Extending Web-Based Applications to Incorporate Autonomous Behavior

Ingrid O. Nunes
PUC-Rio, LES
Rio de Janeiro, Brazil
ioliveira@inf.puc-rio.br

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

Camila Nunes, Elder Cirilo, Carlos Lucena
PUC-Rio, LES
Rio de Janeiro, Brazil
{cnunes,ecirilo,lucena}@inf.puc-rio.br

ABSTRACT
Web applications are popular nowadays due to the ubiquity of the client and also because user experience is becoming each time more interactive. However, several tasks of these applications can be automated. Agent-oriented software engineering has emerged as a new software engineering paradigm to allow the development of applications that present autonomous behavior. In this work, we present two case studies of web-based systems, on which we added autonomous behavior by means of software agents. We also discuss some design and implementation issues found on the development of those systems and propose an architectural pattern as a consequence of our case studies.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: Software Architectures; I.2.11.d [**Artificial Intelligence**]: Multiagent systems

General Terms
Web system, Agents

Keywords
Web Application, Autonomous Behavior, Multi-agent Systems, Architectural Pattern

1. INTRODUCTION
The World Wide Web has begun providing navigation through static web pages. However, now navigation can also provide an interactive experience, presenting pages with dynamic content, which are generated according to information submitted by users or based on their preferences. Over the last years, several technologies have emerged to make the user interaction with the web more interactive and to produce richer web pages. This is accomplished by code running on the client-side, written in scripting languages, such as JavaScript or ActionScript, as also on the server-side, written in languages/frameworks, such as JSP, ASP and PHP, which allow the development of web applications that can process information received from users and give a specific response for them. Moreover, several new technologies came out improving even more the user experience, such as AJAX, which is a group of inter-related web development techniques used for creating interactive web applications and avoiding unsightly complete-page reloads, and Flex, an open source framework for building and maintaining expressive web applications. All of these new technologies and characteristics of web applications make them very popular. Another reason for their popularity is that web applications can be updated and maintained without distributing and installing software on potentially thousands of client computers.

On the other hand, over the last years, agent-oriented software engineering has emerged as a new software engineering paradigm to allow the development of distributed complex applications that are characterized by a system composed of many interrelated sub-systems [11]. A software agent is an abstraction that enjoys mainly the following properties [19]: autonomy, reactivity, pro-activeness and social ability. Current web applications have several tasks that need human interaction to be executed. The addition of autonomous behavior to these web applications can bring a lot of advantages for them, providing the automation (or semi-automation) of tasks that need human intervention and the incorporation of intelligent services. There are plenty of examples that illustrate this, such as: (i) recording metrics data as the user interacts with an application. The recorded data can be sent to a database for future analysis of user interaction patterns [1]; (ii) sending emails to inactive users [3]; and (iii) automatically suggesting sales on e-commerce systems based on the products that users usually buy. Software agents are a strong candidate to provide these new kinds of autonomous behavior required by web applications.

Nevertheless, the possibility of redesigning existing systems as multi-agent systems or making big changes on their architecture is impracticable. Some works [18, 3] have proposed architectures to incorporate software agents into web applications architecture. However, these approaches present some deficiencies, such as: (i) they propose a complex solution for the problem, which requires the understanding of new concepts and techniques; or (ii) these approaches do not provide an effective integration between the agents and the application. In our work, we aim at proposing a simple and efficient way to integrate agents and existing web applications in order to provide autonomous behavior for them.

In this work, we present an exploratory study of incorporating autonomous behavior into web based applications. We have developed two case studies in which traditional web-based systems structured according to the Layer architectural pattern [8] are extended to adequate new autonomous functionalities to their architecture by means of the agent technology. The first one is the ExpertCommittee, a conference management system for the web domain. The second

one is the OLIS, a system that provides different services to the user, such as calendar and events announcement. We then discuss several challenges we faced during the addition of software agents to the web applications, and, as a consequence of the development experience of our case studies, we also propose an architectural pattern to integrate web applications and software agents.

The remainder of this paper is organized as follows. In Section 2, we present the two case studies that guided our research. Section 3 discusses design and implementation issues found when integrating software agents into web applications. The architectural pattern that emerged from our case studies is also presented in this section. Some works related to the integration between web applications and software agents are described in Section 4. Finally, the conclusions and directions for future works are presented in Section 5.

2. EXTENDING WEB-BASED SYSTEMS: AN EXPLORATORY STUDY

Our research started with the development of two case studies to investigate the challenges related to the addition of autonomous behavior into web-based applications by means of agent technology. Both case studies are presented in this section. Each of them represents a traditional web system structured in layers, in which specific autonomous behavior was added with the aim of automate (or semi-automate) existing system functionalities. Section 2.1 presents the ExpertCommittee, a conference management system, which was evolved to automate some user tasks. Section 2.2 details the other case study, the OLIS, a web application that provides some services to the user, such as calendar, on which we added new autonomous behavior features, e.g. automatic detection of event conflicts.

2.1 ExpertCommiittee Case Study

The ExpertCommittee (EC) [14, 15] is a conference management system for the web domain developed to support the paper submission and reviewing processes from conferences. The EC system provides functionalities to support the complete process of conference management, such as: (i) create conferences; (ii) define conference basic data, committee members, 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) delegate the paper review to an additional reviewer; (viii) review paper; (ix) accept / reject paper; (x) notify authors about the paper review; and (xi) submit camera ready. Each of these functionalities can be executed by an appropriate user type of the system, such as, conference chair, program committee members, authors and reviewers.

The EC web-based system was structured according to the Layer architectural pattern [2, 8] and is composed of the following components/layers: (i) `GUI` - this layer is responsible to process the web requests submitted by the system users. It was implemented using the Struts[1] framework; (ii) `Business` - is responsible to structure and organize the business services provided by the EC system. The transaction management of the business services was implemented using the mechanisms provided by the Spring[2] framework; and (iii) `Data` - aggregates the classes of database access of the system, which was implemented using the Data Access Object (DAO) design pattern. The Hibernate[3] framework was used to make persistent the objects in a MySQL[4] database.

Figure 1 illustrates the architecture of the EC web-based system and highlights the base architecture. This implementation corresponds to the first version of the EC. It already provides all the functionalities necessary to manage the conference process. However, there are some tasks in the EC that could be automated, such as automatically distributing the submitted papers of a conference to the committee members, instead of the chair making it manually. Thus, we evolved the system, adding autonomous behavior to it. We consider autonomous behavior actions that the system automatically performs and previously needed human intervention. The introduction of autonomous behavior in the original system was accomplished using multi-agent system technology, such as agents, roles and their associate behaviors. Next section details how the software agents were introduced in the ExpertCommittee core architecture.

2.1.1 Automating User Tasks in EC

The EC system was evolved to incorporate new functionalities that reduce the user effort to manage the conference. These functionalities are: (i) *automatic paper distribution* - papers are automatically distributed among the committee members to be reviewed, according to some predefined rules; (ii) *task management* - the system controls the pending tasks of the user, and the ones that he/she already done; (iii) *deadline monitoring* - the conference deadlines are monitored, and the system takes the appropriate actions when the deadline expires or is about to expire; (iv) *user notifications* - the system sends notifications to the user about the conference status; (v) *conference suggestion* - the authors of submitted papers receive suggestions of conferences that are related to the conference that they submitted the paper.

Software agents were implemented into the EC architecture to address these new autonomous behavior functionalities. The architecture of the new version of the EC is illustrated in Figure 1. The JADE[5] framework was used as the base platform to implement our agents. These agents are responsible for monitoring the execution of different functionalities of the EC in order to trigger new actions related to the autonomous behavior functionalities from the system. The integration between the original web based system and these new agents comprises a multi-agent system. Details about each agent that are part of the system are listed below:

Environment Agent. This agent monitors the EC system by observing the execution of specific business services. These monitored events of the EC system represent the environment in which the user agents are situated. Each user agent is specified to perceive changes in the environment and make actions according to them. When the environment agent is initialized, it registers itself as an observer of

[1]http://struts.apache.org

[2]http://www.springframework.org/.

[3]http://www.hibernate.org/.

[4]http://www.mysql.org/.

[5]http://jade.tilab.com/.

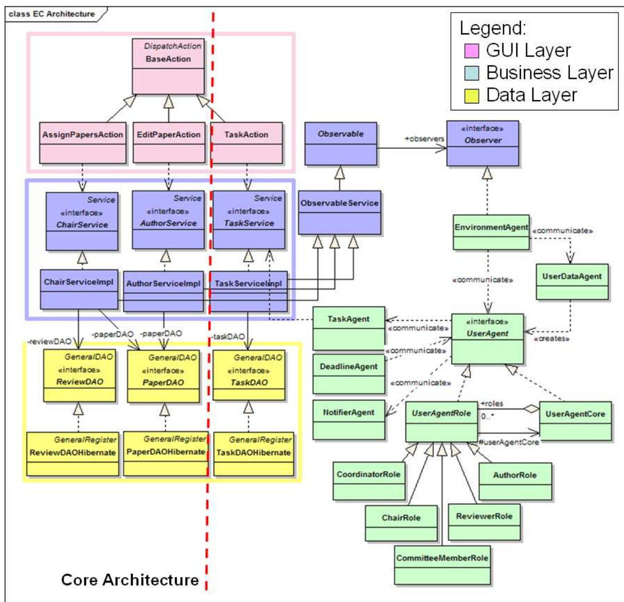

Figure 1: The ExpertCommittee Architecture.

the services that compose the `Business` layer. These services are observable objects that allow the observation of their actions. That means that, for each call of the system business methods, the services not only execute the requested methods, but they also notify their respective observers. The only observer in our implementation is the `EnvironmentAgent`, whose aim is to notify the other agents of the system about the changes;

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. Each agent role performs specific actions according to the role that the user plays in the conference, such as chair, coordinator and author. 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 notifications.

Deadline Agent. This agent is responsible for monitoring the conference deadlines. This monitoring serves basically 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. It also triggers some actions in the user agents, by sending messages informing them about the deadlines that expired.

Task Agent. This agent is responsible for managing the user tasks. It receives requests for creating, removing and setting the execution date of tasks. The requests are made by the user agents.

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.

Thus, the integration between the web architecture and the agents in the EC system was accomplished by introducing an agent called `Environment Agent`. This agent receives notifications of the business layer about the relevant operations executed by system users and broadcasts them to the other agents. Then, the agents change their behavior according to the information that they received and was interesting for them. The Observer pattern [9] was used to keep the `Environment Agent` and the business services loosely coupled. A class called `ObservableService` extends the `Observer` class (see Figure 1). All the services that compose the business layer extend this class, which provides some common methods to all of them, besides inheriting the methods that are part of the Observer pattern.

2.2 OLIS Case Study

The OLIS (**O**n**L**ine **I**ntelligent **S**ervices) case study is a web application that can provide several personal services to users. The first version of the system is composed mainly by two services: the Events Announcement and the Calendar Services. However, the OLIS was designed in such a way that the system can be evolved to incorporate new services without interfere the existing ones. The system has different flavors according to the type of event that it manages: generic events, academic events and travel events. In this paper, we detail the OLIS system version with travel events.

The Events Announcement service allows the user to announce events to other system users through an events board. The events have some common basic attributes, such as - subject, description, location, city, start and end dates, frequency that it happens - and some specific attributes, which are, in travel events, the place type and the activities that can be done in the event. The Calendar service lets the user to schedule events in his/her calendar. Besides the information of the events published in the events board, calendar events have a list of users that participate of it. Announced events can be imported to the users' calendar.

Figure 2 presents the OLIS architecture, with a dashed line delimitating which components belong to the core architecture. It can be noticed that it is very similar to the EC core architecture. The OLIS web-based system was also structured according to the Layer architectural pattern [8], which is the pattern usually used to structure web applications. The layers that compose the architecture are exactly the same of the EC - `GUI`, `Business` and `Data` layers. The responsibilities attributed to each one of the layers were also the same. The only difference is that we used different web application frameworks in the GUI Layer, in EC we used the first version of the Struts framework, and Struts 2 in OLIS.

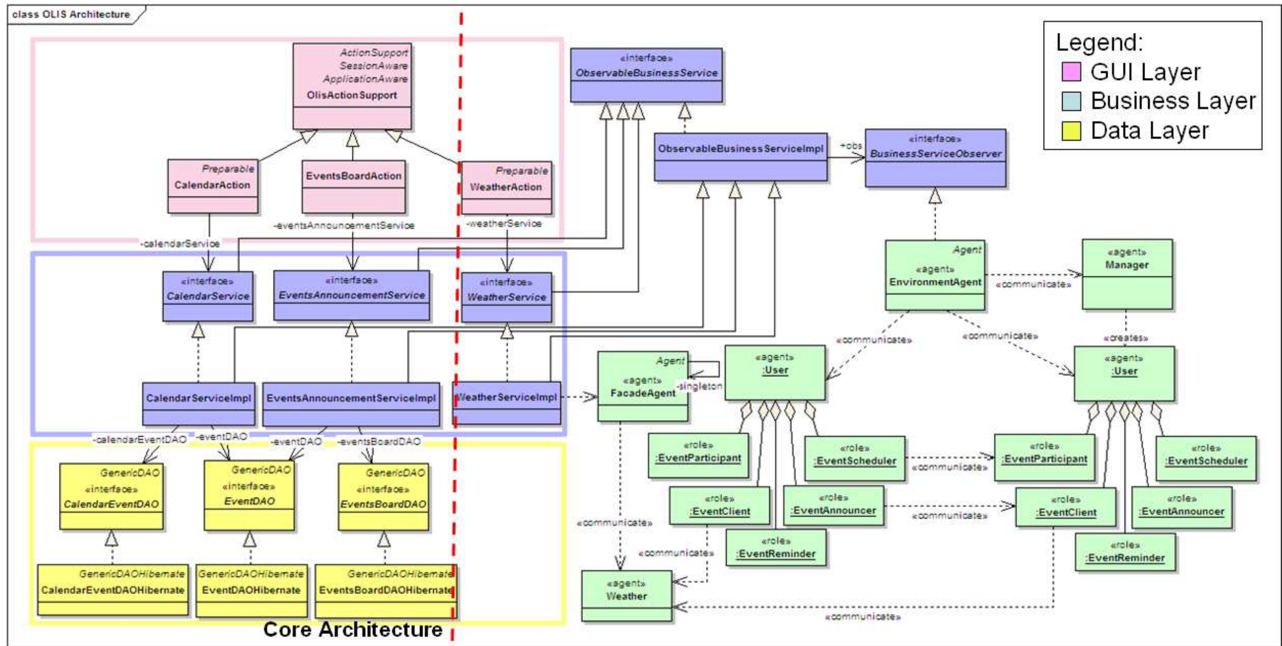

Figure 2: The OLIS Architecture.

After developing the first version of OLIS web application, we have identified that new autonomous behavior features could be introduced to automatize some tasks in the system. An example is to store user preferences about travel events and then automatically suggest users about new events announced, according to their preferences. Another improvement in the application is the addition of new features that retrieve information processed by software agents, such as weather information. So, we evolved the OLIS application, adding new features to it, which take advantage of the agents technology. We present the features that were added to OLIS, and how we developed them in the next section.

2.2.1 *Adding Intelligent Services to OLIS*

The services that compose OLIS application are presented in several applications. They provide the typical operations - create, read, update and delete - to the user manage information in the system. We have evolved these services, aggregating autonomous behavior, to process the data stored in the system, analyze it, take conclusions from it and perform actions that were previously done manually. The services become intelligent services.

The new features incorporated to the OLIS first version are: (i) *Events reminder* - the user configures how many minutes he/she wants to be reminded before the events, and the system sends notifications to the user about events that are about to begin; (ii) *Events scheduler* - when an user adds a new calendar event that involves more participants, the system checks the other participants' schedule to verify if the event conflicts with other events. If so, the system suggests a new date for the calendar event that is appropriate according to the participants schedule; (iii) *Events Suggester* - when a new event is announced, the system automatically recommends the event after checking if it is interesting to the users based on their preferences. The system also checks if the weather is going to be appropriate according to the place type where the event is going to take place; (iv) *Weather* - this is a new user service. It provides information about the current weather conditions and the forecast of a location. This service is also used by the system to recommend announced travel events.

The evolution of the OLIS web application was accomplished by the introduction of software agents and agent roles on the architecture. Figure 2 shows the architecture integrated with the agents. The implementation of this version of the system is a hybrid agent architecture: the Environment agent and the Facade agent were implemented with JADE, and the other agents with Jadex[6]. JADE agents are Java classes that are subclasses of the `Agent` class from the JADE platform. Thus, we could add new methods to the agent classes to provide access to the objects of the system by passing the agent reference. However, Jadex implements the BDI (belief-desire-intention) model, specifying agents in an XML file in terms of their believes, goals and plans, and it provides a reasoning engine, which is necessary for the other agents of the system. Next we describe each one of the OLIS agents:

Environment Agent. This agent receives notifications about the execution of business operations and it propagates them to the other agents of the system. Its behavior is similar to the Environment agent from EC.

Manager Agent. This agent is responsible for creating new user agents when a new user is inserted in the database. It also starts an user agent for each user already stored in the database during the application start up. It is equivalent

[6]http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

to the User Data agent from EC.

Facade Agent. This agent is the access point of the web application to get information from the agents. It hides the other system agents from application, so that the application only needs to know about the Facade agent to get the information from the agents.

Weather Agent. This agent provides the weather information. It looks for the current weather conditions and the forecast of a specific location.

User Agent. Each user of the system has an user agent that represents him/her. Each User agent has five different roles: (i) *Event Reminder Role* - reminds the user about events that are going to begin; (ii) *Event Scheduler Role* - invites other users to a calendar event and finds a time for the event that is compatible with the participants' schedule; (iii) *Event Participant Role* - accepts or rejects an invitation to participate of an event and, in case of reject, provides a time that is appropriate for the user according to his/her schedule; (iv) *Event Announcer Role* - announces new events to the other user agents; (v) *Event Client Role* - checks if the event announced is interesting according to the user preferences. This role also checks if the weather will be good according to the place type, consulting the Weather agent. Eventually, the user agents can access the business services to perform changes in the data model.

3. DESIGN AND IMPLEMENTATION ISSUES

During the development of our case studies, we identified some design and implementation issues when integrating the existing web application and the software agents. In this section, we present and discuss some relevant questions about this integration and we also propose an architectural pattern for incorporating autonomous behavior into web applications. We aimed at proposing a solution that: (i) is quite simple to be implemented; and (ii) have a minimum impact in design and implementation of existing web systems.

3.1 Integrating Agents Into Web-Based Applications

Software agents are situated in an environment, in which they are able to perceive its events in order to take actions in a timely fashion according to changes that occur in the environment [19]. When extending the web systems to incorporate the autonomous behavior, the first problem that we had to solve was how the software agents could perceive changes in the environment. We consider our environment the data model with its current data information. Changes on this model happen as a consequence of the user interactions with the system. So each time the user performs an action that changes the data model, the agents should detect it or be notified about this fact, and then they take the appropriate actions. In BDI agents, we can think the data model, or a part of it, as the believes of the agents, and the agents should perceive when their believes change.

A possible solution to this problem is that can consult the database in periodic times, as it is proposed in [3]. However, this solution can cause a big overhead in the system if it is done in a short period, or changes will be perceived with a large delay if it is done in a large period, which can cause undesired situations, such as missing a change if more than one change happen in the same data. Therefore, we used the Observer pattern to let the business services notify the agents about operations they execute. This allows the agents process information only when something actually changed in the environment. Besides receiving information, agents must have an access to the business services to perform changes in the data model. This is easy to be done through method invocation to the interfaces provided by the business services.

Another common situation is that system functionalities may retrieve information from the agents. Software agents exchange messages, and objects call methods; so it was a problem for an object from the system to retrieve information from the agents. Usually, the agent platforms do not provide an easy way for objects from the system interacting with agents. For example, in Jadex, the agents are specified in XML files, and an object is not able to access them and does not know their interface to call a specific method. However, this is something desired, because agents can provide information that needs reasoning and learning to be produced, e.g. the weather agent provides information about the weather in OLIS case study. Furthermore, it is common that it takes some time to get information from the agents, as this can require a lot of processing and messages exchanges. Thus, delayed answers should also be considered. In OLIS, we developed the Facade agent to let the objects of the web application access and get information from the agents. The Facade agent was implemented with JADE, so it is was designed as a common Java class that provides an interface to objects access information of interest from the system agents. We provided both synchronous and asynchronous methods; thus the object can make a request providing a callback function and continue its processing, while it is waiting for the answer for its request. The Facade agent is a singleton instance; therefore its reference can also be easily retrieved.

3.2 Improving Modularization using Aspect-Oriented Programming

An important characteristic that was taken into consideration when integrating the web application and the agents was to keep them loosely coupled in order to make possible to easily insert and remove the software agents from the web application. It also helps to incorporate the autonomous behavior (agents), with a minimum impact into an existing web application and it improves the reusability and maintenance of the system.

The implementation of the Environment agent using the Observer pattern allowed the agents perceiving changes in the environment. Moreover, the agents can be easily (un)plugged to the system. We can remove the agents from the application without impacting its normal operation. The Observer pattern provides an abstract coupling between the subject and the observer; the services of the Business layer does not know who the concrete observers are. Furthermore, there are some application program interfaces (APIs), such as the Java API, that already provides the Observable class and the Observer interface.

The implementation of the Observer pattern using aspect-oriented programming could make the addition of agents to the web application even less intrusive. Aspect-oriented programming (AOP) [12] enables to separate code that implements crosscutting concerns and modularize it into aspects. It provides mechanisms and techniques to compose crosscutting behaviors into the desired operations and classes during compile-time and even during execution. The source code for operations and classes can be free of crosscutting concerns and therefore easier to understand and maintain. The code of the Observer pattern presents an invasive nature, resulting on a scattered and tangled code among several classes. Therefore, this leads the pattern to "disappear into the code", and it also bring difficulties to the understanding, maintenance and documentation of the pattern, and consequently of the application.

We are currently exploring the refactoring of some existing autonomous behavior functionalities from the EC and OLIS systems using AOP. In the current perspective, aspects have been useful in the implementation of these systems to not only observe the execution of business services, but also in the modularization of existing agency fine-grained features that are encountered spread and tangled along different classes from Agent layer [14, 15].

3.3 Transaction Management and Performance

Besides the questions already presented, there are two important implementation details that should be mentioned. The first one is the transaction management. In typical web applications, the execution of each business methods is under a transaction. Thus, if an error occurs during the method execution, the operations that were already executed are undone. With transaction management, the information stored in the database cannot achieve an inconsistent state. The design issue is if the notification to the observer should be done inside or outside the transaction scope. If the business methods should be committed even though an error occurs during the notification to the agents, this notification should be out of the transaction or the exception thrown should be caught and treated. If the business method execution must rollback when something wrong happens during the notification to the agents, the notification must be inside the transaction.

Another design and implementation issue is related to the system performance. The inclusion of notifications on the business methods implies an overhead to the system, in particular when an entity is deleted, because its state is read and kept in memory before its deletion. Though, only the methods that impact in the behaviors of the agents should propagate its execution. Usually, methods that only retrieve information from the data model do not need to notify the observers.

3.4 Towards a Web-MAS Architectural Pattern

In this section, we present the Web-MAS architectural pattern. This pattern was derived from our case studies based on the common elements identified when integrating the web based systems and their respective software agents. The proposed pattern provides a general structure to add autonomous behavior to existing web applications using agent technology. This extension has a minimum impact on the architecture of web-based systems. Moreover, the agents can be easily removed after being introduced on the system.

The pattern addresses applications that follow the typical web application architecture, i.e. the Layer architectural pattern [2]. Although, it would also be adapted to consider other alternative implementations of web-based systems. This pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction. The proposed solution is composed of these components: (i) the presentation, business and data layers, which comprise the web application; (ii) the agents layer; (iii) the business layer monitor; (iv) and the agents layer facade. The structure of these components is depicted in the Figure 3. Next we describe each one of these components:

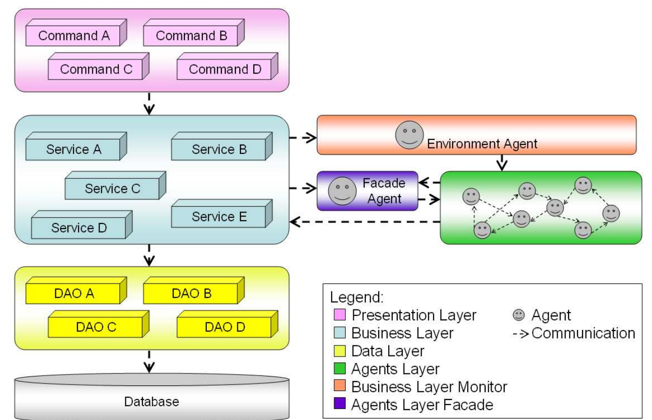

Figure 3: WEB-MAS Architectural Pattern.

Presentation Layer. This layer can also be called Graphical User Interface (GUI) layer. The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand. Usually, this layer follows the model-view-controller (MVC) pattern [8]. This pattern considers three roles: (i) *model* - an object that represents some information about the domain; (ii) *view* - represents the display of the model in the user interface; and (iii) *controller* - takes user input, manipulates the model and causes the view to update appropriately. Commonly, Web Application Frameworks (WAF) are used to implement this layer;

Business Layer. This layer is also known as Logic layer. It coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers. Typically, there is transaction control in this layer;

Data Layer. In this layer, the information is stored and is retrieved from a database or a file system. It is then passed back to the Business layer for processing, and then eventually back to the user. The information is

represented in a data model, on which there are objects and relationships among them;

Agents Layer. This component is responsible for the autonomous behavior *de facto*. It is composed by software agents. The agents provide intelligent services and automate tasks that were previously done directly by users. Instead of being simple objects with attributes and methods, they have believes, goals and plans. The agents receive messages from the Environment agent about the execution processes that this agent detects by monitoring the services of the Business Layer. According to the messages received, the agents take appropriate actions and can also perform changes in the data model by using the business services;

Business Layer Monitor. This component is responsible for monitoring the business operations from the web application. The business operations to be monitored are the ones that are related to the autonomous behavior. The Business Layer Monitor aggregates the Environment agent, which receives notifications about the operations executed in the Business layer and propagates them to the other agents;

Agents Layer Facade. This component is the access point of the web application to the Agents Layer. Besides the information that is stored in the data model, agents can also generate information through some processing and exchanging messages with other agents. Then, this facade provides an interface for the business services get information from the other agents of the system. This component is composed by the Facade agent, which receives a request from a business service, forward it to the appropriate agent and pass the result back to the service. When this agent starts up, it registers itself as a singleton instance. Then the business services can access this agent and make a request. There are three ways of communication: (i) Synchronous - the business service calls the Facade agent and waits for the response; (ii) Asynchronous with pooling - the business service calls the Facade agent, continues its processing, and periodically checks if the response arrived; and (iii) Asynchronous with callback approach - the business service calls the Facade agent, continue its processing, but it is notified through a callback function, which is passed as parameter when the Facade agent was called.

The communication between the business services and the Environment agent is accomplished by means of the introduction of the Observer design pattern [9]. The intent of this pattern is to define a one-to-many dependency between objects so that when one object performs an action or changes state, all its dependents are notified automatically. By the use of this pattern, we keep a loose coupling between the application and the Agents layer. In the Observer pattern, the concrete subject is the object that sends a notification to its observers when its state changes or performs an action; thus all the services that compose the Business layer are concrete subjects. They must implement the `Observable` interface, which allows the observation of their actions. For each call of the business methods, the services not only execute the requested method, but they also notify their respective observers. The concrete observer implements an updating interface to receive notifications from the subject. In our architecture, there is only one concrete observer, which is the Environment agent. This agent registers itself as an observer of the services that compose the Business layer when it is initialized. When some action is performed in the Business layer, the Environment agent is notified about this event and it broadcasts the event to all other agents of the system.

3.5 Automatic Derivation of Applications

Our case studies are composed of (i) a core - that represents the existing web application; and (ii) agency features - which are extensions to the web applications that add new autonomous behavior functionalities to the system. We are currently investigating the different kinds of variability related to the software agents, and how to better modularize them, enabling an automatic product customization. In particular, we are exploring how existing software product lines techniques [14, 15, 13] can: (i) improve the modularization of the different features/services available in the system; and (ii) help the automatic customization of these features/services.

Software Product Lines [7, 17] (SPL) is a new trend in the software reuse [16], which addresses the development of applications that share common functionalities and maintain specific functionalities that vary according to specific systems being considered. We are exploring the use of the SPL technology to better modularize the features and/or services of web applications. This allows an easy customization of different versions of the system. Furthermore, the products can be automatically derived by means of model-based tools: software factories [10], GenArch [5, 4, 6], pure::variants[7].

4. RELATED WORK

There are some approaches in the literature that address the challenge of adding software agents on existing web applications. Stroulia & Hatch [18] propose a software framework called TaMeX, which supports the development of intelligent multi-agent applications that integrate existing web-based applications offering related services in a common domain. The architecture of the framework is based on an extensible integration-specification language and a run-time environment consisting of reflective intelligent agents able to interpret and execute integration specifications, defined in the language. The TaMeX architecture is a distributed multi-agent architecture consisting of two types of agents: task agents, which are responsible for interacting with the end users; and application wrappers, which are responsible for executing existing web applications and translating between the domain model of the integrated application and the individual domain models of the wrapped applications. Our work also proposes the addition of software agents on existing web applications. However, we aimed at proposing a simple solution for that founded on existing object-oriented design techniques and that integrates with current adopted web technologies. On the other hand, the use of the TaMeX framework implies learning of a new programming language

[7]http://www.pure-systems.com/

and specific methods. Besides, it obligates the use of its language to implement the software agents.

Choy et al propose in [3] the use of software agents to make the communication in a distance learning community more effective. They focus on the communication between teachers and students. Their software agents are designed to work on behalf of teachers, assisting them in communicating more effectively and closely with students, saving lot of time by delegating routine jobs. Basically, the software agents monitor the system and send alert e-mails in specific situations. The main elements that were introduced in the web application are: (i) Schedule Control Class, which controls the agent's life and behavior; (ii) Job Listener Classes that generate notifications when exceptional events occur; (iii) Data Retrieval Class, which retrieves all required information at once for the Job Central Processing Class; and (iv) Job Central Processing Class, which hosts the criteria to generate e-mail alerts and sends them. This work does not actually integrate agents into the web application. The agents run in a parallel way with the system and consult the same database that is manipulated by the web application. Agents do not communicate with the rest of the system. Moreover, it does not allow the web application to access information directly from the agents. Both situations are addressed by our work.

5. CONCLUSIONS

This paper presented an exploratory study about the incorporation of autonomous behavior into existing web applications. This addition is achieved by the introduction of software agents in existing web applications. We presented two case studies that guided our research. The first case study was the ExpertCommittee, a conference management system that provides the functionalities to help users with the paper submission and reviewing processes. The second case study, OLIS, is a system the provides different services to the user, such as a calendar service. Both case studies are typical web applications that we evolved to incorporate new features, which automate tasks that previously needed user information and generate information that needs a reasoning engine.

The paper also showed the issues that we found during the case studies development: (i) how the software agents percept changes in the data model; and (ii) how the business service can retrieve information from the agents. It was presented a preliminary version of an architectural pattern derived from our case studies to address these design issues. The pattern proposes the following main elements: (i) a Business Layer Monitor, presented by the Environment agent, which is an observer of the business services that notifies system agents about changes in the system; and (i) the Agents Layer Facade, composed by the Facade agent, which is the web application access point to get information from the other agents. Following the pattern guidelines, a web system is structured in such a way that agents can be easily inserted and removed from the system.

We are currently working in the development of other case studies to validate the Web-MAS architectural pattern and check if our pattern is sufficiently generic to be applied to other systems. We are also implementing our case studies using aspect-oriented programming, to allow the (un)pluggability of the agents from the web applications.

6. REFERENCES

[1] Adobe. Adobe - flex 3, 2008. http://www.adobe.com/products/flex/.

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

[3] S.-O. Choy, S.-C. Ng, and Y.-C. Tsang. Building software agents to assist teaching in distance learning environments. In *ICALT '05*, pages 230–232, Washington, DC, USA, 2005. IEEE Computer Society.

[4] E. Cirilo, U. Kulesza, R. Coelho, C. Lucena, and A. von Staa. Integrating Component and Product Lines Technologies. In *ICSR 2008*, China, 2008.

[5] E. Cirilo, U. Kulesza, and C. Lucena. GenArch: A Model-Based Product Derivation Tool. In *SBCARS 2007*, pages 17–24, Campinas, Brazil, Agosto 2007.

[6] E. Cirilo, U. Kulesza, and C. Lucena. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. *Journal of Universal Computer Science*, 14:1344–1367, 2008.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2002.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

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

[10] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley and Sons, 2004.

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

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, June 1997. Springer-Verlag.

[13] C. Nunes, U. Kulesza, C. Sant'Anna, I. Nunes, and C. Lucena. On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study. In *SBCARS 2008*, Porto Alegre, Brazil, 2008.

[14] I. Nunes, C. Nunes, U. Kulesza, and C. Lucena. Developing and evolving a multi-agent system product line: An exploratory study. In *AOSE 2008*, Estoril, Portugal, 2008.

[15] I. Nunes, C. Nunes, U. Kulesza, and C. Lucena. Documenting and modeling multi-agent systems product lines. In *SEKE 2008*, Redwood City, San Francisco Bay, USA, 2008.

[16] D. L. Parnas. On the design and development of program families. pages 193–213, 2001.

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

[18] E. Stroulia and M. P. Hatch. An intelligent-agent architecture for flexible service integration on the web. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 33(4):468–479, 2003.

[19] M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In *AOSE 2000*, volume 1957, pages 1–28. 2000.