Proactive Computing Based Implementation of Personalized and Adaptive Technology Enhanced Learning over Moodle^{TM}

Sergio Marques Dias, Sandro Reis, Denis Zampunieris
Faculty of Sciences, Technology and Communication
University of Luxembourg
Grand-Duchy of Luxembourg
{sergio.marquesdias, sandro.reis, denis.zampunieris}@uni.lu

In this paper, we show how to implement a system that adds proactivity to Moodle^{TM} to reach a personalized and adaptive support for both students and teachers, by providing an engine to run proactive rules, based on events and non-events, to enhance the users' e-learning process.

Personalized and Adaptive Software Systems; Web-based Learning Systems; Proactive Computing

I. Introduction

Learning Management Systems (LMS) are established tools that have proven their usefulness, but they are limited since they only react to events (user's actions on the interface). On the other hand, proactive computing [2] allows to work on behalf of the user and to act on its own initiative, instead of just reacting to events.

As first introduced in [3], Proactive LMS (PLMS) is a new concept that adds proactivity to an existing LMS, to create a combined tool that analyses users behaviour in order to reach a personalized, adaptive and intelligent support (see [1] for full theoretical background) that enhances the users e-learning process.

The prototype explained in this paper was designed to be the proactive engine embedded into Moodle^{TM}, the current e-learning platform used in our University. This decision allows the research team to focus solely on the design and implementation of the Proactive Engine, using the existing LMS as a framework without modifying it and taking advantage of its functionalities (e.g. sending emails) and of its data. Any new data that the engine requires will be saved on its own database. The idea is that the proactive engine can easily be added to any LMS, by implementing a specialization of the database wrapper to provide the specific data access queries.

In this paper we will start by giving an overview of the whole system and then explain in detail its core modules: the rules engine, the rules execution and the user's interface. We provide a running example of the whole system, draw some conclusions and briefly describe the future works.

II. System Overview

The interaction with Moodle^{TM} is done exclusively via its database, where the Proactive System checks state changes relevant to the scenarios' logic. To that effect, we developed an abstract database wrapper, with two implementations: MySQL (with the specific SQL queries needed to access Moodle's data) and Proxy MySQL with a cache on top of the first one (applying the Proxy Design Pattern).

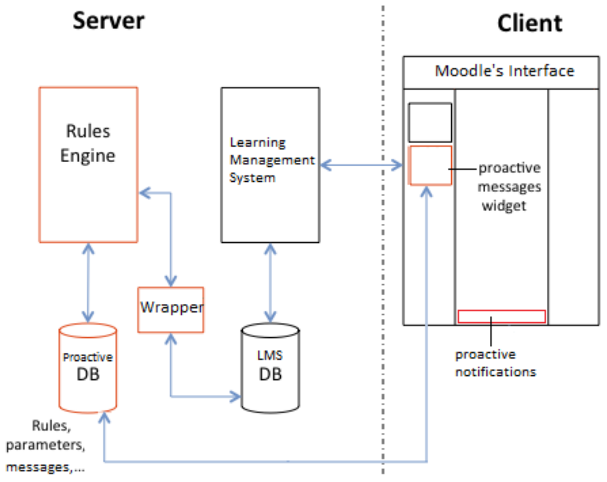


Figure 1 – System Architecture

As shown in Figure 1, the core of the Proactive System is the Rules Engine in addition to the rules themselves. The engine is responsible for the control flow of the rules execution where the rules represent the implementation of the proactive scenarios' logic. As mentioned before, the Rules Engine's goal is to add proactive behaviour to the LMS, neither to change its code, its data nor to restrict its execution in any way.

III. Rules Engine

Our engine acts as it was initially defined in [3], where it is responsible for storing and executing the set of rules, by managing two queues of rules. The 'current queue' holds the rules to be executed in the current activation period (the amount of time allowed for the Proactive Engine to run its rules in a sequence). The second holds the rules generated in the current activation period by the executed rules, depending on their logic.

We ensure the persistence of the Rules Engine by using the Hibernate^{TM} framework to save the rules' queue in each activation period, the generated messages, system statistics and parameters, in a specific database schema.

After some initialization, the engine enters the main loop: it is a process that runs forever. The two conditions that stop the execution of the engine are: when the list of rules is empty; or it receives a STOP signal from the system administrator. Explicitly - the admin requested a STOP in the "admin interface", or implicitly - the administrator changed the system parameters, thus the system must restart to take into account the new settings. An iteration represents an activation period where the engine enters another loop responsible for the actual running of the rules. This second loop has three exit conditions: the current queue is empty; the number of executed rules in this activation period has reached the maximum allowed value (given by the system parameter N); the current activation period execution is taking more time than allowed (given by the system parameter F). Reaching any one of these three conditions is enough to leave the loop of rule execution. Inside this second loop, it dequeues (gets and removes) the first rule from current queue, treats the statistics on the execution of this rule, and executes that rule by calling its execute() method. After exiting the execution, it prepares for the next activation period by: adding the (elements of the) next queue to the current queue; emptying the next queue; clearing database cache (non-permanent cache); replacing the queue on the database with data from current queue; saving statistics for this iteration; updating statistics on execution.

IV. Rule Execution (Abstract Rule)

Since the Proactive System is a goal-oriented mechanism, it uses a pre-defined set of scenarios, each with a goal of providing help to the user or the tutor according to their activity on LMS. The scenarios have been divided into two categories: Meta-Scenarios and Target Scenarios. The main characteristic of the first is that it is a context aware continuous never-ending rule. As soon as one Meta-Scenario detects the corresponding event on the LMS, it activates the relevant Target Scenarios, in which the Meta-Scenario delegates the specific job to the appropriate scenarios. The Target Scenarios take care only of one specific situation, after having performed its individual job each scenario is dismissed. Each scenario is implemented by one rule in a separate java class file. Every rule implements an Abstract Rule. The rules execution follows the algorithm defined in [3] and shown in Figure 2.

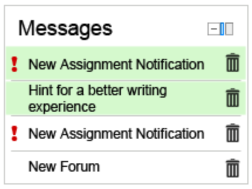
```
activated = false;
dataAcquisition();
if (activationGuards())
    activated = true;
    if (conditions())  actions();
return rulesGeneration();
```

Figure 2 - Algorithm for AbstractRule.Execute()

V. Proactive User Interface

The user interface of our prototype on Moodle is composed of 3 different parts: the Messaging block; a Message pop-up; and a Notification bar (see Figures 3-5).

The user gets notified of changes in the system, by emails and the messaging block (module) added to Moodle's menu. Through Moodle's messaging block, each user can read and manage the messages he/she received from the proactive engine. Each message is presented in a row where the user can find information like the title, importance, if the message is read or unread and a trash-bin to delete it. The unread messages have a different background colour to be easily identifiable.



Messages

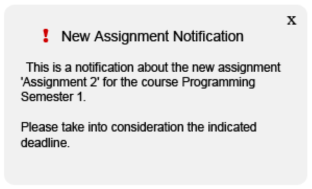
! New Assignment Notification

Hint for a better writing experience

! New Assignment Notification

New Forum

Figure 3 – Messaging Block

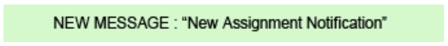


! New Assignment Notification

This is a notification about the new assignment 'Assignment 2' for the course Programming Semester 1.

Please take into consideration the indicated deadline.

Figure 4 – A Message Popup



NEW MESSAGE : "New Assignment Notification"

Figure 5 – The Proactive Notification Bar

When a user chooses to read a message, he clicks on the message title to open a popup with the message. Since the Message block can be out of sight, whenever a new message is sent to the user, a notification message pops up to get the users' attention.

VI. Example of Proactive Scenarios

A very small example of a possible execution flow of one simplified scenario is explained next. The actual scenarios as used in our prototype are more complete and complex.

Since Moodle^{TM} doesn't provide any built-in notification to the students upon an assignment creation, our team considered this to be essential. Thus, we implemented a scenario (MTA001) to that effect, with a proactive rule that is continuously checking for new assignments. Upon detecting a new assignment (A) on the LMS, this rule then generates two new rules, implementing two notification scenarios (NTF001, NFT002): the first goes through the list of enrolled students (S) on the course to which (A) is related, and generates in turn a new messaging rule with the appropriate subject and body to each (S_i). The second generates a new messaging rule to Prof. P, the creator of (A) informing him/her about the previous students notifications.

This example needs 3 activation periods (iterations) to fully execute all tasks: in iteration 1, MTA001 generates NFT001, NTF002, MTA001 (it clones itself); in iteration 2, NFT001 generates MSG for students (S_1 to S_n); NTF002 generates a MSG for (P), MTA001 clones itself; in iteration 3 all messages are sent and MTA001 clones itself once more.

VII. Conclusion and Future Work

In this paper, we showed how our proactive system can enhance the Moodle™ LMS by providing an engine to run proactive rules next to it and with regard to its state. We are currently developing the Moodle^{TM} add-on, which allows the user to interact with our system messages. We are also currently testing the system in a limited set of courses, and with appropriate measurement tools in order to validate and improve the concept for the future.

References

[1] S. Marques Dias, S. Reis and D. Zampunieris, "Personalized, Adaptive and Intelligent Support for Online Assignments Based on Proactive Computing", Proc. of DULP&SPeL Workshop in IEEE Conference ICALT 2012.

[2] D. Tennenhouse, "Proactive computing", Communications of the ACM, 43(5), 2000.

[3] D. Zampunieris, "Implementation of a proactive learning management system". Proc. of E-LEARN Conference 2006.