5
Lessons Learned

In this chapter, we present and discuss some lessons learned from our experience of development and evolution of the two case studies presented in the previous chapter, the EC and OLIS MAS-PLs. Our lessons learned are divided into two parts: the first one (Section 5.1) describes design and implementation guidelines for developing MAS-PLs, and the second details limitations of our approach (5.2).

5.1
Design and Implementation Guidelines

The implementation of modularized features is a very important issue in the SPL development in order to provide an architecture that supports the variability. In addition, some techniques can be used to improve this modularization allowing a better evolution of the SPL and a (semi-)automatic product derivation. In this section we present guidelines to design and implement agent features. These guidelines resulted from our experience while developing MAS-PLs. They are mainly related to: (i) how to structure an architecture that allows an integration of typical web applications architectures and agents in a decoupled way (Section 5.1.1); (ii) how to implement agent roles in some agent platforms that do not provide this concept (Section 5.1.2); and (iii) how to improve modularization using Aspect-oriented Programming (Section 5.1.3).

5.1.1
Web-MAS Architectural Pattern

In this section, we present the Web-MAS architectural pattern (Nunes 2008b). This pattern was derived from our MAS-PL case studies (Chapter 4) 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, allowing an (un)plugability of agent features.

When extending the web systems to incorporate the autonomous behavior, we have identified mainly two problems. The first problem is how software agents can perceive changes in the environment. We consider the 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 beliefs of the agents, and the agents should perceive when their beliefs change. A possible solution to this problem is that agents can query the database in periodic times, as it is proposed in (Choy 2005). However, this solution can cause a big overhead in the system if it is done in short intervals, or changes will be perceived with a large delay if it is done in long intervals, which can cause undesired situations, such as missing a change if more than one change happen in the same data.

The second identified problem is that system functionalities may retrieve information from the agents. However, software agents exchange messages, and objects call methods; so it is 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 to interact 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. Nevertheless, this is something desired, because agents can provide information that needs reasoning and learning to be produced. 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.

The Web-MAS architectural pattern was proposed to solve both problems. The pattern addresses applications that follow the typical web application architecture, i.e. the Layer architectural pattern (Buschmann 1996). However, 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 the following 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 5.1. Next we describe each one of these components:

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 MVC pattern (Fowler 2002). 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, WAFs 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 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, in which there are objects and relationships among them;

Agents Layer. This component is responsible for the autonomous behavior *de facto*. It is composed of 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 beliefs, 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 of 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 to 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 requests. 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 (Gamma 1995). 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.

An important implementation detail related to the Business layer is the transaction management. In typical web applications, the execution of each business method 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 implementation issue is whether if the notification to the observers 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 outside 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 their execution. Usually, methods that only retrieve information from the data model do not need to notify the observers.

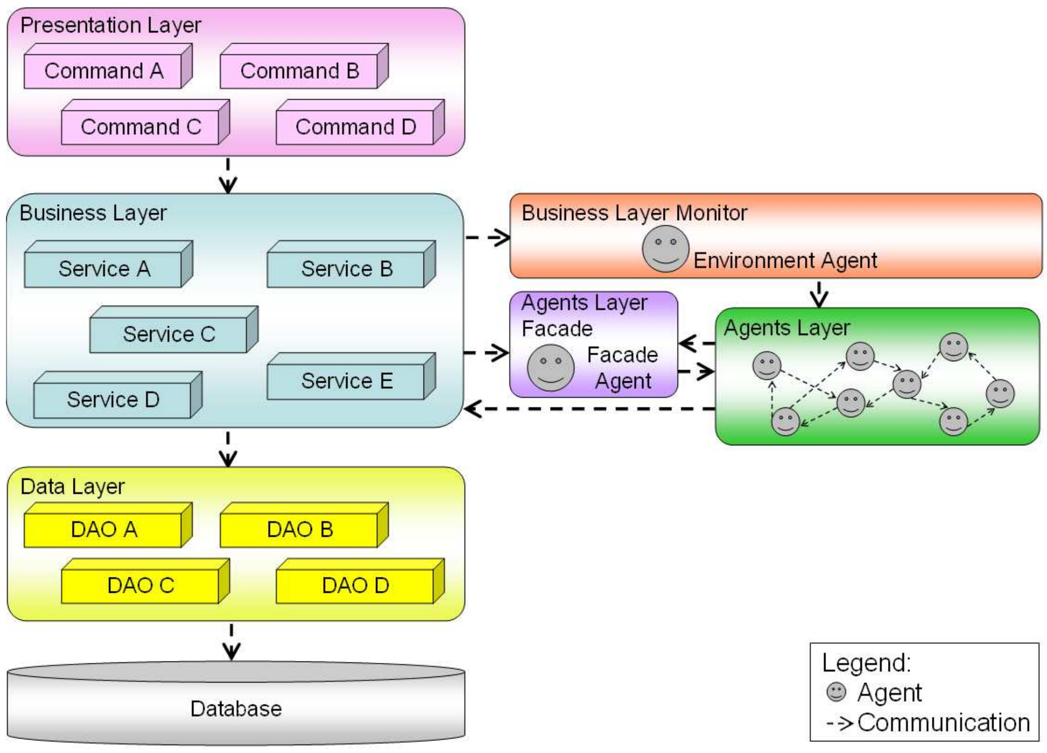

Figure 5.1: Web-MAS Architectural Pattern.

5.1.2 Roles Implementation

The concept of roles is adopted in most of MASs, meaning that an agent can play roles in organizations. Several MAS methodologies define ways of modeling and specifying roles, such as Gaia (Wooldridge 2000a, Zambonelli 2003) and PASSI (Cossentino 2005), which were described in Chapter 2. However, not all frameworks for implementating agents provide this concept. In this Section, we describe how to implement the role concept with JADE (Bellifemine 2007) and Jadex (Pokahr 2005) frameworks in a modularized way, based on our experience while developing MAS-PLs. Both solutions model roles not only as an "interface" of the role that an agent can play, but also as an implementation of the necessary behavior to the agent play that role.

Implementing Roles with JADE

JADE is a framework implemented in Java language, which simplifies the implementation of MASs through a middleware. Basically, the developer should extend the `Agent` class to implement agents and extend the `Behavior` class to define behaviors that the agent should have. These are the main first class elements provided by the framework.

As agents are implemented with JADE framework only by extending Java classes, typical object-oriented techniques can be used when using this framework, such as design patterns and polymorphism. In this context, the Role Object pattern (Baumer 1997) is a design pattern whose purpose is "*to adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified*". Besides allowing the implementation of roles using JADE, this pattern keeps the role modularization, which is an important characteristic of the SPL development.

According to the pattern, we have the following participants: (i) Component – models a particular key abstraction by defining its interface and specifies the protocol for adding, removing, testing and querying for role objects. This interface represents the agent interface; (ii) ComponentCore – implements the Component interface including the role management protocol. This is the agent itself, which extends the `Agent` class of JADE; (iii) ComponentRole – it is an abstract role of the agents; (iv) ConcreteRole – concrete roles are the agents' roles, which are dynamically attached to agents.

An example of the use of the Role Object pattern can be seen in the ExpertCommittee case study, described in Chapter 4.

Implementing Roles with Jadex

In the same way of JADE, Jadex framework is also implemented in Java. However, instead of extending classes, the developer specifies agents in XML files, which contain mainly beliefs, goals and declaration of plans. Plans are implemented in classes that must extend the `Plan` class. In addition, Jadex provides the capability concept, which is declared in a similar XML file and can be incorporated to agents and other capabilities. It provides a reuse mechanism. Moreover, Jadex follows the BDI model (Rao 1995) and has a reasoning engine that chooses plans in order to achieve goals. Nevertheless, Jadex does not provide the role concept.

The solution proposed for implementing roles with Jadex is based on capabilities. The beliefs and goals of a specific role are modularized into a capability, and optionally the plans necessary to achieve the goals. If the plans are not declared into the capability, the role will just define the "interface" that the agent will have while playing a certain role; and if the plans are declared, the role will provide both "interface" and behavior for the agent, like the solution adopted with JADE. The capabilities are statically declared into an agent; however it is possible to change the agent model on runtime, what allows to add and remove capabilities dynamically.

An example of implementing roles with Jadex can be seen in the OLIS case study, described in Chapter 4.

5.1.3
Improving Modularization using Aspect-Oriented Programming

Recent research work presents the benefits of adopting AOP techniques to improve the modularization of features in SPLs (Alves 2006, Figueiredo 2008), framework based (Kulesza 2006b) or MASs (Garcia 2004) architectures. The increasing complexity of agent-based applications motivates the use of AOP. AOP has been proposed to allow a better modularization of crosscutting concerns, and as a consequence to improve the reusability and maintenance of systems (Kiczales 1997). Among the problems of crosscutting variable features, we can enumerate: (i) *tangled code* – the code of variable features is tangled with the base code (core architecture) of a SPL; (ii) *spread code* – the code of variable features is spread over several classes; and (iii) *replicated code* – the code of variable features is replicated over many places. All these problems can cause difficulties regarding the management, maintenance and reuse of variable features in SPL.

In order to promote improved separation of concerns, some crosscutting features that present the problems mentioned above are natural candidates to be designed and implemented using AOP. During the development of our MAS-PL case studies, we have found the following interesting situations to adopt AOP techniques:

– *Modularization of the glue-code that integrates the web-based system (base code) with the agent features.* In order to integrate agents with the web application, the Web-MAS architectural pattern (Section **??**) uses the Observer design pattern to observe/intercept the execution of business methods of the services of the Business layer. This 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 provide the `Observable` class and the `Observer` interface.

  However, AOP (Kiczales 1997) can be used to modularize the intercepted code that allows the agents monitor the execution of the web-based system. It facilitates the (un)plugging of the agent features in the system and makes the addition of agents even less intrusive. AOP 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; and

– *Modularization of the agent roles.* The use of the Role Object pattern (Baumer 1997) to implement agent roles with JADE framework provided the role modularization by the use of object-oriented techniques. However, the use of this pattern cannot provide an improved isolation of the agent role features, which is essential to SPL variability management. The implementation of the agent classes (e.g. `UserAgentCore` class) requires, for example, the activation and deactivation of the agent roles over different points of the execution of the agent behavior, such as agent initialization, execution of specific plans, etc. The adoption of AOP to modularize agent roles (Garcia 2005, Kendall 1999) is thus a better option to improve the modularization and evolution of the agent roles features.

Empirical studies were performed with MAS-PLs in order to analyze the (dis)advantages of the use of AOP to implement agent features. The MAS-PLs were implemented using different agent platforms (JADE and Jadex) and different implementation techniques (conditional compilation, design patterns, configuration files and AOP). Results of these studies can be seen in (Nunes 2008c) and (Nunes 2009a).

5.2 Limitations of Our Approach

Our approach was developed in a bottom-up fashion. The development of the EC case study allowed us to identify the deficiencies in the MAS-PL development. Next, we have made comparison studies among SPL approaches and MAS approaches, and have modeled agent features of the EC MAS-PL with them. The result was the selection of PLUS, PASSI and MAS-ML to incorporate our process. In addition, new activities and adaptations to these approaches were proposed to address their drawbacks in the development MAS-PLs. Later, the EC was developed using our process, and this helped to refine our process. Finally, the OLIS case study was developed to evaluate our process. In addition, some activities of our approach, mainly related to the modeling of agent features, were adopted in the development of design and implementation projects in the Software Systems Design graduate course at PUC-Rio.

The proposed process aims at addressing the problems stated in Chapter 1, most of them identified with our exploratory studies. However, in this section we point out some limitations of our approach to developing MAS-PLs. First, our approach is the definition of a domain engineering process; nevertheless application engineering is the other process of SPLE, which defines a prescribed way in which all models and reusable assets will be used to derive applications. Although we provide means for deriving products by keeping features traceability, we do not explicitly define means of performing the application engineering. Even though this process of the SPLE is out of the scope of this dissertation, it is important to consider that it is also needed and should be addressed in future work.

In our process, we have adopted stereotypes to model variability in the models, and have proposed new models to trace variability. The stereotype concept is typically used for modeling variability, because it is the standard extension mechanism offered by the UML, and thus allows variability information to be made explicitly visible in diagrams (Muthig 2002). Although the use of stereotypes has these advantages, one drawback of their use is keeping consistency among models. The variability information is spread along several models, and no technique is used to verify if this information is consistent. One scenario that this is important is when a mandatory feature becomes optional, and this change must be propagated to all models.

A potential solution for this problem is to incorporate techniques from Model-driven Development (MDD). According to (Beydeda 2005), in MDD, models are used to reason about a problem domain and design a solution in the solution domain. Relationships between these models provide a web of dependencies that record the process by which a solution is created, and help to understand the implications of changes at any point in that process. In addition to creating these models, we can define rules for automating many of the steps needed to convert one model representation to another, for tracing between model elements, and for analyzing important characteristics of the models. Exploiting MDD approaches to improve our process is part of future work.

Our case studies explored the scenario of incorporating features with autonomous and pro-active behavior in web applications with typical web architectures, which use widespread patterns. We have chosen this scenario, because we believe this is a potential application of agents, which can help with the automation of users' tasks. As a consequence, in all case studies, agents are not part of the MAS-PL core. However, MAS-PLs can have agents as part of their core; therefore it is important to validate if our approach covers this situation. Nevertheless, there is no restriction in our approach to have agents as part of the core – it is necessary only to use the appropriate stereotypes – but there is the need to explore this scenario to prove it.

5.3 Final Remarks

This chapter presented lessons learned with our experience with the MAS-PL development. We first described guidelines to help in agent features modularization in the development of MAS-PLs. We presented the Web-MAS architectural pattern (Nunes 2008b), whose purpose is to integrate traditional web applications architectures with software agents in such way that they can be easily (un)plugged. Therefore, the pattern provides the extension of existing web applications to incorporate new agent features with a low impact and provides an architecture that supports optional agent features. We also showed how the role concept can be implemented in two agent platforms: JADE and Jadex. Both of them do not provide this concept. Finally, we presented and discussed the adoption of Aspect-oriented Programming in some situations in order to provide a better modularization. The second lesson learned is related with the limitations of our approach, mainly showing parts of the MAS-PL development, which were not covered by our approach.