Sequencing Engine

The following white paper describes the steps required to integrate the Icodeon Sequencing Engine into existing learning technology systems.

The sequencing engine is a pure Java software component and integrates seamlessly into J2EE environments, typically web application frameworks or Java clients.

- Read in an IMS Content Package manifest file. If the manifest file contains no sequencing information, default sequencing behaviours are instantiated.
- Create an activity tree from the manifest file.
- Pass the tree to an instance of the sequencing engine.
- Issue a Start, Continue, Previous, Choice or Exit navigation request to the sequencing engine to trigger the overall sequencing process.
- Access the current activity identified by the sequencing engine.
- Render the current activity resources to the user.
- Place the activity tree in session state to maintain sequencing state between discrete user requests.

These steps are described in more detail on the following pages, with exemplar code that uses the high level sequencing API from Icodeon.

Visit: Icodeon Ltd, **www.icodeon.com**

Integrating the Icodeon Sequencing Engine

The following description explains the addition of the Icodeon Sequencing Engine to a software application. Implementers should be familiar with the following concepts, particularly from the IMS Simple Sequencing Specification:

- Navigation Requests
- Activities
- Activity Tree
- Local and Global Objectives
- Overall Sequencing Process and Loop
- IMS Content Package manifest file, imsmanifest.xml
- Assessment items (QTI Items or SCOs)

Sequencing state is maintained between **navigation requests** by keeping a unique instance of the sequencing **activity tree** for each user in session state.

1. Import the sequencing APIs into the Java class. Only the APIs and their associated factory methods are required. Implementation classes are specified as String representations of their class names passed as arguments to factory methods:

```
import com.icodeon.ims.ss.info.state.api.*;
import com.icodeon.ims.ss.engine.api.*;
import com.icodeon.ims.ss.persistence.api.*;
import com.icodeon.shared.api.*;
```

2. Use the StateModelFactory to create an instance of the sequencing **activity tree** for each user based on the imsmanifest .xml file in the root of a IMS Content Package. A class name for an implementation of a global objective reader and global objective writer is also passed to the **activity tree** to allow **local objectives** to read from and write to sequencing **global objectives**.

```
ActivityTree activityTree =
      stateModelfactory.createActivityTree(
            imsManifest,
            userID,
            readerClassName,
            writerClassName);
```

3. Use the EngineFactory to create an instance of a **sequencing engine**. Each **sequencing engine** is a JavaBean that is created, used and then disposed during each **navigation request**:

```
EngineFactory engineFactory =
      (EngineFactory) FactoryManager.newInstance(
            factoryImplClassName);
SequencingEngine sequencingEngine =
      engineFactory.createSequencingEngine();
```

4. Pass the **activity tree** as an argument of a mutator method to the **sequencing engine**:

```
sequencingEngine.setActivityTree(activityTree);
```

5. Issue a **navigation request** to trigger the **overall sequencing process**. Two arguments are required. First, the navigation request type, such as start, previous, continue, choice and exit all. The second argument is the activity identifier if a choice request is specified. The identifier is the **item identifier** in the **IMS Content Package manifest**:

```
sequencingEngine.setNavigationRequestType(
      RequestType.START,
      null);
```

6. Get the **current activity** discovered by the **overall sequencing process**:

```
      Activity currentActivity =
sequencingEngine.getCurrentActivity();
```

7. Get the required (usually relative) URL from the **activity**:

```
Resource resource = currentActivity.getResource();
String href = resource.getHref();
```

8. Put the **activity tree** into session state:

```
session.setAttribute(
      SessionKeys.ACTIVITY_TREE_SESSION_KEY,
      activityTree);
```

9. Get any error messages from the sequencing engine:

```
lastException = sequencingEngine.getLastException();
```

10. Dispose the transient instance of the **sequencing engine** bean:

```
sequencingEngine = null;
```

11. Wait for next **navigation request**:

If the activity is a test item, then the result of the test will need to be passed to the appropriate activity to update the **objective progress information**. For QTI (Question and Test Interoperability) items, a QTI response processor will be required. For SCOs (Shareable Content Objects), the run time environment will provide the key/value pairs from the CMI datamodel.

Updating **Objective Progress Information**:

```
            Objectives objectives =
currentActivity.getSequencing().getObjectives();
            Objective objective = objectives.getPrimaryObjective();
            objective.setProgressStatus(true);
            objective.setSatisfiedStatus(true);
```

The learning technology system into which the sequencing engine is integrated may also update the **activity** and **attempt progress information**.

Updating **Activity Progress Information**:

```
      currentActivity.setActivityAbsoluteDuration(231.4)
```