Journal of Web Engineering, Vol. 1, No.1 (2005) 000-000


# BOOSTING THE ADOPTION OF COMPUTER MANAGED INSTRUCTION FUNCTIONALITIES IN E-LEARNING SYSTEMS

GENNARO COSTAGLIOLA, FILOMENA FERRUCCI, VITTORIO FUCCELLA

*Dipartimento di Matematica e Informatica, Università degli Studi di Salerno*
*{gcostagliola, fferrucci, vfuccella}@unisa.it*

Received (to be filled by the JWE editorial)
Revised (to be filled by the JWE editorial)

Standardization efforts in *e-learning* are mainly aimed at achieving interoperability among *Learning Management Systems* (*LMSs*) and *Learning Object* (*LO*) authoring tools. In particular, the main standard producers are giving special attention to a set of functionalities, referred to as *Computer Managed Instruction (CMI)* and also known as *SCORM Run-Time Environment*. Their adoption is crucial in the achievement of full interoperability among *LMSs* and *LO* authoring tools since they allow *LOs* to be launched in the *LMS* and to exchange data with it. Even desirable, standard compliancy and guideline adoption are difficult to obtain for *LMS* producers. This paper presents two design solutions aimed at boosting the adoption of *CMI* functionalities in Object-Oriented and Message-Oriented *LMS* systems, respectively. The former is a framework, named *CMIFramework*, which allows *LMS* developers to rapidly adopt *CMI* functionalities in Object-Oriented systems. The latter is a *Service Oriented Architecture (SOA)*-based reference model for offering the *CMI* functionalities as a service, external to the *LMS*. We investigate several case studies concerning the adoption of *CMI* functionalities, using our solutions, in different *e-learning* contexts.

*Key words*: E-Learning, standardization, Computer Managed Instruction, CMI, SCORM RTE, framework, Service Oriented Architecture, SOA, Web Services
*Communicated by*: (to be filled by the JMM editorial)

## 1 Introduction

*E-learning* has spread rapidly in recent years. Many authors remark that the birth of *e-learning* coincides with the time in which hypermedia resources started to be distributed through the Internet, that is, with the birth of the World Wide Web. Since then its growth has been strongly influenced by the development of the Web technologies. In fact, all of the *e-learning* systems are now developed as Web applications. These Web applications present some specific requirements and features that should be taken into account by suitable Web engineering tools and methodologies. In particular, interoperability among e-learning systems is a very important issue. In the context of these systems, the most important aspect related to interoperability is the possibility of running LOs produced with any authoring tool on any *Learning Management System (LMS)* compliant to the standard specifications.

Recently, in order to obtain a stronger interoperability among e-*learning systems*, great efforts have been made to define standards, reference models, and guidelines for *e-learning*. Despite the presence of several detractors [15, 33], the importance of *e-learning* standards has been ratified by several institutional initiatives in many countries. For example, in Italy, a decree of the Ministry of Instruction establishes that a crucial requirement for academic institutions for being accredited as “distance courses providers” is the support of several standard specifications [10].

At present, the main specifications are focused on the proposal of common formats for *LO* metadata and for resource interchange. Metadata for *LOs* can be used to describe them from several points of view. Several models have been defined. Noteworthy among them are *Dublin Core* [11] and *Learning Object Metadata* [24], the latter being the first international standard issued by *IEEE Learning Technology Standard Committee (LTSC)* [17]. A resource interchange format for *LOs* is proposed in the *Content Packaging* specification, issued both by *SCORM* [37] and *IMS* [19].

Common metadata and interchange formats are not enough to allow the *LMSs* to fully interoperate: it is necessary that there is a standard environment in which the *LOs* can be launched and can exchange data with the *LMS*. The definition of a model for this standard environment is currently proposed in several specification documents, such as *AICC CMI Guidelines for Interoperability* [2], *SCORM Run Time Environment (RTE)* [38] and *IEEE CMI* [23]. We will refer to the functionalities proposed in these documents using the acronym *CMI*, an abbreviation of *Computer Managed Instruction*.

Unfortunately, standard compliancy and guideline adoption are difficult to obtain for *LMS* producers. Adopting standards is onerous for several reasons: a large amount of time is required for studying documents, understanding their contents and implementing them properly [27]; the specifications have a technical nature and the availability of reference material and supporting tools is limited [3]; the scarce availability of courses in a standard format prevents developers to adequately test the developed solutions [40]. Furthermore, there are numerous standard producers, whose specification documents differ in some aspects. Those differences are often a source of confusion and incompatibilities. This is the case of *CMI*, whose basic architecture of launch and communication model is accepted by all of the producers, but the specifications differ in the definition of the data models of the information exchanged during the communication and of the *API* exposed to the *LOs* to perform the communication. Consequently, the adoption of the *CMI* model, whose importance is attested by the attention of the three main producers of standards, of many software vendors and of several authors in the literature [6], [36] has been insufficient. Thus, there is the need of design solutions for boosting the adoption of *CMI* functionalities in *LMSs*. In this paper, we present two approaches*:* the former is a framework, named *CMIFramework,* and has been conceived for OO systems.

*CMIFramework* consists of a set of reusable components that can be easily configured and extended to obtain an environment in which *LOs* that are compliant with any issue and version of the specifications can be launched without incurring incompatibility problems. The basic idea is to match the changes among the various issuers’ specifications with the variability and extension points (hot spots) of the framework. *CMIFramework* is configurable to allow the developer to decide the data formats and the interfaces to support and has been conceived flexible enough to address further modifications in the specifications. To show the effectiveness of the proposal, we describe how a well-known *LMS*, Sakai, has been extended to support *CMI* functionalities.

The second solution, suitable for *Message-Oriented* systems, is a *Service Oriented Architecture* (*SOA*)-based reference model for offering the *CMI* functionalities as a service, external to the *LMS*. The necessity for externalizing the *CMI* functionalities from the *LMS* is motivated by the following factors:

- The high cost of being up-to-date with the specifications.
- The high cost of hardware and software resources necessary for offering the functionalities.
- The necessity for having a standard model, for which *SOA* represents a valid choice.

Our model can be useful for *LMS* producers to avoid the above costs and to develop the *LMS* independently from the external module, which can be provided by third party efforts. Starting from a technical discussion of the requirements of the model, we propose a decomposition of an *LMS* system in order to establish the separation of roles between the basic *LMS* and the identified external service. Then, we outline the architecture of the system by explaining the interactions among the identified services. Lastly, the whole process is defined, identifying the activities involving the LMS and the external process. The proposed model is validated through a prototype system, in which a popular *LMS*, developed with the *PHP* language, is enhanced with the support of *SCORM RTE* functionalities, provided by an external Web service based on Java technology.

The rest of the paper is organized as follows: the next section discusses the state of art in the adoption of standards in *e-learning* systems. Some concepts of the *CMI* model are reported in section 3; section 4 outlines *CMIFramework*, including its functionalities and architecture and a case-study of its application; the *SOA*-based model for offering *CMI* functionalities as a service external to the *LMS* is presented in section 5. Work in literature related to ours is the subject of section 6. Some final remarks and comments on future work conclude the paper. Some code segments for the configuration of the framework and the definition of the SOA-based model are reported in the appendixes.

## 2 The Interoperability Issue and the Adoption of Standards in E-learning Systems

Interoperability among software systems can be generically defined as “the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units” [18]. For e-learning systems interoperability is a fundamental issue. In the context of these systems, in fact, there is the necessity of exchanging various categories of data, such as information about the learners and the *LO*. The most important aspect concerns the re-use of the *LOs*, since, the production and development of educational contents for e-learning is usually a higher cost process, compared to the production of course material for traditional learning [34]. Once full interoperability among *LMS* and authoring tools is achieved, it will be easier to share *LOs*, and, consequently, re-use them, with considerable time and resource savings for the content developers. As mentioned in the introduction, in order to obtain a stronger interoperability among e-*learning systems*, standards, reference models, and guidelines for *e-learning* have been defined.

To have a deep knowledge about the state of art in the support of standards in e-learning systems, a survey has been carried out on both authoring tools and *LMSs*. We realized that authoring tools are more advanced than *LMSs* in this context. In particular, tools such as *Macromedia Authorware* [4] and *Toolbook Instructor* [45], offer an almost complete support of the existing specifications, including many versions of several issuers.

| LMS | | Supported Specs | | | |
|---|---|---|---|---|---|
| **Product** | **Web Site** | **LO Metadata** | **Content Packaging** | **CMI** | **Others** |
| ANGEL Learning Management Suite, V7.1 | http://www.angellearning.com/products/lms/default.html | IMS 1.3<br>SCORM 1.2 | IMS 1.1.4<br>SCORM 1.2 | SCORM 1.2 | IMS QTI 1.2.1 |
| ATutor 1.5.3.2 | http://www.atutor.ca/ | SCORM 1.2 | IMS 1.1.3<br>SCORM 1.2 | SCORM 1.2 | / |
| Blackboard Learning System CE 6.1 EL | http://www.blackboard.com/products/Academic_Suite/Learning_System/CE.htm | SCORM 1.2 | IMS 1.1.3<br>SCORM 1.2 | SCORM 1.2 | IMS QTI 1.2.1 |
| Blackboard Learning System Vista 4.1 EL | http://www.blackboard.com/products/Academic_Suite/Learning_System/vista.htm | SCORM 1.2 | IMS 1.1.3<br>SCORM 1.2 | SCORM 1.2 | IMS QTI 1.2.1 |
| Claroline 1.8.1 | http://www.claroline.net | IMS 1.2.2<br>IMS 1.2.3<br>SCORM 1.2<br>SCORM 1.3 | IMS 1.1.3<br>IMS 1.1.4<br>SCORM 1.2<br>SCORM 1.3 | SCORM 1.2<br>SCORM 1.3 | IMS QTI 2.0 |
| Desire2Learn 8.1 | http://www.desire2learn.com/ | IEEE LOM<br>Dublin Core<br>SCORM 1.2<br>SCORM 1.3 | IMS 1.1.2<br>IMS 1.1.3<br>SCORM 1.2<br>SCORM 1.3 | AICC<br>SCORM 1.2<br>SCORM 1.3 | IMS QTI 1.2.1 |
| eCollege | http://www.ecollege.com/indexflash.learn | SCORM 1.2 | SCORM 1.2 | SCORM 1.2 | / |
| KEWL | http://kngforge.uwc.ac.za | SCORM 1.2 | SCORM 1.2 | SCORM 1.2 | / |
| LON-CAPA | http://www.lon-capa.org | / | IMS 1.1.3<br>IMS 1.1.4 | / | / |
| Moodle 1.6.1 | http://moodle.org/ | SCORM 1.2<br>SCORM 1.3 | SCORM 1.2<br>SCORM 1.3 | SCORM 1.2<br>SCORM 1.3 | / |
| Sakai 2.3 | http://sakaiproject.org/ | / | / | / | / |
| TeleTOP Virtual Learning Environment | http://www.teletop.nl/en/ | IMS 1.2.2<br>IMS 1.3<br>SCORM 1.2<br>SCORM 1.3 | IMS 1.1.3<br>IMS 1.1.4<br>SCORM 1.2<br>SCORM 1.3 | SCORM 1.2<br>SCORM 1.3 | / |
| The Blackboard Academic Suite | http://www.blackboard.com/products/academic_suite/index.Bb | IMS 1.2.1<br>SCORM 1.2<br>SCORM 1.3 | IMS 1.1.2<br>SCORM 1.2<br>SCORM 1.3 | SCORM 1.2<br>SCORM 1.3 | / |

Table 1- State of Art of Support of Standards in LMSs (last up-date: 9th Jan 2007)

As for the *LMSs*, the analysis has been carried out using data from the publicly-available *Edutools* [12] system: thirteen products out of the most popular *LMSs*, accompanied either with an Open Source or a Commercial license, have been included in the survey. The analysis, summarized in table 1, only concentrates on the support of standards and excludes other features. The table shows the supported specifications for each *LMS*. These are divided by functionality (*LO Metadata*, *Content Packaging*, *CMI* and others). Each cell in the table reports the supported specification, including issuer and version, for each product and for each set of functionalities.

From the study, it emerged that the producers of the most popular systems are very interested in standardization: twelve systems out of thirteen support some standard specifications at present. As has been previously pointed out, there are some difficulties for the *LMS* producers in adopting specifications which regard the same functionalities produced by different issuers. Challenges reside in overcoming incompatibilities among them. From this point of view, *CMI* functionalities are in a disadvantaged position in respect to other functionalities. By analyzing the data, we can notice that the variety of the specifications supported in *LMSs* for *CMI* is inferior than others: all of the producers of *LMSs*, minus one (*Desire2Learn*), have preferred to adopt *SCORM*, ignoring specifications as those issued by *IEEE* and *AICC* [1].

Lastly, as also noticed by Buendia & Hervas [7], there are several difficulties for producers in being up-to-date with the latest versions of the specifications: in our survey, the producers have up-graded their *LMS* to the last version (the 1.3, issued in 2004) in only five of the eleven cases in which the *SCORM* model has been adopted.

## 3 The Computer Managed Instruction Model

The *Computer Managed Instruction (CMI)* model defines a set of functionalities which allow *LOs* to be launched in the *LMS* and to exchange data with it. The issuers (*AICC*, *IEEE* and *SCORM*) propose a very similar model, even though several differences are present even among different versions of the same issuer. Almost all of them are aimed at defining the following common aspects regarding the *LO – LMS* communication:

- *Launch*: the set of rules under which an *LO* can be launched in a Web-based environment
- *API*: the interface of methods to be invoked by an *LO* in order to communicate with the *LMS*
- *Data Model*: the data set on which the communication is based.

Only a limited set of *LOs* can communicate with the *LMS*. These *LOs*, according to the *SCORM*, are called *Sharable Content Objects (SCOs)*, and their communication capability is due to the fact that they contain a specialized software module, called *ECMAScript*, which consists of several *Javascript* functions in the *ECMAScript* standard format.

The core of the *CMI* model contains the description of the *LO - LMS* communication mechanism. The way in which it takes place is shown in figure 1, which depicts a Web based scenario where a *LO* has already been launched in a Web browser window and the *LMS* runs within a Web Server.

The *LO (SCO)*, equipped with the *ECMAScript* module, can communicate with another module running on the client side: the *API Adapter*. We will refer to a running instance of the *API Adapter* with the term of *API Instance*. The *API Adapter*, even though it runs on the client side, must be provided by the *LMS*. Therefore, it has often been implemented through a browser plug-in, an *Active-X* object, or, more frequently, through a Java applet. Java applets technology fits the needs of the *RTE* model well, since it can provide a module deployed on a server (the *LMS*), but running on the client (the Web browser). The *API Adapter* module exposes an interface of methods to the *LO*. By invoking them, the *LO* can exchange data with the *LMS* server. In practice, the *API Adapter* works as a broker between the *LO* and the *LMS*, since the former lacks the capability to connect with the *LMS* server directly, due to its nature of a plain document readable through a Web browser.

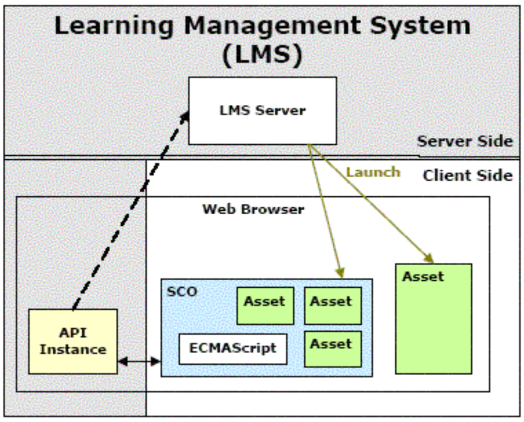


Figure 1 - CMI Architecture (SCORM RTE)

The *LO* has the duty of starting and terminating the communication session and of leading the data exchange with the *LMS*. On the *LMS* side, an instance of the communication data must be kept. As mentioned before, the *LO* can perform the communication invoking several *ECMAScript* methods exposed by the *API Adapter*. With reference to the 2004 (1.3) version of the *SCORM*, the methods for starting and terminating the communication are, respectively, *initialize*() and *terminate*(). The methods to set and get the *run-time data* (an instance of the data model) on the *LMS* are, respectively, *getValue(<element_name>)* and *setValue(<element_name>, <value>)*.

The *API Adapter* must handle error conditions which can occur during the communication, and notify the *LO* about them by returning a specific value on a method invocation. Furthermore, the *API Adapter* provides the *LO* with further methods for obtaining information on the errors, in case any of them have occurred.

The *Data Model* is the set of data exchanged between the *LO* and the *LMS* during the communication. For each element, the name, the data type, the access mode (*read only*, *write only*, *read/write*), the multiplicity and other information have been defined. This set of data includes, but is not limited to, information about the learner, interactions that the learner has had with the *LO*, objectives, success status and completion status of the *LO*. The set of data that can only be read *(RO)* by the *LO* is typically information which must be passed from the *LMS* to the *LO* to be shown to the user, such as the learner's name and identifier. The set of data that can be both read and written (*RW*) is information which must be available at the *LO* at its launch and updated by the *LO* at the end of the session. An example of this information is the progress level of the lesson. Finally, an example of data which can only be written (*WO*) by the *LO*, is the time spent by the learner in the session. Generally, there is an instance of the *Data Model* (the run-time data, in the sequel) for each (learner, LO) couple, if the learner has accessed the *LO* at least once. The same instance can be shared throughout the session of the learner on the *LO*, otherwise a new instance can be generated, according to the needs of the *LMS*. All data model elements are identified by a name, composed using a dot-notation (e.g., *cmi.success_status*).

## 4. The CMIFramework

An Object-Oriented framework can be defined as a reusable, "semi-complete" application that can be specialized to produce custom applications [21]. Frameworks are one of the most exploited solutions to achieve software reuse, together with software libraries and design patterns [30]. The primary benefits of OO application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers [14].

The *CMIFramework* is an Object-Oriented Java framework which can be instantiated in order to alleviate the work of *LMS* developers in adopting *CMI* functionalities in their systems, thanks to the software re-use principle. It also solves the incompatibility problems among different specifications, allowing the launch of *LOs* compliant with any *CMI* specification in the same environment. Presently, *CMIFramework* can support most of the specifications produced so far, avoiding the time-consuming task of up-grading all the *LO* compliant with older versions of the specifications. Nevertheless, it has been designed flexible enough to address several future changes in the specifications. Furthermore, it goes beyond the standard functionalities, allowing the developers to define customized solutions, not necessarily adhering to them strictly. Once the framework has been instantiated to support *CMI*, adding a newer version of the specifications can be done only by editing the configuration of the framework.

To elaborate, among the features of *CMIFramework*, we can find the support for *LMS*-defined *API Interfaces* with the related error handling system and for *LMS*-defined data models. These *LMS*-defined solutions can be combined to standard ones, providing all these functionalities in a unique environment. Other interesting features of the *CMIFramework* is the caching of the *LO - LMS* communication and the server-side persistence of the *Run-Time* data.

*CMIFramework* has an innovative architecture. On the client-side, it allows the deploying of any number of *API* Interfaces. This is simply done mainly by editing the XML-based configuration and coding the interface. The configuration also allows the designer to completely define the elements of the *Data Models*.

On the server-side, a small amount of code must be written in order to customize the *LMS* behavior on the occurrence of the main events of the communication: the actions to undertake on initialization, commit and termination, can be handled by customizing the server-side module.

### *4.1. Main Functionalities*

As mentioned before, the reference model has been subject to several modifications through the many documents that discuss *CMI* functionalities. The most important ones take into account the definition of the following aspects: the interface of the *API Adapter*, the error handling model and the *Data Model* elements. The main configuration features and the extension points of the framework concentrate above all on these aspects. In particular, our framework supports the following main functionalities:

- Full implementation of all the *CMI* specifications produced so far;
- Caching of the *API Instance – LMS* communication;
- Server side persistence of the run-time data;
- Support for more than one *API Instance* interfaces and for their related error handling systems;
- Support for more than one *Data Model;*

The way in which the *API Instance – LMS* communication takes place follows a consolidated set of rules. The framework supports these rules and, moreover, it implements a sort of communication caching: the run-time data, initialized by the *LMS*, is sent to the *API Instance* before the communication takes place. Later on, all the read and write operations are performed locally. Only at the end of the communication, the instance is sent back to the *LMS*, avoiding delays in the communication due to possible slow communications.

The specifications consider some cases in which the run-time data could be persisted, to be possibly loaded and re-used in more than one communication sessions. In order to support this feature, the framework has a support for the persistence of run-time data on the server side.

The interface exposed to the *LO* by the *API Instance* has been subject to some changes which have regarded, particularly, the name with which the *API Instance* object is identified in the Web page and the method prototypes definition. Even though such modifications could seem trivial, they have been the main source of incompatibility between *LMS* and *LO* authoring tools. To face this problem, we chose to give the framework the chance to expose more than one interface to the *LO*. These interfaces must be defined in the framework configuration. The framework generates them at run-time, just before the launch of the *LO*.

The error handling system is strictly dependent on the *API Instance* interface and, in fact, for each invocation of one of its methods, a set of exceptions can occur. During the evolution of the specifications, the code and the name bound to each of them, have been subject to changes. Moreover, time after time, new error conditions have been defined. An exception mainly occurs when wrong values for the parameters are passed to the invoked methods or when the *API Instance* state does not allow the invocation of a given method. In the event of an error, the method execution is interrupted and an atypical value is returned to the *LO*. The framework allows the *LMS* designer, through a simple configuration mechanism, to list all the exceptions that could occur for a given method of a deployed *API Interface*, providing a large set of predefined *checks*. Furthermore, it could be extended by adding new *check* definitions. All the listed *checks* are triggered before the execution of the methods, to assure that the invoked method can be safely executed. The internationalization support assures that the name and the text of for the error messages are viewed in the language selected in the browser's settings. The *Validator* framework [46], provided by *Apache Group*, has been used for aiding the error handling process.

The data model on which the *LO-LMS* communication is based has been in the centre of the debate since its first definition by *AICC*. Later on, several more data models have been defined. Even in this case, in order to support all the specification documents, the framework has been designed to support more than one data model. For each data model, the elements that constitute it, can be defined by the *LMS* administrator, setting their identifier and type in the configuration. Moreover, other information concerning them can be defined, such as: the access rules to the element, some constraints on its value and some eventual dependencies on other elements. Finally, it is possible to define some derived elements, calculated on the basis of the values of other elements. Even in this case, some predefined classes to calculate derived elements have been made available to the *LMS* developer. The framework can be extended as well, writing the code of the method that calculates the derived element.

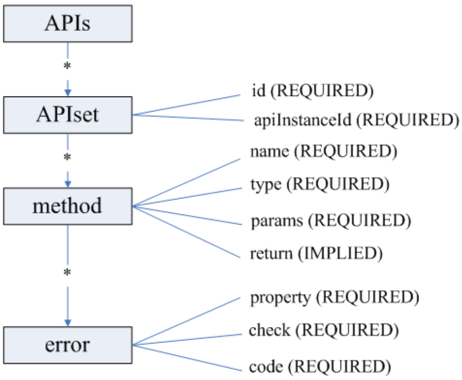


Figure 2 - Information model of the apis.xml configuration file

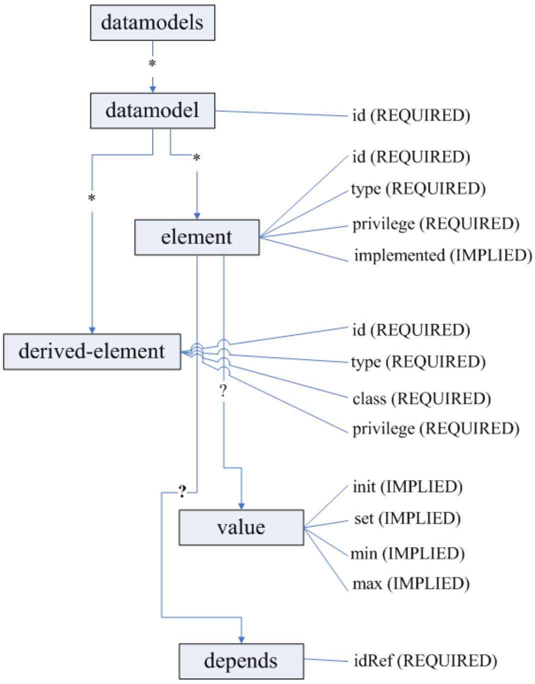


Figure 3 - Information model of the datamodels.xml configuration file

The framework configuration is easily performed by editing two *XML* files. Their formats are shown in figure 2 and 3. The first (named *apis.xml*, see figure 2) allows us to define the *API Instance* interfaces. This can be done by adding a new *APIset* element. The interface methods can be defined through the *method* element. The error conditions which must be checked on a method can be defined by using the *error* element.

The *datamodels.xml* configuration file allows us to define the supported data models. Each of them, represented by a *datamodel* element, can contain both simple elements and derived elements, represented, respectively, with the *element* and the *derived-element* XML elements. Constraints on the elements can be defined by using the *value* element, while, in order to define dependencies between them, the *dependency* element must be used.

*4.2. Architecture*

The framework is composed of two main components: a client-side one and a server-side one. The former is composed of modules running in the Web browser (Java applets), archived in a *JAR* (*Java ARchive*) compressed file. The latter is a Java library of modules for use in Web-based applications.

The client component provided by the framework is, in practice, an *API Adapter* slightly modified, compared to the one shown in figure 1, in order to communicate with *LOs* conformant to different versions of the specifications and to allow the deployment of *LMS*-defined *API Interfaces*. As stated in section 2.1, the *API Instance* runs on the client in the LMS main page. This characteristic has been modeled with the composition relation in figure 4a.

As stated before, the *API Adapter* is usually implemented through a Java Applet. Unfortunately, this solution only allows a limited set of methods to be exposed to the *LO*. Furthermore, these methods' signatures must be established at the development-time. For example, a *SCORM 1.2* conformant *API Adapter* will contain the *LMSInitialize*() method, while this method is simply called *initialize*() in *SCORM 1.3*.

To support the definition of the *API Interface* at the deployment-time (by editing the configuration) we use the following pattern: the *LO* does not invoke methods directly on the *API Adapter*, but on a new *API Interface* module, implemented through a *Javascript* class. The *API Adapter* is still implemented through a Java applet. The *API Interface* Javascript class is built at run-time by a server-side module which reads the configuration of the framework. It is worth noting that the most recent versions of the most popular Web browsers do support the execution of Javascript Object Oriented code. Just before the launch of the *LO*, the issue of the specifications to which the *LO* is conformant (this information is contained in the *Content Package* from which the *LO* was imported) is determined. Then, the framework looks for the definition of the right *API Interface* in the configuration, generates the Javascript class and instantiates a new object of that class.

The proposed pattern is shown in the *UML* class diagram of figure 4b. The relation of *API Adapter* with the *LMS Main Page* has been modelled as a composition. Before the launch of the *LO*, the server-side module called *APIInterfaceGenerator.jsp* builds the *APIInterface* Javascript class. Then, the *LMS Main Page* launches the *LO*, which contains (modelled through a composition relation) the *ECMAScript* module, as the specifications prescribe. The *LO* can invoke the *APIAdapter* methods indirectly through the *APIInterface*.

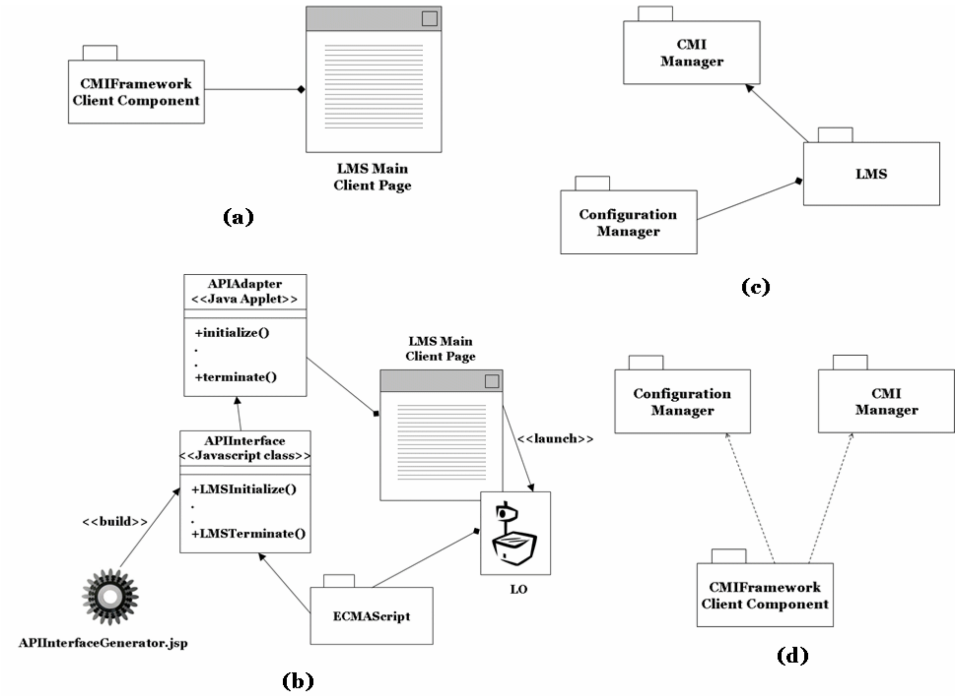


Figure 4 - Architecture of CMIFramework

To compel the *LO* to interact with the *APIInterface* Javascript class instead of the *APIAdapter*, the framework makes use of the following trick: it is the *APIInterface* object and not the *API Instance* to be inserted in the *LMS Main Page* with the standard identifier used by the *LO* to locate the interface (the identifier's value is *API* in the *SCORM 1.2* and *API_1484_11* in the *SCORM 1.3*).

The server component is composed of two main modules: *ConfigurationManager* and *CMIManager*. The former must be instantiated in the *LMS* by composition, the latter by inheritance, as shown in figure 4c. The *ConfigurationManager* is responsible for reading the configuration from the *XML* files and for sending it as serialized objects on demand to the client component. Besides the definition of the *API Interfaces*, the remaining part of the configuration consists of the definition of the data models to support, including the whole set of its elements and derived elements. As for the *CMIManager* module, its core has been implemented as an abstract Java Servlet, such that it can be customized for *LMS* implementations. The *CMIManager* is responsible for handling the server side duties of the *LO-LMS* communication. The customizable features are *LMS*-defined actions to undertake before the start of the communication, after the end of the communication and on the commit of the changes made to the run-time data, through the implementation of *onInitialize*(), *onTerminate*() and *onCommit*() methods, respectively. In practice, the developer only needs to create a Java servlet which extends the one provided by the framework and to implement the above methods. For a typical implementation, before the communication starts, the run-time data could be initialized with some information kept by the *LMS*, such as the personal information of the learner and the progress status of the learner on the lesson. Furthermore, the run-time data, received from the client, could be used to update the *LMS* records after the communication or on every commit event. The interactions just described, between the client component and the two server modules, are shown in figure 4d.

*4.3. Case-Study: A SCORM Module for Sakai*

The *Sakai Project* [35] is a community source software development effort to design, build and deploy a new *Collaboration and Learning Environment* for higher education. The *Sakai* application framework has been customized by our developers in order to obtain a learning environment called *Running Platform (RP)*, which has been used at our department for the management of the courses, in a blended learning style.

A prototype for a new tool for *SCORM RTE* has been developed in order to test the effectiveness of our framework in creating an environment in which *LOs*, compliant with different versions of the *SCORM* specification could have been launched. The module, originally designed as a stand-alone application, was later integrated into the *RP*. The stand-alone application, called *CMILMS*, is a minimal system, which is only able to launch pre-loaded *LOs* conformant to the versions *1.2* and *1.3* of the *SCORM*. *CMILMS* adopts the classical three-tier (Presentation-Logic-Data) architecture of Web applications. The *CMI Framework Client Component* has been deployed in the *Presentation* layer of the application by simply putting the JAR file among the Web content of the application. Furthermore, it has been instantiated in the JSP pages of the application through the use of a *Tag Library* [43] developed ad hoc.

The server component has been instantiated in the *Logic* tier of *CMILMS* as explained in the previous chapter: the *CMIManager* module by inheritance and the *Configuration Manager* module by composition. The *CMIManager* has been extended in order to customize the server side behavior of the application. In this case, both the methods *onInitialize*() and *onTerminate*() have been implemented. In the former, the data model used for the communication has been initialized with the data to pass from the *LMS* to the *LO*. The latter has been used for the opposite purpose. In both cases, simple *JDBC* code has been added to these methods. The server-side persistence of *run-time* data, provided by the framework, has been used to share the data model instances across multiple sessions of the same learner on the same *LO*. The framework has been configured by declaring the *API Interfaces* and the data models for both the supported versions of the *SCORM*. Extracts from the configuration files are shown in appendix A.

*Sakai* offers a suitable container for tools and associated services. Its architecture is quite flexible to allow different levels of integration for the tools. The most loosely coupled integration level allows the developer to integrate stand-alone applications. At the scope, two main rules must be followed:

1. The request must be intercepted and dispatched to the application by a module called *Sakai Web-App Gateway*
2. Basis services, such as authentication and authorization management, must be provided by an interface called *Sakai API Gateway*.

In light of these arguments, the main integration programming activity has consisted in the modification of the *CMILMS* application in order to dialog with the *Sakai APIs*. The architecture of the integrated system is shown in figure 5. The whole application runs in a Java Web Container (Apache Tomcat, in our test deployment). The *CMILMS* application uses the *Sakai* framework as a container for all the needed services: for the handling of the HTTP requests and responses and for the use of basic services such as authentication, authorization and user group handling.

To plug the *Sakai WebApp Gateway* in the application, actually, there was no need to modify the application: we just needed to develop a *servlet filter* for the requests and the responses. A *filter* entry was added to the *deployment descriptor* (the *web.xml* file) of the application.

At the *Presentation* tier, some work was necessary in order to harmonize the aspect of the final application: the *Cascading Style Sheets* of *RP* have been applied to the Web pages of the application. Then, the original main page of the *CMILMS* application has been linked to an *iFrame* in the *RP* page which launches the *LOs*.

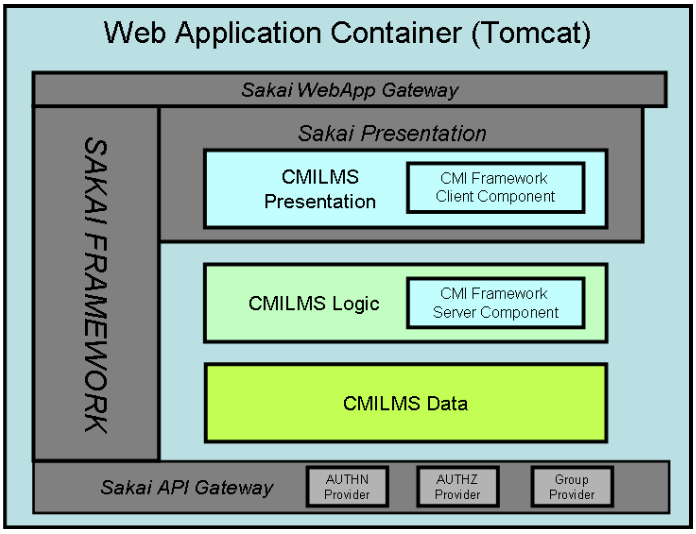


Figure 5 - Architecture of the *Running Platform* with the integration of the SCORM module (*CMILMS*)

The work necessary to plug the *Sakai API Gateway* in the application has been slightly more complicated: the handling of the user accounts, based on *JDBC*, of the stand-alone version have been substituted with some calls to the *Sakai API Gateway*. This has been done in every part of the application dealing with the user handling (both in the *Logic* and in the *Data* tier). Additionally, some user accounts and information have been imported from the application database to the *RP* one.

## 5. The SOA-based Model for CMI

According to *OASIS* [28](the *Organization for the Advancement of Structured Information Standards*), *SOA* can be defined as *a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations*. *SOA* offers several advantages to developers (reusability, composability, autonomy, optimization and discoverability) and is very useful when loose-coupling, that is, a low dependency among systems, is needed. For the latter requirement, OO systems appear inadequate, since objects are strictly tied in the applications, thus it can be very difficult to offer functionalities as "services". Even under the point of view of interoperability, SOA overcomes the OO model, since the most common Object-based distributed system technologies (i.e CORBA or J2EE) are based on quite different and incompatible object models [13]. *SOA* is strictly related to Web services: actually, Web services can be regarded as a realization of *SOA*.

This section defines a *SOA*-based architecture for offering the *CMI* functionalities from a service external to the *LMS*. Our solution is valid for a generic *LMS*. A real-world application, based on our model, is contained in the last sub-section. We propose a decomposition performed at two different levels: at a higher level, the separation of concerns between the *LMS* and the external service is specified; at a lower level, the modules composing each service are identified. Only the basic functionalities of the *CMI* model, such as the launch of *LOs* and the *LO-LMS* communication, together with basic *LMS* functionalities, such as the management of *LO*, are considered. Other services which can be found in a common *LMS* or other standard functionalities, which are not pertinent to our research, are not considered in this work. This choice does not prevent us from applying our model to wider systems.

### *5.1. Definition of the Services*

The main objective of this phase is the definition of the services to build and of the logic encapsulated in each of them. Most of our work in this phase consists of establishing how to span the *CMI* functionalities among the identified services. Our aim is to alleviate the duties of the *LMS* as much as possible in the handling of *CMI* functionalities. Most of the work will be provided by an external service, which will be referred to as *CMI Service*.

In order to support the *CMI* model, the basic functionalities of an *LMS* are the following:

- managing users (above all, learners and tutors) and keeping an *LO* database;
- launching and dismissing *LOs* on learner's demand;
- communicating with the *LO*, providing the learner's user-agent with The *API Instance*;
- handling the run-time data: the *LMS* must create an instance of it using names and types defined in the *Data Model*, keep it up-to-date during the communication and save it for future sessions.

The handling of users, including registration, authentication and authorization services, must be a duty of the *LMS*. Digital repositories of *LOs* can be external to the *LMS*. For example, the solution proposed in [48] uses a dedicated server for keeping LOs. Other solutions integrate them on the same server as the *LMS* which launches them. We prefer to deal with the separate servers option because it is flexible enough to include the integrated one: once an external service is identified to keep *LOs*, it can still be placed on the same server as the *LMS*. We will refer to the service which keeps *LOs* and provides them to the *LMS* as *LO Repository service*.

According to the *CMI* model, among the operations provided to the learner by the *LMS*, there are the launch, the suspension, the resume and the dismissal of an *LO*. The communication between the *LO* and the *LMS* must start on the *launch* or *resume* events and must end on the *suspend* or *dismiss* events.

While it is quite clear that the *CMI Service* is in charge of hosting the server-side module which handles the communication with the *LO*, more doubts can arise as to which service should provide the *API Adapter* to the user-agent. As pointed out in section 2.2 this is a duty of the *LMS*. The *API Adapter* must be downloaded and run on the client-side. Due to these requirements, a common solution is to implement the *API Adapter* as a Java applet, which can be packed in a *JAR* file and downloaded through the *HTTP* protocol. As before, we will refer to the instance of the *API Adapter* running on the user-agent as *API Instance*. To avoid complications, the following reasons suggests the inclusion of the *API Adapter* as a module of the *RTE Service*:

- The *API Instance* must interact with the server-side module responsible for the communication. Putting the *API Adapter* on a separate service from this module gives no practical benefits and would compel us to define a standard protocol for the communication.
- A security limitation of Java applets prevents them from establishing network connections with other servers than the one from which they have been downloaded. This limitation, however, can be overcome by using signed applets or changing user-agents security policies.

The last considerations concern how and where to keep the communication run-time data and, if they are kept by a service external to the *LMS*, how to make this data available to the latter during the communication. It is widely accepted that run-time data is not part of the *LMS* database. In the past, a poor design choice, adopted in some systems, was to design the *LMS* database in conformity with the *Data Model* of the *CMI* model. This choice should be avoided for the following reasons: firstly, the *Data Model* has a hierarchical structure, which does not fit well with the relational model that is almost always used by *LMSs*; secondly, the definition of the data model has been subject to changes across the versions of the *SCORM* specifications. To be up-to-date, it would have been necessary to re-engineer the systems designed with the database conformant to the *Data Model*.

In light of the previous observations, our choice is to keep the run-time data on the *CMI Service*. In the next section we will explain how to make the run-time data available to the *LMS* when needed. The above reasoning led us to identify the services model for *CMI* functionalities shown in figure 6. It identifies the services and the operations for each of them. Including only the *CMI* functionalities, the *LMS* must only supply the operations for the learner to make use of the *LOs*. The *LO Repository Service* provides the operations related to the administration of the *LO* repository, such as listing, searching and downloading of the *LOs* contained in it. The *CMI Service* is responsible for all the operations to perform the *CMI* communication with the *LO*, for making the run-time data available to the *LMS* and, finally, for making the *API Adapter* available for download to the learner's user-agent.

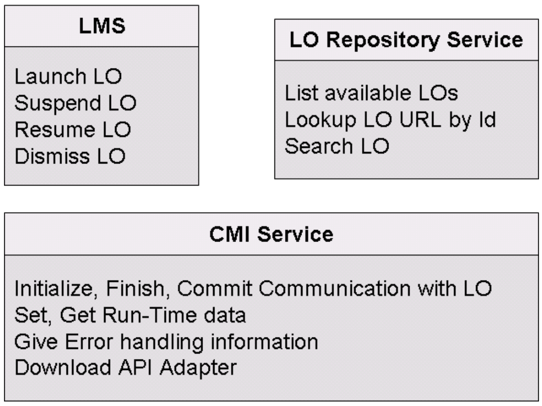


Figure 6 - Services Model

### *5.2. Architecture*

The main objective in this phase is to define the low-level architectural decomposition of an *LMS* system which offers *CMI* functionalities, using the services identified in the previous section. The interactions among them, with the specification of the message exchange patterns, are shown.

Figure 7 illustrates the “actors on the scene” and their interactions. They are the *LMS*, the *CMI Service*, the *LO Repository Service* and the *User-agent*. The interactions among them are shown with arrows. Wide arrows show Web services-based interactions. The following channels have been defined:

1. The channel through which the *User-agent* downloads the *API Adapter* from the *CMI Service*
2. The channel for requests and responses from the *User-agent* to the *LMS* to perform operations (launch, suspend, resume and dismiss) related to the *LOs*
3. The channel used by the *LMS* to locate the requested *LO* on the *LO Repository Service* and to forward the user-agent’s request to the given *URL*
4. The channel used by the *API Instance* (running on the *User-Agent*) to perform the *CMI* communication with the *CMI Service*
5. The channel through which the *CMI Service* and the *LMS* communicate to allow the *LMS* to access run-time data when needed

Channels from 1 to 4 can use a simple *HTTP* request/response message pattern. The message pattern for channel 5, instead, requires a more detailed explanation on the events which cause the *LMS* to access the run-time data. In our model, the run-time data is kept by the *CMI Service*. According to the *CMI* model, the run-time data can be read and written by the *LO* during the communication through the invocation of the methods *getValue*() and *setValue*() respectively, exposed by the *API Instance*. Besides the *LO*, the run-time data must also be read and written by the *LMS*. This happens on the occurrence of several events, for the following reasons:

1. After run-time data is instantiated and just before the communication starts, the data must be initialized with *LMS*-specific settings
2. After the communication is finished the *LMS* can read the run-time data to up-date its internal database with information gathered during the communication
3. Whenever a *setValue*() or *getValue*() or *commit*() is performed, the *LMS* could undertake some customized actions.

It is worth noting that, since the *CMI* communication is performed between the *API Instance* and the *CMI Service*, the *LMS* is unaware of the events listed above. Thus, the channel 5 is used to inform the *LMS* of the occurrence of these events.

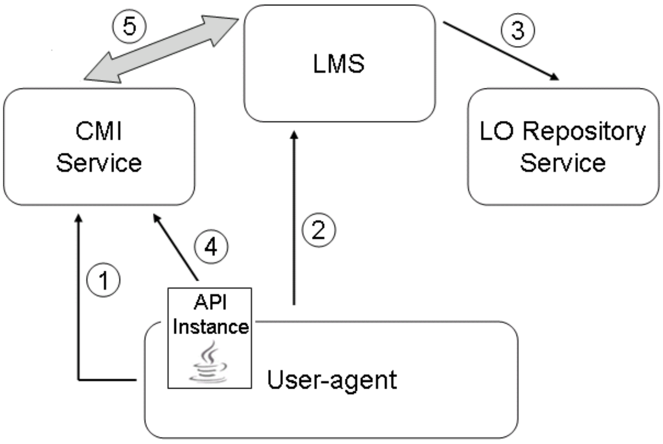


Figure 7 - Interactions among services

*5.3. Definition of the Process*

This section describes in detail the communication process between the *CMI Service* and the *LMS*, performed whenever a user-agent asks for an *LO* to be launched. The communication can be based on *SOAP* formatted messages and must be *conversational*: the services keep a state of the conversation during the message exchange. In other words, the messages must be part of a session. To perform this message exchange, the *LMS* must equipped with a service callback endpoint. We will refer to this module as the *LMS Endpoint*.

The basic idea is that the LMS, on a LO launch request from the user-agent, delegates all the duties of the CMI communication to the CMI Service, and only requires to be notified on the occurrence of the desired CMI events (a sub-set of [*initialize*, *setValue*, *getValue*, *commit*, *terminate*]). On the occurrence of those events, the LMS could undertake some actions. For example, some information contained in the run-time data can be persisted by the LMS on each invocation of a *commit* type method. The only event the LMS should compulsorily manage is the *terminate* one, on which it should receive and use in some way the run-time data registered during the communication.

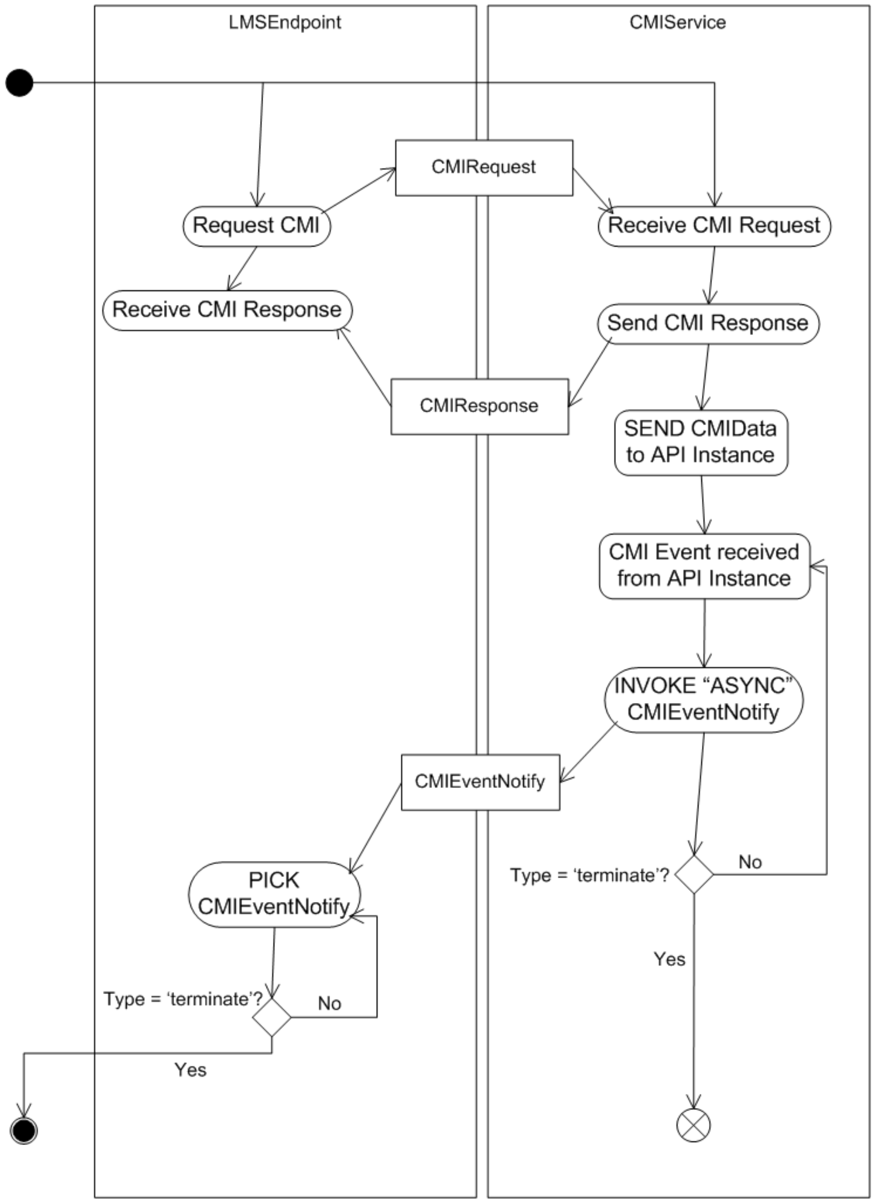


Figure 8 – UML activity diagram of the process

The process has been shown with an UML activity diagram (figure 8): the swim-line on the left shows the activities performed by the *LMS Endpoint*, while the activities performed by the *CMI Service* are shown on the right. The scenario starts when the LMS has received a LO Launch request from the user-agent and has already instantiated and initialized the run-time data. Furthermore, the *API Adapter* has already been downloaded from the CMI Service and an instance of it is running in the user-agent. Firstly, the *LMS* requests the services of the *CMI Service*, sending a synchronous *CMIRequest* message. This message carries configuration information, such as, the issue of the CMI specifications, and, most important, the set of CMI events on whose occurrence it wants to be notified. Furthermore, the entire run-time data are sent with the request. The *CMI Service* replies with a *CMIResponse* message. The above defined operations follow the *request-reply* message pattern (*solicit-response* from the point of view of the *CMI Service*). Lastly, the *CMIRequest* message can, optionally, carry authentication information from the LMS.

Now, the LMS can launch the LO, and the CMI communication can start. The CMI communication takes place between the *API Instance* and the *CMI Service*. Firstly, the *CMI Service* must send the run-time data, just received from the LMS, to the *API Instance*. On the occurrence of the CMI communication events, the *CMI Service* notifies that to the LMS, attaching the whole run-time data to the asynchronous *CMIEventNotify* message. Messages are asynchronous for performance reasons: the communication must not stop on every method invocation and the LMS can undertake the desired action. The message exchange pattern is *notification* from the point of view of the *LMS Endpoint* and *one-way* from the point of view of the *CMI Service*.

On the occurrence of the *terminate* event type, after the notification message, the process terminates.

### *5.4. Case-Study: A SCORM RTE Module for Moodle*

In this section we show how the reference architecture presented in the previous sections has been applied to add *SCORM RTE* functionalities to *Moodle 1.8* [26], a popular *Open Source LMS* developed using *PHP* server-side language. A prototype of the *CMI Service* has been implemented using *Java 2 Enterprise Edition (J2EE)* technology. The choice of such cross-technology system is not the fruit of coincidence, but has been made in order to show the language independency of our solution. Furthermore, the *CMI Service*, developed as a prototype, can be completed to offer its services to more than one *LMS*, based on whatever technology, at the same time.

The *CMI Service* has been built as a *J2EE* Web Application, packaged in a *WAR* file. It can be deployed in any *J2EE* Web container. The availability of *CMIFramework* has allowed us to make little effort in developing the *CMI Service*. We should recall from the previous section that, among the others, *CMIFramework* provides the following components:

- An implementation of the *API Adapter* as a Java applet
- Full implementation of the modules involved in the *LO-LMS* communication
- Run-time data persistence handling module
- A module, implemented as a Java Servlet, which provides methods to override in order to handle the events of the communication.

Thanks to the availability of the above modules, it has been necessary to develop only some modules of the *LMS Endpoint* from scratch. The *Apache Axis* [5] *SOAP* library has been used to compose the messages to carry run-time data to and from the *LMS*, on the occurrence of the events of the communication. To elaborate, these events have been handled by overriding the *onInitialize*() and *onTerminate*() methods, provided by the server side module of *CMIFramework*. In these methods, the code to compose *SOAP* messages has been added. The information carried by these messages include: the event type, a session identifier, to keep a conversational state and the entire run-time data, represented as a list of (name, value) couples. It is worth noting that the caching of the communication has been used: in our implementation we have avoided the *API Instance* and the *CMI Service* to communicate on every single *setValue*() and *getValue*() method invocation. Instead, the run-time data has been changed locally on the *API Instance*, thus sending it to the *CMI Service* only on the termination of the communication.

The *LMS Endpoint* has been developed as an extension of the *Moodle* system. *Moodle* comes with a mechanism to develop extensions to the basic *LMS*: a new module can be developed and integrated by modifying a template provided with the Moodle documentation. Actually, a *SCORM* player for *Moodle* already exists, but it is entirely built as an internal module. Our prototype, however, is aimed at demonstrating how to provide *SCORM RTE* functionalities using an external service. Moodle has an internal *LO* repository, thus, the operations of searching an *LO*, getting its *URL* and so on, are based on the simple invocation of *Moodle API* methods. Furthermore, the *forward* operation with which the *LMS* launches an *LO*, has been implemented as an action internal to the Web server which hosts the *LMS* system. The support for external *LO* repositories has been announced for the 2.0 version of *Moodle*.

Summarizing, our development activity consisted of the following two steps:

1. Preparing the environment in which the *LOs* are launched
2. Developing the *LMS Endpoint* for *Moodle*.

The activities related to the first point have consisted in simple *PHP* page coding: a *PHP* Web page has been created. The *API Adapter* has been inserted in it as an applet to download from the Web server which hosts the *CMI Service*. Furthermore, this page has been designed to contain a form with the buttons to launch, resume, suspend and dispose a previously selected *LO*.

The development of the *LMS Endpoint* has been quite simple: a free library of *PHP* functions has been used to manage the *SOAP* messages sent to and received from the *CMI Service*. In our simple prototype, the LMS only requires the *CMI Service* to be notified on the *terminate* event. The function which handles the launch operation, contains the code to send a *SOAP* message to register to the *CMI Service*, as described in the previous section. Applying a common pattern, suggested by the *CMI* specifications, the *LO* downloaded from the *LMS* is launched in a child Window of the user-agent. A single function has been created to decode the message from the *CMI Service*, read the event type and locally persist the run-time data.

To handle the conversational state of the communication we have adopted the 1.0 version of the *SOAP Conversation Protocol* [41]. This protocol makes it easy to conduct stateful conversations between two parties. In appendix B, the definition of the *CMI Service* is shown through a *WSDL* document. Furthermore, the appendix contains the BPEL code for the definition of the *CMI Service* process.

## 6. Related Work

Challenges in the adoption of standards have been the main motivation for the investigation of approaches which insure the re-use of standard functionalities [47]. To this extent two main solutions have been explored:

- Providing LMS developers with frameworks [27, 9, 32] and reference implementations [25, 31, 39] of standard functionalities.
- Proposing architectures and reference models [8, 16, 29, 44, 47, 48, 49, 50] to adopt in real systems in order to establish a widely accepted decomposition for *e-learning* systems. Once established, these models should facilitate the independent development of the identified components.

In order to simplify the duty of the developers, some reference implementations have been developed. The most important of them is *SCORM Sample RTE*, freely downloadable through the SCORM website. In [25] an implementation of the *SCORM*, adapted to present contents on mobile devices, is presented. Beside the implementations based on Java, such as the ones cited so far and as the further product described in [31], some systems have been developed using the *Microsoft .Net* framework, such as *DotNetSCORM* and the one described in [39], which implements the *LO – LMS* communication using Web Services.

As for frameworks and libraries, an experiment is shown in [27], which proposes a library of re-usable components and testing tools for *WBT* systems. A wider-ranging work, which has as an objective the development of a framework for the adoption of the whole *SCORM* model, is described in [9]. Another framework for the support of several functionalities not directly connected to the adoption of standards and guidelines in *e-learning* systems is presented in [32].

Despite recognizing the importance of supporting all of the different existing specifications, none of the cited works propose a solution to the problem of the incompatibility between *LO* and *LMS* supporting different specifications or different versions of the same specification: the only solution seems to be the proposal of conversion utilities to update the *LOs*. For example, the *SCORM* web site claims that “the *ADL Technical Team* is currently developing several conversion utilities that can be used to update *SCORM* Version *1.2* conformant content to *SCORM* Version *2004* conformance”. Some third party conversion utilities are already available on the Internet and some authoring tools, such as the one proposed in [42], have been developed to upgrade *LOs* in order to support the latest version of the specification. It is worth noting that the effort of upgrading LOs can be avoided using an LMS developed with the framework we propose.

Some researchers propose a *SOA*-based architecture for defining a decomposition of a generic *e-learning* system [47], [48], [49], [50]. Also e-learning standards and guidelines producers, as *IMS*, have started to focus their attention on Web services, proposing ad hoc specifications [20]. Authors in [50] propose a service architecture to integrate *LMS* and *Learning Content Management System* functionalities. All the identified modules are services that offer their functionalities using Web Services technology. Vossen & Westerkamp propose an architecture of a generic *e-learning* system [47], [48], whose functionalities are provided by a set of Web Services, external to the main *LMS* application. In [49] a Grid-based layered architecture for the support of collaborative learning is proposed.

Other *SOA*-based architectures are more focused on the search of *LOs*, which may or may not use standard functionalities. In [44] a Web Services-based architecture is proposed in order to allow *LMS* servers to share learning-related information, such as learning material, learner data and learning strategies. Each of the previous category of information is kept by a different sub-system. According to Hussain and Khan [16], Web Services can be used in the field of content repositories, in order to obtain an infrastructure for the centralized search and discovery of *SCORM*-based learning contents. The work proposed in [29] is based on the *LTSA* [22] architecture, which is adapted to a *SOA*-based model. The authors intend to use this model to allow for a flexible integration of educational components. *LOs* can be discovered using the metadata annotation of the *LOM* and then assembled together in a Web-services based platform. Casella et al. [8] propose several modifications to the approach described by the *SCORM RTE*. The use of the *API Adapter*, which could not run in devices with limited capabilities, is substituted by the use of a suitable *Middleware* component in a Web Services-based architecture.

A work closely related with ours is [9]. It presents a framework for the adoption of the whole SCORM model in a SOA-based architecture. Most of the functionalities are provided by external services. A service which offers the functionalities specified in the RTE model is called Tracking Service. In the authors' opinion, such a service should be local to the LMS, for performance reasons. This argument is valid in their architecture, due to their decision to fuse RTE functionalities with other tracking functionalities. Otherwise, in our opinion, there would not have been valid reasons for preventing the externalization of the RTE functionalities from the LMS.

Reference implementations give scarce opportunities for software re-use, since their components are tightly coupled with the whole system of which they are a part. Frameworks overcome this problem, being loosely coupled with the system in which they are instantiated. Unfortunately, under several circumstances, several problems still arise with frameworks. First of all, in most cases they are adoptable only in systems developed with the same technology: an Object Oriented framework developed in Java could not be used in a .NET or LAMP (Linux, Apache, MySQL, PHP)-based LMS. Secondly, even though the use of a framework allows for the easy extensibility of a system with new functionalities and has more customization margins, when instantiated in a monolithic system, frameworks become part of it, increasing its size. The drawbacks in this case are related to the maintenance, testing, and workload of the resulting system, since most enterprises, educational organizations cannot afford high systems handling [9]. The latter problem can be overcome defining standard architectural models based on distributed *e-learning* systems. Among them, solutions based on *SOA* are more and more widely adopted. Offering a way to externalize functionalities from the LMS, they allow LMS producers to gain several benefits, such as better software re-use and easier integration and complexity management, with a consequent cost reduction. Furthermore, these solutions are language independent and interoperable. Offering functionalities as services external to the LMS often poses technical and practical problems depending on the specific service offered. The lack of existing systems or prototypes based on the proposed architectures prevents us from effectively validating them. Furthermore, there is no agreement on the decomposition. As a consequence, we are quite far from obtaining a standardized architectural model of a generic and comprehensive *e-learning* system, which could effectively help in the re-use of functionalities.

## 7. Conclusions

The production of *e-learning* content is a very onerous task, compared to the production of course material for traditional learning. Thus, there is the necessity of reusing the developed content and to launch it on any *e-learning* system. The possibility of running *LOs* produced with any authoring tool on any *LMS* is one of the most important aspect of interoperability. To this aim, standard formats and guidelines for e-learning have been defined. From an analysis of the state of art in their adoption, a certain difficulty from *LMS* producers in supporting the specifications has emerged and, in particular, in being up-to-date with the most recent of them. Problems arise in adopting specifications which regard the same functionalities produced by different issuers, due to incompatibilities among them. From this point of view, *CMI* functionalities are in a disadvantageous position in respect to other functionalities.

Nevertheless, the adoption of the *CMI* model has been identified as a necessary choice in order to make these systems fully interoperate. To facilitate the *LMS* designers, we have proposed two solutions to the above problems, in order to boost the adoption of the *CMI* model.

*CMIFramework* is helpful in rapidly adopting the *CMI* functionalities in Object-Oriented systems. A simple prototype has been developed in order to demonstrate its power and ease of use.

Then, a further solution, useful in those cases in which the high cost of implementing the CMI specifications suggests the necessity of externalizing its functionalities from the *LMS*, has been presented: a *SOA*-based architecture which can be adopted by *LMS* systems in order to support the *CMI* functionalities, using a service external to the *LMS*.

Both solutions allow *LMS* producers not to consume time and resources in implementing and being up-to-date with the standards. Furthermore, the *SOA*-based model allows *LMS* producers to use external resources for offering *CMI* functionalities, which can be more resource-intensive compared to other simpler standard functionalities, such as metadata and packaging. Nevertheless, compared the framework-based solution, in implementing the *SOA*-based model, a greater effort is required for the development of the wrapper for message exchange. A complete system which implements the *CMI Service* is planned as future work.

## References

[1] AICC (2007), Aviation Industry CBT Committee, http://www.aicc.org/

[2] AICC CMI (2004), *CMI Guidelines for Interoperability AICC rev. 4.0*, http://www.aicc.org/docs/tech/cmi001v4.pdf

[3] Andreev, R., Ganchev, I., O'Droma, M. (2005), Content metadata application and packaging service (CMAPS) - innovative framework for producing SCORM-compliant e-learning content, *Proceedings of the 5th International Conference on Advanced Learning Technologies*, pp. 274 – 278

[4] Authorware (2007), Macromedia Authorware 7, http://www.adobe.com/products/authorware/

[5] Axis (2006), *Apache Web Services – Axis*, http://jakarta.apache.org/axis /

[6] Bohl O., Schellhase J., Sengler R., Winand U. (2002), "The Sharable Content Object Reference Model (SCORM) – A Critical Review", *Int. Conf on Computers in Education*, pp. 950-951

[7] Buendia, F., Hervas, A., (2006). An Evaluation Framework for e-Learning Platforms Based on Educational Standard Specifications, *Proceedings of the 6th International Conference on Advanced Learning Technologies*, pp. 184 - 186

[8] Casella, G., Costagliola, G., Ferrucci, F., Polese, G., Scanniello (2006) G., A SCORM Thin Client e-learning Systems Based on Web Services, *To appear in International Journal of Distance Education Technology*

[9] Chu, C.P., Chang, C.P., Yeh C.W., Yeh Y.F. (2004), A Web-service oriented framework for building SCORM compatible learning management systems, *Proceedings of International Conference on Information Technology: Coding and Computing*, pp. 156 - 161 Vol.1

[10] *Criteri e procedure di accreditamento dei corsi di studio a distanza delle università statali e non statali e delle istituzioni universitarie abilitate a rilasciare titoli accademici di cui all'art. 3 del decreto 3 novembre 1999, n. 509* - G.U. n. 98 del 29/04/2003 - http://www.unimercatorum.it/Documenti/Decreto17_04_03.pdf (in Italian)

[11] Dublin Core (2005), Dublin Core Metadata Initiative, http://dublincore.org

[12] Edutools (2007), EduTools Course Management System Comparisons – Reborn, http://www.edutools.info/static.jsp?pj=4&page=HOME

[13] Emmerich, W. (2000), *Engineering Distributed Objects*. John Wiley and Sons

[14] Fayad, M.E., Schmidt, D.C. (1997), Object-Oriented Application Frameworks, *Communications of the ACM*, 40 (10), pp. 32-38

[15] Friesen, N. (2004), Three Objections to Learning Objects and E-learning Standards, McGreal, R. (Ed.). *Online Education Using Learning Objects*. London: Routledge, pp. 59-70.

[16] Hussain, N., Khan, M.K. (2006), SCASDA: SCORM-based Centralized Access, Search and Discovery Architecture, *Proceedings of the International Conference on SCORM 2004*, Taipei, Taiwan, pp. 137-140

[17] IEEE LTSC (2007), IEEE Learning Technology Standards Committee, http://ieeeltsc.org/

[18] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

[19] IMS (2007), IMS Global Learning Consortium, http://www.imsglobal.org/

[20] IMS WS (2005), *IMS General Web Services Final Specification v. 1.0*, http://www.imsglobal.org/gws/index.html

[21] Johnson, R.E., Foote, B. Designing reusable classes. *J. Object-OrientedProgramming* 1, 5 (June/July 1988), pp. 22-35.

[22] LTSC WG1 (2006), IEEE LTSC, WG1, *Architecture & Reference Model*, http://ieeeltsc.org/inactive/arch/

[23] LTSC WG11 (2002), IEEE LTSC, WG11, *Computing Managed Instruction*, http://ltsc.ieee.org/wg11/index.html

[24] LOM (2002), Learning Object Metadata, IEEE LTSC, WG12, http://ltsc.ieee.org/wg12/

[25] Lin, N.H., Shih, T.K., Hui-huang, H., Chang, H. P., Chang, H. B., Ko, W. C.; Lin, L.J. (2004), Pocket SCORM, *Proceedings of 24th International Conference on Distributed Computing Systems Workshops*, Tokyo, Japan, pp. 274-279

[26] Moodle (2005), *Moodle, A Free, Open Source Course Management System for Online Learning*, http://moodle.org/

[27] Nakabayashi, K., Kubota, Y., Yoshida, H., Shinohara, T. (2001), Design and Implementation of WBT System Components and Test Tools for WBT content standards, *Proceedings of 1st IEEE Int. Conf. on Advanced Learning Technologies*, Madison, USA, pp. 213-214

[28] OASIS (2007) - Reference Model for Service Oriented Architecture v1.0 - http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[29] Pahl, C., Barrett, R. (2004), A web services architecture for learning object discovery and assembly, *Proceedings of the 13th int. World Wide Web conference on Alternate track papers*, New York, USA, pp. 446-447

[30] Pree W., *Design patterns for object-oriented software development*, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1995

[31] Qu C., Nejdl W. (2002), "Towards Interoperability and Reusability of Learning Resource: a SCORM-conformant Courseware for Computer Science Education", *Proceedings 2nd IEEE Int. Conf. on Advanced Learning Technologies*, Kazan, Russia, pp. 525-530

[32] Redol J., Simões D., Carvalho A., Páscoa H., Coelho J., Grave P., Luís R., Horta N. (2003), "VIANET – A New Web Framework for Distance Learning", *Proc. of 3rd IEEE Int. Conf. on Advanced Learning Technologies*, Athens, Greece, pp. 258-259

[33] Rehak, D. (2002), SCORM is not for everyone. Kraan, W. & Wilson S. (Eds.) http://www.cetis.ac.uk/content/20021002000737

[34] Rosemberg, M. J. (2002) *e-Learning*. Makron Publishing, So Paulo.

[35] Sakai (2005), *Sakai Project*, http://www.sakaiproject.org

[36] Santos J.M., Anido L., Llamas M. (2003), "On the Use of E-learning Standards in Adaptive Learning Systems", *Proceedings of 3rd IEEE Int. Conf. on Advanced Learning Technologies*, Athens, Greece, p. 480

[37] SCORM (2007), Advanced Distributed Learning – SCORM, http://www.adlnet.gov/scorm/index.cfm

[38] SCORM RTE (2004), *The Scorm Run-Time Environment ver 1.3.1*, http://www.adlnet.org/scorm/history/2004/documents.cfm

[39] Shih, T.K., Chang, W. C., Lin, N.H., Lin, L.H., Hsu, H. H., Hsieh, C. T. (2003), Using SOAP and .NET web service to build SCORM RTE and LMS, *Proceedings of Advanced Information Networking and Applications*, Xi'an, China, pp. 408-413

[40] Simek, H., Akpinar, Y., (2005). Overcoming scormification difficulties in implementing a learning content management system, Proceedings of the 6th International Conference on Information Technology Based Higher Education and Training, pp. T3A/11 - T3A/16

[41] SOAP CP (2006), *SOAP Conversation Protocol 1.0*, http://dev2dev.bea.com/pub/a/2002/06/SOAPConversation.htm

[42] Su J. M., Tseng S. S., Weng J. F., Chen K. T.; Liu Y. L., Tsai Y. T. (2005), An object based authoring tool for creating SCORM compliant course, *Proceedings of 19th Int. Conf. on Advanced Information Networking and Applications*, Taipei, Taiwan, pp. 209-214 vol.1

[43] TagLibs – The Jakarta TagLibs Project, http://jakarta.apache.org/taglibs/index.html

[44] Tamura Y., Yamamuro, T. (2006), Distributed and Learner Adaptive e-Learning Environment with Use of Web Services, *Proceedings of the International Conference on SCORM 2004*, Taipei, Taiwan, pp. 11-15

[45] Toolbook Instructor (2007), e-Learning Software Content Authoring Tool, http://www.toolbook.com/

[46] Validator (2006), *The Jakarta Project – Commons Validator*, http://jakarta.apache.org/commons/validator/

[47] Vossen, G., Westerkamp, P. (2003), E-learning as a Web service, *Proceedings of Seventh International Database Engineering and Applications Symposium*, pp. 242 – 249

[48] Vossen, G.; Westerkamp, P. (2006). Towards the Next Generation of E-Learning Standards: SCORM for Service-Oriented Environments, *Proceedings of the 6th International Conference on Advanced Learning Technologies*, pp. 1031 - 1035

[49] Wang G.L., Li Y.S., Yang S.W., Miao C.Y., Xu J., Shi M.L. (2005), Service-oriented grid architecture and middleware technologies for collaborative e-learning, *Proceedings of IEEE International Conference on Services Computing*, Orlando, FL, USA, pp. 67 - 74 vol.2

[50] Xiaofei L., El Saddik, A., Georganas, N.D. (2003), An implementable architecture of an e-learning system, *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*, pp. 717 - 720 vol.2

### Appendix A: Configuration of CMIFramework to support SCORM 1.2 and SCORM 1.3

The code fragments in this section show how the *CMIFramework* has been configured for the case-study presented in section 4.3, in order to support both 1.2 and 1.3 versions of the *SCORM*. The former fragment is from the *apis.xml* configuration file, while the latter is from *datamodels.xml*.

In the *apis.xml*, two *API Interfaces* are declared with the attribute *apiInstanceId* set at *API* and *API_1484_11*, as required respectively by the *1.2* and *1.3* versions of the SCORM (lines 3 and 32). The definition of the interface consists of the enumeration of all the supported methods. For brevity, the configuration of the error handling system is not shown for all of the methods. In lines 12 through 19, the list of errors which can occur on the invocation of *LMSSetValue()* method are defined. The first two of them (lines 12-13), declare a check on the state of the API Instance. The following three lines (14-16) declare checks on the first parameter with which the method is invoked. Lastly, in lines 18 and 19, checks on the second parameter are declared. Checks are defined in the V*alidator* framework configuration file, where a class method that performs the validation is linked to it.

```
1   <APIs>
2
3      <APIset id="SCORM1.2" apiInstanceId="API">
4         <method name="LMSInitialize" type="initialize" params="1" return="false">
5            <error property="apiState" check="not_terminated" code="104"/>
6            <error property="apiState" check="not_running" code="103"/>
7         </method>
8         <method name="LMSGetValue" type="getValue" params="1">
9          ...
10        </method>
11        <method name="LMSSetValue" type="setValue" params="2">
12           <error property="apiState" check="not_initialized" code="132"/>
13           <error property="apiState" check="not_terminated" code="133"/>
14           <error property="param1" check="required" code="401"/>
15           <error property="param1" check="defined" code="401"/>
16           <error property="param1" check="implemented" code="402"/>
17           <error property="param1" check="read_only" code="404"/>
18           <error property="param2" check="type_match" code="406"/>
19           <error property="param2" check="range" code="407"/>
20        </method>
21        <method name="LMSCommit" type="commit" return ="false" params="1">
22         ...
23        </method>
24        <method name="LMSFinish" type="terminate" return ="false" params="1">
25         ...
26        </method>
27        <method name="LMSGetLastError" type="getLastError" params="0"/>
28        <method name="LMSGetErrorString" type="getErrorString" params="1"/>
29        <method name="LMSGetDiagnostic" type="getDiagnostic" params="1"/>
30     </APIset>
31
32     <APIset id="SCORM1.3" apiInstanceId="API_1484_11">
33        <method name="initialize" type="initialize" params="1" return="false">
34           <error property="apiState" check="not_terminated" code="104"/>
35           <error property="apiState" check="not_running" code="103"/>
36        </method>
37             ...
38     </APIset>
39  </APIs>
```

As done for the *API Interfaces*, a similar work was requested and performed on the *datamodels.xml* file, in order to define the data models for both the versions of the *SCORM*. The data model for *SCORM* 1.2 is defined through lines 4 to 9, while that for version 1.3 through lines 11 to 33. The changes in the names of the elements compel us to define separately elements with the same meaning. This is the case of the elements representing the identifier and the name of the student (line 6-7 for SCORM 1.2 and line 16-17 for SCORM 1.3). Lines 19 to 21 show the definition of a derived element, that is, an element which is not explicitly set, but calculated on the basis of the value of other elements. The element *cmi.comments_from_learner._count* expresses the size of the a collection containing the comments from the learner. As declared, the suitable *CountManager* class calculates this value. Lines 23 to 26 show an element whose value is initialized (and, in this case, never changed) at the time of the definition of the data model. As defined in the specifications, all the elements with the *_children* suffix can be read in order to obtain the sub-elements composing a structured data, which, in the case of the *cmi.comments_from_learner* element are *comment*, *location* and *timestamp*. Lastly, the dependency of an element from two other elements is defined in line 31.

```
1    <?xml version="1.0" encoding="UTF-8"?>
2
3    <datamodels>
4        <datamodel id="SCORM1.2">
5             ...
6            <element id="cmi.core.student_id" type="long" privilege="RO" />
7            <element id="cmi.core.student_name" type="string" privilege="RO" />
8             ...
9        </datamodel>
10
11       <datamodel id="SCORM1.3">
12           <element id="cmi._version" type="string"  privilege="RO" >
13               <value init="1.0"/>
14           </element>
15            ...
16           <element id="cmi.learner_id"  type="long" privilege="RO" />
17           <element id="cmi.learner_name"  type="string" privilege="RO" />
18            ...
19           <derived-element id="cmi.comments_from_learner._count" type="int"
20               class="org.l3.CMIFramework.client.error.utility.CountManager"
21               privilege="RO"/>
22            ...
23           <element id="cmi.comments_from_learner._children" type="string"
24                    privilege="RO">
25               <value init="comment,location,timestamp"/>
26           </element>
27            ...
28           <element id="cmi.completion_status" type="string" privilege="RW">
29               <value set="completed,incomplete,not_attempted,unknown"
30                      init="unknown"/>
31               <depends idRef="cmi.completion_threshold,cmi.progress_measure"/>
32           </element>
33       </datamodel>
34   </datamodels>
```

**Appendix B: Description and definition of *CMIService* process**

The code fragments in this section show the description, through *WSDL* code, and definition, through *BPEL* code, of *CMIService* process.

As for the description of the process, only the reusable abstract part is presented: the concrete part containing the service binding and implementation is not included as it is related to implementation details. The following document contains the definition of types, messages and port types. Among the types (lines 3-22), a *stringMap* type is defined (lines 5-14). This data type is composed of a sequence of (key, value) couples, whose values are strings, and is used to represent both a part of the run-time data (line 18) and the configuration (line 25).

The three message types exchanged between *CMI Service* and its partners (the *LMS Endpoint* and the *API Instance*) are defined in lines 24-34. The *CMIRequest* message (lines 24-27) carries both the configuration data and the just instantiated and initialized run time data. Its corresponding response (*CMIResponse* message, lines 28-30), just carries a response string. The *CMIEventNotify* message (lines 31-34) carries the event type and the whole run-time data.

The process defines two port-types: *LMSEndpointPT* (lines 36-44) and *APIInstancePT* (lines 45-47). They are both composed of two operations (*CMIRegistration* and *CMIEventNotification*), and are used to interact with the *LMS Endpoint* and the *API Instance*, respectively. The only difference between them is the inversion of the input and the output messages. E.g. the *CMIRequest* message (line 38) of the *CMIRegistration* operation is received (input) from the *LMS Endpoint* and lately forwarded (output) to the *API Instance*.

Lastly, the *WSDL* document defines the partner link types to be used in the *BPEL* process definition: *LMSEndpointPLT* (lines 49-56) and *APIInstancePLT* (lines 57-59).

```
1     <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
2
3        <wsdl:types>
4           <xsd:schema targetNamespace="...">
5              <xsd:element name="stringMap">
6                 <xsd:complexType><xsd:sequence>
7                    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
8                       <xsd:complexType><xsd:sequence>
9                          <xsd:element name="key" type="xsd:string" />
10                         <xsd:element name="value" type="xsd:string" />
11                      </xsd:sequence></xsd:complexType>
12                   </xsd:element>
13                </xsd:sequence></xsd:complexType>
14             </xsd:element>
15             <xsd:element name="RTDataType">
16                <xsd:complexType><xsd:sequence>
17                   <xsd:element name="version" type="xsd:string"/>
18                   <xsd:element name="RTData" type="xsd:stringMap"/>
19                </xsd:sequence></xsd:complexType>
20             </xsd:element>
21          </xsd:schema>
22       </wsdl:types>
23
24       <wsdl:message name="CMIRequest">
25          <wsdl:part element="tns:stringMap" name="configData" />
```

```
26          <wsdl:part element="tns:RTDataType" name="RTData" />
27       </wsdl:message>
28       <wsdl:message name="CMIResponse">
29          <wsdl:part element="xsd:string" name="response" />
30       </wsdl:message>
31       <wsdl:message name="CMIEventNotify">
32          <wsdl:part element="xsd:string" name="eventType" />
33          <wsdl:part element="tns:RTDataType" name="RTData" />
34       </wsdl:message>
35
36       <wsdl:portType name="LMSEndpointPT">
37          <wsdl:operation name="CMIRegistration">
38             <wsdl:input message="tns:CMIRequest" />
39             <wsdl:output message="tns:CMIResponse" />
40          </wsdl:operation>
41          <wsdl:operation name="CMIEventNotification">
42             <wsdl:output message="tns:CMIEventNotify" />
43          </wsdl:operation>
44       </wsdl:portType>
45       <wsdl:portType name="APIInstancePT">
46              ...
47       </wsdl:portType>
48
49       <plnk:partnerLinkType name="LMSEndpointPLT">
50          <plnk:role name="CMIService">
51             <plnk:portType name="LMSEndpointPT" />
52                </plnk:role>
53                <plnk:role name="LMSEndpoint">
54             <plnk:portType name="tns:CMIServicePT" />
55          </plnk:role>
56       </plnk:partnerLinkType>
57       <plnk:partnerLinkType name="APIInstancePLT">
60              ...
61       </plnk:partnerLinkType>
62
63     </ wsdl:definitions>
```

The *BPEL* document for the definition of the *CMI Service* process follows. The document imports the definition of the previously reported *WSDL* document (line 2). In particular, the partner link types of the *WSDL* are used to define the two partner links to interact with the *LMS Endpoint* (lines 4-5) and the *CMI Service* (lines 6-7). The declaration of the variables follows. Two variables are declared: *CMIRequestVar* (line 11), which is bound to the *CMIRequest* message and *CMIEventNotifyVar* (line 12), bound to the *CMIEventNotify* message.

The process is composed of a main sequence, defined through lines 15 to 38. The first activity is the reception of a *CMIRequest* message from the *LMS Endpoint* (line 16). On its occurrence the process is instantiated. A reply (line 19) message follows. Then, the process informs the *API Instance* of the registration. This is done by forwarding (the invoke action in line 22) the registration message to the *API Adapter*.

To handle the *CMI* communication, a *repeat-until* block (lines 25-37) is used. Inside it, there is a nested sequence of two activities. With the former (line 27), a *CMI* event is received from the *API Instance*. Through the latter (line 30), the same message is forwarded to the *LMS Endpoint*. The iteration is interrupted on the reception of a *terminate* message type (line 34).

```
<bpws:process ...>
   <bpws:import importType="CMIService.wsdl" />
   <bpws:partnerLinks>
      <bpws:partnerLink myRole="CMIService" name="LMSEndpointPL"
             partnerLinkType="cmi:LMSEndpointPLT" partnerRole="LMSEndpoint" />
      <bpws:partnerLink myRole="CMIService" name="API Instance"
             partnerLinkType="cmi:APIInstancePLT" partnerRole="APIInstance" />
   </bpws:partnerLinks>

   <bpws:variables>
      <bpws:variable messageType="CMIRequest" name="CMIRequestVar" />
      <bpws:variable messageType="CMIEventNotify" name="CMIEventNotifyVar" />
   </bpws:variables>

   <bpws:sequence name=Main Sequence">
      <bpws:receive createInstance="yes" name="Receive CMI request"
                   operation="CMIRegistration" partnerLink="LMSEndpointPL"
                   portType="LMSEndpointPT" variable="CMIRequestVar" />
      <bpws:reply name="Reply CMI response"
                   operation="CMIRegistration" partnerLink="LMSEndpointPL"
                   portType="LMSEndpointPT"/>
      <bpws:invoke inputVariable="CMIRequestVar" name="Send RT-Data"
                   operation="CMIRegistration" partnerLink="APIInstancePL"
                   portType="APIInstancePT" />
      <bpws:repeatUntil name="RepeatUntil event.type = 'terminate' ">
         <bpws:sequence name="Repeat-Until Sequence">
            <bpws:receive name="Receive CMI Event"
                 operation="CMIEventNotification" partnerLink="APIInstancePL"
                 portType="LMSEndpointPT" variable="CMIEventNotifyVar" />
            <bpws:invoke inputVariable="CMIEventNotifyVar"
                 name="Notify CMI Event" operation="CMIEventNotification"
                 partnerLink="LMSEndpointPL" portType="LMSEndpointPT" />
        </bpws:sequence>
        <bpws:condition> <![CDATA[bpws:getVariableData('CMIEventNotifyVar',
                 'eventType','//') = 'terminate']]>
        </bpws:condition>
      </bpws:repeatUntil>
  </bpws:sequence>

</ bpws:process>
```