CopperCore Service Integration


Open University of the Netherlands, Valkenburgerweg 177, 6419 AT Heerlen

This article describes a framework for the integration of e-learning services. There is a need for this type of integration in general, but the presented solution was a direct result of work done on the IMS Learning Design specification (LD). This specification relies heavily on other specifications and services. The presented architecture is described using the example of two of such services: CopperCore, an LD service and APIS, an IMS Question and Test Interoperability service. One of the design goals of the architecture was to minimize the intrusion for both the services as well as any legacy client that already uses these services.

Keywords: Service Integration, CopperCore, e-learning, IMS Learning Design, IMS Question and Test Interoperability, APIS

Introduction

This article describes the design and implementation of a generic integrative service framework, called CopperCore Service Integration (CCSI) (Vogten & Martens, 2006), for the IMS Learning Design specification (LD) (IMS, 2003). This work was done as part of the JISC ELF (Wilson, Blinco, & Rehak, 2004) (JISC E-Learning Framework: Technical Framework and Tools Strand 2006) toolkit strand project called SLeD2 (Service Based Learning Design System 2004) as a joint effort of both the Open University and the Open University of the Netherlands. The project extended earlier work which involved building an LD runtime service and a corresponding web based client application called SLeD.

The LD runtime service, called CopperCore (Martens, Vogten, Rosmalen, & Koper, 2004), processes units of learning (UOLs) which are IMS content packages containing a learning design defined in LD. CopperCore does not make any assumptions about the type of user interface used by the calling party. This allows CopperCore to be integrated in web clients as well as rich client platform applications. In fact, CopperCore does not provide any user interface at all, and all methods are only available through an Application Programming Interface (API). Therefore CopperCore cannot be used as a standalone product and must be used as a service integrated into a larger framework or Learning Management System (LMS). CopperCore relies on the provisioning of other services by this framework or LMS for parts of the LD processing.

Some of the services on which CopperCore relies are generic and may be used by other services as well. Examples of such common services are authorization and authentication. Although technically challenging, these types of services are not the focus of our work as they apply to all service oriented architectures. However, there are a number of e-learning oriented services that are tightly integrated with the LD specification that provide our focus. Typically, these can be found in the service section of the LD environment. Note the LD term service refers to the functional concept of a learning service supporting a user in the learning process. The LD term service does not refer to the technical notion of a service as in the term web service although the technical implementation of a LD service could well be achieved by a web service.

*Corresponding author. Email: hubert.vogten@ou.nl
The LD specification includes a number of services such as a mail service, synchronous and asynchronous conferencing service and an index and search service. LD also allows additional services to be specified when needed.

Furthermore LD specifies how other IMS specifications should be integrated. Examples of such specifications are the IMS Question and Test Interoperability specification (QTI) (IMS, 2006a) and the IMS Simple Sequencing specification. Although these specifications are quite clear on the authoring aspects of their integration, they are not particularly clear on their runtime aspects. An example is the integration of QTI items in the unit of learning. During runtime there must be a means of reacting to outcomes of QTI assessment items within the learning design workflow.

These implications are not well understood. The CCSI framework provides an extensible solution for the tight integration of loosely coupled services. The cross service concerns in particular are targeted by CCSI, alleviating the calling process from the burden of dealing with these concerns. In the remainder of this article the CCSI framework will be further elaborated by focusing on the integration of the CopperCore service and a QTI service which is called Assessment Provision through Interoperable Segments (APIS) (Barr, 2006). APIS is an implementation of a computer aided assessment service conforming to QTI and is also funded under the JISC ELF toolkit strand.

### Integrating IMS Learning Design and QTIv2

With the release of the second version of QTI guidelines for the integration of LD and QTI were described (IMS, 2006b). The integration of LD and QTI revolves around aligning LD properties and QTI variable names. Essentially, when property identifiers and variable names are declared to be lexically identical at design time (i.e. in LD-based and QTI-based XML), they are considered to be a shared variable in run-time software environments that involve LD and QTI-based processing.

One implementation strategy for the guidelines above could be to build an integrated system combining the functionality of both the CopperCore and APIS service. However, given the considerable efforts that have been invested in the CopperCore and APIS services, this may not be an economically viable solution. Another approach would be an adaptation of both CopperCore and APIS allowing them to directly communicate with each other. This approach has two major drawbacks. First of all this introduces undesired dependencies between services. Secondly, this solution is not scalable as each new service being integrated requires an ever growing integration effort required to support communication with all the others. In the next section the architecture for CCSI is described that has none of the above drawbacks, together with a number of benefits.

### CopperCore Service Integration Architecture

In order to make the service integration viable it is essential that the underpinning architecture is not intrusive, meaning adaptation to this architecture should only require minimal changes in the code of the existing services, like CopperCore and APIS and the existing clients using these services. Service and client implementers are unlikely to make it a priority to adapt their code solely for CCSI.

By the introduction of an intermediate service layer composed of a dispatcher and adapters we can meet the above requirements. This approach is a well known in the software industry and is described by the adapter design pattern (Gamma, Helm, Johnson, & Vlissides, 1995). The adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces. In case of CCSI, each adapter is a software component encapsulating a single service implementation. The dispatcher is the central component, responsible for the orchestration between these services. To make this orchestration possible, all adapters share a common API providing the dispatcher a standard interface to all integrated services. Each adapter implements specific code to access the underlying
service by implementing this common interface. This way the required code adaptations needed for the service integration are now encapsulated in the adapters, leaving the services untouched.

For each type of service (LD services, QTI services or conferencing services) multiple implementations may exist. In order to make these service implementations interchangeable a contract between the client and the adapter is introduced for each service type in the form of an interface. This principle is described by the bridge pattern, another well known design pattern (Gamma et al., 1995). The bridge decouples an abstraction from its implementation so that the two can vary independently. In the case of CCSI the bridge is the interface that describes the common functionality for the aforementioned service types. Adapters are allowed to extend this functionality by exposing the complete API of the underlying service implementations. Not only does this provide a richer system, it also makes the adapter transparent for any client using the original service. However, clients that make use of the extended functionality will need to be modified when another service implementation is used that does not provide this functionality.

Each interface is accompanied by an abstract adapter. Each abstract adapter implements the default hooks for the dispatcher. This alleviates the implementers of specific adapters from re-implementing these hooks over and over again. The implementations of these abstract adapters can act as proxy for the service preventing however when needed additional actions may be added by the implementations of the abstract adapters. This principle is also known as the proxy pattern (Gamma et al., 1995).

Figure 1 depicts the CCSI architecture. The Dispatchers most important role is the propagation of events through all defined adapters. It is the responsibility of the adapters to listen for these events. Vice versa, it is the responsibility of each adapter to trigger the Dispatcher when an event occurs that has potential cross service repercussions.

The Dispatcher is also responsible for returning an adapter of the requested type to the client, thereby acting as an adapter factory corresponding to the abstract factory pattern (Gamma et al., 1995). This adapter factory is necessary because the types and implementation of the adapters are not known in advance, and may vary even during deployment by simply adding or replacing adapters. Adapters can come in two flavors depending on the way the client wishes to access the adapter. This can be done either via native Java calls or via SOAP...
web services. All adapters are declared in the CCSI service definition file. This file contains information about the base service type, the implementing Java class and WSDL URL. Furthermore Figure. 1 depicts two adapter types; an adapter for the LD service and an adapter for the QTI service. Note that there could have been additional adapters for other services as well. The common interfaces for these service types are defined by the interfaces ILDAdapter and IQTIAdapter. Each adapter must implement the interface for its base type. The figure also shows two abstract classes: LDAdapter and QTIAdapter These classes implement the hooks for the Dispatcher. They act as extension points for any implementation of the LD or QTI services. Both the CopperCoreAdapter and the APISAdapter provide an interface that can be used by client applications. This interface is a replication of the original interface provided by the service that is being integrated, hence the dependency relationship between ICopperCoreAdapter and ICopperCoreService and between IAPISAdapter and IAPISService. By maintaining this relationship between the interfaces the impact for existing clients migrating to CCSI is limited to a minimum. Vice versa, when a service implementation is modified the impact is limited to the adapter acting as proxy for that particular service.

Figure. 2. Sequence diagram showing the processing of a QTI item and the resulting event handling by the dispatcher.

Figure. 2 depicts a sequence diagram representing the processing of a QTI item within the context of a UOL run. The client (e.g. SLeD) creates a new instance of the Dispatcher. The Dispatcher reads the CCSI service definition file and is informed about all available adapters. In the case of the example we only have the CopperCoreAdapter and the APISAdapter. Next, the client will request a handle for an LDAdapter. Depending on the technology used, an instance of the CopperCore adapter or a URL to the WSDL of the CopperCore adapter is returned. The Dispatcher provides the client with an identical API in the CopperCoreAdapter compared to the original CopperCore service. So legacy clients, like SLeD, only have to be slightly modified. At some stage in the process the client retrieves QTI content and reacts by requesting the Dispatcher to provide a handle to a QTI adapter. In our example the handle for the APIS adapter is returned. The client makes a request for the rendered content of the QTI item to the APIS adapter. The user response to this item is passed on to the APIS adapter. The APIS adapter processes this response, which results in a change of one of the variables defined by the QTI item’s response section. It is the responsibility of the
QTIAdapter to notify the Dispatcher about this property event. In turn the Dispatcher propagates this event to all adapters that have registered themselves as listeners for this event type allowing them to react to this event. In order to synchronize the value of the QTI outcome variable, a corresponding LD property needs to be defined in the UOL. The CopperCoreAdapter will verify if this property exists and if so the value of the LD property will be set to the value of the QTI outcome. After all adapters have been informed about the property event, the result of the APIS adapter is finally returned to the client.

![Two consecutive screenshots of the SLeD client are captured while processing an UOL containing a QTI item. The scenario is very similar to the one elaborated in the UML sequence diagram in Figure 2. The first screen shows the QTI item being rendered by SLeD. After the user has entered the answer, the response is visible in the second screen. Notice that CopperCore was informed about this change and provided the user with a new activity to perform.](image)

**Figure. 3.** Two consecutive screenshots of the SLeD client are captured while processing an UOL containing a QTI item. The scenario is very similar to the one elaborated in the UML sequence diagram in Figure 2. The first screen shows the QTI item being rendered by SLeD. After the user has entered the answer, the response is visible in the second screen. Notice that CopperCore was informed about this change and provided the user with a new activity to perform.

**Integration of other Services**

CCSI was developed with the integration of different kind of services in mind, especially those defined in the service section of LD although other types of services are conceivable too. In fact, in SLeD2 a number of adapters for these services were developed such as a search adapter and a forum adapter. The principle of integration is exactly the same as was done for the QTI adapter. However the type of events that are dispatched may differ per adapter type. For example, for the forum adapter it is relevant to be informed about new runs (Tattersall et al., 2005) being created for a UOL. A run is a runtime instantiation of a UOL and involves the enrollment of individual users to the defined roles in the UOL thereby populating the UOL. Similarly, it is relevant for the forum adapter to be informed about user subscriptions and role changes within the run of a UOL. The events are generated by the CopperCore adapter and can be picked up by any forum adapter. The forum adapter implementations may react to these events by creating new topics and granting users the corresponding access rights. Momentary two asynchronous forum adapters are developed, one for
Moodle and one for Knowledge Network a proprietary system of the Open University, which can be switched by merely changing the deployment configuration.

Another example of a service currently integrated through CCSI is the search service. Like with the forum adapter, there are two adapter implementations available. One adapter uses the Google API as search service provider. The other adapter uses the aforementioned Knowledge Network as service provider.

**Related Work**

In the field of learning service integration some interesting related work has emerged. The IMS Tools Interoperability Guidelines (TIG) (IMS, 2006d) is worth mentioning here. TIG deals with the interoperability of tools and LMS and is a first attempt to any standardization in this area. It shows some resemblance to the solution presented in this paper although there is a significant difference. The focus of SIG is mainly on technical aspects of the integration and less on the functional integration of the different services. TIG will not deal with any functional inter service dependencies, like the orchestration of property values between services, as shown in our example.

Another interesting, closely related development is the Business Process Execution Language (BPEL) (IBM, BEA Systems, Microsoft, SAP AG, & Siebel Systems, 2006) for Web Services. BPEL primary focus is the orchestration of SOAP web services. All logic for this orchestration is declared in an XML file which is interpreted by a BPEL engine. Recently tools for BPEL, like engines and editors have become widely available. Assis (Sherrat & Jeyes, 2006), also a JISC funded project, is worth mentioning in this context. In the Assis project BPEL was used for the orchestration of web services handling IMS Simple Sequencing (IMS, 2006c) and QTI.

BPEL holds some promising advantages over the presented approach in the paper, like standardization of the workflow solution and the separation of the workflow description from the actual implementation. However, at the same time extra overhead and complexity is introduced by the use of BPEL and one can argue if this outweighs the simplicity of CCSI. Especially in cases where services are not SOAP compliant the approach taken by CCSI has the advantage that it can make use of these services directly. This advantage should not be underestimated as the battle between SOAP and its light weight counterpart ReST (Fielding, 2006) is not yet decided (zur Muehlen, Nickerson, & Swenson, 2004) either way.

**Conclusion**

Interoperability specifications like LD and QTI are having an ever growing impact on the e-learning community. As a result the number of implementations is steadily growing; initiatives such as the JISC ELF have demonstrated this via the delivery of several services dealing with these specifications (e.g. APIS and CopperCore). However at the same time, runtime inter-specification operability issues are not yet understood. In this article, an approach was presented that deals with the interoperability of e-learning services within the context of LD. As basis for presenting the CCSI solution two service implementations were chosen; CopperCore and APIS. The need for integrating these two components can be explained by the fact that QTI is a natural complement to LD. Both CopperCore and APIS were independently developed as part of the JISC ELF and both are already being used by legacy systems. The latter introduced an additional requirement as the identified solution must deal with legacy services and legacy clients. The switch to the new architecture should cause minimal intrusions in any existing code. Furthermore, the provided solution should be robust for new developments as the integrated services have their own development dynamics.

The CCSI architecture, informed by a number of design patterns, deals with these requirements by seamlessly inserting itself between the service and client. By replicating the original API the consequences for the client...
are limited to a switch of services factory. The underlying services do not have to be changed at all. All inter-service issues are dealt with in the adapter and dispatcher. We have seen that there is an adapter for each service type and that an adapter has a contract enforced by an interface per service type. The latter concept makes the adapter robust for changes in the services; it makes it possible to completely switch service implementations with minimal consequences.

Finally, as highlighted above the CCSI architecture is not limited to the integration of CopperCore and APIS. Other services such as the forum and search service can and in fact have already been integrated in a very similar manner although the types of events are different. For these services multiple adapter implementations exist which can be interchanged without any need to change any client code. The work on CCSI will be taken up by the European Commission funded TENCompetence (TENCompetence2006) programme. All code for CCSI is available as open source and may be downloaded from SourceForge at http://sf.net/projects/ccsi. For an easy up and running example of CCSI the CopperCore Runtime Environment, also known as CCRT, can be downloaded from http://coppercore.org. This runtime contains deployable versions of the CopperCore service, the APIS service and the CCSI integrative service. Additionally, the SLeD2 player downloaded from http://sourceforge.net/projects/lldplayer. Finally, the example UOL can be downloaded from http://dspace.ou.nl/handle/1820/555.

References


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley.


IMS. (2006b). *IMS Question and Test Interoperability Integration Guide*. Retrieved [http://www.imsglobal.org/question/qti_v2p0/imsqti_intgv2p0.html](http://www.imsglobal.org/question/qti_v2p0/imsqti_intgv2p0.html)


zur Muehlen, M., Nickerson, J. V., & Swenson, K. D. (2004). Developing Web Service Choreography Standards; The case of REST vs. SOAP. *Decision Support Systems, 37*