[Pohl et al., 2005] Pohl, K., Böckle, G., van der Linden, F. J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, New York, USA, 2005.

[Sant'Anna et al., 2003] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In XVII Brazilian Symposium on Software Engineering (SBES'03), 19-34, Manaus, Brazil, 2003.

[Sant'Anna et al., 2007] Sant'Anna, C., Figueiredo, E., Garcia, A. F., Lucena, C. J. P.: On the modularity of software architectures: A concern driven measurement framework. In Software Architecture, First European Conference (ECSA'07), volume 4758, 207-224. September, 2007.

[Sant'Anna et al., 2008] Sant'Anna, C., Garcia, A., Lucena, C.: Evaluating the Efficacy of Concern-Driven Metrics: A Comparative Study. In Proceedings of the 2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM'08), 25-30, Nashville, 2008.

[Soares et al., 2006] Soares, S., Borba, P., Laureano, E.: Distribution and persistence as aspects. Software Practices Experience, 36(7):711-759. 2006.

[Together, 2008] Borland Together. http://www.borland.com/us/products/together/index.html. Borland 1994 - 2008. Borland Software Corporation

[Voelter and Groher, 2007] Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In Proceedings of the 11th international Software Product Line Conference (SPLC'07). IEEE Computer Society, Washington, DC, 233-242. September, 2007.

[Wooldridge and Ciancarini, 2000] Wooldridge, M. Ciancarini, P.: Agent-Oriented Software Engineering: The State of the Art. First International Workshop on Agent-Oriented Software Engineering (AOSE'00), V. 1957, 1-28. Springer-Verlag, Berlin, 2000.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers.

[Yau and Collofello, 1985] Yau, S. S., Collofello, J. S.: Design Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering, 11(9), 849-856 (1985).