In the Eye of the SCORM

An introduction to SCORM 2004 for Content Developers

Claude Ostyn

For the most recent version of this book, see www.ostyn.com/resources.htm

Update 0.9-8.8– March 2007

Abstract

This book explains SCORM 2004 to content developers. It contains an overview of SCORM, as well as practical examples. The function and structure of a package manifest are described. Various aspects of the behavior of a shareable content object (SCO), and in particular how a SCO communicates with a runtime environment, are explained with working examples. A template for a manifest is provided, and is used to explain the process of assembling a basic SCORM package. An introduction to sequencing and navigation is also included.

Copyright

Copyright © 2005, 2006, 2007 Ostyn Consulting. All rights reserved.

License

Unless otherwise expressly stated, all original material of whatever nature created by Claude Ostyn and included in this document and associated software samples is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Commercial licensing terms are available for implementers who cannot abide by the Attribution or ShareAlike provisions of the Creative Common License. For additional information or questions regarding copyright, commercial use, distribution and reproduction, contact:

Ostyn Consulting, PO Box 2362, Kirkland, WA 98083-2362, USA

Representations, Warranties and Disclaimer

OSTYN CONSULTING OFFERS THIS WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE.

Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL OSTYN CONSULTING OR CLAUDE OSTYN BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THE USE OF THIS WORK, EVEN IF OSTYN CONSULTING OR CLAUDE OSTYN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Trademarks

Any trademarks or service marks used in this document are the property of their respective owners.

Table of Content

Chapter 1 - Introduction

Who should read this book

This book is intended for content developers, for developers of authoring tools for SCORM content, and for anyone who has to manage or deal with SCORM content. It assumes no prior knowledge of SCORM, but it assumes that the reader has some basic understanding of things like file and directory structures, and of how web content is delivered.

This book is intended to provide conceptual overviews and examples. It is not intended to provide a complete reference or to replace the SCORM specifications.

If your interest is not technical, you can stop reading before reaching the final chapters, which include some detailed technical examples to illustrate various SCORM concepts.

Specialized terms and acronyms are unavoidable. The glossary on page 91 may be useful if you encounter a mysterious term or acronym.

How to use this book

This book contains both conceptual overviews and technical overviews. The technical examples are, by their very nature, written using programming languages that are not really fit for normal human beings, but I tried to make them understandable even for people with only a passing acquaintance with JavaScript and XML.

If you want to use this book for a SCORM project, or if it seems hard to understand, recommend that you print out the ADL SCORM 2004 documents or otherwise keep them available for reference when reading this book. Each SCORM book begins with a useful conceptual introduction. In various places, this book will paraphrase what is in the ADL documents. This is by design. Often, viewing the same dense information from different angles can make it easier to understand.

If you want to try the provided examples, you do not need an authoring tool. All you need is a text editor like Notepad.

Acknowledgements

This document would not exist without the relentless truth seekers, implementers, critics and the "usual suspects" who have been raising all kinds of questions about learning objects, interoperability, content packaging, metadata and related matters over the last few years. It would also not exist without the diligence of the ADL and the SCORM technical team and technical working group for providing a workable specification that finally allows content to interoperate reliably so we can all focus on more interesting things, like performance and the human side of learning processes. Humble thanks also to all those who generously shared their research and wisdom about why and how we learn, with or without technology. Special thanks to Steve Alessi, Corrie Bergeron, Bill Blackmon, Jennifer Brooks, Judy Brown, Andrew Chemey, Philip Dodds, Erik Duval, Dexter Fletcher, Jason Haag, Wayne Hodgins, Peter Hope, Jack Hyde, Tom King, John Kleeman, M. David Merrill, Boyd Nielsen, Angelo Panar, Nina Deibler, Frank Polster, Tom Reeves, Daniel Rehak, Jeff Rhodes, Tyde Richards, Eric Roberts, Robby Robson, Eric Rosen, Harvey Singh, Roger St-Pierre, Schawn Thropp, Stanley Trollip, Jeff Webb, Eamonn Webster, Ian Wright.

...as becomes the ignorant, I must learn from the wise...

Plato, *The Republic*

Chapter 2 - A short history of the SCORM

The basic problem

For many years, one of the major problems with e-learning has been the creation and deployment of quality e-learning content. E-learning content is actually software. Its development has typically been subject to the same issues as other software projects. In other words, e-learning content is expensive and time-consuming to develop.

To deliver and especially track results from the use of the content, it typically had to be custom programmed to work in a specific delivery environment. Different learning management systems had very different delivery environments. If an enterprise wanted to upgrade a learning management system or change vendors, often that meant abandoning very expensive content and starting over. If a content vendor wanted to distribute content widely, it was very expensive. A completely different version was often required to accommodate each different learning management system. Big content vendors, on the other hand, specified their own delivery environment and forced each learning management system to implement different delivery modules for each large content vendor.

As learning requirements changed and overlapped within enterprises and government agencies, the importance of reusable content modules became apparent. In education as well, the usefulness of content objects or "content nuggets" to support all kinds of learning activities was demonstrated by various projects.

There has thus been a strong market incentive for content that is durable, portable between systems and reusable in a modular fashion. In other words, content that is interoperable.

With interoperable content, content developers win because the same content can work in more different systems without modification. LMS vendors win because they can focus on the management aspects of learning, without having to constantly adapt the delivery environment to various content libraries. Enterprises and agencies using the learning management systems and the content win because instead of wasting money and time on integration of different libraries of content, they can easily mix off the shelf content with their own custom content using the same delivery environment.

Precursors of the SCORM

AICC

A long time ago, when the delivery environments were still mostly DOS machines and Windows was still vaporware, the Aviation Industry Computer-Based Training Committee (AICC) published guidelines and recommendations for interoperable content. The AICC has since been updating these specifications to work in a web based environment, sometimes with mixed results because of an ongoing concern for legacy content and systems. The AICC specifications did not provide a way to guarantee cataloguing metadata, or a robust way to package the content to make it portable. However, good lessons were learned from those early efforts and contributions.

IMS Global Learning Consortium

New useful specifications have emerged from other communities of practice, such as the Content Packaging specification and Content object Metadata profile from the IMS Global Learning Consortium.

IEEE

An accredited international standards organization, the IEEE has been busy turning specifications into high quality standards. The standards initiatives often cooperate. For example, the IMS metadata specification was based on an early draft of the IEEE metadata standard, and other IEEE standards incorporate some of the functionality first contributed by the AICC.

Enter the ADL

Several years ago, the White House Office of Technology, the Department of Defense and the Department of Labor launched the Advanced Distributed Learning Initiative (ADL) in the United States. Industry and education partners, as well as various entities in other countries also joined the effort. One of the first ADL projects was for a practical profile of existing specifications and standards for content. A profile is a document that specifies a particular interpretation of a standard or specification. In the process, some gaps had to be filled in, and the end result was a set of "books", each describing a different aspect of the solution. The result was called the Shareable Content Object Reference Model, or SCORM.

The SCORM was born to take the best from the early efforts, specifications and standards, and achieve the goals of durability, portability, reusability, interoperability and accessibility for content.

Many people from the e-learning industry were involved in the genesis of the SCORM, along with the technical team funded by ADL. Each release of the SCORM has been tested in "Plugfest" events. A Plugfest is a meeting where competing vendors and developers get together to verify that their content and implementations interoperate as expected, and to iron out problems in a cooperative atmosphere. Plugfest events have attracted a very wide international participation, which is a clear indication of the impact and adoption of the SCORM well beyond its North American origins.

SCORM 1.1

The first release of the SCORM was a trial balloon, intended to discover unresolved issues. Test bed implementations revealed that SCORM 1.1 was less than fully functional, and interoperability was still mostly hit and miss. The lessons from SCORM 1.1 were put to good use in the subsequent releases.

SCORM 1.2

The first "real" release of the SCORM was SCORM 1.2. This was the first version for which a test suite was available, and thus the first version for which conformance could be verified. SCORM 1.2 proved that content can be made portable and interoperable. For example, one leading LMS vendor has seen the cost and time of new content integration with their SCORM conformant drop to almost nothing when the content was verifiably SCORM conformant. Any remaining issue was a result of not conforming to SCORM, or the result of a misinterpretation of a SCORM feature that was still not quite pinned down completely.

The current version: SCORM 2004

SCORM 2004 improves significantly on SCORM 1.2, by eliminating even more ambiguities in the specification, and by making SCORM conformant with robust IEEE standards. The API now supports the wide range of human languages supported by ECMAScript. Besides improving on SCORM 1.2, SCORM 2004 also adds optional features for sequencing and navigation.

The addition of sequencing is a major functional milestone. SCORM 1.2 was all about making content portable, but left it to the learner to choose which part of the content to run. SCORM 2004 adds the ability to deliver activity-centered content packages that support guided or adaptive sequencing behavior.

Compatibility between versions

Many content Learning Management System vendors will probably continue to support SCORM 1.2 content for a long time, along with SCORM 2004.

Tools are available or can be built relatively easily to convert SCORM 1.2 content packages to SCORM 2004.

It is possible to launch unmodified SCORM 1.2 content objects in a SCORM 2004 environment by using a "wrapper" provided by the ADL.

It is also possible to launch SCORM 2004 content objects in a SCORM 1.2 environment through such a "wrapper". However, in that case, and depending on the content, some tracking or session data may be lost because SCORM 1.2 does not support the full IEEE data model used by SCORM 2004. Obviously, SCORM 1.2 environments do not support SCORM 2004 sequencing.

Another approach is to create content objects that can work in either SCORM 1.2 or SCORM 2004, with graceful degradation if the environment is SCORM 1.2. This approach is of course more expensive.

This book focuses on SCORM 2004 because this is the current version. There is little point in fighting the old battles of SCORM 1.2 in new implementations.

Compatibility with other specifications

Since SCORM is defines profile of IEEE standards and IMS specifications, SCORM content packages conform to those standards or specifications. However, packages that comply with those standards or specifications are not necessarily SCORM compliant. For example, an IMS compliant package that does not include the SCORM extension elements in its manifest is not SCORM conformant.

Although AICC has a long term plan to conform to the same IEEE standards as SCORM, most current AICC content is not SCORM conformant. For one thing, the AICC Guidelines and Recommendations do not include a packaging specification that supports the inventory of all components of a package, as provided by the IMS content packaging manifest. Also, most AICC compliant content uses an older, non-standard AICC defined communication protocol (called HACP by AICC), which is not compatible with the IEEE standard used by SCORM. Unlike SCORM, the AICC's HACP protocol is not compatible with offline delivery of content, because it requires an active web server component. However, some newer AICC content that uses the so-called "JavaScript" protocol can often be adapted and repackaged to work in a SCORM delivery environment.

Chapter 3 - Overview of the SCORM

What SCORM means

SCORM is an acronym for Shareable Content Object Reference Model.

The SCORM specifies a framework for content that meets the following requirements for e-learning content:

- Durability – Content should last long enough to amortize its cost, and be usable as long as it is relevant.
- Portability – It should be possible to move the content easily from one delivery environment to another. The same content should work without modification in different delivery environments, as long as the delivery environment includes a web browser.
- Reusability – It should be possible to build the content in small, reusable modules that can be recombined in different ways. Different communities of practice should be able to share reusable content.
- Interoperability – The same content should work the same way when it is deployed in different environments.
- Accessibility – It must be possible to find the content in a repository. This requires that some standard cataloguing data be associated with the content.

The SCORM also defines minimum conformance requirements for systems that can deliver such content to a learner.

The SCORM consists of several "books", each of which specifies some technical aspects of shareable content. Some software is provided along with the books to verify conformance and to allow demonstrations of some of the SCORM functionality.

SCORM conformance does not automatically mean you have a usable system. It would be like saying that fuel makes a transportation system. But SCORM conformance is a powerful enabler. It was designed to work behind the scenes, unseen by users. The real payoff of SCORM is in the applications it enables, and removing some of the major cost and time barriers to content integration in learning management systems or performance support systems.

What the SCORM specifies

Aggregations of content objects for portability

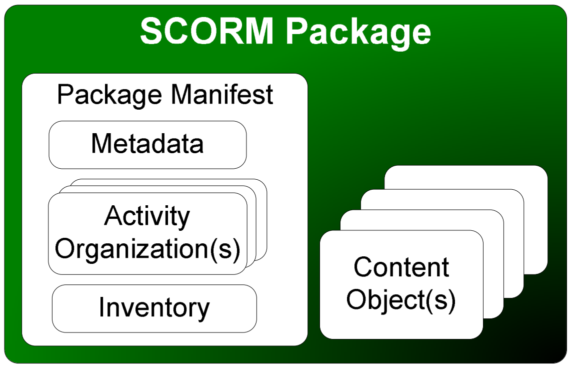


Figure 1 - SCORM Package

The SCORM specifies how reusable web-based content objects can be aggregated into a portable package that includes a manifest to form a larger self-contained content object.

A SCORM manifest provides a detailed description of the content of the SCORM package, as well as some prescriptions for the use of the package. The prescription typically specifies activities and sub-activities that use the content objects in the package. The manifest also includes descriptive metadata. It is conceptually similar to the shipping manifest used when one ships physical goods in one or more boxes.

A SCORM package can exist as a persistent package, which is never modified after publication, or it can be assembled and customized on the fly by an automated system for a particular individual. The SCORM does not specify the process by which a package is aggregated, but it specifies the result of such a process.

Launching and tracking of content objects in a package

A runtime environment (RTE) must be used to launch the individual content objects in a SCORM conformant package. The runtime environment is typically provided by a LMS, a performance support system, or a competency management system. The learner interacts with the runtime environment and the web content through a standard web browser with JavaScript enabled.

The runtime environment is completely independent of the content. However, some parts of it must be constructed in a particular way so that some of the content objects will be able to exchange data with the runtime environment. Typically, the runtime environment is split across a network connection, with parts of it on a server and part of it running in the user's browser.

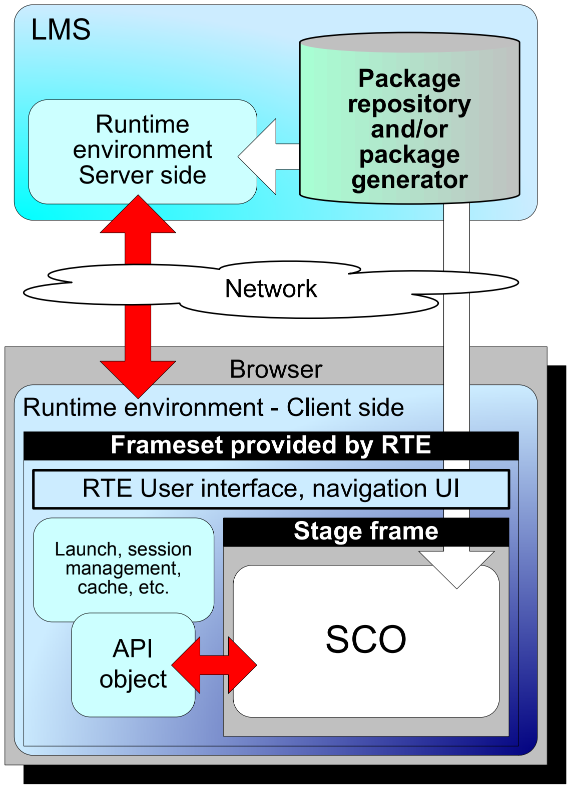

LMS
Runtime environment Server side
Package repository and/or package generator
Network
Browser
Runtime environment - Client side
Frameset provided by RTE
RTE User interface, navigation UI
Launch, session management, cache, etc.
Stage frame
API object
SCO

Figure 2 – SCORM Runtime environment

The content objects that can exchange data with a SCORM conformant runtime environment are called Shareable Content Objects (SCOs). The runtime environment launches the SCOs one at a time, according to a particular activity prescription included in the package. Unless the activity prescription forbids it, the user can also navigate from SCO to SCO through controls provided in the runtime environment's user interface.

The SCORM specifies in detail how a SCO must behave within the runtime environment:

The SCO must establish a communication session with the runtime environment, and there is a standard set of data elements that the SCO can use during the communication session. This includes tracking data that allows the SCO to report success and progress, as well as other information about the status of content objectives, results of interactions, and so on.

Offline content delivery

The SCORM does not specify that a web server is required. It only specifies that the runtime environment must be able to launch the SCOs in a web browser, and that a SCO must be able to find an API object in another browser window that is related to the launch window in a particular way. The SCO may actually come from a remote server, from a local server or from the local file system. For example, it is possible to deliver the same SCORM conformant content on a CD-ROM or through a web based learning management system. Of course, offline delivery through a CD-ROM would require the installation of an appropriate player on the delivery system if it is necessary to track SCORM data in a persistent way. For example, an offline SCORM player might be a native Windows application that uses the Internet Explorer object built into Windows to display the SCOs, and that can synchronize data with a LMS when a connection is available.

Content aggregation package vs. content resource package

The SCORM defines two kinds of packages.

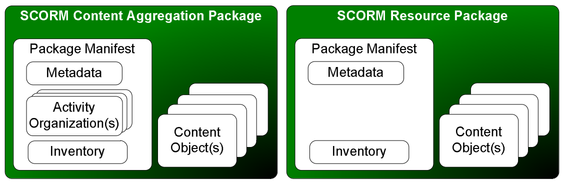


Figure 3 - Two kinds of SCORM packages

Content aggregation package

The most common kind of package is intended for delivery to a learner. In this kind of package, a special section of the manifest describes how the content objects are organized for delivery. This prescription takes the form of a manifest element named organization. An organization element defines a tree of activities and sub-activities that use the content objects.

This is the kind of package this book will focus on.

Content resource package

Another kind of SCORM package does not include any organization information. It is not intended for delivery to a learner. Rather, it is used to move amorphous collections of content objects from one system to another, or to archive a collection of content objects.

This book does not describe that kind of package, because no interesting behavior can be associated with it.

Adaptive sequencing behaviors for activities

By default, there is no sequencing information associated with the activity tree. In that case, a runtime environment must show all the activities and let the learner choose what to do.

However, the creator of a package may add sequencing rules to the activity tree to prescribe guided flows through the content, adaptive sequencing and other navigation options. The SCORM specifies how to add and implement those rules. It also specifies how the tracking data reported by content objects when they are used can affect adaptive sequencing.

What the SCORM does not specify

How to design learning content

The SCORM is neutral when it comes to pedagogy. Many of the readily available SCORM examples have been based on very traditional programmed instruction models. This is due more to a failure of imagination than to intrinsic constraints or prescriptions of the SCORM.

Look and feel

The SCORM does not specify what content should look like, what a runtime environment looks like, and in particular what the user interface for navigation between SCOs looks like. It does however assume that certain navigation facilities will be available.

What to do with tracking data

The SCORM does not specify how a LMS uses and reports tracking data collected while running SCORM content.

Granularity of SCOs and other content objects

The SCORM does not specify a particular granularity, size or duration for SCOs and other content objects. One SCO can be arbitrarily large and take several days to get through, while another SCO could be a single item in a test.

Some communities of practice are fixated on specific levels of granularity. Others allow total flexibility. For example, a policy might specify that each SCO should correspond to an enabling content objective in a particular training model. However, the SCORM makes no such assumption. Interoperability with content from different sources may be compromised if the granularity policies are too rigid.

You do not have to learn all of SCORM to use SCORM

A SCORM content developer should not be concerned about how the runtime environment is built. The LMS vendor or developer will provide the runtime environment. The SCORM was designed to put most of the complexity burden on the runtime environment. This means that there are few requirements on the content. The assumption is that there will be many more pieces of content than there will be runtime environments. Therefore it makes sense to make the content lightweight and delegate the heavy lifting to the shared environments.

For example, the runtime environment must exchange data with the LMS across the network or the Internet, which requires complicated protocols, careful timing, and advanced error management. On the other hand, all a SCO has to do is make simple JavaScript calls to the runtime environment to get or send data.

SCORM content and objective tracking

SCORM 2004 enables tracking of status for learning objectives associated with the content, using globally unique identifiers. A LMS may use this data according to policies that are outside the scope of the SCORM. For example, this data could be used for various learning management purposes, possibly in connection with reusable competency definitions. The IMS RDCEO (Reusable Definition of Competency or Educational Objective) specification describes how to create reusable competency definitions. This specification is the base for the IEEE P1484.20 Reusable Competency Definitions standard project.

Basically, a reusable competency definition describes the part of competency data that can be reused for more than one learner and more than one context. This can just be a summary title, for example, "knows how to tie a shoelace", or it may be a very detailed specification for a skill, knowledge or ability, or for a learning objective that corresponds to a skill or specific knowledge. Whatever the content of a reusable definition is, its unique identifier can be used to reference it. For example, metadata for a learning object can include the identifier of a reusable competency definition that describes the intended learning outcome for the learning object.

The identifiers for objectives specified in SCORM compliant content could match the identifiers of such competency definitions. This would in turn allow the LMS to record the success of a particular learner in mastering the objective described by a particular reusable competency definition. This information could then be used again later if the learner attempts another learning activity that involves the same objective, to provide a personalized learning experience.

SCORM 2004 sequencing can also use the objective status information to control sequencing rules. This means that a LMS could preset the status information for objectives referenced in SCORM sequencing rules, based on prior experience of the learner, to personalize the sequencing. For example, a learner might be allowed to skip topics already mastered in a completely different course. Note that the setting and use of objective status information is not currently defined in the SCORM beyond the scope of what happens during the delivery of SCORM packages. Therefore whether and how it is implemented will vary from LMS to LMS. This is however an exciting potential opportunity that leverages the standard features of the SCORM.

Chapter 4 - Anatomy of a SCORM package

Overview

A SCORM package is basically a collection of files, with a manifest that describes how the files fit together and how to deliver them, and with metadata that can be used to describe the package in a catalog.

Where in the SCORM?

The SCORM Content Aggregation Model (CAM) document defines the components of a SCORM package. The SCORM Conformance Requirements document specifies in detail how to verify that a package is conformant.

Directory structure

The files in a SCORM packages must all be in a directory structure under a single root directory. Nothing prevents a content developer from putting all the files in the same directory. Otherwise, the directory structure may be arbitrarily deep, and it can be constructed in any way a content developer desires.

When the content is deployed for delivery on a web site, the files are placed in a physical or virtual directory structure that exactly mirrors the directory structure of the original package. However, the root may be anywhere within a physical or virtual file system. This means, of course, that any links to other files within the package must be relative. For example, a link like `<img src="/pictures/img1.gif">` in a web page will not work, because the root directory for the package will almost certainly not be the root of the web site's virtual path structure.

Manifest

The manifest is a file that resides in the root directory of the package's directory structure. The manifest is an XML file that contains metadata about the package, organization structures that describe the structure of the content, and an inventory of the content resources in the package.

To allow verification that the manifest file is valid according to the schemas defined by SCORM, copies of the standard XML schema files must also be included in the root directory of the package. These copies may not be modified.

The name of the manifest file is always "`imsmanifest.xml`".

Metadata

The first element inside a SCORM manifest is named "metadata". Metadata means "data about data" – in this case data about the manifest itself and about the package.

The manifest metadata element includes metadata that identify the manifest as a content packaging manifest built to conform for SCORM 2004. It should also include descriptive and administrative metadata according that conform to the IEEE Standard 1484.12.1 for Content object Metadata.

For convenience, the SCORM allows the IEEE conformant metadata to be provided as a separate XML file rather than included in the manifest itself. In that case, the metadata element contains a reference to that other file rather than the actual metadata.

When a package is imported into a LMS or other repository, some cataloguing information must be provided to find the package in the repository. This can be automated for a SCORM compliant package. The importing system can inspect the manifest and mine the metadata for the needed cataloguing information.

In order to enable systematic cataloguing and discovery of SCORM packages, the SCORM requires that some specific metadata elements be provided, such as title, rights, and so on. These elements are specified and described in detail the SCORM Content Aggregation Model book.

Sometimes, it is desirable to document some aspects of various components of the package. To allow this, metadata can also be included with many of the other elements in an XML manifest. This is however strictly optional, and one should weigh the advantages of fully documenting every part of the manifest with metadata against the resulting file size.

Organizations

The manifest in a package intended for delivery must contain some prescriptive information as to how to deliver the content of the package for active use by a user. At a minimum, a user must be able to browse the content, and this normally requires some information about how the content objects in the package are organized.

The manifest must contain at least one organization element. The organization contains one or more activities that can be nested to any depth as sub-activities. This tree of activities represents the structure of the content, as the package author intends it to be delivered.

Each activity has a title. The title will typically be used if the package structure is shown in a table of content. The title may also be used in other places, like reports that show the status of the activity.

Each activity in the tree is either the "parent activity" in a cluster of sub-activities, or a leaf activity with no children.

Leaf activities reference a content object which is used when the activity is started. When the package is delivered to a user and the user chooses to "run" a leaf activity, the corresponding content object is launched. A leaf activity may reference only a single content object. Some parameters to be passed to the content object can also be specified for the activity.

Resources

A manifest contains a list of resource elements. Each resource element describes a content object. Typically, a resource element contains a list of one or more files required to deliver the resource. A SCO is always represented by such a resource element.

Resources are either "launchable" or not. A launchable resource has an attribute named "`href`" whose value is a URL. This URL is used to launch the content object. For example, a resource may contain several HTML files. The URL specifies which one to launch, and may also contain some launch parameters.

A "non-launchable" resource is just a container for a list of shared files used by one or more other resources. Since it will never be launched, it has no URL attribute. The leaf activities in an organization may only reference launchable resources.

The SCORM defines are two types of launchable resources: `sco` or `asset`. A SCO is a content object that will use the SCORM API to interact with the runtime environment when it is launched and while it is running. An asset is a content object that will not use the SCORM API but that can still be used for an activity. For example, it might be a text document or an image.

A resource is launched when an activity that references that resource is started. Multiple activities can reference the same resource.

Different activities that reference the same SCO may have different parameters that will be passed to the SCO when it is launched on behalf of those items. For example, one activity might use a SCO, telling it to use one level of difficulty, while another activity might use the same SCO, but specify a different level of difficulty.

Sequencing rules

The author of a package may add sequencing rules to the activity organization. This is entirely optional. If no rules are specified, there is a default sequencing behavior, which is that the user gets to choose any activity at will.

If sequencing rules are defined, the package should only be delivered in a runtime environment that supports SCORM sequencing and navigation. If no sequencing rules are defined, the package can be delivered as intended in any runtime environment that does not implement SCORM sequencing and navigation, but that allows the user to choose any activity at will.

When a SCO is launched, it typically provides tracking data. The tracking data can in turn influence the result of sequencing rules. For example, a passing score for a SCO may result in skipping some other activity.

The rules are associated with individual activities, at any level of the activity tree. Sometimes, a "cluster activity" is created just for the purpose of defining sequencing rules for a part of the content. For example, a lesson may contain a fixed sequence, followed by an exploration phase in which the learner may choose between a variety of activities, followed by a check on learning. These different behaviors within the same lesson can be implemented as sub-activities within the lesson, and each of those sub-activities may in turn specify whether and how to sequence its sub-activities. Sequencing will be described in more detail in a later chapter.

Where in the SCORM?

The SCORM Content Aggregation Model (CAM) document only specifies how the resources are aggregated with one or more activity trees.

The SCORM Runtime Environment (RTE) document defines how a content object is launched, and how it can communicate tracking data to the runtime environment.

The SCORM Sequencing and Navigation (SN) document defines which sequencing rules can be added, and the behaviors that should occur at runtime when sequencing rules exist in the package.

The SCORM Conformance Requirements document spells out in detail how to conform to the specifications defined in the other documents.

Errata documents are published occasionally between updates of the main SCORM documents.

Sub-manifests

There are currently some serious open issues regarding sub-manifests, and the ADL Technical Working Group decided in August 2005 that use of sub-manifests should be avoided until those issues are resolved by the ADL technical team. In the meantime, this brief introduction may be helpful since you will encounter the topic of sub-manifests when you read the specification documents.

Sometimes a package can become so complex that it is useful to break the manifest into smaller parts. Or sometimes it is necessary to aggregate several packages into a larger package, but it is not practical to analyze all the components of each package to create the optimal manifest. In other cases, a chunk of activity tree may be reused in more than one place in a larger activity tree. This is where sub-manifests come in handy.

A sub manifest is basically a manifest included in another package manifest. It describes a "package within a package". Each sub-manifest has its own metadata, at least one organization, and a collection of resources.

Instead of referencing a resource, an activity may reference a sub-manifest. The activity tree is then extended into the activity tree defined by the default organization in the sub-manifest.

Shared resources

Sharing resources within a package

The content package manifest model provides an efficient way to share content resources within the same package. For example, the same script or graphic may be used by several content objects described by manifest resources.

Sharing resources between packages

However, at present the SCORM does not support sharing of resources between packages. For example, it is sometimes desirable to use the same script file or company logo file by reference in all the packages produced by a particular publisher. There is at present no interoperable way to share such a file between packages, because the SCORM does not specify how repositories should function. In particular, the SCORM does not specify where the files for a content package should be installed in a repository. For example, a repository may typically use a physical or virtual directory structure and security policies that make it illegal for content in a package to access anything outside the scope of the package's own directory structure. The SCORM also does not address the unavoidable security and version control issues that may arise with content assets shared between packages that may be installed at different times and come from different sources. For example, a change to a shared script may break some dependent packages unless validation and testing processes and policies are in place; this is far beyond the scope of SCORM at this time.

A first step in the direction of addressing this issue is the ADL CORDRA project, which defines an architecture that uses a system of abstract handles to identify objects registered in repositories. A side benefit of this project is that it will allow the use of handles instead of URLs in content asset references. When a resolution mechanism is provided, the handle can be dynamically translated to the URL to use to access the asset. For example, in an offline or firewall protection scenario the handle might be translated to a URL into a local repository, while in other scenario the handle is translated to the URL for the closest Internet location of the content asset, wherever this may be.

Summary of SCORM package dos and don'ts

What a SCORM package must contain

- A manifest that inventories the content of the package and specifies how the content is organized.
- Metadata, if not included inline in the manifest.
- All the files required to launch the content of the package.

What a content package manifest must contain

- Metadata describing the package.
- At least one organization element that specified an activity tree for the use of the content of the package.
- A collection of one or more resource elements that specify launchable content objects. Each resource contains an inventory of the files in the package that are required to launch and run the resource.

What a content package manifest may also contain

- Resource elements that specify additional resources files shared by one or more other resource elements
- Additional metadata for any component that is intended to be extracted from the package for use separately from this package or inclusion in another package.
- Sub-manifests representing sub-packages embedded in the package
- Sequencing rules and information specified for the activity tree represented by an organization element.

What a content package manifest may not contain

- References to anything in a directory "above" the root directory structure of the content of the package.

What a content package manifest may not contain

- References to anything in a directory "above" the root directory structure of the content of the package.

What the files in a content package are not allowed to contain

- References to anything in a directory "above" the root directory structure of the content of the package.
- Downloadable runtime components for which an installation that requires administrative rights is required.

What a content package manifest should not contain

- References to a file or other asset identified by a fully qualified URL (domain + path), since there is no guarantee that such a file or resource will be available at the time of package delivery.
- References to a file or resource that requires an active web server component, such as .asp, .jsp or .php web pages, since there is no guarantee that the corresponding active server component will be available on the web server or file system used to deliver the content package.

What the files in content package manifest should not contain

- References to a file or resource identified by a fully qualified URL (domain + path).
- References to a file or resource that requires an active web server component, such as .asp, .jsp or .php web pages.

Packages that depend on external content or an external server are, by definition, brittle. They will not work in an offline environment or behind some strictly configured firewalls. See also the note about server-dependent SCOs on the next page.

.

Chapter 5 - Understanding SCOs

What is a SCO?

A Shareable Content Object, or SCO, is a special kind of content object that knows how to communicate with the runtime environment in which it is launched.

A SCO is web content, meaning that it can be launched in a web browser by using a URL. It may consist of a single HTML page, or it may be a large collection of web pages and include simulations, Flash assets, or other media rich content. A SCO is basically a small portable web site that can be copied from place to place by gathering all its files and capturing them in a SCORM package. To be portable, a SCO must be compatible with any generic web server. In other words, it cannot depend on special services that might exist on one web server but not on another.

There has been some talk about SCOs that would not be fully portable. For example, some forms of learning experience, dynamic content or simulations may require specific advanced server functionality that goes beyond what a generic web server provides. However, there is currently no standard that specifies how to package, transport and install such content across servers or technology platforms.

Therefore, such server-dependent SCOs are not SCORM conformant today. SCORM content should be compatible with any web server, and also run in an offline environment without requiring the installation and configuration of a web server.

The SCO must be designed so that it can be launched in a standalone web window, or in a frame in a HTML frameset. Many SCORM runtime environments launch SCOs in a frameset, with other frames containing user interface elements of the runtime environment. Typically, if there is more than one content object available through the activity tree, the learning environment will show the activity tree in the form of an outline. The runtime environment may also display user interface elements for value-added features specific to that environment, like help or collaboration features.

Context is king, the SCO is only content

In the SCORM, the context rules and SCOs are used at will. In other words, the runtime environment decides whether and how to launch a SCO, and SCOs have no say in the matter. What drives the learning environment is typically a larger instructional or performance support context. This may require some conceptual adjustment for some content designers who are not used to build highly modular content that is driven by a single unified design. The SCORM was designed to favor the requirements of the new paradigms for advanced learning, in which the context is what matters, and content is only a tool used in learning activities.

How reusable are SCOs?

In practice, SCOs are more or less reusable, depending on the design of the SCO. Some SCOs are designed to be usable in any context. Others make sense only in a particular context. Not all SCOs can be aggregated into arbitrary packages, and some SCOs separated from a package that provides their context may not be useful. In any case, however, all SCOs use the same interface and launch method and can thus be shared without modification among learning management systems regardless of the learning management system implementation.

The "ransom letter" controversy

Many instructional designers and other SCORM stakeholders have expressed concern about what they perceive as a problem of visual continuity. The argument goes like this: If SCOs are assembled from various sources, they may have vastly different look and feel, like a ransom letter, and that does not look good.

It is indeed possible to achieve this visually jarring effect. However it may not be as serious as it seems. For one thing, one should not confuse functional continuity and visual continuity. It is important to provide consistent or at least predictable user interfaces, but that does not mean they have to look the same. For another, most people today are successfully extracting information and learning from vastly different content they gather with Google or other search engines. In fact, many younger learners, raised on MTV and the short attention span culture of television, are probably more turned off and bored by visual continuity than by interesting clashes of look and feel.

In the end, what really matters is that the active process of learning takes place, not which color or font is used. The time may have come to do away with the confusion between good instructional design and good graphic design.

This being said, there are cases where visual continuity is important, for reasons of enterprise politics, or because of specific characteristics of the audience. Nothing prevents a content developer or a community of practice from agreeing on design policies, style sheets and templates to ensure that all the SCOs they will deploy have a consistent look and feel.

As of now, however, the SCORM contains no provision to enforce such a policy. There is also no provision to allow content to control the look and feel across SCOs or between SCOs and the runtime environment.

SCORM 2004 does however specify some means for SCOs to display their own navigation controls and request that the runtime environment turn off the corresponding controls. This is specified in the Navigation section of the Sequencing and Navigation document. In the end, though, those are only requests and the runtime environment will decide what to show and how.

What a SCO does

As far as the SCORM is concerned, a SCO is not required to do much. At a minimum, though, it must communicate with the runtime environment once it has been launched.

Except for this basic communication requirement, the SCO developer can do anything he or she likes as long as it the behavior is self-contained in the SCO. For example, links to other SCOs are not allowed.

See below for a more detailed list of dos and don'ts.

The more advanced authoring tools for SCORM content hide the underlying code a SCO use for communication completely, so that an author never needs to see any of it. Other tools may include templates that encapsulate the necessary code.

Communicating with the runtime environment

The IEEE communication API

SCORM communication between a SCO and the runtime environment is an implementation of an international standard, IEEE 1484.11.2: ECMAScript API for Content to Runtime Services.

The runtime environment must make an ECMAScript-compatible API object available in the DOM (Document Object Model) context of the browser before it launches the SCO. The SCO must then look for an instance of this API object, by searching frames and windows in a very specific order defined by the IEEE standard. Once the SCO has found the object, it calls functions of the object to start a communication session with that object.

Neither the standard nor SCORM specify that any particular technology, such as JavaScript, Java, etc. must be used to implement the API object. The only requirement is that the object must be compatible with any ECMAScript implementation. For this reason, all parameters and return values must be strings or represented as strings, because otherwise there may be compatibility issues with the binary representations for data types. The SCORM documents refer to this API object as an *API instance*, to reinforce the idea that this object must be instantiated as a DOM element in the context of the browser. Once a SCO discovers the API instance through the DOM, it must use the same instance for any subsequent API calls within the same session.

ECMAScript is the ISO standard born from earlier versions of Netscape's JavaScript and Microsoft's JScript. Many people still use the term "JavaScript" to mean ECMAScript as implemented in the current mainstream browsers, such as the Mozilla based browsers and Microsoft's Internet Explorer.

Managing the communication session

The SCO must initialize a communication session by calling the corresponding function of the API instance to open a communication session. Once the session has been successfully initialized, the SCO can get and set data through corresponding functions of the API instance. Finally, the SCO must terminate the communication session by calling the corresponding function.

Only one communication session is allowed for every launch of the SCO by the runtime environment. If a SCO tries to initialize a new communication session after terminating the session, this will cause an error.

Managing unexpected unloading

Frequently, a SCO is unloaded before running its course to a point where it would normally terminate the communication session. For example, the learner might choose another activity, which causes premature termination of the SCO, or the learner might close the browser window.

It is still the responsibility of the SCO to manage the communication session properly in this case. Typically, a SCO developer defines a handler for the `onunload` browser event to do this. When the SCO is running in Internet Explorer (IE) or Mozilla FireFox, a more robust way to handle an unexpected unloading can be to define a handler for the `onbeforeunload` browser event. Both Internet Explorer and Mozilla FireFox trigger `onbeforeunload` before it actually begins to unload the web page and its associated resources and scripts. Unfortunately some other features of the SCO such as may trigger `onbeforeunload` prematurely and therefore `onbeforeunload` must be used with caution.

The handler for the unexpected unloading must attempt to send any unsaved tracking data to the runtime environment by using the `SetValue` function of the API instance, and then call the Terminate function of the API instance.

Being ready for unexpected unloading has another implication for SCOs. For example, if a SCO is made of a series of linked web pages, and the SCO may be unloaded at any time, every page must be ready to be abnormally unloaded at any time and terminate the communication session. This really requires some form of state management for the entire SCO, rather than on a page per page basis. For this reason, a frameset that can maintain state is often used to implement SCOs that use multiple web pages.

Optimizing bandwidth and reliability with Commit

The SCORM profile for the IEEE API uses a lot of function calls to get and set values through the API instance, one data element at a time. This works very well in the local communication between the SCO and the API instance, but it is not practical for communication across the Internet. For this reason, the IEEE API also provides a Commit function. When the API function is called, it is a signal to the runtime environment that the data sent by the SCO should be committed to persistent storage. Many SCORM runtime environments use Commit as a signal to transfer a batch of data to the LMS across the Internet. Terminating the communication session implicitly invokes Commit. Some SCO developers wait until the end of the communication session to send and commit all their data. However, this may result in a lot of data being transmitted while the page is being unloaded. Worse, if this happens while the browser window is closing, the browser may never finish the operation. It is better to spread out the data communication over the duration of the session. For example, it is a good idea to call Commit after a "batch" of individual data has been sent to the runtime environment, or if some significant data has been sent and it may be a while before there will be more data.

Mitigating catastrophic failure

Running an application in a browser is always a "best effort" proposition, because the content does not control the browser. For example, the user might just shut down the browser. If the SCO design allows it, information necessary to recover in case of catastrophic failure can also be sent and committed early in the session and then after every significant event. For example, tentative status information may be sent early, and updated as the learner progresses through the SCO. Suspend data to allow resumption of the SCO in a later learner session can also be updated at regular intervals throughout the session.

The communication data models

The API standard does not specify any particular data model for the data exchanged during a communication session. It can be used with any data model, and more than one data model may be used in the same communication session.

The SCORM specifies both an IEEE standard communication data model and a custom data model for navigation data. The first is identified in the SCORM dot notation by the "`cmi.`" prefix, and the second is identified in the SCORM dot notation by the "`adl.nav.`" prefix.

The standard data model is defined by the IEEE standard 1484.11.1: Data Model for Content to Learning Management System Communication. The SCORM calls this data model the "CMI" data model because the standard is mostly compatible with the older AICC CMI data model used in SCORM 1.2.

The CMI data model

The IEEE standard for the CMI communication data model specifies data that a content object may query from the runtime environment as well as data that the SCO may send to the runtime environment.

The SCORM profile of the IEEE standard adds some specific requirements. For each data element, it specifies whether a SCO may get the data from the runtime environment by using a `GetValue` API function, send the data to the runtime environment by using a `SetValue` function, or both get and set data.

One-way data from the runtime environment

This includes data that a SCO may request about how it is being launched, other initialization data, and some data about the learner, such as the learner's name.

One way data from the SCO

This includes data used to signal various things to the runtime environment, such as the time elapsed in the SCO, or whether the SCO requests to have some data preserved so that it can resume from a suspended state in a later session.

Two-way data

This includes most of the data elements defined in the data model. For example, a SCO may send a score to the runtime environment, and get the score back later in the same communication session. Or, it may get existing status information about a content objective for this particular learner, update the information, and send that to the runtime environment.

The CMI data model contains several categories of data:

- Status data, about the status of the SCO, including completion and success status.
- Score data.
- Thresholds for passing score, and for how much progress is required for completion
- Data about content objectives and their status.
- Data about various types of interactions and their status and learner responses. This can be used for traditional test items as well as for more complex interactions, such as simulations.
- Comments
- Limited learner information
- Some common learner preference items
- Suspend data and location, which can be used to resume an interrupted session.
- Entry and exit status, used to determine how the SCO was launched and how the SCO believes it is being terminated.

Note that a SCO is under no obligation to use every one of those data elements. However, a runtime environment must implement all of them, in case a SCO needs to use any of them.

The ADL navigation data model

The ADL navigation data model is intended for use when SCORM 2004 sequencing is implemented. It allows the SCO to request certain navigation actions. This in turns allows a SCO to display its own navigation elements. For example, a SCO might contain a "Continue" button that will cause the runtime environment to continue to the next activity, whatever that activity happens to be. Note that the general rule of SCORM still applies: Context is king. So, a SCO may request something, but the runtime environment decides what will actually happen.

Working with the SCORM data models

Scope and persistence of the data

The data in the SCORM communication data model are always good for a particular learner in a particular communication session in support of this specific activity. Data about other learners, about other uses of the SCO in other activities, or about other communication sessions is not available.

The SCORM specifies that the data from previous between activity attempts that use the same SCO is not available. In other words, if a learner attempts to use a SCO, and then later makes a new attempt, the runtime environment must initialize a fresh set of data.

The only exception to this is data about the status of objectives. This status information for objectives may persist between attempts if the runtime environment or the LMS stores it in persistent records. Whether and how this happens, however, is outside the scope of the CMI data model.

A new specification from the IMS, the Sharable State Persistence specification, defines a model to make data persist between invocations of a SCO, even when the SCO is used by different activities, and to make data available to other SCOs. However, as of this writing there is no requirement for a SCORM conformant runtime environment to implement this specification. This may change if enough stakeholders require the functionality.

Suspend and resume functionality

The CMI data model supports the ability for a SCO to request that its session be suspended when it is terminated. The SCO can send some private data to be stored by the runtime environment. This data is then available to the SCO if the session is resumed later. For example, imagine that a complex SCO regularly sends location information and requests that the ability to suspend be enabled. If the session is resumed later, the SCO can use the location data to return to the same place.

The size allowed for the suspend data is fairly small. This limitation is necessary because content should not be allowed to store arbitrarily large chunks of data on a LMS without negotiations. Past experience has shown that without such limits some content developers wanted to store things like multi-megabyte voice recordings. In a LMS with a few thousand users this might quickly lead to a storage meltdown, not to mention the bandwidth issues.

An interesting new feature of SCORM 2004 that was not present in SCORM 1.2 is that values in the data model must persist between sessions that are suspended. So, data like completion status, interactions data and so on that may have been set by the SCO will still be there along with suspend data and location if the SCO is resumed. This addresses a longstanding frustration of SCORM 1.2 content developers, because in SCORM 1.2 only suspend data and location values were guaranteed to be available on resume, and interaction data was write only. It also significantly reduces the need to shoehorn all the state information in suspend data.

In practical terms, for example, it means that a simulation may use an interaction record within the data model to store the path taken by a user through a simulation, by specifying the appropriate interaction type (sequencing or performance). The simulation may then retrieve this data to reconstruct state (or at least the path) if the SCO is resumed later.

This is also true for objectives in the CMI data model, but with a twist. If an objective's identifier is mapped to other activities in the activity sequencing, the status of that objective may be modified as the result of another activity while the activity that uses the SCO is suspended. On resume, the status for that objective could then be different than the status as it had been set by the SCO prior to suspend. This may seem a little disconcerting, but it opens a whole new range of functional possibilities.

The IMS Shareable State Persistent specification mentioned above may allow content developers to work around other size limitations in the future. In particular, it allows the content developer to "warn" the LMS about how much data it needs to store.

The dot notation binding

Because the IEEE communication data model standard does not specify a particular binding, the SCORM team "invented" a dot notation binding for it, based on proven existing practice using the AICC CMI data model in SCORM 1.2. A similar dot notation binding was also used for the navigation data model.

The dot notation conundrum

Using dot notation to represent elements in an unordered collection of data elements sometimes can be awkward. The dot notation forces the use of an index value to reference a particular element in such a collection, but the index value itself is not part of the data model. For example, the data model contains records with various data elements about objectives. The order of objectives is meaningless, but the dot notation requires that each record be addressed by an index number, because there is no other way.

It is important to remember that some data index values are not meaningful and exist only as a way to walk through data records. This means that a runtime environment is not required to use the same index values if it makes the data records available in a subsequent communication session.

As a rule of thumb, if the record includes an identifier, the identifier is meaningful to identify the record, but the index number is only a way to access a particular record in this particular communication session. For example, a SCO that is trying to read existing objective should first determine the proper index by walking the objective records to find the one with the desired identifier, and then use that index to access the record in order to get the other elements of the record.

SCO dos and don'ts

What a SCO must do

- A SCO must locate the API instance and then initiate and manage a communication session with the API instance.

What a SCO should do

- A SCO should be designed to work in different window sizes, and ideally adjust itself to the size of the window in which it is being run. The runtime environment, not the SCO, dictates the size of the "stage" window or frame in which the SCO will be played.
- A SCO should be designed to function in different visual contexts.
- A SCO should be designed to meet standard accessibility guidelines. Communities of practice may impose specific accessibility requirements (WAI, Section 508, and so on).
- A SCO should be implemented in such a way that no critical user data is lost if the SCO is unloaded unexpectedly.
- A SCO should send data at least about its completion status. If relevant, it should also send data about its success status.
- A SCO that needs to send a lot of data should not send it all at once at the end of the communication session. It should spread out the sending of data throughout the session, and call the API's Commit function when appropriate.

What a SCO is not allowed to do

- A SCO may not contain hyperlinks to other SCOs in the package.
- A SCO may not attempt change the size or appearance of the runtime environment.
- A SCO may not interact with the runtime environment through any means other than through the SCORM API.
- A SCO may not close the top level window of the browser, unless it is the only thing running in that window.
- A SCO may not create additional windows, unless it can close them reliably when the SCO is terminated.

What a SCO should not do

- A SCO should not use or reference any file that is not listed or specified through a dependency in the corresponding resource element the package manifest. A content server is under no obligation to provide any file that is not properly listed in the manifest when the SCO is actually launched.
- A SCO should not use hyperlinks to content outside the SCO, even if the content is in the same package, unless the target files are accounted for in the resource element that defines the SCO, either by listing the target files within the resource, or by listing dependencies on other resources.
- A SCO should not use hyperlinks to web pages or resources that may not be available when the SCO is run in a different context than the original context. For example, a package may be deployed behind a firewall, or on an offline player on a computer with no network access. Hyperlinks to content outside the package are obviously not manageable. No system that can manage or archive the package would be able to guarantee that such outside hyperlinks will work in the future.

File dependencies and server-dependent SCOs

Server-dependent SCOs are a special case. There is currently no SCORM approved standard way to specify the content of a server-dependent SCO and to ensure that all the dependencies for a server-dependent SCO will be resolved as expected at run time.

Except for file dependencies, server-dependent SCOs are subject to the same dos and don'ts as portable SCOs.

Chapter 6 - SCORM Sequencing

Introduction to sequencing

Sequencing is optional

Developers of SCORM conformant runtime environments must implement sequencing. But developers of SCORM conformant content do not have to implement sequencing. The default behavior for a SCORM delivery environment is to let the learner choose from among all the activities represented by items specified by an Organization element in the manifest. If the content designer is happy to allow learners free rein over the content, no sequencing needs to be specified.

Why sequencing is optional

There are many kinds of SCORM conformant content, and some of the content may be amenable to full control by the learner. In fact, the issue of locus of control—who controls what happens, the learner or the content designer—has been a major philosophical debate for years. If the locus of control is with the learner, the content developer does not need to worry about imposing a sequence.

However, it is often desirable to provide at least some guidance, or to make the content adaptive. Some designers believe that unless their content is used according to a particular activity sequence it will not work. Therefore, SCORM 2004 allows you to specify sequencing behavior in a content package.

Another reason to use the sequencing features of SCORM 2004 is to control the rollup behavior for tracking information. By default, a SCORM 2004 runtime environment "rolls up" the scores from sub-activities in a simple and predictable way. Sometimes, however, this is not what you want. For example, in a test the scores on some activities may have more weight than the scores on other activities. Or you may specify that some activity is not required to consider a course complete. In those cases, you need to use the SCORM 2004 sequencing rules that control rollups, even if you do not specify a particular sequencing behavior.

How do you specify sequencing?

Sequencing and rollup rules are specified by adding data to the Organization element and to the Item elements in the package manifest. The Organization represents an activity tree, where the Organization itself represents the main activity, and Items within the organization represent sub-activities. If an activity has children, some rules associated with it will govern how to manage its children. For example, a rule may specify that you have to do the sub-activities sequentially. If an activity is a child of another activity, some rules associated with it will govern how it is managed as a sub-activity. For example, it may specify the weight of the sub-activity when tracking data are rolled up. Sequencing in the SCORM can be quite different from the way sequencing is done in traditional computer-based training, and may require “unlearning” some traditional sequencing techniques. In the next section, we will take a brief look at some of the sequencing capabilities of SCORM 2004.

Logic will get you from A to B. Imagination will take you everywhere.

Albert Einstein

What you can do with sequencing

General learning process strategies

You can do a lot of very complicated things with SCORM sequencing. You can also blow your mind and get lost in the intricacies of the specifications. It may be more useful to look at it from a learning process perspective, rather than the traditional computer based training approach. So, let us step away from the computer and programming for a moment, and look at the functional picture.

Human learning is an activity. This activity typically encompasses a number of sub-activities. There are also some recognizable, common patterns in learning activities. Typical patterns are things like:

- Try until you succeed or fail
- Skip what has already been learned
- Follow a guided sequence of steps
- Try different learning approaches until one succeeds

These patterns occur every day in every one of our learning activities. SCORM 2004 sequencing allows us to capture some of those patterns in a content package.

Think activity, not content

This is a big shift from the content-centric model. The SCORM sequencing model is activity-centric, not content centric. Activities happen to use resources. In SCORM 2004, the only types of resources supported are SCOs and other digital content assets, or resources that can be represented in digital form.

Because the activities are what matter, the SCOs don't have much control over what happens. In other words, the rules that govern activities take precedence over anything defined in a SCO. Once a SCO is launched, it may have its own internal sequencing, of course, but any sequencing between SCOs is governed by the activities. There are a couple mechanisms that allow as SCO to *request* some particular navigation, but those requests can only be honored if they do not conflict with rules defined for the activities. For example, a SCO might request to continue to the next SCO—whatever that is—but an activity rule may exist that specifies that this is not allowed until the learner has achieved a sufficient score.

Adaptive learning

A typical approach for systematic learning is to follow a sequence, and adapt to the learner's characteristics. For example, if the learner already knows something, there is no need to go through a tutorial on the subject. If, on the other hand, the learner fails in an assessment, remediation is typically desirable. This may involve one or more tries until the learner either succeeds or fails.

It does help not to think of adaptive learning using SCORM 2004 as a traditional CBT branching problem. Rather, think of it as a learner-centric activity tree model that can accommodate different learning styles.

Different pedagogical models

The systematic learning approach described above is not the only desirable one. In fact, some prominent educators believe that in many cases a constructivist or discovery approach is more conducive to real learning. This is more difficult to design, which is why a lot of computer-based learning follows the systematic learning approaches. But in any case SCORM 2004 does not preclude this approach. In fact, it also allows you to mix the approaches so that you are not forced into a single strategy. For example, you might force the learner to go through an orientation, but then go into a discovery environment in which the learner can choose among an array of activities of different types, and follow this by an assessment sequence. Or, you might make it all a discovery environment by simply not specifying any sequencing at all.

A simple way to support different learning styles, for example, is to enable guided flow without disabling free choice in navigation between the activities. For field-dependent learners, the defined flow provides the guidance they need. For field-independent learners, the ability to choose lets them learn through discovery.

Mapping objectives and competencies

Every activity has an implicit objective, and success in achieving that objective can be recorded. Sometimes it does not matter for the big picture. For example, some activities may be included in a SCORM activity tree only for necessary administrative tasks, or they may be practice activities with no lasting consequence. In other cases, success in one activity can influence another activity. For example, if the learner successfully aces out a pre test on a particular skill, it may not be necessary to go through the tutorial for that skill. This connection between activities is done by tracking the status on shared objectives. If the pretest showed that you master skill X, then the tutorial to teach X can be skipped.

SCORM 2004 allows you to make the implicit objective for an activity explicit, and to explicitly map it to other objectives that may be associated with other activities in the activity tree. Any objective referenced in such a map must be given an explicit identifier. This identifier may also be the identifier of a reusable competency definition, such as a learning objective or skill definition. The IMS RDCEO (Reusable Definition of Competency or Educational Objective) specification describes how to create reusable competency definitions.

Each entry in an objective map specifies a target objective identifier, and whether status information about that target objective can be shared, and in which direction. For example, an activity may be allowed to affect the learner's score on X, or it may be allowed only to read the existing score for X.

Mixing strategies in the activity hierarchy

SCORM 2004 allows unlimited nesting of sub-activities within activities. This means that you are not constrained to use the same strategy throughout the activity tree. For example, you might have a formal sequence of activities that, when launched, turn out to be discovery activities, within which the learner has free choice, but some of those activities encountered in the discovery environment might themselves be structured. What you can do is limited only by your imagination.

Cautions and warnings

In some SCORM 2004 sequencing templates available from certain SCORM related sources, objective maps are also used for something that has nothing to do at all with learning objectives and competencies. Objective maps are overloaded to provide some primitive "menu" navigation features which are not yet supported by the IMS Simple Sequencing specification at the core of SCORM sequencing.

Objective identifiers used for this purpose, which has nothing to do with competency, should be clearly constructed so that they are not confused with actual objectives related to learning and competency.

For example, confusion might be avoided by using a prefix like "var_" in such objective identifiers, and a namespace prefix in "real" objective identifiers. So far, however, there is no standard ADL defined way to address this issue in an interoperable way.

So, this is a problem if you intend to use global objectives in connection with a LMS or competency management system to report only on the objectives that matter to you.

One way to avoid garbage data in reports and competency tracking may be to exchange data only for objectives whose identifier is also the identifier of a formal definition. This might be a standard reusable competency definition.

.

Simple sequencing templates

Why use templates?

Simple Sequencing, the IMS specification on which SCORM 2004 sequencing is based, is all but simple. One must specify sequencing by defining various data elements which are added to the nodes in the activity tree. Some of these data items specify behaviors that are often difficult to understand. Mastery of this very intricate model requires a deep understanding of the behaviors, how they interact or modify each other and how a runtime environment is supposed to implement them.

However, useful behavior patterns can be predefined in templates. The user of the template does not need to understand the intricacies of the encoding of the behavior. What you lose in flexibility by using templates, you gain through ease of use. Instructional designers may understand the templates without having to understand the mind-numbing details of how they are implemented under the hood.

Templates also draw on centuries of experience in the history of learning. In many cases, they can embody proven pedagogical or instructional strategies or assessment strategies adapted to the needs of the learner and the specific characteristics of the desired learning outcomes.

Components of a template

Sequencing can be built in different ways, depending on whether and how they will be supported by a higher level authoring environment or workflow automation. A crude form of template consists of a commented sub-tree of activities which can be copied and pasted into a manifest, and edited manually. A more sophisticated template may be an XML document that is designed to guide an authoring tool that will generate the actual manifest.

Generally speaking, a sequencing template would include:

- A tree of activity nodes (items) designed to be used as the activity tree in a manifest, or nested at any level of depth in an activity tree.
- Placeholders for the learning resources used to implement the activities. The placeholders may be designed to accept a SCO, an asset, or an activity sub-tree that may be defined by another template. Each placeholder typically has an associated specific purpose. For example, a template may have placeholders for a pre-assessment, a tutorial, a remediation and a post-assessment. Another template may have placeholders for learning resources that can be explored in a discovery approach.
- Rules that govern the use and instantiation of the template. For example, a template may be designed to allow the author to change various parameters. Rules may govern whether some placeholders can be left empty, in which case the generation of an activity tree for the template will be automatically adjusted to work around the missing optional components.

A template also needs documentation or some application that can mediate its use by the author, content creator or aggregator.

Suggested small starter set

A small starter set of templates might include the following:

- Simple sequential guided flow, which traverses an activity tree in a linear manner.
- Learning activity followed by optional practice followed by assessment with remediation including optional practice if the assessment fails.
- Set of learning activities, each with a specific objective, followed by an optional practice for each objective and by an assessment that includes each objective, with remediation including optional practice for the failed objectives only.
- Simple exploratory learning module, flowing from an introduction into an activity which affords the learner unlimited choice and exploration of sub-activities, followed by an optional practice and an optional assessment

More complicated templates

More complicated templates can be built to embody combinations of proven learning patterns. For example, an instructional sequence template should provide alternative methods of remediation if the first remediation failed.

Chapter 7 - The Navigation data model

History of the navigation data model

Early SCORM content developers have sometimes been frustrated by the fact that they could not provide their own user interface for navigation within SCOs. It is important to remember that in SCORM the control resides firmly with the runtime environment, and the SCOs are just resources used in the fulfillment of learning activities.

However, sometimes it is useful to be able to embed a navigation control in the SCO itself. For example, a content developer might want to provide a package in which a "Continue" button . goes from event to event inside a SCO. When navigation inside the SCO reaches the end of the SCO—whatever that means is up to the designer of the SCO—the same button would also trigger the continuation to the next SCO in the sequence, wherever that SCO happens to be in the activity sequence.

SCORM 2004 added a navigation data model to enable some of this functionality.

How the data model works with sequencing

This data model works with the standard SCORM communication API and allows the SCO to interrogate the runtime environment about navigation status and to signal a desired navigation behavior. For example, this allows the SCO to display or enable a "Continue" button only if the runtime environment reports that continuing to another SCO will be allowed.

It also allows the SCO to communicate to the runtime environment a navigation event that should take place when the SCO calls `Terminate` to end its communication session. When Terminates is called, the runtime environment commits any remaining data set by the SCO, and then it updates the status of the activity tree, and then is looks at a possible pending navigation signal from the SCO.

It is important to note that this signal is only a request. The runtime environment, based on its interpretation of the sequencing rules and on the status of the activity tree, may deny the request. For example, this could happen if the SCO sends new data that result in a status change that then causes the request to be denied.

Besides the basic navigation commands, the Navigation data model also allows the SCO to request navigation to a specific activity. This feature should be used with extreme caution, since it depends on an activity identifier that may make sense only in the context of a very specific activity tree. If the SCO is reused in another activity tree, this may lead to mysterious bugs that could be extremely difficult to resolve. For example, if the activity identifier is "activity1.1" someone might reuse the SCO in another tree where "activity1.1" has a completely different meaning. If you use this feature, you should use globally unique identifiers for the activity items, so that there is at least a fighting chance that the SCO will not end up requesting navigation to some random activity. The SCO should also use the navigation data model to request whether the target activity is available before assuming it can request to navigate to it.

Chapter 8 - Practical SCO construction

After reading this chapter, you should have a better understanding of how to construct a practical SCO, including how to work with the SCORM API and data models. It contains a series of working examples that introduce progressively more powerful features.

You may be able to use or adapt code from these examples to create your own SCOs, or to create a tool that generates SCOs. Or, you might be using a high level authoring tool that is generating some similar code, in which case this chapter may help you understand what is going on behind the scenes.

The HTML and JavaScript samples in this chapter are tested and functional, and should work in any SCORM 2004 conformant environment and with any browser that implements ECMAScript 262, the standard behind JavaScript.

These samples do not run the gamut of SCORM API and data model functionality. This document is not intended to replace the SCORM 2004 Runtime Environment book, which does describe that gamut, but rather to introduce concepts and methods that can be useful in implementing content object that take advantage of the SCORM.

Some readers might not like the programming style or naming and formatting conventions used in the script and HTML examples. However, that is often a matter of taste or software design policy and really is not very important here. These samples were designed to help you understand what makes SCORM content tick, and how you can exploit the functionality of SCORM for good learning experiences. They are not intended to be used as a software design tutorial or a guide for programming style.

NOTE: PLEASE SEND FEEDBACK TO TOOLS AT OSTYN.COM IF YOU ENCOUNTER A BUG!

A minimal generic, reusable SCO script

This minimal script for a web page finds the API instance. See section 3.3.1 of the SCORM Run-Time Environment (RTE) Version 1.3.1 document for an illustration of how it does this. The minimal script also provides functions to initialize and terminate the communication session automatically if a SCORM 2004 API instance is available. It also sets the completion status of the SCO to "`completed`" when the SCO terminates.

```
// For SCORM 2004 only
var gAPI = null;
var gnScormSessionState = 0; // 0=not initialized; 1=initialized; 2=terminated
function ScanForAPI(win)
{
  var nFindAPITries = 500;
  while ((win.API_1484_11 == null) && (win.parent != null) && (win.parent != win))
  {
    nFindAPITries--;
    if (nFindAPITries < 0) return null;
    win = win.parent;
  }
  return win.API_1484_11;
}
function GetAPI(win)
{
  if ((win.parent != null) && (win.parent != win))
  {
    gAPI = ScanForAPI(win.parent);
  }
  if ((gAPI == null) && (win.opener != null))
  {
    gAPI = ScanForAPI(win.opener);
  }
}
function ScormInitialize()
{
  if (gnScormSessionState == 0)
  {
    GetAPI(window);
    if ((gAPI != null) && (gAPI.Initialize("") == "true"))
    {
      gnScormSessionState = 1;
    }
  }
}
function ScormTerminate()
{
  if (gnScormSessionState == 1)
  {
    gAPI.SetValue("cmi.completion_status", "completed")
    if (gAPI.Terminate("") == "true") gnScormSessionState = 2;
  }
}
```

Code Sample 1: Simplest reusable SCO script (minisco.js)

A very simple SCO using the generic script

This is a complete SCO that uses the simple generic script above. It is a single page and does not carry much learning content, but it is a SCORM conformant SCO. The `onload` handler in the `<body>` tag for the web page ensures that SCO will automatically start a communication session with the SCORM API as soon as it is loaded. See the more detailed explanation below for the `onunload` handler.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Claude's simplest SCO</title>
<script type="text/javascript" src="minisco.js">
</script>
</head>
<body onload=ScormInitialize() onunload=ScormTerminate()>
<em>... as simple as possible, but not simpler.</em><br />Albert Einstein
</body>
</html>
```

Code sample 2: Simplest SCO using a generic script (minisco.html)

Scripting for the unexpected

Unexpected unloading

Things don’t always work as expected. A SCO is always launched in a browser window, over which it has no control. Therefore, it must be ready to be unloaded unexpectedly as a result of user actions such as closing the browser window or choosing another activity. This could happen at any time.

By scripting the SCO to detect that it is being unloaded, the SCO can at least try an orderly shutdown if this happens unexpectedly. This is particularly important if the SCO still holds unsaved tracking data to send through the SCORM API. As part of the orderly shutdown, the SCO can send the data and terminate the communication session cleanly.

.

Sending data to the runtime environment

Look again at the generic simple script in Code Sample 1. Do you see where some data is sent to the runtime environment? It is done by the statement

```
gAPI.SetValue("cmi.completion_status",
"completed")
```

Sending data to the Runtime Environment requires two string parameters: The identifier for a data element, and the value to be ascribed to that data element. In this case, the element is part of the CMI data model. The value is one of the predefined tokens that are valid values for that element.

A generic helper function that sets any value and does basic error handling could be added to the simple script above. It could look something like Code Sample 3.

```
function ScormSetValue(what, value)
{
  if (gnScormSessionState == 1)
  {
    return gAPI.SetValue(what, value);
  }
  else
  {
    return false;
  }
}
```

Code sample 3: A helper function to send data to the API

Why use a generic function, rather than calling the API directly? Because it makes it easy to implement additional error handling or debugging code in a single place. Also, using a generic function allows your generic script to be adapted to deal with future or past versions of the SCORM API without having to change anything in the SCOs that uses the generic script.

Getting data from the runtime environment

Getting data from the runtime environment only requires one parameter: A string that specifies the data element whose value you need. First, let us make a generic helper function to get the value. At the same time, let us add some way to deal with errors. If we call the `GetValue` function, but the result is an empty string, we may want to check the error status to determine whether the data element really had an empty string value, or whether a null string was returned because an error occurred. The API function `GetLastError` returns a number that represents the error state of the communication session after the last call to the API. A return value of 0 means that no error was detected.

```
function ScormGetLastError()
{
  var sErr = "-1"; // -1 is our own private error status to mean "no SCORM API".
  if (gAPI) sErr = gAPI.GetLastError();
  return sErr;
}
function ScormGetErrorString(sErr)
{
  var strErr = "SCORM API not available";
  if (gAPI)
  {
    if (isNaN(parseInt(sErr))) sErr = gAPI.GetLastError();
    strErr = gAPI.GetErrorString(sErr.toString());
  }
  return strErr;
}
function ScormGetValue(what)
{
  var v = null;
  if (gnScormSessionState == 1)
  {
    v = gAPI.GetValue(what);
    if ((v == "") && (ScormGetLastError() != 0)) alert(ScormGetErrorString());
  }
  return v;
}
```

Code sample 4: Helper functions to get data from the API and check the error state.

A more complete reusable SCO script

Like the script we saw earlier in Code Sample 1, this more complete script finds the API instance and provides functions to initialize and terminate the communication session if a SCORM 2004 API instance is available. It also includes generic helper functions to get and send data through the communication API, and to get both the error status and the error text that may correspond to an error status.

```
// For SCORM 2004 only
var gAPI = null;
var gnScormSessionState = 0; // 0=not initialized; 1=initialized; 2=terminated
function ScanForAPI(win)
{
  var nFindAPITries = 500;
  while ((win.API_1484_11 == null)&&(win.parent != null)&&(win.parent != win))
  {
    nFindAPITries--;
    if (nFindAPITries < 0) return null;
    win = win.parent;
  }
  return win.API_1484_11;
}
function GetAPI(win)
{
  if ((win.parent != null) && (win.parent != win))
  {
    gAPI = ScanForAPI(win.parent);
  }
  if ((gAPI == null) && (win.opener != null))
  {
    gAPI = ScanForAPI(win.opener);
  }
}
function ScormInitialize()
{
  if (gnScormSessionState == 0)
  {
    GetAPI(window);
    if ((gAPI != null) && (gAPI.Initialize("") == "true"))
    {
      gnScormSessionState = 1;
    }
  }
}
function ScormTerminate()
{
  if (gnScormSessionState == 1)
  {
    if (gAPI.Terminate("") == "true")
    {
      gnScormSessionState = 2;
      return "true";
    }
  }
  return "false";
}
```

```
function ScormGetLastError()
{
  var sErr = "-1";
  if (gAPI) sErr = gAPI.GetLastError();
  return sErr;
}
function ScormGetErrorString(sErr)
{
  var strErr = "SCORM API not available";
  if (gAPI)
  {
    // Note: Get Error functions may work even if the session is not open
    // (to help diagnose session management errors), but we're still careful,
    // and so we check whether each function is available before calling it.
    if ((isNaN(parseInt(sErr))) && (gAPI.GetLastError)) sErr = gAPI.GetLastError();
    if (gAPI.GetErrorText) strErr = gAPI.GetErrorText(sErr.toString());
  }
  return strErr;
}
function ScormGetValue(what)
{
  var strR = "";
  if (gnScormSessionState == 1)
  {
    strR = gAPI.GetValue(what);
    if ((strR == "") && (ScormGetLastError() != 0)) alert(ScormGetErrorString());
  }
  return strR;
}
function ScormSetValue(what, value)
{
  if (gnScormSessionState == 1)
  {
    return gAPI.SetValue(what, value);
  }
  else
  {
    return "false";
  }
}
```

Code Sample 5: A more complete reusable SCO script (simplesco.js)

You may notice that the statement that sets the completion status has disappeared from this reusable script. As we are getting more ambitious, we will now put a call to set completion status in the SCO itself. After all, only the SCO knows for sure when it is completed. When the SCO in Code sample 6 is launched, it immediately initializes a communication session. Then, it sets the completion status to "`incomplete`" but only if the runtime environment does not report that it was already completed. When the SCO is unloaded, it sets the completion status to "`completed`" before terminating the session.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Claude's simplest SCO</title>
<script type="text/javascript" src="simplesco.js"></script>
<script type="text/javascript">
function init()
{
  ScormInitialize();
  if (ScormGetValue("cmi.completion_status") != "completed")
  {
    ScormSetValue("cmi.completion_status","incomplete");
  }
}
function terminate()
{
  ScormSetValue("cmi.completion_status","completed");
  ScormTerminate();
}
</script>
</head>
<body onload="init()" onunload="terminate()">
<em>Perfection is achieved not when there is nothing left to add, but when
 there is nothing left to remove.</em><br />-- Antoine de Saint-Exup&eacute;ry
</body>
</html>
```

Code sample 6: Simple SCO using a generic script to report completion status (simplesco.html)

Sending basic tracking data to the runtime environment

A typical SCO reports information about its status, so that a LMS can track the user's success and progress toward completion of the activity that uses the SCO as a resource.

Some people care most about whether the activity implemented in the SCO was completed; other care most about whether it was successful. In some cases, completion may not be relevant if the learner was successful; for example, a learner may test out of a content object that includes both a pre-assessment and a tutorial. In some other cases, completion is required for success; for example, a drill and practice exercise may require a set number of drills to acquire an automated response mechanism for some type of stimulus.

The SCORM does not specify which basic tracking data a SCO should send, but it enables whatever policy a designer or community of practice would like to follow.

Progress information may include summary completion status, such as incomplete or completed, and a measure of progress. The measure of progress is the ratio between what has been done and what can be done. It is a floating-point value between 0, meaning not attempted or nothing done and 1.0, meaning completed.

Success information may include summary success status, such as pass or fail, and a measure of success in the form of a numeric score. The measure of success is reported as a score scaled to the range of –1.0 to 1.0, where 0 represents no success and 1.0 represents total success. Negative values are allowed for the cases where it is necessary to represent abject failure with a "penalty score".

In addition to the scaled score, information about a SCO-specific scoring range may be reported; this includes the minimum and maximum scores that define the range for a raw score. In practice, however, only the scaled score will be used by SCORM sequencing. The scaled score should also be used by a LMS to report the score, typically as a percentage value. For example, a scale score of `0.75` can be displayed as 75%.

In the sample SCO in Code Sample 7, the summary data values are hard coded, to keep the sample brief. In a real SCO, the scripts that manage the actual behavior of the SCO would of course probably determine those values, based on what actually happens when a learner uses the SCO.

This sample does not report minimum, maximum and raw scores because they are typically ignored in systematic learning management. For example, SCORM 2004 Sequencing rules reference on only the scaled score value that may be reported by a SCO.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Claude's simple tracking SCO</title>
<script type="text/javascript" src="simplesco.js"></script>
<script type="text/javascript">
function init()
{
  ScormInitialize();
  if (ScormGetValue("cmi.completion_status") != "completed")
  {
    ScormSetValue("cmi.completion_status","incomplete");
  }
}
function terminate()
{
  ScormSetValue("cmi.completion_status","completed");
  ScormSetValue("cmi.progress_measure","1.0");
  ScormSetValue("cmi.success_status","passed");
  ScormSetValue("cmi.score.scaled","1.0");
  ScormTerminate();
}
</script>
</head>
<body onload="init()" onunload="terminate()">
<em>Perfection is achieved not when there is nothing left to add, but when
 there is nothing left to remove.</em><br />-- Antoine de Saint-Exup&eacute;ry
</body>
</html>
```

Code Sample 7: SCO that reports completion and success

Relating score and success status

The same SCO can be used in different contexts, with different threshold values for what would constitute a passing score.

The SCO may query the runtime environment to find out whether such a value is available. The SCO can then use this information to determine that the success status is "`passed`" or "`failed`", depending on the scaled score. In any case, even if the SCO reports success status as "`passed`", the runtime environment makes the final decision. The SCO is only used as a resource in a learning activity; it does not run the show. If the runtime environment might not provide a success threshold value, the SCO may provide its own.

In the sample code in Code Sample 7, the `SCOSetValue` function has been modified to take into account the success threshold value to update success status when a score is reported. If no threshold value is available from the runtime environment, the SCO uses its own value.

```
// script fragment
var gnPassingScore = 1.0; // Default value for this SCO
var gbPassingScoreAlreadyQueriedFromRTE = false; // True if RTE queried for value
var gbPassingScoreIsFromRTE = false; // True only if RTE did provide value

function ScormSetValue(what, value)
{
  var err = "false"
  if (gnScormSessionState == 1)
  {
    err = gAPI.SetValue(what, value);
    if ((err == "true") && (what == "cmi.score.scaled"))
    {
      ScormSetSuccessStatusForScore(parseFloat(value));
    }
  }
  return err;
}
function ScormSetSuccessStatusForScore(nScore)
{
  if (!gbPassingScoreAlreadyQueriedFromRTE)
  {
    var nThreshold = parseFloat(ScormGetValue("cmi.scaled_passing_score"));
    gbPassingScoreAlreadyQueriedFromRTE = true;
    if (IsValidScaledScore(nThreshold))
    {
      gnPassingScore = nThreshold;
      gbPassingScoreIsFromRTE = true;
    }
  }
  if (IsValidScaledScore(nScore)) && (IsValidScaledScore(gnPassingScore))
  {
    (nScore >= gnPassingScore? ScormSetValue("cmi.success_status","passed"):
       ScormSetValue("cmi.success_status","failed"));
  }
}
function IsValidScaledScore(nScore)
{
  return ((!isNaN(nThreshold)) && (nThreshold >= -1.0) && (nThreshold <=1.0))
}
```

Code Sample 8: SCO functions to report success status automatically when reporting a score

Maintaining state across multiple pages

A SCO that is made of multiple HTML pages must still keep track of the status of the communication session and other variables. Since there can be only one communication session for each launch of a SCO, you cannot have each page initialize and terminate its own communication session with the API instance. One reliable solution is to use an invisible frameset within which the pages can be changed without losing the state data stored in the frameset itself. The SCO is the frameset. This is more reliable than using cookies.

For example, let us say that a SCO uses 3 pages. Going from page to page basically just changes the source of a "stage" frame in the frameset (Figure 4) that is launched when the SCORM runtime environment launches the SCO. The frameset itself can be invisible to the learner, as shown in the example below. The frameset can also act as a basic runtime or sequencing environment for the pages, with a script that manages navigation. Functions in the script of the frameset can also be used to relay calls from the different pages to the API instance provided by the SCORM runtime environment.

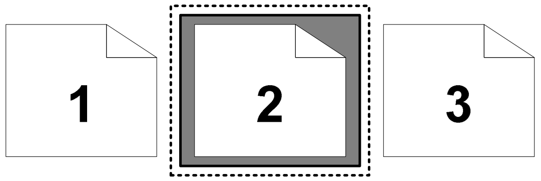


Figure 4: Playing pages in a frame

One important consideration when using a frameset is compliance with accessibility guidelines. You are probably working under policies that require your content to comply with some standard accessibility guidelines. You also need your SCO to be compatible with special browsers, such as browsers for visually impaired users. Make sure that the "stage" frame in which the pages are displayed has a meaningful name, and that each page has a meaningful title. The title of each page will be invisible in a normal browser, since the page is displayed in a frame without adornments, but is important to provide this normally hidden title for accessibility-enabled browsers. See the appendix for useful references on how to achieve accessibility while using frames.

Multi-page SCO using the generic script

The SCO in Code sample 9 uses the same generic script as a simple SCO. However, it is a little different. It will only set completion status to "`completed`" after every page has been visited. It also implements some simple functions to provide navigation between the pages. You can use this basic script and add any number of pages by just changing a couple of variables.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Claude's simplest multiple page SCO</title>
<script type="text/javascript" src="simplesco.js"></script>
<script type="text/javascript">
function init()
{
  ScormInitialize();
  if (ScormGetValue("cmi.completion_status") != "completed")
  {
    ScormSetValue("cmi.completion_status","incomplete");
  }
}
function terminate()
{
  MarkIfCompleted();
  ScormTerminate();
}
var gnPage = 1;
var gnMaxPages = 3; // Change this if you add more pages
var gnPagesNeeded = gnMaxPages; // Change this if fewer pages are needed for
completion
var gaPagesCompleted = new Array(true,false); // Used to keep track of pages viewed
for (i=1; i < gnMaxPages; i++) gaPagesCompleted[i] = false;
function GoToPage(n)
{
  if (gnPage != n)
  {
    gnPage = n;
    DisplayFrame.location.href = "page" + gnPage + ".html";
    gaPagesCompleted[gnPage-1] = true;
    MarkIfCompleted();
  }
}
function GoPreviousPage()
{
  if (gnPage > 1) GoToPage(gnPage - 1);
}
function GoNextPage()
{
  if (gnPage < gnMaxPages) GoToPage(gnPage + 1);
}
function MarkIfCompleted()
{
  var nCompleted = 0;
  var bCompleted = false;
  for (i=0; i < gaPagesCompleted.length; i++)
  {
    if (gaPagesCompleted[i]) nCompleted++;
  }
  bCompleted = (nCompleted >= gnPagesNeeded)
  if (bCompleted) ScormSetValue("cmi.completion_status","completed");
  return bCompleted;
}
</script>
</head>
<frameset rows="100%,*"
 onload="init()" onunload="terminate()">
  <frame id="DisplayFrame" name="DisplayFrame" src="page1.html" />
  <frame id="DummyFrame" name="DummyFrame" src="about:blank" />
</frameset>
</html>
```

Code sample 9: Frameset SCO to display multiple pages status (multisco.html)

Each page of the SCO contains the same basic navigation links. This code sample shows page 2. Of course, page 1 should not contain the "Previous" link, and the last page should not contain the "Next" link. The check for the function in the parent page is not strictly necessary, but it prevents disconcerting error messages if the page is used by itself, outside the context of the frameset.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Page two</title>
<script type="text/javascript">
function previous()
{
  if ((window.parent)&&(window.parent.GoPreviousPage)) window.parent.GoPreviousPage()
}
function next()
{
  if ((window.parent)&&(window.parent.GoNextPage)) window.parent.GoNextPage()
}
</script>
</head>
<body>
<p>This is page 2</p>
<p><a href="javascript:previous()">Previous</a></p>
<p><a href="javascript:next()">Next</a></p>
</body>
</html>
```

Code sample 10: Page 2 of a simple multiple page SCO (page2.html)

When to use Commit

We may want to ensure that the information about the user progress is safely recorded, even if something catastrophic like a communication failure should happen. Although there is no way to control the proprietary mechanism by which the runtime service stores persistent data, at least the SCO can ask it do so, by calling the Commit function of the API instance. A good time to do this in this example would be whenever we want to record a page change in the frameset.

First, we add a generic Commit function to our reusable script. Such a generic function allows us to add error handling and debug code without having to modify the SCOs that use the function.

```
...
function ScormCommit()
{
  if ((gnScormSessionState == 0) && (API != null))
  {
    return gAPI.Commit("");
  }
  return "false";
}
...
```

Code sample 11: A generic function to commit data sent to the runtime environment

Then, we add the `Commit` call if we recorded completion while the SCO is running. Code Sample 12 shows how we can add this in the frameset script's `GoToPage` function, to be called if the function resulted in completion of the SCO.

```
...
function GoToPage(n)
{
  if (gnPage != n)
  {
    gnPage = n;
    DisplayFrame.location.href = "page" + gnPage + ".html";
    gaPagesCompleted[gnPage-1] = true;
    if (MarkIfCompleted()) ScormCommit();
  }
}
...
```

Code Sample 12: Calling Commit while the SCO is running

Note that calling `Terminate` implicitly calls `Commit`. In other words, a SCO does not need to call `Commit` if one or more calls to `SetValue` are immediately followed by a call to `Terminate`.

Using suspend and resume data

It would be nice if the user could come back to the same page if the attempt to finish the activity that uses the SCO is interrupted and then resumed later. This requires that the SCO notify the runtime environment that it wants the session to be suspended. The SCO can then store some specific data into the data elements `cmi.location` and `cmi.suspend_data`, and if the session is later resumed it will be able to use that data to resume its own state.

No change is required in the reusable script we have used so far, but we need to modify the SCO's script to save state information and invoke suspend before terminating. We also need to modify it so that it can detect whether it is resuming a prior session when it is launched, and use the suspend data that may have been stored during that previous session. Only the SCO understands the data values in `cmi.location` and `cmi.suspend_data`. The runtime environment and the LMS do not and should not try to interpret the values in those data elements. This is private data that only makes sense to the SCO. Still, the SCO can be a little paranoid. It makes sense to validate the values the SCO gets from the runtime environment before using them. Suspend and location data may also need to be converted to a string for sending through the API, and back to some other data model when received back through the API. In this example, the location is a number and the suspend data is used to restore an array.

Since this SCO always wants to be suspended if possible, and would like to resume where it left off in case of catastrophic failure, we will update the suspend data as the user goes from page to page, rather than waiting for the last bitter moment when the SCO detects that it is being unloaded to do so. Saving suspend data as you go may use some bandwidth, but it allows for graceful recovery if things go bad.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Claude's simplest multiple page SCO</title>
<script type="text/javascript" src="simplesco.js"></script>
<script type="text/javascript">
function init()
{
  ScormInitialize();
  if (ScormGetValue("cmi.completion_status") != "completed")
  {
    ScormSetValue("cmi.completion_status","incomplete");
  }
  if (ScormGetValue("cmi.entry") == "resume") RestoreFromSuspend();
}
function terminate()
{
  MarkIfCompleted();
  ScormTerminate();
}
var gnPage = 1;
var gnMaxPages = 3; // Change this if you add more pages
var gnPagesNeeded = gnMaxPages; // Change if fewer pages are needed for completion
var gaPagesCompleted = new Array(true,false); // Used to keep track of pages viewed
for (i=1; i < gnMaxPages; i++) gaPagesCompleted[i] = false;
function SaveSuspendData()
{
  ScormSetValue("cmi.exit","suspend");
  ScormSetValue("cmi.location", gnPage.toString()); //SCORM requires string type
  ScormSetValue("cmi.suspend_data", gaPagesCompleted.join(","))
}
```

```
function RestoreFromSuspend()
{
  var a = ScormGetValue("cmi.suspend_data").split(",");
  if (a.length == gaPagesCompleted.length) gaPagesCompleted = a;
  var n = parseInt(ScormGetValue("cmi.location"));
  if (!isNaN(n)) GoToPage(n)
}
function GoToPage(n)
{
  if (gnPage != n)
  {
    gnPage = n;
    DisplayFrame.location.href = "page" + gnPage + ".html";
    gaPagesCompleted[gnPage-1] = true;
    MarkIfCompleted();
  }
}
function GoPreviousPage()
{
  if (gnPage > 1) GoToPage(gnPage - 1);
}
function GoNextPage()
{
  if (gnPage < gnMaxPages) GoToPage(gnPage + 1);
}
function MarkIfCompleted()
{
  var nCompleted = 0;
  var bCompleted = false;
  for (i=0; i < gaPagesCompleted.length; i++)
  {
    if (gaPagesCompleted[i]) nCompleted++;
  }
  bCompleted = (nCompleted >= gnPagesNeeded)
  if (bCompleted) ScormSetValue("cmi.completion_status","completed");
  return bCompleted;
}</script>
</head>
<frameset rows="*,100%"
 onload="init()" onunload="terminate()">
  <frame id="DummyFrame" name="DummyFrame" src="about:blank" />
  <frame name="DisplayFrame" name="DisplayFrame" src="page1.html" />
</frameset>
</html>
```

Code sample 13: Frameset SCO to display multiple pages status (multisco.html)

Reporting interaction data

One could write quite a few books about learning interaction design and the applications of the interaction data element in the CMI data model. This is not one of those books, but we will try to provide some flavor of how to use interaction data efficiently. Obviously, interaction data can be used to track traditional test item types, like multiple choice questions. But they can also be used to track richer interactions that may not feel like a traditional test item, but that result in evaluation models that are identical to various types of test items. For example, you could use the good old multiple choice data model to track and evaluate which pages among a set of page the user has bookmarked as being relevant to the learning task. The data models for sequencing and performance interaction types are designed to accommodate data from various forms of simulations.

This example will be far less ambitious. It will however expose some of the technical issues and solutions that work. A usual, we will start with some reusable utility functions.

Walking the array of interaction records

Understanding interaction records

SCORM 2004 allows a SCO to create up to 250 interaction records, as defined in the IEEE data model. Each interaction record must contain an identifier for that interaction. The identifier is the only reliable way to identify an interaction record according to the IEEE data model. During a communication session, the runtime environment will store the records in an indexed array, and the "dot notation" used by SCORM requires the index number to access one of those records. Note however that this indexed array is not part of the IEEE data model and is only an artifact of the "dot notation" used in SCORM to access elements of the data model.

Journaling vs. status

There has been debate about whether the interactions data model is best used to record a journal of interaction response events or the state of a finite set of interaction objects in a content object. Recording each response as a separate interaction record allows for analysis of how a learner may have changed responses during an attempt. However, if the interactions are intricate the limit of 250 records may be reached fast, and there is no defined way to deal with the overflow. On the other hand, recording the status of each interaction object is more like observing the results of a traditional paper test. Only the most current response status is shown, regardless of whether the learner changed responses or not. Unless there are extenuating circumstances that require journaling, I would strongly recommend using the interaction object status approach. If event journaling is required for after action analysis, it can be implemented more efficiently by logging every data change sent by a SCO in a customized delivery environment. If, on the other hand, some form of journaling is required to provide after action analysis and feedback within the SCO itself, there are other methods such as using the standard data model for a specific interaction type like “`performance`”, which already includes journaling capability. The code samples below assume the more common and simpler model where only the most current status of interaction objects is recorded.

No intrinsic order to the interactions

The IEEE standard defines the collection of interaction records as a "bag", which means a collection with no specified order. In the SCORM, the interactions array only persists during the communication session, and the runtime environment may use a completely different approach to store those records between sessions. For example, it might store them with the identifier as the key in a database table. Like the array itself, the array index is not part of the IEEE data model and is only an artifact of the "dot notation" used in SCORM to access elements of the data model.

Avoiding interactions index problems

Since the collection of interaction records is a bag, in a subsequent communication session, the interaction records might appear in a different order. This means that if you use interaction records from a previous session, you must first find out what array index got assigned to each record in this new session. How do you do that? By searching the records for the interaction identifiers first. The easiest way to do this is to use helper functions that hide the complexity of the array dot notation and use the interaction identifiers to walk the array on behalf of the SCO.

Generic helper functions for interactions

We will therefore add the following helper functions to our reusable SCORM script:

- `ScormInteractionAddRecord` takes as a parameter the identifier of the interaction and a valid interaction type. It returns an integer which is the index of the interaction record in the dot notation array index, or –1 if an error occurred. If an interaction record with the same identifier and the same type already exists, the function does not do anything and just returns the index of the existing record. If an interaction record with the same identifier but a different interaction type already exists, the function fails. If the number of allowed interaction records would be exceeded by the addition of a new record, the function fails.
- `ScormInteractionGetCount` takes no parameter and returns an integer value representing the number of existing interaction records.
- `ScormInteractionGetData` takes as parameter the identifier of the interaction, and the data model element within interaction record, and returns a value. If the value is an empty string, which may indicate a possible error, the normal SCORM API error status can be examined for diagnostic. The string that identifies the data element is what appears to the right of "`interactions.n.`" in the SCORM documentation for the data model.
- `ScormInteractionGetIndex` takes as a parameter the identifier of the interaction. It returns an integer which is the index of the interaction record in the dot notation array index, or –1 if no such record exists. This function can be used to check whether a record already exists for an interaction.
- `ScormInteractionSetData` takes as parameter the identifier of the interaction, the data model element within interaction record, and the value to set. It returns "`true`" or "`false`". If "`false`", the normal SCORM API error status can be examined for diagnostic. Because no data can be stored in an interaction record until the identifier is stored in the record, and some of the data cannot be store properly unless the interaction type is known, this function will fail if the interaction record has not been previously created by a call to `ScormInteractionAddRecord`. The string that identifies the data element is what appears to the right of "`interactions.n.`" in the SCORM documentation for the data model.

These functions could of course be heavily optimized, for example by caching the interaction count and last index used, or by creating a separate index array. However, for the sake of simplicity, such optimizations have not been applied here. Some error checking has also been omitted for brevity.

Caution: If you use journaling, more than one record may exist with the same identifier. If each record contains a timestamp, you can use that to distinguish between records with the same identifier. You may want to consider whether journaling is worth the added complexity. The sample helper functions shown here assume that each record has a unique identifier. If you use journaling, you will have to create much more complex helper functions to retrieve interaction data. You also need to add error handling for the cases where you might exceed 250 interaction records, maybe because a user changes her mind a lot.

```
// fragment
function ScormInteractionAddRecord (strID, strType)
{
  var n = ScormInteractionGetIndex(strID);
  if (n > -1) // An interaction record exists with this identifier
  {
    if (ScormGetValue("cmi.interactions." + n + ".type") != strType) return -1;
    return n;
  }
  n = ScormInteractionGetCount();
  var strPrefix = "cmi.interactions." + n + ".";
  if (ScormSetValue(strPrefix + "id", strID) != "true") return -1;
  if (ScormSetValue(strPrefix + "type", strType) != "true") return -1;
  return n
}
function ScormInteractionGetCount()
{
  var r = parseInt(ScormGetValue("cmi.interactions.count"));
  if (isNaN(r)) r = 0;
  return r;
}
function ScormInteractionGetData(strID, strElem)
{
  var n = ScormInteractionGetIndex(strID);
  if (n < 0)
  {
    return ""; // No interaction record exists with this identifier
  }
  return ScormGetValue("cmi.interactions." + n + "." + strElem);
}
function ScormInteractionGetIndex(strID)
{
  var n = ScormInteractionCount();
  for (i = 0; i < n; i++)
  {
    if (ScormGetValue("cmi.interactions." + i + ".id") == strID)
    {
      return i;
    }
  }
  return -1;
}
function ScormInteractionSetData(strID, strElem, strVal)
{
  var n = ScormInteractionGetIndex(strID);
  if (n < 0)
  {
    return "false"; // No interaction record exists with this identifier
  }
  return ScormSetValue("cmi.interactions." + n + "." + strElem, strVal);
}
```

Code sample 14: Helper functions to work with interaction records (in ostyn2004sco.js)

Code Sample 15 is a SCO that will do some simple interaction tracking. The SCO will also record a score. Note that there is NO requirement that the score for the SCO be in any way tied to interaction results.

This example uses a simple form. This has one major advantage, which is that the SCO will automatically meet at accessibility guidelines for Web content. By using CSS, you can of course dress up a form to be much richer visually than what is shown here.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A simple SCO with interaction tracking</title>
<script type="text/javascript" src="ostyn2004sco.js"> </script>
<script type="text/javascript">
function evalQChoice(thisForm,strCorr)
{
  var n = 0;
  var strID = thisForm.id;
  var aResp = new Array();
  for (i = 0; i < thisForm.elements.length; i++)
  {
    if (thisForm.elements[i].checked) aResp[aResp.length] = thisForm.elements[i].id;
  }
  var strResp = aResp.toString();
  (strResp != strCorr ? strResult = "incorrect" : strResult = "correct");
  (strResp != strCorr ? alert("Oops!") : alert("Well done!"));
  ScormInteractionAddRecord(strID, "choice");
  ScormInteractionSetData(strID, "description", "Some sample multiple choice");
  ScormInteractionSetData(strID, "correct_responses.0.pattern", strCorr);
  ScormInteractionSetData(strID, "learner_response", strResp);
  ScormInteractionSetData(strID, "result", strResult);
  ScormCommit();
  return false; // this is a local submit - we don't want to unload the page
}
</script>
</head>
<body><!-- onload and onunload are managed by ostyn2004sco.js -->
<form id="urn_ostyn_q_sa20041101_foo" onsubmit="return evalQChoice(this,'A1,A3')">
Which of these functions <b>must</b> be used by a SCO to be SCORM conformant?<br />
<input type="checkbox" id="A1" size="50" />Initialize<br />
<input type="checkbox" id="A2" size="50" />Commit<br />
<input type="checkbox" id="A3" size="50" />Terminate<br />
<input type="submit" value="Done" /></form>
</body>
</html>
```

Code Sample 15: Simple SCO reporting interaction data (interactionsco.html)

This sample SCO includes a more complete generic script than the previous examples. This script handles much of the tedious housekeeping that otherwise would have to be recreated for every SCO. For example, you may have noticed that the `<body>` tag in Code Sample 15 does not include event handlers for `unload` and `onunload`. The generic script manages those events automatically. The generic script is documented in detail in the Appendix *Overview of the generic SCO* script.

Interaction identifiers

You may notice that SCORM 2004 uses a form of "long identifier" data type for interaction record identifiers. This supports the use of this data element for universally unique identifiers. For example, this allows recording of data for interactions that may be drawn from question banks. The identifier can, without any ambiguity, specify a particular version of a particular question in a particular version of a particular question bank. Even if you don't use question banks, a unique identifier can ease with record keeping and analysis of the data in the future.

Cross version compatibility

SCORM 2004 and SCORM 1.2 are slightly different in the details and syntax of interaction data. This may require reshaping some data elements and values. If you need to create

SCOs that that work with both SCORM 1.2 and SCORM 2004, or if you need to update existing SCOs, functions like `ScormInteractionSetData` and `ScormInteractionGetData` provide a convenient place to hide the necessary data re shaping. These functions can be embedded in a generic reusable script to isolate your SCO from differences between SCORM 1.2 SCOs and SCORM 2004. Note that the samples included with this book do not perform this data shaping for cross-version compatibility.

Since the data model for interactions in SCORM 1.2 is more limiting than in SCORM 2004, cross-version SCOs should only use the SCORM 1.2 CMI data model functionality, which can be fully represented in the SCORM 2004 data model functionality. Also, keep in mind that in SCORM 1.2 the CMI interaction data are write-only. They cannot be read back from the API by a SCO.

Assessment security

An issue that has been raised frequently is the security of a SCORM assessment. It should be noted that although this code sample makes it easy to determine the correct answer, it is possible to code scripts or content in such a way as to make it very difficult to determine the correct answer. It is also possible to use the IEEE data model to only communicate the learner responses to some service that evaluates the response; however this is outside the current scope of SCORM.

“Man in the middle” attacks can be prevented by running the entire SCORM session in a secure (HTTPS) environment. Of course, in that case the runtime implementation must also use a secure, encrypted method for communication with the back end server. Client side hacking is possible in theory, but quite difficult in practice, and would probably be worth the effort only for high stakes assessments.

In any case, SCORM security or the lack thereof is the least of the problems for high stake exams or assessments, which require physical security of the workstation and its software configuration, as well as reliable authentication of the person tested.

In high stakes assessments, you need a way to ascertain that the right person is being tested and that the person does not get undue help from documents, online resources and friendly helpers. Ensuring this is beyond the scope of the SCORM and indeed beyond what most online assessment systems, whether they use SCORM or not, can provide today.

Working with objective data

Implicit vs. explicit objectives

Any SCO is assumed to have an implicit objective, which typically is some form of learning outcome. The status regarding this implicit objective is reported as status data for the SCO itself: Whether and how much of it is completed, and whether and how well the learner is succeeding. This was described in an example above. However, a SCO may also be setting or getting status information about other objectives. Those objectives must be explicitly identified through a unique identifier. The status for each objective can then be stored in "objective records" that include that identifier.

Objective identifiers

The identifier for an explicit objective should be globally unique, because the SCORM Sequencing and Navigation specification includes a way to specify objective maps for activities. An objective map allows the content author to specify whether and how status data about objective referenced by the SCO is also shared by other activities. This allows a SCO to influence the status of objectives that are referenced by sequencing rules; it also allows a SCO to find out about the status of objectives that are influenced by other activities. For example, a pre-test might set the status of a series of objectives to pass or fail, and the sequencing may then skip the tutorials designed to achieve mastery of the objectives that have been deemed to be already mastered.

Objective data vs. interaction data

Note that the data model for interactions includes optional references to objectives. However, such references are purely informative. A reference to an objective in the data for an interaction does not imply any way to influence the status data for the objective. If you want to set objective data as a result of interactions in a SCO, it is up to you to define this in the coding or scripting of the SCO. The SCORM does not specify how to do that. For example, if a SCO contains an assessment with questions whose status is reported using the data model, several questions might be used to assess the learner's mastery for a particular objective, and these questions might influence the status for the objective in different ways as determined by the designer of the assessment.

Walking the array of objective records

Understanding objective records

SCORM 2004 allows a SCO to access or create up to 100 objective status records, as defined in the IEEE data model. Each objective record must contain an identifier for that objective. The identifier is the only reliable way to identify an objective record according to the IEEE data model. During a communication session, the runtime environment will store the records in an indexed array, and the "dot notation" used by SCORM requires the index number to access one of those records. Note however that this indexed array is not part of the IEEE data model and is only an artifact of the "dot notation" used in SCORM to access elements of the data model.

Predefined objective records

When an attempt is started on a SCO, some objective records may already exist in the collection of objective records for the SCO. This will occur when the corresponding objective identifiers are specified in an objective map in the sequencing properties for the activity that launched the SCO.

The objective records are not ordered

The IEEE standard defines the collection of objective records as a "bag", which means a collection with no specified order. In the SCORM, the objective array only persists during the communication session, and the runtime environment may use a completely different approach to store those records between sessions. For example, it might store them with the identifier as the key in a database table. Like the array itself, the array index is not part of the IEEE data model and is only an artifact of the "dot notation" used in SCORM to access elements of the data model. In a subsequent communication session, the objective records might appear in a different order.

Avoiding objective record index problems

If you use objective records from a previous session, you must first find out what array index got assigned to each record in this new session. How do you do that? By searching the records for the objective identifiers first. The easiest way to do this is to use helper functions that hide the complexity of the array dot notation and use the objective identifiers to walk the array on behalf of the SCO.

Generic helper functions for objectives

We will therefore add the following helper functions to our reusable SCORM script:

- `ScormObjectiveAddRecord` takes as a parameter the identifier of the objective. It returns an integer that is the index of the objective record in the dot notation array index, or –1 if an error occurred. If an objective record with the same identifier already exists, the function does not do anything and just returns the index of the existing record. If the number of allowed objective records would be exceeded by the addition of a new record, the function fails.
- `ScormObjectiveGetCount` takes no parameter and returns an integer value representing the number of existing objective records.
- `ScormObjectiveGetData` takes as parameters the identifier of the objective, and the data model element within objective record, and returns a value. If the value is an empty string, which may indicate a possible error, the normal SCORM API error status can be examined for diagnostic. The string that identifies the data element is what appears to the right of "`objectives.n.`" in the SCORM documentation for the data model.
- `ScormObjectiveGetIndex` takes as a parameter the identifier of the objective. It returns an integer that is the index of the objective record in the dot notation array index, or –1 if no such record exists. This function can be used to check whether a record already exists for an interaction.
- `ScormObjectiveSetData` takes as parameters the identifier of the interaction, the data model element within interaction record, and the value to set. It returns "`true`" or "`false`". If "`false`", the normal SCORM API error status can be examined for diagnostic. Because no data can be stored in an interaction record until the identifier is stored in the record, this function will automatically attempt to call to `ScormObjectiveAddRecord if necessary to attempt to create the record`. The string that identifies the data element is what appears to the right of "`objectives.n.`" in the SCORM documentation for the data model.

These functions could of course be heavily optimized, for example by caching the objective count and last index used, or by creating a separate index array. However, for the sake of simplicity, such optimizations have not been applied here. Some error checking has also been omitted to keep the example short.

```
function ScormObjectiveAddRecord (strID)
{
  var n = ScormObjectiveGetIndex(strID);
  if (n > -1) // An objective record exists with this identifier
  {
    return n;
  }
  n = ScormObjectiveGetCount();
  var strPrefix = "cmi.objectives." + n + ".";
  if (ScormSetValue(strPrefix + "id", strID) != "true") return -1;
  return n
}
```

```
function ScormObjectiveGetCount()
{
  var r = parseInt(ScormGetValue("cmi.objectives.count"));
  if (isNaN(r)) r = 0;
  return r;
}
function ScormObjectiveGetData(strID, strElem)
{
  var n = ScormObjectiveGetIndex(strID);
  if (n < 0)
  {
    return ""; // No objectiverecord exists with this identifier
  }
  return ScormGetValue("cmi.objectives." + n + "." + strElem);
}
function ScormObjectiveGetIndex(strID)
{
  var n = ScormObjectiveGetCount();
  for (i = 0; i < n; i++)
  {
    if (ScormGetValue("cmi.objectives." + i + ".id") == strID)
    {
      return i;
    }
  }
  return -1;
}
function ScormObjectiveSetData(strID, strElem, strVal)
{
  var n = ScormObjectiveGetIndex(strID);
  if (n < 0) // If no objective record with this ID
  {
    n = ScormObjectiveAddRecord(strID);
    if (n < 0) return "false"; // No objective record and failed to create one
  }
  return ScormSetValue("cmi.objectives." + n + "." + strElem, strVal);
}
```

Code sample 16: Helper functions to work with objective records (in ostyn2004sco.js)

Code Sample 17 is a SCO that will report some status data about a couple of objectives. This SCO is hard wired to always set the status the same values when it is run.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A simple SCO with interaction tracking</title>
<script type="text/javascript" src="ostyn2004sco.js"> </script>
<script type="text/javascript">
function MarkObjectiveSuccess(strID)
{
  ScormObjectiveSetData(strID, "success_status", "true")
  ScormCommit();
}
</script>
</head>
<body> <!—- onload and onunload are managed by ostyn2004sco.js -->
<form id="urn_ostyn_rcdid_sa20041101_bar" onsubmit="MarkObjectiveSuccess(this.id)">
Click the button to mark this objective as successful.
<input type="submit" value="Done" /></form>
</body>
</html>
```

Code Sample 17: Simple SCO reporting objective data (objectivessco.html)

Turning a passive asset into a SCO

Why turn assets into SCOs

Sometimes, a learning activity uses a simple passive resource, such as a document, a picture, or an ordinary web page.

SCORM 2004 allows you to specify such an asset as the resource to launch instead of a SCO for an activity. This is done by specifying an attribute of the resource descriptor element in the package manifest. However, often it may be desirable to track the use of the asset. In that case, the asset can be turned into a SCO.

Asset + wrapper = SCO

The easiest way to turn an asset into a SCO is to "wrap" it into a HTML page. Some asset authoring tools, such as Adobe/Macromedia Flash, can do this semi-automatically through the template publishing method. For other assets, the trick is to create a page that contains a placeholder for the asset. Many assets, such as .pdf files, images, or text files can be launched directly by a browser. However, in order to use them as SCOs some scripts are required.

Generic SCO as asset wrapper

A generic SCO that can be used as a wrapper for any asset can be built as a frameset. The frameset's script manages its behavior as a SCO, and a "stage" frame is used to display the passive asset.

The only customized aspect of the wrapper is the relative URL of the passive asset, which must be provided as the source for the stage frame.

Code Sample 18 shows a SCO that wraps a JPEG image, named "foo.jpg" in this sample. It uses default behaviors built into a generic script to report the following data to the RTE:

- Completion status of `incomplete` when displaying the asset.
- Completion status of `completed` when the SCO is unloaded.
- The session time, which is the time elapsed during the SCO's communication session with the RTE. This corresponds roughly to the time the asset was displayed. For some types of assets, such as a .pdf file this includes the loading time for the browser's helper application and the asset data, because there is no way to automatically measure the time more precisely.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A simple SCO wrapping a passive asset</title>
<script type="text/javascript" src="ostyn2004sco.js"></script>
</head>
<frameset rows="100%,*">
<frame id="StageFrame" src="foo.jpg" /> // here replace src value with URL of asset
<frame id="DummyFrame" src="about:blank" />
</frameset>
</html>
```

Code Sample 18: Simple SCO wrapping a passive asset (pictureassco.html)

Closing the window when the SCO finishes

The perils of closing

There are two ways for a LMS to launch a SCO: In a frame or in a popup window. If the SCO is launched in a popup window, it "owns" the window and therefore it can close it safely. But if the SCO is launched in a frame, it does not own the context of the frame, which will be a frameset or a web page. This context is part of the runtime environment. If the SCO closes the browser window that contains that part of the runtime environment, this can have catastrophic consequences such as the loss of tracking data. If the frame is part of the main interface for the LMS, closing down the window effectively throws the user out of the LMS, which is probably not a good user experience.

The wild windows experience

In a sequencing scenario, popping up and closing a window for every activity may be a rather jarring experience, especially if the activities are short. Every time the user chooses to continue to the next activity, the window for one activity closes and then another one opens for the next activity. There is a way for a LMS implementer to avoid this, but so far most LMS implementers have implemented either the frameset approach in which the "stage" frame remains constant, and the SCOs are sequenced into it, or a simple popup approach.

To close or not to close, that is the question

If the runtime environment creates a popup window to run the SCO and launches the SCO directly in that popup window, the SCO "owns" the window and it can safely close it when it finishes. In fact, this is often the expected behavior.

If the runtime environment launches the SCO in a frame, the SCO is not allowed to close the window that contains that frame.

Automating the closing of the window

A SCO can easily determine whether it is running in a popup window or not. It can check whether the window in which it is launched has a parent. If it has one, it is running in a frame or frameset. If it does not, then it is safe to close the window.

Only the SCO designer knows when a SCO is "finished". Typically, this corresponds to some event like the user clicking an Exit button. There might not be a formal Exit button; the SCO may have some internal trigger that signals that it is time to end the SCORM API communication session.

User interface considerations

If the SCO cannot close its window but the activity is finished, the SCO should prevent further interaction by the user with its content. The most effective way to do this is to switch to a neutral display that is visually harmonious with the rest of the SCO but only contains a prompt to confirm to the user that the activity has finished. For example, in a multi-page SCO, this could entail going to an "exit page" that has no interactive elements. You should do this before closing the communication session.

Quirky behaviors

Experience has shown that with some LMS that use a popup window there is an implementation quirk that results in the window not closing even if the SCO can determine that it can be closed safely. For example, there might be a dialog box or even a LMS scripting error that prevents the script that would normally close the window from executing. We can work around that by using a recurring timer to trigger the closing of the window—only if allowed of course.

Timing and sequence considerations

There are a few situations where using a script call to close window is not a good idea, even if it is allowed. For example, if the window is already closed or is closing but the script is somehow still executing. The sample code fragment below uses flags and extra statements to avoid this problem. In this script fragment, we modify `ScormTerminate` to automatically try to close the window if allowed. If the SCO's user interface contains an "Exit" button or similar control, that control can display the neutral user interface and immediately call `ScormTerminate()` to attempt to close the window if allowed. The modification to our generic script requires the addition of some variables to avoid undesired states. Notice that the timer to trigger the attempt to close the window if started before calling `Terminate()`. The timer is a workaround for a problem encountered with a couple of LMS implementations where `Terminate()` results in a strange state. This may be the result of a silent scripting error or of some other implementation quirk. Although the SCO has no control over how the LMS is implemented, it can still act preemptively. The timer is recurring because single-shot timers sometimes fail to trigger due to system load or other conflicts.

Note: A more robust version of this script is included in the current version of the file ostyn2004sco.js.

```
var gbEnableAutoCloseWindow = true; // Preset value to control the behavior
var gTimerOwnWindowClose = null;
var gbAlreadyTriedToCloseOwnWindow = false;

function ScormTerminate()
{
  var result = "false";
  PrepareCloseWindowIfAllowed();
  if (gnScormSessionState == 1)
  {
    if (gAPI.Terminate("") == "true")
    {
      result = "true";
    }
  }
  return "false";
}
function IsClosingWindowOK()
{
  if (!gbEnableAutoCloseWindow) return false;
  // Tweaking of the rule may be required for some LMS
  // that use what looks like a popup window but is actually a frameset.
  // A function to try to detect such a situation might be inserted here.
  return (window == window.top);
}
function PrepareCloseWindowIfAllowed()
{
  if ((!gbAlreadyTriedToCloseOwnWindow) && (IsClosingWindowOK())
       && (gnScormSessionState == 2))
  {
    gTimerWindowClose = setInterval('SCOCloseOwnWindow()', 1500);
  }
}
function SCOCloseOwnWindow()
{
  if (gTimerOwnWindowClose)
  {
    clearInterval(gTimerOwnWindowClose);
    gTimerOwnWindowClose = null;
  }
  if (gbAlreadyTriedToCloseOwnWindow) return;
  gbAlreadyTriedToCloseOwnWindow = true;
  if (!window.closed) window.close();
}
```

Code Sample 19: Safe and sane automatic window closing when the SCO finishes

Using Flash to make a SCO

Flash can be nice

While a lot of learning content has been developed using high level tools like Flash or ToolBook, they are not required for SCORM content. For example, capabilities like drag and drop, simple vector graphics and dynamic graphing can be implemented using only cross-platform DHTML that does not require any plug-in. For examples of how SCORM content can be developed without plug-in dependencies, see the open source JavaScript libraries by Walter Zorn, or the way ToolBook generates DHTML code.

If you use Flash to create a SCO, the Flash movie must be played in a Flash object embedded in a web page. The web page itself is the SCO, and the resource element that describes the SCO in the SCORM package manifest specifies the URL of the HTML page.

You can do this embedding manually, or you can use a Flash publishing template with the appropriate placeholders.

Sequencing using Flash

You cannot sequence SCOs using Flash. The LMS provides the sequencer, not the content package. In fact, SCOs in the same package may use different technologies, ranging from simple HTML to DHTM to Flash to ToolBook.

You can however use Flash to create any sequencing you want *inside* a SCO. People have used different techniques to do this. It often takes the form of a single master Flash "wrapper" that loads different `.swf` files into the Flash object embedded in the page. The archives of the forums at adlnet.gov contain quite a few helpful pointers on this topic.

ActionScript communication with the API

It is highly recommended to use JavaScript in the host HTML page handle the communication session management, because timing issues are tricky in ActionScript, while JavaScript functions can respond immediately and synchronously to browser load and unload events. For example, it is trivial for a page script to detect that the page is being unloaded, but by the time the Flash movie gets notified of such an event it is typically too late to communicate any data.

It is also recommended that all communication between ActionScript and the API implementation be done through JavaScript relay functions in the host HTML page. Sending data through the relay functions that call SCORM API is relatively easy, and can be done by either FSCommand or `GetUrl`, or by using the newer Flash/JavaScript Integration Kit. Flash 8 introduces the ExternalInterface facility as a new way to communicate synchronously between ActionScript and JavaScript that removes a lot of the asynchronicity problems that arise when using FSCommand or GetUrl.

It is a good idea to send the data whenever something significant is happening, rather than waiting for the end of the movie, as this increases reliability, and the end of the movie might in reality never get reached if the SCO is unloaded by a sequencing event. For example, a Flash movie might report its frame as a SCORM data model location whenever the frame changes, so that it can be resumed from the same frame. The host page JavaScript can ensure that this is committed before the communication session is terminated.

Getting data into ActionScript from the SCORM API can be more complicated unless you master the new features in Flash 8. Again, JavaScript helper functions may help here by caching data or raising Flash events with return values.

Chapter 9 - Practical package assembly

Workflow

Pulling the components together

To assemble a SCORM package, you need the following components:

- All the files that will be required to run the package. These are the web content that makes up the SCOs and launchable assets.
- An XML manifest file that inventories the package content and describes how it is organized.
- The XML schema files (`.XSD` files) required to validate the XML manifest. These files can be found in any of the SCORM 2004 sample packages, including those included in the test suite to test a LMS for conformance.
- Metadata describing the package. This is a best practice, not required for conformance.

Describing resources

Each SCO or launchable asset is described by a Resource element in the manifest. You need to know which files each SCO or launchable asset requires, and these files must be inventoried in the Resource element.

If some of the SCOs or assets are using shared files, you may group the inventory of those files into one or more non-launchable Resource elements, and reference those with a Dependency element within each dependent resource. This will be explained in more detail later in this chapter.

Good authoring environments keep track of all the required files and should be able to construct the Resource elements automatically. More advanced environments may also be able to optimize the sharing of files and resources automatically.

Directory structure for package components

All the files that make up a package must fit within the same directory tree. The common directory at the root of that tree will become the root directory of the package. For example, you might build your package in a directory named "`c:\temp\scormbuild`". Any file in your package must be in that directory or in a subdirectory that depends on that directory.

Your package may group everything into a single directory, or it may use a base directory with subdirectories. It is entirely up to you. For example, a common approach is to use separate subdirectories for each resource, as well as subdirectories different types of shared resources

Good authoring environments automate the mustering of the required files into an appropriate directory structure. Unless you use a single directory, the main challenge is to keep track of all the relative paths in various links and references.

Adding metadata

Although SCORM 2004 does not require metadata, it is a good idea to provide metadata for your package. If there is any chance the package might be disaggregated or audited later, you might consider providing metadata for components as well as for the whole package. A full discussion of metadata is beyond the scope of this book, but we will look at some useful minimal metadata.

The metadata may be provided as a separate file within the package, or it may be included in the package manifest itself. There are advantages and disadvantages to both methods. One disadvantage of including metadata directly into the manifest is that if you have a lot of metadata it can make the manifest file very big. On the other hand, including the metadata inline in the manifest is often simpler. You should choose the method that suits your purpose best.

Good authoring environments automate the generation of the metadata, and you only need to worry about metadata if you are building a package by hand or if your tool does not provide adequate support.

The metadata are not considered resources. In other words, metadata files are not inventoried in the resources section of the manifest.

Building the manifest

You could build a manifest by hand, but XML was not really intended for people to edit by hand. It is much too easy to introduce simple errors. A good authoring tool or environment should build the manifest and musters all the needed files automatically for you.

However, for the purpose of this book, we will assume that you don't have such a tool handy, and so we will show you how to build a simple manifest from a template.

Some manifest elements are optional

Some elements are optional in a manifest. The SCORM books and the SCORM 2004 Conformance Requirements document list which manifest elements are mandatory and which are optional.

For example, unless you want to specify a particular behavior for navigation or rollup or tracking information in your package, you do not need to include any sequencing information. The SCORM specifies appropriate defaults. If you do add such behavior data, however, you must include this information as specified by SCORM, and also include the appropriate schemas. This will be explained in more detail in the chapter about sequencing. The important thing to remember is that this sequencing information is optional.

Some optional elements may be added to the organization items to add various data that can be communicated to the SCO. This includes launch parameters and launch data. These are described in detail in the SCORM Runtime Environment document.

Sample manifest

```
<?xml version="1.0" standalone="no"?>
<manifest identifier="ostyn.com/2005/pckgs/12345-23456-ABCD"
 version="1.0"
  xmlns="http://www.imsglobal.org/xsd/imscp_v1p1"
  xmlns:adlcp="http://www.adlnet.org/xsd/adlcp_v1p3"
  xmlns:adlseq="http://www.adlnet.org/xsd/adlseq_v1p3"
  xmlns:adlnav="http://www.adlnet.org/xsd/adlnav_v1p3"
  xmlns:imsss="http://www.imsglobal.org/xsd/imsss"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.imsglobal.org/xsd/imscp_v1p1 imscp_v1p1.xsd
   http://www.imsglobal.org/xsd/imsss imsss_v1p0.xsd
   http://www.adlnet.org/xsd/adlcp_v1p3 adlcp_v1p3.xsd
  http://www.adlnet.org/xsd/adlseq_v1p3 adlseq_v1p3.xsd
  http://www.adlnet.org/xsd/adlnav_v1p3 adlnav_v1p3.xsd">
<metadata>
   <schema>ADL SCORM</schema>
   <schemaversion>2004 3rd Edition</schemaversion>
   <adlcp:location>packagemetadata.xml</adlcp:location>
 </metadata>
 <organizations default="org_1">
   <organization identifier="org_1">
     <title>your_title_here</title>
     <item identifier="activity_1"
        identifierref="ostyn.com_-_2005_-_scos_-_12345-23456-ABCD ">
       <title>One activity</title>
     </item>
     <item identifier="activity_2"
        identifierref="ostyn.com_-_2005_-_scos_-_12345-23456-ABCE ">
       <title>Another activity</title>
     </item>
     <imsss:sequencing>
       <imsss:controlMode choiceExit="true" flow="true"/>
     </imsss:sequencing>
   </organization>
 </organizations>
 <resources>
   <resource identifier="ostyn.com_-_2005_-_scos_-_12345-23456-ABCD"
      adlcp:scormType="sco"
      href="scos/sco1.htm" type="webcontent">
     <file href="scos/sco1.htm"/>
     <file href="scos/simplesco.js"/>
   </resource>
   <resource identifier="ostyn.com_-_2005_-_scos_-_12345-23456-ABCE"
      adlcp:scormType="sco"
      href="scos/sco2.htm" type="webcontent">
     <file href="scos/sco2.htm"/>
     <file href="scos/simplesco.js"/>
   </resource>
 </resources>
</manifest>
```

Code Sample 20: A sample manifest

Code Sample 20 shows the XML manifest for a small SCORM 2004 package.

This package contains one activity tree with two activities and no sequencing information, meaning that the user should be free to choose any activity in any order. Each activity uses a different SCO.

If you are not familiar with XML, you will probably need to get a good XML primer to understand everything in this manifest. However, even if you understand XML this sample can help clarify a few things.

Most of what you see in this XML file is defined by the IMS content packaging schema or by an extension schema defined by SCORM. Things that may be different are shown in this ***`format`***. They are the values that you need to specify when you create your own manifest. Let us look at those in a little more detail.

The value for the identifier attribute of the `<manifest>` element should be a globally unique identifier for your package. If your package includes metadata—which is definitely recommended—the identifier in the General element of the metadata should be consistent with this identifier. The globally unique identifier can be a Uniform Resource Identifier (URI), which may be a "handle" as defined in the Handle System (www.handle.net). In any case, it must conform to IETF RFC 2396, which specifies which characters are allowed. For example, the identifier may not contain whitespace characters (space, tab, etc.) unless the characters are URL encoded. Making this identifier unique is important because you don't want this package to be confused with any other package.

The value for the version attribute of the `<manifest>` element can be anything you want, but a value is required.

The `xsi:schemaLocation` attribute of the `<manifest>` specifies the name of the XML schema file (.xsd) that must exist in the “root folder” of your package for each namespace used in the manifest.

The `<adlcp:location>` element is a SCORM specific element. You use this element if metadata are provided in a separate file. The value in the element is the URL of the file relative to the manifest file. Alternatively, metadata to describe the package may be included inline rather than as a separate file.

The `<organization>` element represents the activity tree. It requires an identifier. This identifier is specific to the manifest and although you could make it globally unique that does not really add much value, unless there is a chance that the package may be disaggregated or merged with another package at some future time. The organization element also requires a title. This may typically be shown in user interfaces as the title of the activity tree.

Each `<item>` element specifies an activity or sub-activity. It requires an identifier. This identifier is specific to the manifest and although you could make it globally unique that does not really add much value, unless there is a chance that the package may be disaggregated or merged with another package at some future time. The item element also requires a title. This may typically be shown in user interfaces as the title of the activity. A leaf item (an item with no children) also has an `identifierref` attribute. This attribute specifies the identifier of a resource to use to deliver the activity. This resource must be defined in the same manifest as the item. More than one item may reference the same resource; in other words, you can use the same resource for different activities. In that case, the item might also include parameters that are passed to the resource when it is launched.

Every `<resource>` element in the manifest describes a SCO, a launchable asset, or a collection of one or more files used by other resources.

The `<resource>` element requires an identifier. This identifier is specific to the manifest, but it is recommended to make the identifier globally unique, because resources are often designed to be reusable and may be cannibalized from one package to another, or may be imported from a repository or collection of resources to use in different packages. Making the resource identifiers globally unique is also useful if packages get merged or reworked—if two resources have the same identifiers they could reasonably be assumed to be identical. Like all the elements named “`identifier`” in the manifest schema, the value of this identifier is a string that conforms to the XML schema type `xsd:ID`. This means that it must meet requirements very similar to those of a variable name in many programming language: It must begin with a letter, and may contain only alphanumeric characters, hyphens and underscores.

A launchable resource has a `href` attribute whose value is the URL the SCORM Runtime Environment will use to launch the resource. This is a URL, which means that it may include parameters. For example, "`sco.htm?a=1&b=5`". The URL is relative to the location of the imsmanifest.xml file. Another attribute is used to indicate another relative path, but this is beyond the scope of what we can explain here. Typically, when the resource is launched, the Runtime Environment will prepend a path to that URL and also add any parameters that may have been specified for the `<item>` element that describes the activity that launches the resource.

The resource element contains an inventory of the files required to launch the resource. Each file is specified by a `<file>` element with a `href` attribute. The value of the `href` attribute is the path and name of the file, relative to the location of the imsmanifest.xml file, and is expressed as a URL without any parameters. This value will be used to verify that the file is placed in the correct location when the package is installed on a web server for delivery.

Note that URLs that specify the launch URL for a resource or the location of a file must be valid cross-platform URLs as defined by the IETF. In other words, they must use "`/`" as path separators and not "`\`". Also, they may not begin with a "`/`" which would indicate the root of a file system, since there is no way to predict where in a web browser's directory structure the package will be installed for delivery.

All the files for your package can be in the same directory as the imsmanifest.xml file, or some files may be arranged in various subdirectories. However, all the XML schema files (`.xsd` files) that are required to validate the XML manifest file must be in the same directory as imsmanifest.xml, and that file must be in the "root" directory for the package. In other words you cannot tuck the manifest and schema files into a subdirectory. Except for these constraints, there is no particular requirement to organize your directories or subdirectories in any particular way.

A reusable manifest template

The following listing is an XML manifest template that can be easily edited to suit your purpose. It has provisions for a single SCO. It includes no navigation data.

The namespace declarations and schema location declarations in the manifest element are very important. Typos here are one of the leading causes of problems with validation. Copy and paste rather than trying to type them in is recommended. Even better, automatic generation of the manifest by a tool rather than by hand can help prevent costly and frustrating errors. Although this particular manifest does not use all of them, the namespace declarations for all the schemas and navigation schemas you might find in a standard SCORM package have been copied and pasted here along with the other namespaces.

In this template, the parts that you may change to suit your purpose are indicated with bold type in this listing.

```
<?xml version="1.0" standalone="no"?>
<manifest identifier="(Note 1)" version="(Note 2)"
  xmlns="http://www.imsglobal.org/xsd/imscp_v1p1"
  xmlns:adlcp="http://www.adlnet.org/xsd/adlcp_v1p3"
  xmlns:adlseq="http://www.adlnet.org/xsd/adlseq_v1p3"
  xmlns:adlnav="http://www.adlnet.org/xsd/adlnav_v1p3"
  xmlns:imsss="http://www.imsglobal.org/xsd/imsss"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.imsglobal.org/xsd/imscp_v1p1 imscp_v1p1.xsd
   http://www.imsglobal.org/xsd/imsss imsss_v1p0.xsd
   http://www.adlnet.org/xsd/adlcp_v1p3 adlcp_v1p3.xsd
  http://www.adlnet.org/xsd/adlseq_v1p3 adlseq_v1p3.xsd
  http://www.adlnet.org/xsd/adlnav_v1p3 adlnav_v1p3.xsd">
  <metadata>
    <schema>ADL SCORM</schema>
    <schemaversion>2004 3rd Edition</schemaversion>
  </metadata>
  <organizations default="org_1">
    <organization identifier="org_1">
      <title>(Note 3)</title>
      <item identifier="(Note 4)" identifierref="(Note 5)">
        <title>(Note 6)</title>
      </item>
      (Note 7)
      <imsss:sequencing>
        <imsss:controlMode choiceExit="true" flow="true"/>
        (Note 8)
      </imsss:sequencing>
    </organization>
  </organizations>
  <resources>
    <resource identifier="(Note 5)" adlcp:scormType="sco"
       href="(Note 9)" type="webcontent">
      <file href="(Note 10)"/>
      (Note 11)
    </resource>
  </resources>
</manifest>
```

Code sample 21: A very simple manifest template

Note 1 – The value of `identifier` is a globally unique identifier for this package. Although conformance does not require that you use different identifiers for different versions, it is still a good idea because it is one less thing that might confuse content repositories. The identifier must conform to the XML schema type `xsd:ID`, which means that must begin with a letter and may not contain any spaces or characters that would not be allowed in a variable name in most programming languages. It is not intended to be seen by normal human beings.

Note 2 – The value of `version` can be any string. Better keep it short, though, with no line breaks in case it is displayed somewhere in a user interface. Note that normally an end user interface should use the version information in the package metadata, which also includes a human readable title for the package, rather than this value.

Note 3 – This it the title of the activity tree. If your package contains only one organization, this title can be the same as the title you specify in the metadata. If your package contains more than one organization, some LMS will show the title to end users to allow them to choose between organizations.

Note 4 – A unique identifier for this activity. This identifier must be unique within the manifest, but you should make it globally unique if you ever intend to reuse this item in another activity tree, through disaggregation, re-aggregation or cannibalization. Like all elements named "identifier" in a manifest, it must also be of type `xsd:ID`. It is not intended to be seen by normal human beings.

Note 5 – A unique identifier for the resource element. This identifier must be unique within the manifest, but it is recommended to make it globally unique, because resource elements typically define reusable components. The same resource element may be referenced by more than one activity item.

Note 6 – This is the title for the activity. If the runtime environment shows this activity in a menu or tree view, this is what it will show.

Note 7 – You can of course add more item elements here. Item elements can be nested within item elements, with no limit on depth.

The sequencing specification specifies a flow behavior in addition to the default choice behavior. It is necessary to specify flow here because of a defect in the current sequencing specification that results in not showing any SCO in a choice situation unless a flow is defined also. This may be corrected in a future revision of the specification, but in the meantime having a guided flow for the activities is usually harmless.

Note 8 – By default, the SCORM sequencer will set completion and success to true automatically if the SCO does not explicitly set a value for `cmi.success_status` and `cmi.completion_status`. This allows very simple SCOs that only call Initialize and Terminate to be tracked as completed and successful just by virtue of having been launched and having communicated with the API. If your SCO does set the success status and completion status, but only when the SCO finishes normally, you may have to add an element here to turn off that automatic behavior. If you don't there is a risk that the activity will automatically recorded as completed and successful even if the learner does not go through the activity successfully. See Code sample 22 and refer to the *<deliveryControls> Element* clause in the SCORM Content Aggregation Model book for specific directions. The other way to avoid this behavior is to fix the SCO itself. For example, set `cmi.success_status` and `cmi.completion_status` to a temporary value such as "failed" and "incomplete" in your SCO immediately after the SCO is launched and if the value for those elements is not already set. Later, when the status changes for either success or completion as the SCO runs, you can set the element again to the new value.

Note 9 – This is the relative URL to use to launch the resource. It must be a valid URL, but it may not be a fully qualified URL, i.e. it may not include a fully qualified domain name unless this resource element describes something that is available on the Internet and is not included in the package. A fully portable package contains all the files it needs, and thus the URLs in a fully portable package are always relative to wherever the imsmanifest.xml file happens to be. The runtime environment prepends the fully qualified domain name and typically a path to this URL when it launches the resource.

Note 10 – This is a relative URL to the file itself. It is a "bare" URL without parameters. It is always relative to wherever the imsmanifest.xml file happens to be.

Note 11 – Add a file element for every file required by the resource. If some files are shared with other resources, you can specify a dependency to a common resource element that contains those files. This helps keep the manifest shorter. However, it is not an error to list a file more than once in different resource elements. Every file in your package must be listed in a resource element, and every file listed in a resource element must be present in the package.

```
...
      <imsss:sequencing>
        ...
        <imsss:deliveryControls
            completionSetByContent="true"
            objectiveSetByContent="true"/>
      </imsss:sequencing>
...
```

Code sample 22: Preventing automatic setting of completion or success by the sequencer. See Note 8 above.

Validating your package

You should always validate your packages before trying them out in a real runtime environment.

A convenient way to do this is to install the SCORM test suite available at the ADL web site. Be sure to read the installation instructions the readme.html file carefully *before* you run the setup program. You will probably have to download and install the appropriate older version of the Sun Java environment before installing the test suite. If you do this in the wrong order you may run into strange test suite errors.

In the SCORM test suite, the test you want to run is for a Content Package Conformance Test. When prompted, choose the Content Aggregation test. The rest is pretty much a matter of following the prompts.

Do not get discouraged if the test suite shows errors, especially if you built any part of the package or the manifest by hand. It takes a little practice to get the hang of it. If you are using a higher level tool, of course, you should not get any error. If you do get errors anyway, check your documentation and the support options for your tool.

Adding metadata

Metadata about the package

Although SCORM 2004 3rd edition removed the requirement for metadata describing the content of the package, it is still is a very good idea to add metadata that describe your package. This allows anyone, or any system that can read the manifest, to understand what the package is about. It also allows the automation of some processes, like importing the package in a repository. The import process can scan the metadata and create a catalog entry with the information it found. If you do specify metadata, SCORM specifies a profile with recommended metadata elements, and also recommends that the metadata conform to the IEEE Learning Object Metadata (LOM) standard. This helps ensure that SCORM packages can be used in automated processes, like search, by having a consistent set of metadata.

You may also add metadata to other elements of the manifest. The SCORM Content Aggregation Model book specifies which elements of the manifest can have attached metadata, and which of the metadata elements are required when metadata are attached to some elements. Note however that this is in no way required by SCORM, and there is currently no consensus of best practices to use such metadata.

Metadata about activities

In some cases, you might want to add metadata describing the activities represented by the organization elements and the item elements in the manifest. This is particularly useful if several people collaborate on the creation of the activity tree, as a way to explain the purpose and design of the activities to others who might try to interpret this later.

Metadata about SCOS and assets

If the package is intended for disaggregation in the future, or for archival, you might also want to add metadata to document the resources and assets inventoried in the manifest. However, just because you can do something does not mean that it is a good idea, and this is why the SCORM does not require. If the manifest is too loaded with metadata, it can become very large and cumbersome, with little benefit if the metadata is never actually used. Some communities of practice might require more metadata than required by SCORM, but one must be careful not to confuse useful data and datarrhea.

Light vs. heavy manifests

You might want to make two versions of the manifest—one that goes into an "archival" version of the package, and one that goes into a "delivery" version of the package. The archival version contains metadata that document the elements of the package as completely as possible. The delivery version, on the other hand, contains only the minimum metadata required to support automatic installation in a delivery environment. This is typically just the package level metadata, which are inserted at the top level of the manifest.

You probably noticed in the manifest examples above that there was already a metadata element in the manifest. This element provides metadata about the manifest itself, as well as a placeholder for additional metadata that describe the package.

```
<metadata>
  <schema>ADL SCORM</schema>
  <schemaversion>2004 3rd Edition</schemaversion>
</metadata>
```

Code sample 23: Required metadata about the manifest itself

The values for the `schema` and `schemaversion` elements are fixed. Those metadata elements must be present to specify that this manifest conforms to SCORM 2004 3rd Edition. If you add other metadata to describe the package, it will be in the form of another element within the `metadata` element. That element will then contain your descriptive metadata. SCORM allows you to put these additional metadata inline within the manifest itself, or to put them in the package as a separate XML file. When metadata are in a separate file, the metadata element in the manifest must contain a reference to that file.

The example below shows a reference to a separate metadata file. The next section will examine a sample metadata file in more detail.

```
<metadata>
  <schema>ADL SCORM</schema>
  <schemaversion>2004 3rd Edition</schemaversion>
  <adlcp:location>metadata.xml</adlcp:location>
</metadata>
```

Code sample 24: Referencing an external metadata file.

Sample metadata file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<lom xmlns="http://ltsc.ieee.org/xsd/LOM"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ltsc.ieee.org/xsd/LOM lom.xsd">
  <general>
    <identifier>
      <catalog>Anything</catalog> <entry>12345/12345-12345-12345</entry>
    </identifier>
    <title>
      <string language="en">My Package</string>
    </title>
    <description>
      <string language="en">Description of my package</string>
    </description>
    <keyword><string language="en">something</string></keyword>
    <keyword><string>package</string></keyword>
    <structure>
      <source>LOMv1.0</source><value>atomic</value>
    </structure>
  </general>
  <lifeCycle>
    <version>
      <string>1.0</string>
    </version>
    <status>
      <source>LOMv1.0</source><value>final</value>
    </status>
  </lifeCycle>
  <metaMetadata>
    <identifier>
      <entry>anything</entry>
    </identifier>
    <metadataSchema>LOMv1.0</metadataSchema>
    <metadataSchema>SCORM_CAM_v1.3</metadataSchema>
  </metaMetadata>
  <technical>
    <format>text-html</format>
  </technical>
  <rights>
    <cost>
      <source>LOMv1.0</source><value>no</value>
    </cost>
    <copyrightAndOtherRestrictions>
      <source>LOMv1.0</source><value>yes</value>
    </copyrightAndOtherRestrictions>
  </rights>
</lom>
```

Code sample 25: Sample metadata for a content package.

SCORM 2004 metadata must conform to the IEEE standard for Learning Object Metadata, usually called "IEEE LOM". In that standard, everything is optional. However, the SCORM CAM book defines conformance profiles for various elements of the manifest. Each conformance profile specifies that some elements are mandatory, and also specifies mandatory fixed values for some elements.

Special SCORM packaging issues

Shareable resources

If two or more SCOs are using the same files, you may want to inventory those files into a separate Resource element, which can then be referenced by a dependency element defined in the Resource element that defines each SCO. This may be best understood with an example. In this example, there are two SCOs. The SCOs share a script and some graphics.

```
<?xml version="1.0" standalone="no"?>
<manifest identifier="your_identifier_here" version="your_version_here"
  xmlns="http://www.imsglobal.org/xsd/imscp_v1p1"
  xmlns:adlcp="http://www.adlnet.org/xsd/adlcp_v1p3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.imsglobal.org/xsd/imscp_v1p1 imscp_v1p1.xsd
                      http://www.adlnet.org/xsd/adlcp_v1p3 adlcp_v1p3.xsd">
  <metadata>
    <schema>ADL SCORM</schema>
    <schemaversion>2004 3rd Edition</schemaversion>
  </metadata>
  <organizations default="org_1">
    <organization identifier="org_1">
      <title>your_title_here</title>
      <item identifier="activity_1" identifierref="SCO_1">
        <title>your_title_here</title>
      </item>
      <item identifier="activity_2" identifierref="SCO_2">
        <title>your_title_here</title>
      </item>
    </organization>
  </organizations>
  <resources>
    <resource identifier="SCO_1" adlcp:scormType="sco"
       href="your_file_name.html" type="webcontent">
      <file href="your_file_name.html"/>
      <file href="some_other_file_maybe"/>
      <dependency identifierref="sharedScript"/>
      <dependency identifierref="sharedImages"/>
    </resource>
    <resource identifier="SCO_2" adlcp:scormType="sco"
       href="another_file_name.html" type="webcontent">
      <file href="another_file_name.html"/>
      <file href="etc."/>
      <dependency identifierref="sharedScript"/>
      <dependency identifierref="sharedImages"/>
    </resource>
    <resource identifier="sharedScript" adlcp:scormType="asset"
       type="webcontent">
      <file href="ostyn2004sco.js"/>
    </resource>
    <resource identifier="sharedImages" adlcp:scormType="asset"
       type="webcontent">
      <file href="foo.gif"/>
      <file href="bar.gif"/>
    </resource>
  </resources>
</manifest>
```

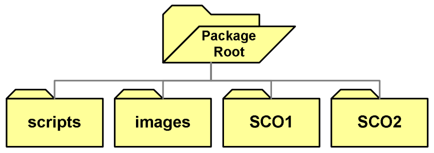
Code sample 26: A manifest with two SCOs sharing some assets

To keep the example simple, the same path was assumed for all the files in this package.

As mentioned earlier, you may organize your files into a more complex structure. That structure must then be reflected in the description and inventory of resources.

The next listing shows what the Resources element of the example manifest above would look like with relative paths. All paths are relative to the root directory of the package, wherever that directory may be in an actual file system structure.

This example assumes this structure:



```
<resources>
  <resource identifier="SCO_1" adlcp:scormType="sco"
     href="SCO1/your_file_name.html" type="webcontent">
    <file href="SCO1/your_file_name.html"/>
    <file href="SCO1/some_other_file_maybe.html"/>
    <dependency identifierref="sharedScript"/>
    <dependency identifierref="sharedImages"/>
  </resource>
  <resource identifier="SCO_2" adlcp:scormType="sco"
     href="SCO2/another_file_name.html" type="webcontent">
    <file href="SCO2/another_file_name.html"/>
    <file href="SCO2/etc.html"/>
    <dependency identifierref="sharedScript"/>
    <dependency identifierref="sharedImages"/>
  </resource>
  <resource identifier="sharedScript" adlcp:scormType="asset"
     type="webcontent">
    <file href="scripts/ostyn2004sco.js"/>
  </resource>
  <resource identifier="sharedImages" adlcp:scormType="asset"
     type="webcontent">
    <file href="images/foo.gif"/>
    <file href="images/bar.gif"/>
  </resource>
</resources>
```

Code sample 27: Sharing resource assets with relative paths

Note that all paths are using the URL form of separator, which is a "/" rather than a "\".

Note also that all references must be relative. So, for example, a URL to an image, as it would appear in a file in SCO1, would be something like img src="../images/foo.gif". You may never use "/" as the leading character in the path for a reference, because that indicates the root of the file system, and it is very unlikely that the package will be installed in the root of the file system in its delivery environment.

SCORM does not support sharing of resources between packages

SCORM 2004 does not support the sharing of resources between packages. This is in part because there is currently no standard that resolves the complicated version control, security and referential integrity issues that would arise when packages get updated or installed at various times. Also, packages are not allowed to be installed "beyond their root": For security and sanity reasons, nothing in a package is allowed to reference anything closer to the file system root than the base directory of the package itself.

Some delivery environments might be entirely under the control of the content creator, and use some method of resource sharing that works around this limitation. However, content packages that exploit this are typically not SCORM 2004 conformant because they cannot be installed safely in other delivery environments.

Server-dependent content

Many content developers would like to be able to claim SCORM conformance for content that is delivered through an active server, such as .php, .asp or .jsp pages. Obviously these are not files that can be included in the package, since they are generated on the fly.

Conformance and portability issues

Since such content is not self-contained, it cannot be reliably archived or transported and it cannot be used offline without recreating the server functionality on the client machine. The content does not conform to the spirit of the SCORM because it is not portable and cannot delivered by generic Web servers without installing some specific active components on the server. There is currently no standard way to specify such an installation.

Leveraging the SCORM

Although server-dependent content is not fully SCORM conformant, it can still take advantage of the SCORM. The content can be installed, delivered, sequenced and tracked in a SCORM runtime environment. The SCORM packaging manifest can be used to represent and communicate the activity tree, and to manage the sequencing of activities--it just contains empty resources that reference the actual server pages. The SCORM launch model and the SCORM API and data model can of course be used for communication and tracking.

Conformance testing issues

Maybe one of the ways to deal with server-dependent would be to have two different test suite flavors, each describing a different flavor of conformance. One flavor is for "self-contained", fully portable content and one for "server-dependent content" which is not portable.

The difference could be ascertained by a different test procedure. Content could be credibly verified as "self-contained" only if the test is run on a "virgin" machine that is not connected to a network (and to the Internet by extension). A "virgin" machine has to be different from the machine on which content was developed, contain nothing but the operating system and the system components required by the test suite, as well as the test suite itself of course, and the physical or virtual directory structure used as repository for content must be cleansed before unpacking of content packages.

In addition, the metadata for the package or for a resource could include, along with the already defined technical metadata elements, a list of all the external dependencies, in the form of a list of URLs.

Packaging server-dependent content

It might or might not make sense to require that the external resources be also copied into the package, because the packaging specification has no way to deal with installation issues beyond the simple unpacking of a file collection into a directory structure. Interoperable installation specifications for server-dependent content can easily become very complex, considering the different platform and configurations that must be considered. For example, .asp or .aspx pages will typically not run on anything but a Windows server with the proper engines installed and enabled. Also, how "deep" the server dependencies go can vary. For example, server-dependent content usually assumes access to a particular database back end that is typically governed by its own administrative rules, security configuration and dependencies. At this point, this is still only a research project.

The cross-server delivery issue

An issue that arises most often with server-dependent content is security. Browser security prevents cross-frame communication between scripts that come from different servers. This is a critical security issue and some loopholes have been progressively closed in the last couple of years as browsers get hardened against increasingly aggressive hacking exploits that were using those loopholes.

There are several solutions to this problem. The simplest is to make the content fully portable and install it on the same server that delivers the runtime environment, which is typically also the LMS server. When this is not possible, as in enterprise deployments where scalability or other configuration issues require different servers, the solution is to use enterprise class networking infrastructure to make the runtime environment and the content appear to come from the same server. Some open source and off the shelf solutions are also available. See the SCORM technical resources page on the www.ostyn.com web site, as well as the ADL web site for more information.

Appendix 1 - Bibliography

SCORM specification documents

This book is based on the SCORM 2004 3rd Edition specification documents as of November 22, 2006. Get the most current version of the SCORM document set at http://www.adlnet.gov/scorm/

Standards and specifications reference

1484.11.2-2003 IEEE Standard for Learning Technology —ECMAScript Application Programming Interface for Content to Runtime Services Communication (2004). Piscataway, NJ: IEEE.

IEEE 1484.11.1-2004 IEEE Standard for Learning Technology — Data Model for Content to Learning Management System Communication. Piscataway, NJ: IEEE.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax (1998). Retrieved 1 November 2006 from http://www.ietf.org/rfc/rfc2396.txt

Standard ECMA-262 - ECMAScript Language Specification (1999). Retrieved 1 November 2006 from http://www.ecma-international.org/publications/standards/Ecma-262.htm

Corporation for National Research Initiatives. *Handle System Documentation* (2003). Retrieved 1 November 2006 from http://www.handle.net/documentation.html
(Note: As of now, SCORM does not support handles directly. Identifiers in a manifest cannot be handles because of syntax conflicts. However you might consider making your own globally unique identifiers compatible with the name portion of a handle.)

Accessibility

IBM Accessibility Center: Frames: Web checkpoint 9 (2004). Retrieved 1 November 2006 from http://www-306.ibm.com/able/guidelines/web/webframes.html

SCORM in a competency management context

Ostyn, Claude. *SCORM, repositories and competencies* (2005). Retrieved 1 November 2006 from http://www.ostyn.com/rescompetency.htm

Ostyn, Claude. *Competency Data for Training Automation*. (2005). Retrieved 1 November 2006 from http://www.ostyn.com/rescompetency.htm

Appendix 2 - Resources

Sample SCORM related scripts

Ostyn, Claude. *ostyn2004sco.js -- Generic reusable script for SCORM 2004 conformant SCOs* (2006). Retrieved 1 November 2006 from http://www.ostyn.com/standards/scorm/samples/ostyn2004sco.js

DHTML scripts for SCOs without plug-in dependencies

Zorn, Walter. *JavaScript: DHTML API, Drag & Drop for Images and Layers* (2004) Retrieved 1 November 2006 from http://www.walterzorn.com/dragdrop/dragdrop_e.htm

Zorn, Walter. *High Performance JavaScript VectorGraphics Library* (2004) Retrieved 1 November 2006 from http://www.walterzorn.com/jsgraphics/jsgraphics_e.htm

Flash ActionScript to SCORM API communication

Note: The following resources have been superseded by new functionality in Flash player 8 and above.

Ramsey, Chip and Wallace, Richard. *Cross-platform Communication Between Macromedia Flash and SCORM Run-Time Environments* (2005). Retrieved 5 February 2005 from http://www.intellum.org/articles/crossplatform-course-communication.html

Cantrell, Christian and Chambers, Mike. *Intro to the Flash/JavaScript Integration Kit* (2005). Retrieved 1 September 2005 from http://weblogs.macromedia.com/flashjavascript/
A more robust method than GetUrl or FSCommand to facilitate communication between Flash movies and scripts on the host web page launched as SCO.

Content developer resources

Rustici, Mike. *SCORM 2004 Reference Poster*(2006). Retrieved 22 January 2007 from http://www.scorm.com/pages/resources.aspx

The Reload Project. *Reload Editor 2.5* (2006). Retrieved 22 January 2007 from http://www.reload.ac.uk/editor.html
This is a Java based open source editor for SCORM packages.

Testing and diagnostics

ADL. *SCORM 20043rd Edition Self-Test Conformance Test Suite* (2006). Retrieved 22 January 2007 from http://www.adlnet.gov.
Note: An updated version of the Conformance Test Suite that corrects some issues with the 2006 release is expected in early 2007.

ADL. *SCORM 20043rd Edition Sample Runtime Environment* (2006). Retrieved 22 January 2007 from http://www.adlnet.gov.
Note: This is only a sample and does not have the full functionality of a practical production environment.

Ostyn, Claude. *SCO Test wrap to monitor SCO communication with LMS (2007)*. Retrieved 22 January 2007 from http://www.ostyn.com/resscormtech.htm#testing

Appendix 3 - A generic, reusable SCO script

Overview of the generic SCO script

The sample generic SCO script (ostyn2004sco.js) that accompanies this book is designed to be included in the header of the HTML page that implements the SCO. This script can be shared by multiple SCOs in a package. A SCO using this script never needs to call the SCORM API directly. The script contains:

- Automatic management of the SCO to RTE communication session. Your SCO does not have to contain scripts to find the API implementation, or to call `Initialize` or `Terminate`; this is handled automatically by the included generic script when the SCO is loaded and unloaded. Your SCO's custom script may however call the `ScormTerminate` function provided by the script whenever it wants to terminate the communication session before the SCO is unloaded.
- Automatic calls to your own custom SCO initialization and cleanup functions when the page is loaded and unloaded. If you choose not to implement those functions, no error will occur.
- Helper functions that can be called from the SCO. A SCO that calls these functions does not need to locate or call the API implementation directly. The SCO can simply call the helper functions, and everything else is handled behind the scenes inside the generic script.
- Automatic features that help manage some common tracking data without any custom coding on your part. These features include automatic reporting of elapsed time, automatic setting of status when setting a score or progress measure, and automatic detection of maximum time allowed for the SCO. Your SCO script can turn those features on or off by setting some global variables, or you can turn them on or off in the version of the generic script you choose to use by presetting different values and saving the script under a different name.
- A debug flag that can be turned on or off to show alerts for significant API events.

Generic SCO script features and functions

Automatic API session initialization and termination

Default behavior

The script adds handlers automatically to the body or frameset of your SCO for the browser events `onload` and `onunload`.

You may also define those handlers for those events in the body or frameset tag of your HTML page. If your HTML page defines a handler for `onload` or `onunload` in the body or frameset tag of your SCO, you should be aware of the way different browsers handle multiple event handlers. In Internet Explorer, the event handlers added by this script will execute after any event handler defined in the body or frameset tag. In FireFox, on the other hand, the event handler added by this script will execute before any event handler defined in the body or frameset tag.

Overriding initialization order

By default, the initialize function is called as soon as the generic script is instantiated and before the body or frameset in your web page is fully loaded. This allows for a page to dynamically write itself differently depending on whether a SCORM API is available or not, for example. If you need to delay the initialization of the communication session until the browser triggers the `onload` event, you must change the preset value for the variable `gbInitializeBeforeLoad` in the generic script from `true` to `false`.

If the API session was initialized successfully, you have access to the API wrapper functions in your script even before it is loaded. The following HTML snippet shows how a SCO can dynamically alter its appearance while being loaded, depending on whether the SCORM API has been initialized successfully by the generic script.

```
...
<body>
<script language="JavaScript">
  if (ScormIsInSession()) //
  {
    document.write("<p>No tracking data will be recorded for this session.</p>");
  }
  else
  {
    var nam = ScormGetValue("cmi.student_name");
    if (nam != "")
    {
      document.write("<p>Hello, " + nam + ".</p>");
    }
  }
</script>
...
```

As another example of the use of this feature, you can determine whether the value of `cmi.entry` is `resume` and, if so, get the value of `cmi.location` or `cmi.suspend_data` and display the appropriate content right away in a frameset you create on the fly.

Automatic calls to your custom initialization and cleanup functions

Initialization

If the script of your SCO contains a function named `SCOSessionInitializedHandler`, this function is called automatically by the generic script after successful initialization of the communication session. No error occurs if your custom script contains no such function.

This allows your custom script to get any data it needs from the RTE by calling `ScormGetValue` as appropriate before the user can interact with your SCO. It also allows your custom script a chance to override the default control flag values in the generic script, because the automatic behavior controlled by those flags are not invoked until after `SCOSessionInitializedHandler` has been called.

The function is not called until the browser triggers the `onload` event.

For example, your custom SCO script might contain a fragment like this:

```
function SCOSessionInitializedHandler()
{
  gbAutoSuccessStatus = false; // turn off one of the generic script auto behaviors
  if (ScormGetValue("cmi.entry") == "resume")
  {
    SetLocation(ScormGetValue("cmi.location"));
    // where SetLocation is a custom function in this custom script
  }
}
```

Note that the generic script has already initialized the API session before `onload` is triggered by the browser if `gbInitializeBeforeLoad` is true in the generic script. If you change the value of `gbInitializeBeforeLoad` to `false` in the generic script, your `SCOSessionInitializedHandler` function will not be called unless you call `ScormInitialize()` in a body or frameset `onload` handler.

Cleanup

If the script of your SCO contains a function named `SCOSessionTerminatingHandler`, this function is called automatically by the generic script when it detects that the SCO is about to be unloaded, and before the generic script terminates the communication session in response to this event. No error occurs if your custom script contains no such function.

This allows your custom script to send any unsaved data to the RTE by calling `ScormSetValue` as appropriate. It also allows your custom script a chance to override any data values set by automatic behaviors, because the generic script does not set any values after calling `SCOSessionTerminatingHandler`.

For example, your custom SCO script might contain a fragment like:

```
function SCOSessionTerminatingHandler()
{
  SaveInteractionData();
  // where SaveInteractionData is a custom function in this custom script
}
```

Helper functions in the reusable script

These functions may be called by any compatible language technology, such as Adobe/Macromedia Flash ActionScript or VBScript. Because of limitations in the ability of Flash ActionScript to get values back from a call to JavaScript, additional scripting may be required to get values back into ActionScript if you use FSCommand, depending on the platforms that need to be supported.

function ScormVersion()

Parameters	None.
Returns	A string.
Description	If there is a current SCORM session, returns "`SCORM 2004`"; otherwise returns "`none`". Note: Another equivalent generic script might return "`SCORM 1.2`" if that script is capable of working in an SCORM 1.2 runtime environment, and detects that the runtime environment is SCORM 1.2 compatible. This function returns "`unknown`" until `ScormInitialize` has been called successfully.

function ScormInitialize()

Parameters	None.
Returns	The string "`true`" or the string "`false`".
Description	Attempts to initialize a SCORM communication session. Returns "`true`" if succeeded, "`false`" otherwise. Note that by default the generic script calls this function automatically while the SCO is loaded. It is exposed in case you want to create a derived version of the generic script that requires a different initialization model. This function keeps track of the communication session status and can be called more than once without causing an error in the API implementation. Redundant calls to this function have no effect.

function ScormTerminate()

Parameters None.

Returns The string "`true`" or the string "`false`".

Description Attempts to terminate the current SCORM communication session. Returns "`true`" if the session that was in progress was successfully terminated, "`false`" otherwise. Once this function has been called successfully, the session is irremediably closed. Only the RTE can allow a new session by relaunching the SCO. The generic script calls this function automatically as soon as it detects an "`onunload`" event. Your SCO's custom script may also call this function earlier, if the design of the SCO design requires. This function keeps track of the communication session status and can be called more than once without causing an error in the API implementation. Redundant calls to this function have no effect.

function ScormGetLastError()

Parameters None.

Returns A string representing an integer value.

Description Returns the number that corresponds to the current communication session error state, as reported by the runtime environment.

Example
```
sErr = ScormGetLastError();
if (sErr !="0") alert(ScormGetErrorText(sErr));
```

function ScormGetErrorString(strErrNumber)

Parameters A string representing an error number defined for the SCORM API.

Returns A string.

Description Returns the string value provided by the runtime environment for the specified error number. If there is no current runtime environment, or if the runtime environment cannot be queried successfully, returns an empty string.

Example
```
sErr = ScormGetLastError();
if (sErr !="0") alert(ScormGetErrorText(sErr));
```

function ScormGetValue(what)

Parameters A string that identifies a data element.

Returns A string.

Description Returns the value for the specified data element, as returned by the runtime environment. If the data element specification is invalid, the string is empty. If this function returns an empty string and you were expecting something else, you can check the reason by calling `ScormGetLastError()`.

Example
```
sErr = ScormGetValue("cmi.entry");
```

function ScormSetValue(what, value)

Parameters A string that identifies a data element, and a string that contains a new value.

Returns The string "`true`" or the string "`false`".

Description Attempts to set the value for the specified data element. Returns "true" if successful, "false" otherwise. If this function returns "false", you can check the reason by calling `ScormGetLastError()`. This function always casts any value passed to it into a string type, as required by the IEEE API. The example below shows a numeric value; the function will convert it to a string before calling the IEEE API.

Example

```
ScormSetValue("cmi.score.scaled", 0.78);
```

function ScormCommit()

Parameters None.

Returns A string.

Description Asks the runtime environment to commit current data to persistent storage. If there is a current SCORM session and the runtime environment reported that the function was successful, returns "`true`"; otherwise returns "`false`". If your script is calling `ScormSetValue` for a series of elements, do not call this function after each call to `ScormSetValue`, but only at the end of the series.

Example

```
ScormSetValue("cmi.score.min", nScoreMin);
ScormSetValue("cmi.score.max", nScoreMax);
ScormSetValue("cmi.score.raw", nScoreRaw);
ScormSetValue("cmi.score.scaled", nScoreScaled);
ScormCommit();
```

function ScormIsInSession()

Parameters None.

Returns A string with the value "`true`" or "`false`".

Description If there is a current SCORM session, returns "`true`"; otherwise returns "`false`". Returns "`true`" if the session has been successfully initialized and has not been terminated yet. Note that this function "`true`" if the session is in the process of being terminated, but has not been actually terminated yet.

Example

```
if (ScormIsInSession())
{
  if (ScormSetValue("cmi.score.scaled", nScoreScaled) == "true"
  {
    ScormCommit();
  }
}
```

function ScormMarkInitElapsedTime()

Parameters None.

Returns The number of centiseconds since January 1, 1970.

Description Sets or resets the point in time that is used by the generic script as the beginning of the current session, and that is used to set the value of `cmi.session_time` when the session is terminated. This point in time is set automatically by the generic script, but a custom SCO script may call this function during a communication session to reset the beginning of the tracked session time to some later time. For example, if a SCO begins with game instructions to a learner, this function could be called when the game is actually started.

If a time limit is in effect for the SCO, and this function is called before the time limit has been reached, the time limit is extended because the same initial timing mark is used by the generic script's time limit detection feature.

Example

```
if (ScormIsInSession())
{
  ScormMarkInitElapsedTime();
}
```

function ScormInteractionAddRecord (strID, strType)

Parameters A string that identifies an interaction, and a string that specifies an interaction type.

Returns An integer.

Description Attempts to add an interaction record. Takes as parameters the identifier of the interaction and a valid interaction type. Returns an integer which is the index of the interaction record in the interaction records array for the current communication session, or `-1` if the function failed. If an interaction record with the same identifier and the same type already exists, the function does not do anything and just returns the index of the existing record. If an interaction record with the same identifier but a different interaction type already exists, the function fails. If the number of allowed interaction records would be exceeded by the addition of a new record, the function fails.

Example

```
ScormInteractionAddRecord("Q12345678", "performance")
```

function ScormInteractionGetCount()

Parameters None.

Returns An integer.

Description Returns the current count of interaction records.

Example

```
numberOfInteractions = ScormInteractionGetCount();
```

function ScormInteractionGetData(strID, strElem)

Parameters A string that identifies an interaction, and a string that identifies a data element within an interaction record.

Returns A string

Description Returns the value for the specified data element of the specified interaction record, as returned by the runtime environment. If the identifier does not match an existing interaction record, or if the data element specification is invalid, the string is empty. If this function returns an empty string and you were expecting something else, you can check the reason by calling `ScormGetLastError()`.

Example

```
nCorrectResp=
ScormInteractionGetData("Q12345678","correct_responses._count")
```

function ScormInteractionGetIndex(strID)

Parameters A string that identifies an interaction.

Returns An integer.

Description Returns the current index of the interaction record matching the specified identifier in the interaction records array for the current communication session. Typically, your script does not need to call this function because all other interaction helper functions use the identifier as the key to locate the proper record. It is however provided to support some special optimizations.

Example

```
i = ScormInteractionGetIndex("Q12345678")
```

function ScormInteractionSetData(strID, strElem, strVal)

Parameters A string that identifies an interaction, a string that identifies a data element within an interaction record, and a string that contains the value to set.

Returns The string "`true`" or the string "`false`".

Description Attempts to set the value for the specified data element of the specified interaction record. If the function succeeds, it returns "`true`". If the identifier does not match an existing interaction record, or if the data element specification is invalid, the function returns "`false`". If this function returns "`false`", you can check the reason by calling ScormGetLastError().

Example
```
test = ScormInteractionSetData("Q12345678","learner_response", "456");
```

function ScormObjectiveAddRecord (strID)

Parameters A string that identifies an objective.

Returns An integer.

Description Attempts to add an objective record. Takes as parameters the identifier of the objective. Returns an integer which is the index of the objective record in the objective records array for the current communication session, or `-1` if the function failed. If an objective record with the same identifier already exists, the function does not do anything and just returns the index of the existing record. If the number of allowed objective records would be exceeded by the addition of a new record, the function fails.

Example
```
test = ScormObjectiveAddRecord("urn:ADL:interaction-id-0001");
```

function ScormObjectiveGetCount()

Parameters None.

Returns An integer.

Description Returns the current count of objective records.

Example
```
numberOfObjectives = ScormObjectiveGetCount();
```

function ScormObjectiveGetData(strID, strElem)

Parameters A string that identifies an objective and a string that identifies a data element within an objective record.

Returns A string.

Description Returns the value for the specified data element of the specified objective record, as returned by the runtime environment. If the identifier does not match an existing objective record, or if the data element specification is invalid, the string is empty. If this function returns an empty string and you were expecting something else, you can check the reason by calling `ScormGetLastError()`.

Example
```
stat = ScormObjectiveGetData("urn:ADL:interaction-id-0001",
  "success_status");
```

function ScormObjectiveGetIndex(strID)

Parameters A string that identifies an objective.

Returns An integer.

Description Returns the current index of the objective record matching the specified identifier in the objective records array for the current communication session. Typically, your script does not need to call this function because all other objective helper functions use the identifier as the key to locate the proper record. It is however provided to support some special optimizations.

Example

```
i = ScormObjectiveGetIndex("urn:ADL:interaction-id-0001")
```

function ScormObjectiveSetData(strID, strElem, strVal)

Parameters A string that identifies an objective, a string that identifies a data element within an objective record, and a string that contains the value to set.

Returns The string "`true`" or the string "`false`".

Description Attempts to set the value for the specified data element of the specified objective record. If the function succeeds, it returns "`true`". If the identifier does not match an existing objective record, the function attempts to add an objective record automatically. If that fails, or if the data element specification is invalid, the function returns "`false`". If this function returns "`false`", you can check the reason by calling `ScormGetLastError()`.

Example

```
test = ScormObjectiveSetData("urn:ADL:interaction-id-0001",
  "success_status", "passed");
```

function ScormGetSessionState()

Parameters None.

Returns An integer representing the current state of the communication session.

Description Returns the current state of the communication session, from the point of view of the SCO. The returned value can be:

`0` – Session initiation not attempted yet
`1` – Currently attempting to initiate session
`2` – Session successfully initiated and currently in progress
`3` – Currently attempting to terminate session
`4` – Session terminated
`-1` – Initialization failed (no API found, or other initialization error).

This function can be called if your custom SCO scripts need to ascertain the current status of the communication session. The communication session itself is managed automatically by the generic script.

Example

```
if (ScormGetSessionState() > 2) alert("Oops. Cannot reinitialize!")
```

The following functions are helper functions to deal with elapsed time and time interval value conversions. SCORM uses a specific ISO standard format to communicate durations. These functions hide the complexity of converting to and from that format.

function centisecsToISODuration(nCentiSecs)

Parameters A positive number that represents a number of centiseconds.

Returns A string representing the duration in ISO format.

Description C a numeric duration expressed in centiseconds into the ISO format required for communication with the SCORM API. A centisecond is 1/100th of a second, and is the finest time interval resolution supported by SCORM. For example, 41567 is translated to "`PT6M55.67S`" If the parameter is a floating point value, it is rounded to the nearest integer before conversion.

Example

```
nNowMs = (new Date()).milliseconds
nLatencyMs= (nNow - nQuestionDisplayTimeMs); // latency in milliseconds
strLatency = centisecsToISODuration(nLatency / 10);
test = ScormInteractionSetData("urn:ADL:interaction-id-0001",
  "latency", strLatency)
```

function ISODurationToCentisec(strISO)

Parameters A string representing the duration in ISO format.

Returns A positive integer that represents a number of centiseconds.

Description Converts a duration expressed in the ISO format required for communication with the SCORM API into a number of centiseconds. A centisecond is 1/100th of a second, and is the finest time interval resolution supported by SCORM. For example, "`PT6M55.67S`" is converted to 41567. If the parameter is not a valid ISO representation for a duration, the return value is 0.

Example

```
s = ScormInteractionGetData("urn:ADL:interaction-id-0001,"latency");
nLatencySeconds = ISODurationToCentisec(s) / 100;
```

function CentisecsSinceSessionStart()

Parameters None.

Returns A number of centiseconds.

Description Returns the time elapsed since the current communication session with the API was successfully initiated. If the current communication session was not initiated, returns 0. If the communication session has already been terminated, returns the elapsed communication time as it was at the time the session was terminated. Note: This function is not affected by the value of the control variable `gbAutoElapsedTime`.

Example

```
if (ScormGetSessionState() > 1)
{
  nSecondsElapsed = CentisecsSinceSessionStart() / 100;
}
```

function CentisecsSinceAttemptStart()

Parameters None.

Returns A number of centiseconds.

Description Returns the time elapsed in all the communication sessions that took place for this SCO in the current attempt, including the time elapsed in the current session. An attempt may include several launches of the SCO, each of which results in a communication session. If the current communication session was not initiated, returns `0`. If the communication session has already been terminated, returns the total elapsed communication time for all sessions in the attempt, as it was at the time the session was terminated. Note: This function is not affected by the value of the control variable gbAutoElapsedTime.

Example	`if (ScormGetSessionState() > 1)`
	`{`
	`  nSecondsElapsedInAttempt = CentisecsSinceAttemptStart() / 100;`
	`  nSecondsElapsedInThisSession = CentisecsSinceSessionStart() / 100;`
	`}`

Controllable automated features in the reusable script

The automated features listed below can be turned on or off or tweaked by setting the corresponding control variables. If you want to change the default behavior, you may modify the value those variables in a function named `SCOSessionInitializedHandler` in your custom SCO script. If you set the variables from your custom script, there is no need to modify the generic script in any way, because these automated features are triggered only after that function has been called.

The features are:

- Automated elapsed time reporting
- Automated success status reporting
- Automated completion status reporting
- Automated passing score if completed
- Automated time out if maximum time allowed for the SCO is exceeded
- Robust handling of unexpected forced unloading of the SCO by handling `onbeforeunload`
- Automatic closing of the window at end of the communication session (if allowed)

Example

```
// In your SCO script
function SCOSessionInitializedHandler()
{
  // Score will be set automatically to 100% if completion status is "completed"
  gbAutoPassingScoreIfCompleted = true;
}
```

Automated elapsed time reporting

Description	This feature reports to the RTE the time elapsed in the SCO since the beginning of the communication session. It also sets a global variable with the total time elapsed in previous learner sessions in the SCO, as reported by the RTE. For convenience, the value of this variable is in centiseconds (1/100 of a second) rather than the ISO string value used for communication with the API.
Control Variables	gbAutoElapsedTime controls whether the time elapsed in the SCO will be reported automatically by the generic script. It is true by default.
Data variables	gnTotalTime is the value obtained from the RTE for the data element cmi.total_time. If no value is available from the RTE, the value is 0. It never changes during a session. For convenience, the value of this variable is in centiseconds (1/100 of a second) rather than the ISO string value used for communication with the API. For example, to calculate the total session time at any moment, you can add the value of this variable to the number of centiseconds elapsed in the current session.

Automated success status reporting

Description This feature reports to the RTE the success status value of "`passed`" or "`failed`" when a score is reported.

Control Variables `gbAutoSuccessStatus` controls whether the success status value of "`passed`" or "`failed`" when `ScormSetValue` is called to set `cmi.scaled_score`. It is `true` by default.

Data variables `gbPassingScore` is the default threshold score value below which the success status is "`failed`". You can preset it to any valid floating point value you want in the range -1 to 1. Note however that this value will be changed if the RTE returns a different value, and that the threshold value from the RTE overrides the preset value.

Automated completion status reporting

Description This feature reports to the RTE the completion status value of "`not attempted`", "completed" or "incomplete" whenever `ScormSetValue` is called to set `cmi.progress_measure`, according to the following table:

Value of `progress_measure`	`cmi.completion_status`
`0`	"`not attempted`"
> `0` and < value of `gbCompletionThreshold` (see below)	"`incomplete`"
>= value of `gbCompletionThreshold`	"`completed`"

If the SCO never sets `cmi.progress_measure`, then a coarse completion status is set automatically as follows: When the communication session begins, if the completion status as reported by the RTE is "`unknown`" or "`not attempted`", the completion status is set to "`incomplete`". When the communication session ends, the completion status is set to "`completed`". To prevent this from happening, either set `gbAutoCoarseCompletionStatus` to false, or call `ScormSetValue` from your own script to set `cmi.progress_measure` to any valid value.

Control Variables `gbAutoFineCompletionStatus` controls whether the completion status is set when `ScormSetValue` is called to set `cmi.progress_measure`. It is `true` by default.

`gbAutoCoarseCompletionStatus` controls whether the completion status is set automatically to "incomplete" when the communication session is initiated, and to "completed" when the communication session is terminated. It is true by default, but is automatically set to false if `gbAutoFineCompletionStatus` is true and `ScormSetValue` is called to set `cmi.progress_measure`, because a fine value is better than a coarse value.

Data variables `gnCompletionThreshold` is the default threshold progress value below which the completion status is "incomplete". You can preset it to any valid floating point value you want in the range 0 to 1. Note however that this value will be changed if the RTE returns a different value, and that the threshold value from the RTE overrides the preset value.

Automated passing score if completed

Description This feature if disabled by default, but may be useful in some situations. It automatically reports to the RTE a scaled score of 1.0 (100%) if the completion status value is "`completed`" when the session is terminated, and no score has been previously recorded. If automated success status reporting is enabled, this will also set success status to "`passed`". If a score or success status has already been reported, or if the value of completion status is anything else than "`completed`", this feature does nothing even if it is enabled.

This feature can be enabled by a custom SCO script by setting the value of the control variable to true.

Control Variables `gbAutoPassingScoreIfCompleted` controls whether this feature is enabled if the completion status is "completed" when the session is terminated. Its default value is `false`.

Data variables None

Automated time out if maximum time allowed for the SCO is exceeded

Description This feature if enabled by default. If the RTE specifies a maximum allowed time for the SCO, this behavior keeps track of the session elapsed time by checking the elapsed time at regular intervals. If a time out occurs. If the custom SCO script contains a function named "`SCOTimeLimitDetected`", that function is called; if a function named "`SCOTimeLimitDetected`" does not exist, the generic script queries the RTE for a an action to perform when a time limit is reached, and performs the corresponding action. See "`cmi.time_limit_action`" in the SCORM RTE document for the possible time limit actions.

This feature can be disabled by a custom SCO script by setting the value of the control variable to `false`. The interval at which time checks occur can be set by setting the value of another control variable.

Control Variables `gbAutoTrackAllowedTime` controls whether the automated time limit detection feature is enabled.

`gnAutoTrackAllowedTimePeriod` controls the interval for time checks. The value of this variable is a number of `centiseconds`. The default value is `300` (3 seconds).

Data variables None

Optional handling of unexpected forced unloading of the SCO by handling `onbeforeunload`

Description This feature can make the saving of data by a SCO more robust when there is a lot of system latency. However, if disabled by default because it can be triggered prematurely by various things in your SCO, such as Flash content that gets unloaded while the SCO is executing or if your SCO uses popup windows of its own. When it is enabled, the `ScormTerminate` function is called automatically before the browser unloads the SCO. This can be more robust than handling the cleanup and final data saving performed by `ScormTerminate` in an unload handler, because some older browsers have already started unloading the HTML page when `onunload` is called. However, if your SCO uses popup windows or Flash, `onbeforeunload` will be called prematurely and also terminate your communication session prematurely is this feature is enabled. This feature should not be needed with modern browsers like FireFox or IE6, and it has no effect in Safari and Konqueror since they don't implement `onbeforeunload` events.

This feature can be enabled by a custom SCO script by setting the value of the control variable to true.

Control Variables `gbTerminateOnBeforeUnload` controls whether this feature is enabled. Its default value is false.

Data variables None

Automatic closing of the window at end of the communication session (if allowed)

Description This feature if enabled by default, but will have an effect only if the SCO is launched in a popup window. See the SCORM RTE book for specification of the allowable behavior for a SCO. If this feature is enabled and the SCO is running in a top level window, the SCO attempts to close the window automatically when the API communication session is terminated by a call from the SCO to `ScormTerminate`. Note that, depending on how a LMS launches the content, in some cases browser security settings will prevent a script from closing the window even if this feature is enabled and the window is a top level window. The SCO should be designed to handle this contingency by showing a neutral display before calling `ScormTerminate`.

This feature can be disabled by a custom SCO script by setting the value of the control variable to false.

Control Variables `gbEnableAutoCloseWindow` controls whether this feature is enabled. Its default value is true.

Data variables None

Appendix 4 - Glossary

This book unavoidably contains specialized jargon and acronyms. Sometimes the meaning of a term in the SCORM specification is not the same as the common meaning of the term.

Activity – In SCORM, an activity is what happens when a SCORM package is delivered to a learner. Activities can be nested, i.e. an activity may consist of sub-activities.

activity tree – In SCORM, a tree structure that represents the way activities are organized for delivery to a learner. Nodes in the tree structure represent the "root activity" (using the package) and sub-activities (e.g. using a tutorial within the package)

API – Abstract Programming Interface – A well-defined set of methods and protocols by which a software component can communicate with another one. The SCORM API specifies how a SCO communicates with the RTE that launches it.

asset -- In SCORM, asset has 3 different meanings. a. A file, generally speaking. b. A launchable resource used for an activity, but that does not communicate with the runtime environment. c. A file described in the inventory of files in the description of a resource.

Data model – A well-defined set of definitions for an organized collection of data elements. It specifies the data types, value spaces, and, where appropriate, relationships between data elements.

LMS – Learning Management System. A system that manages learner and learner activities. In the SCORM specification, the term LMS is sometimes used to mean Run Time Environment (see RTE).

manifest – An XML document that describes the content of a SCORM package.

metadata – Data about some other data.

package – A collection of digital objects that represent an aggregation of digital content. A SCORM package contains various files and a manifest that describes the rest of the content of the package.

resource – a. Anything that is used for learning. b. In a SCORM package manifest, a data element that represents can represent a SCO or a launchable asset, or that can represent a collection of files used by other resources.

RTE – Run Time Environment. The software environment, usually provided by a LMS, in which SCORM content is delivered and sequenced for an end user.

SCO – Sharable Content Object – A content resource that can be launched by a RTE to deliver an activity specified in a SCORM package manifest. A SCO is expected to communicate with the RTE to provide tracking data or to get data it can use to adapt its behavior.

Index