Hibernation of Advanced Railroad Trains (ArrT)

Schritt Drei – ..... and Communicate them (aCt)

SMS Facility Loaders

Dieser Hibernation Report beschäftigt sich mit den sogenannten Facility Loadern.

Facility Loader sind Prototypen des SMUOS Frameworks, die dabei helfen, dynamische Elemente zu laden, dynamische Elemente werden nämlich auch als „dynamic facilities“ bezeichnet.

Jetzt, während der Implementierung von Step 0033.11, erhält dieser Hibernation Report die Versionsnummer 3.x, die Kapitel 1 – 3 sollten bereits den endgültigen Inhalt besitzen.

Das Kapitel 4 jedoch über „Unbound Objects (UBOs)“ wird sich noch ändern, wenn die endgültige Version der Software von Step 0033.11 freigegeben sein wird.

THIS HR WILL BE TRANSLATED TO ENGLISH LANGUAGE SOON

Table of Content

1 Dynamische Elemente – Dynamic Facilities

1.1 Einleitung

Als „facility“ bezeichnen wir eigentlich jedes Element einer SMS – also einer Simple Multiuser Scene –, das irgendwie für den Benutzer in Erscheinung tritt, sei es nun ein Avatar, ein Modul, ein Modell, ein MIDAS Objekt oder irgendeine Ansammlung von Informationen, die in der Szene ihren Niederschlag finden.

Als „dynamische“ Elemente, also „dynamic facilities“, bezeichnen wir dabei all jene Elemente, die nicht automatisch beim Laden und Initialisieren der Szeneninstanz mitgeladen und -initialisiert werden, sondern erst später im Laufe der Simulation „**nach“-geladen**.

Dabei ist zu berücksichtigen, dass es irgendeinen **Auslöser** geben muss, der das Nachladen des dynamischen Elements bewirkt.

Weiters können dynamische Elemente gelöscht – sozusagen entladen – werden, lange bevor die Szeneninstanz zerstört wird. Hier ist die Besonderheit zu berücksichtigen, dass die meisten X3D Player den Content, der nicht mehr benötigt wird, nicht sofort aus dem Speicher entfernen, sondern noch eine Zeit lang dort belassen.

Deswegen muss man diesen Content **explizit „disablen“**, bevor man ihn aus der Szene entfernt, damit er sich ab dann komplett „passiv“ verhält und die Simulation nicht mehr stören kann.

1.2 Vorgaben des SMUOS Frameworks

Das SMUOS Framework trifft über alle „dynamic facilities“ in einer SMS folgende Annahmen:

- Ein dynamisches Element lässt sich mit Hilfe der Methode Browser.createVrmlFromURL() laden, man benötigt dazu nur den URL des Elements
- Jedes dynamische Element ist ein einzelner X3D Knoten, der mit Hilfe dieser Methode geladen wird (Browser.createVrmlFromURL() hat eigentlich einen MFNode-Wert als Ergebnis, aber in unserem Fall wird nur das Element mit dem Index [0] verwendet)
- Alle dynamischen Elemente einer „Klasse von dynamischen Elementen“ lassen sich indexieren und in einem MFNode – Feld abspeichern
- Dynamische Elemente können – und müssen – entsprechend den Konzepten des SMUOS Frameworks eine „basic initialization“ durchlaufen, bevor sie verwendet werden können
- Dynamische Elemente können – und müssen – mit Hilfe des Felds disable (SFTime) disablet werden, bevor man sie aus der Szene entfernt

1.3 Der SMS Loader

Der SMS Loader ist im **Prototypen SmsLoader** implementiert und bietet grundlegende Unterstützung beim Laden von dynamischen Elementen.

Die dynamischen Elemente werden im Feld **dynElems (MFNode)** gespeichert, wobei unbelegte Indices nicht mit dem Wert NULL belegt werden, sondern mit „irgendeinem Pointer“.

Der Status jedes Elements lässt sich über das Feld **dynElemStates (MFInt32)** feststellen, wobei zu jedem Zeitpunkt garantiert ist, dass **dynElems** und **dynElemStates** dieselbe Dimension haben.

Der SMS Loader ruft die Methode **Browser.createVrmlFromURL()** auf, führt dann selbst die „basic initialization“ des Elements durch und dient danach als „Schrittmacher“ für die Initialisierung des dynamischen Elements, die aber der User im wesentlichen selbst vornehmen muss, wenn auch vom SMS Loader getaktet.

Erst, wenn das dynamische Element fertig initialisiert ist, kann dieser SMS Loader das nächste Element laden. Der SMS Loader führt aber eine Warteschlange, in der die dynamischen Elemente auf das Laden warten, deren Laden bereits getriggert worden ist.

Wenn der SMS Loader den Befehl bekommt, ein Element aus der Szene zu entfernen, dann kümmert er sich darum, dass das Element vorher disablet wird

Wenn der SMS Loader disablet wird, dann sorgt er dafür, dass alle geladenen Elemente disablet und aus der Szene entfernt werden

1.3.1 „Basic Initialization“ des SMS Loader

Da der SMS Loader einer von vielen SMUOS Prototypen ist, bietet er an seinem externen Interface auch die Felder, die alle SMUOS Prototypen bieten, nämlich:

```
<!-- Common fields for the MASTER/DEP state machine →
<field accessType='outputOnly' name='sendLoaded' type='SFBool'/>
<field accessType='inputOnly' name='receivePing' type='SFBool'/>
<field accessType='outputOnly' name='sendPong' type='SFBool'/>
<field accessType='inputOnly' name='receiveBasicInit' type='SFBool'/>
<!-- Common fields for all SMUOS prototypes →
<field accessType='outputOnly' name='objType' type='SFString'/>
<field accessType='outputOnly' name='version' type='SFFloat'/>
```

Die Felder **objType** und **version** sind dazu da, eine Instanz des SMS Loader als solche zu identifizieren, die anderen vier Felder dienen der „Basic Initialization“, d.h. die Szene wartet, bis alle externen Prototypen geladen sind, bevor das Feld **receiveBasicInit** aufgerufen wird.

Durch den Aufruf des Feldes **receiveBasicInit** legt der SMS Loader die Größe der Felder **dynElems** und **dynElemStates** vorläufig fest, und zwar basierend auf dem Feld

<field accessType='initializeOnly' name='**typicalDynElemSpace**' type='SFInt32' value='20'/>,

und gibt an die Umgebung die tatsächliche Größe im Event **dynElemSpace** aus:

<field accessType='outputOnly' name='**dynElemSpace**' type='SFInt32'/>

Jetzt sind folgende zwei Felder initialisiert und können ab sofort verwendet werden:

<field accessType='inputOutput' name='**dynElems**' type='MFNode'></field>
<field accessType='inputOutput' name='**dynElemStates**' type='MFInt32' value=''/>

1.3.2 Initialisierung und Disabling des SMS Loader

Der SMS Loader bietet folgende Felder für Initialisierung und Disabling:

<field accessType='inputOutput' name='**commParam**' type='SFNode' value='NULL'/>
<field accessType='inputOutput' name='**modParam**' type='SFNode' value='NULL'/>
<field accessType='inputOnly' name='**disable**' type='SFTime'/>

Da der SMS Loader eine Referenz auf die Common Parameters (**commParam**) benötigt, muss man ihn entweder mit dem Feld **commParam** oder mit dem Feld **modParam** (Module Parameters) initialisieren, bevor man ihn benützt.

Nach Gebrauch sollte man den SMS Loader mit dem Feld „disable“ wieder in einen passiven Zustand versetzen. Dadurch ist auch sichergestellt, dass etwaige dynamische Elemente, die noch geladen sind, sicher disablet und aus der Szene entfernt werden.

Man kann den SMS Loader mehrmals hintereinander initialisieren und wieder disablen. Das kann man zum Beispiel nützen, um „mit einem Schlag“ alle dynamischen Elemente einer „Klasse von dynamischen Elementen“ zu disablen und zu entladen.

Wenn man den SMS Loader mit den commParam initialisiert, dann speichert er sich diesen Pointer zur späteren Verwendung. Wenn gerade ein Disabling am Laufen ist, wird zuerst gewartet, bis das Disabling abgeschlossen ist.

Wenn man den SMS Loader mit den modParam initialisiert, dann geschieht dasselbe, zusätzlich jedoch hört der SMS Loader ab sofort auf das **disable** Feld der modParam und disablet sich selbst, sobald das Modul disablet wird.

1.3.3 Laden und Initialisieren eines dynamischen Elements

Zur Realisierung dieser Funktion bietet der SMS Loader folgende Felder an seinem externen Interface:

```
<field accessType='inputOutput' name='numOfProcedures' type='SFInt32' value='0'/>
<field accessType='inputOutput' name='urls' type='MFString' value=''/>
<field accessType='inputOnly' name='loadElement' type='SFInt32'/>
<field accessType='outputOnly' name='elementQueued' type='SFInt32'/>
<field accessType='outputOnly' name='elementOccupied' type='SFInt32'/>
<field accessType='outputOnly' name='elementFailed' type='SFInt32'/>
<field accessType='outputOnly' name='initializeElement' type='SFInt32'/>
<field accessType='inputOnly' name='finishProcedure' type='SFInt32'/>
<field accessType='outputOnly' name='elementReady' type='SFInt32'/>
```

Als Vorbereitung für das Laden und Initialisieren eines dynamischen Elements muss der User folgende Felder setzen:

numOfProcedures.....Anzahl der Prozeduren, in die die Initialisierung zerlegt werden soll
urls.....dieser Wert soll der Methode **Browser.createVrmlFromURL()** übergeben werden

Um das Laden tatsächlich zu triggern, muss nun der User das Feld **loadElement** setzen, und zwar auf den gewünschten Wert ***dynElemIdx***, wenn er den gewünschten Index des Elements schon weiss und auf einen Wert kleiner als 0, wenn er diesen Index noch nicht weiss.

Daraufhin fügt der SMS Loader die Informationen über das zu ladende Element in die interne Queue ein (numOfProcedures, urls und dynElemIdx) und meldet im Feld **elementQueued** = ***dynElemIdx*** zurück, dass das Element jetzt in der Queue liegt. Hier wird schon der endgültige Elementindex verwendet und es werden auch die Dimensionen der Felder **dynElems** und **dynElemStates** vergrößert, falls der Index nicht in die alten Dimensionen passt.

Wenn dann das Element an die Reihe kommt, wenn also die vorherigen Elemente in der Queue abgearbeitet worden sind, dann wird das Element tatsächlich geladen.

Zuerst wird überprüft, ob **dynElemStates**[***dynElemIdx***] größer oder gleich Null ist, ob also schon ein anderes Element an diesem Index gespeichert wird. In diesem Fall bricht der SMS Loader ab, meldet **elementOccupied** = ***dynElemIdx*** und macht mit dem nächsten Eintrag in der Queue weiter.

Im Gutfall merkt sich der SMS Loader den Wert numOfProcedures für dieses Element und ruft nun die Methode **Browser.createVrmlFromURL()** tatsächlich auf.

Wenn nun das Element tatsächlich geladen ist, wird es im Feld **dynElems** gespeichert und erfährt die „Basic Initialization“. Dafür muss jedes dynamische Element die vier Felder **sendLoaded**, **receivePing**, **sendPong** und **receiveBasicInit** unterstützen.

Nachdem nun das Element tatsächlich geladen ist und nachdem die „Basic Initialization“ erfolgreich durchgeführt worden ist, wird mit der Initialisierung des Elements begonnen.

Die Initialisierung erfolgt in numOfProcedures Schritten, wobei in **dynElemStates**[***dynElemIdx***] von oben herunter gezählt wird.

Wenn in **dynElemStates**[***dynElemIdx***] der Wert 0 steht, dann ist das dynamische Element erfolgreich geladen UND initialisiert worden.

Für jeden Schritt (für jede „Prozedur“), der innerhalb der Initialisierung durchgeführt wird, passiert folgendes (wenn numOfProcedures den Wert 0 hatte, dann passiert es nie):

1. Mit dem Feld **initializeElement** = ***dynElemIdx*** fordert der SMS Loader den User auf, eine Prozedur durchzuführen (also einen Schritt der Initialisierung)
2. Die Prozedur wird durch den Wert **dynElemStates**[***dynElemIdx***] identifiziert, wobei für die Schritte (Prozeduren) von numOfProcedures bis eins heruntergezählt wird.
3. Das dynamische Element befindet sich bereits an seiner Stelle im Feld **dynElems**[***dynElemIdx***].
4. Nachdem der User die Prozedur (den Schritt) durchgeführt hat, muss er mit **finishProcedure** = ***dynElemIdx*** dem SMS Loader mitteilen, dass die Prozedur beendet ist und dass er mit der nächsten Prozedur weitermachen soll

Nachdem der User alle Prozeduren der Initialisierung durchgeführt hat, meldet der SMS Loader mit **elementReady** = ***dynElemIdx***, dass das Element nun verwendet werden kann.

Für den Fall, dass der User die dynamischen Elemente zur Szene hinzufügen möchte, also zum Beispiel als children eines <Group> Knotens, weil es nicht reicht, die dynamischen Elemente im Feld **dynElems** zur Verfügung zu haben, bietet der SMS Loader einige Felder, die direkt zu den Feldern eines <Group> Knotens geroutet werden können.

Diese werden im Kapitel 1.3.5 beschrieben.

1.3.4 Disablen und Entladen eines dynamischen Elements

Für das Disablen und Entladen dynamischer Elemente bietet der SMS Loader folgendes Feld an:

```
<field accessType='inputOnly' name='discardElement' type='SFInt32'/>
```

Durch ein Event **discardElement** = ***dynElemIdx*** sagt der User dem SMS Loader, dass dieser ein dynamisches Element disablen und entladen soll.

1.3.5 Interaktion mit einem <Group>-Knoten innerhalb der Szene

Der SMS Loader bietet folgende Elemente für die Interaktion mit einem <Group> Knoten innerhalb der Szene:

```
<field accessType='outputOnly' name='initializingElement' type='MFNode'/>
<field accessType='outputOnly' name='addElement' type='MFNode'/>
<field accessType='outputOnly' name='removeElement' type='MFNode'/>
```

Die Felder **addElement** und **removeElement** können direkt zu den Feldern **addChildren** und **removeChildren** eines <Group> Knotens ge <ROUTE>d werden.

Das Feld **addElement** hat die Eigenschaft, dass dynamische Elemente erst zur Szene hinzugefügt werden, *nachdem* sie initialisiert worden sind.

Wenn man statt dem Feld **addElement** das Feld **initializingElement** verwendet, dann werden die dynamischen Elemente schon zur Szene hinzugefügt, *bevor* sie initialisiert worden sind.

2 Module Related SSC Dispatcher

Der SSC Base selbst ist ein Benützer des SMS Loader, wie man an folgenden Auszügen aus dem File SscBase.x3d unschwer erkennen kann.

```
<ProtoDeclare name='SscBase'>
 <ProtoInterface>
  …...
 </ProtoInterface>
 <ProtoBody>
  …...
   <Script DEF='SimpleSceneControllerBase' directOutput='true' mustEvaluate='true'>
     …...
    <field accessType='inputOutput' name='dispatcherLoader' type='SFNode'>
      <ProtoInstance DEF='DispatcherLoader' name='SmsLoader'>
        <fieldValue name='numOfProcedures' value='1'/>
        <fieldValue name='urls' value='"../sms/SscDispatcher.x3d"'/>
      </ProtoInstance>
    </field>
```

Er benützt nämlich den SMS Loader, um für jedes registrierte Modul einen sogenannten „Module Related SSC Dispatcher“ dynamisch zu laden und im Feld **commParam**.**sscDispatchersGroup** abzuspeichern.

Wenn das Modul deregistriert wird, dann wird auch der SSC Dispatcher disablet und entladen.

3 Dynamische Module

Das SMUOS Framework stellt den **Prototypen SmsModuleLoader** zur Verfügung, damit der Rahmen eine gewisse Hilfestellung hat, wenn es darum geht, dynamische Module zu laden und wieder zu entladen.

Der SMS Module Loader stellt an seinem externen Interface folgende Felder zur Verfügung:

```
<field accessType='inputOutput' name='dynModConf' type='SFNode' value='NULL'/>
<field accessType='inputOutput' name='commParam' type='SFNode' value='NULL'/>
<field accessType='outputOnly' name='registerModules' type='MFString'/>
<field accessType='inputOutput' name='registeredModules' type='MFString' value=''/>
<field accessType='inputOnly' name='loadModuleWrappers' type='MFString'/>
<field accessType='outputOnly' name='moduleWrapperLoaded' type='SFNode'/>
<field accessType='inputOnly' name='loadModule' type='SFNode'/>
<field accessType='inputOnly' name='unloadModuleWrapper' type='SFInt32'/>
```

Das Feld **dynModConf** („Dynamic Module Configuration“) muss vom User gesetzt werden, es enthält einen Knoten, dessen Felder die Informationen über alle dynamischen Module enthalten.

Die Felder **commParam**, **registerModules** und **registeredModules** werden am besten mit den gleichnamigen Feldern des SSC Base verbunden:

Sobald der SSC initialisiert ist, meldet dieser die Common Parameters (**commParam**). Mit den **commParam** wird nun auch der SMS Module Loader initialisiert, liest den Inhalt der „Dynamic Module Configuration“ und registriert die dynamischen Module mit **registerModules**.

Während der gesamten Simulation hält der SSC die Liste der registrierten Module aktuell im Feld **registeredModules**, dieses enthält sowohl dynamische als auch statische Module.

3.1 Laden eines dynamischen Moduls

Wenn der Rahmen entscheidet, ein registriertes dynamisches Modul zu laden, dann muss er den Modulnamen im Feld **loadModuleWrappers** an den SMS Module Loader übergeben.

Dieser wird dafür sorgen, dass das Modul, falls es bereits geladen war, vorerst entladen und dann frisch geladen wird.

Sobald der Modulwrapper geladen worden ist, meldet sich der SMS Module Loader mit dem Feld **moduleWrapperLoaded**. Der Wert dieses Events zeigt auf den Modulwrapper und ermöglicht es dem Rahmen, den **„Initial State“** des Modulwrappers zu setzen. Sobald das erledigt ist, muss der Rahmen den Wert an das Feld **loadModule** „zurückspiegeln“.

Jetzt wird das Modul tatsächlich geladen.

3.2 Disablen und Entladen eines dynamischen Moduls

Der Index eines Modules (der sogenannte ***moduleIx***) ist der Index, an dem der SSC den Modulnamen im Feld **registeredModules** ausgibt.

Wenn nun der Rahmen ein dynamisches Modul entladen möchte, muss er den ***moduleIx*** des Moduls an das Feld **unloadModuleWrapper** übergeben.

Weiters entlädt der SMS Module Loader ein dynamisches Modul automatisch, wenn es geladen ist und wenn der SSC den Modulnamen aus dem Feld **registeredModules** löscht (wenn es deregistriert wird).

4 Unbound Objects (UBOs)

UBOs beziehen sich immer auf eine Universal Object Class (UOC). Eine UOC ist so etwas wie eine „grobe Klassifizierung von Objekten“, aus der man darauf schließen kann, welche SSC Extension man in seiner Szene benötigt, um eben eine bestimmte UOC zu unterstützen.

Die grundlegenden Ideen über UOCs und Object Types (Ots) **finden sich in Kapitel 4.1** .

Die UOCs werden von SSC Extensions definiert. Das heisst also, dass jeder Programmierer, der eine oder mehrere neue UOCs anbieten möchte, zuerst einmal eine SSC Extension programmieren muss.

UBOs und UOCs werden aber vom SMUOS Framework unterstützt. Das heisst, der Programmierer, der eine neue UOC für UBOs anbieten möchte, kann auf folgende X3D Prototypen zurückgreifen:

- UOC Dispatcher (SscDispatcher.x3d)
- UBO Loader (SscUboLoader.x3d)
- Basic MIDAS Objekt „Creator“ (MoosCreator.x3d)

Wie diese drei Prototypen zusammenarbeiten, und was noch fehlt, um eine UOC für UBOs zu definieren, **ist im Wesentlichen in Kapitel 4.2 beschrieben**.

Beide Unterkapitel sind in englischer Sprache gehalten, da sie aus einem anderen Paper stammen, welches auf einem englischen Blog veröffentlicht worden war.

4.1 Universal Object Classes (UOCs) and Object Types (OTs)

If you want to create a UBO, then it's not enough to know the UOC. A UOC is only a "rough classification" of an object, e.g

- military helicopter
- rail vehicle
- sports car
- propeller airplane
- and so on

An Object Type (OT) provides the possibility to categorize an object and it provides the possibility to load an object from a URL

- "Boeing AH-64 Apache", cat: "attack,1-rotor", URL: "http://x3d.net/apache.x3d"
- "ÖBB Rh 1044", cat: "1435mm,electric,BoBo", URL: "http://x3d.net/loco1044.x3d"
- "Bugatti-57G", cat: "4743cm3,Inj8,160PS", URL: "http://x3d.net/bugatti57g.x3d"
- "Cessna-182-Skylane-RG", cat: "4seat,1motor", URL: "http://x3d.net/cessna.x3d"

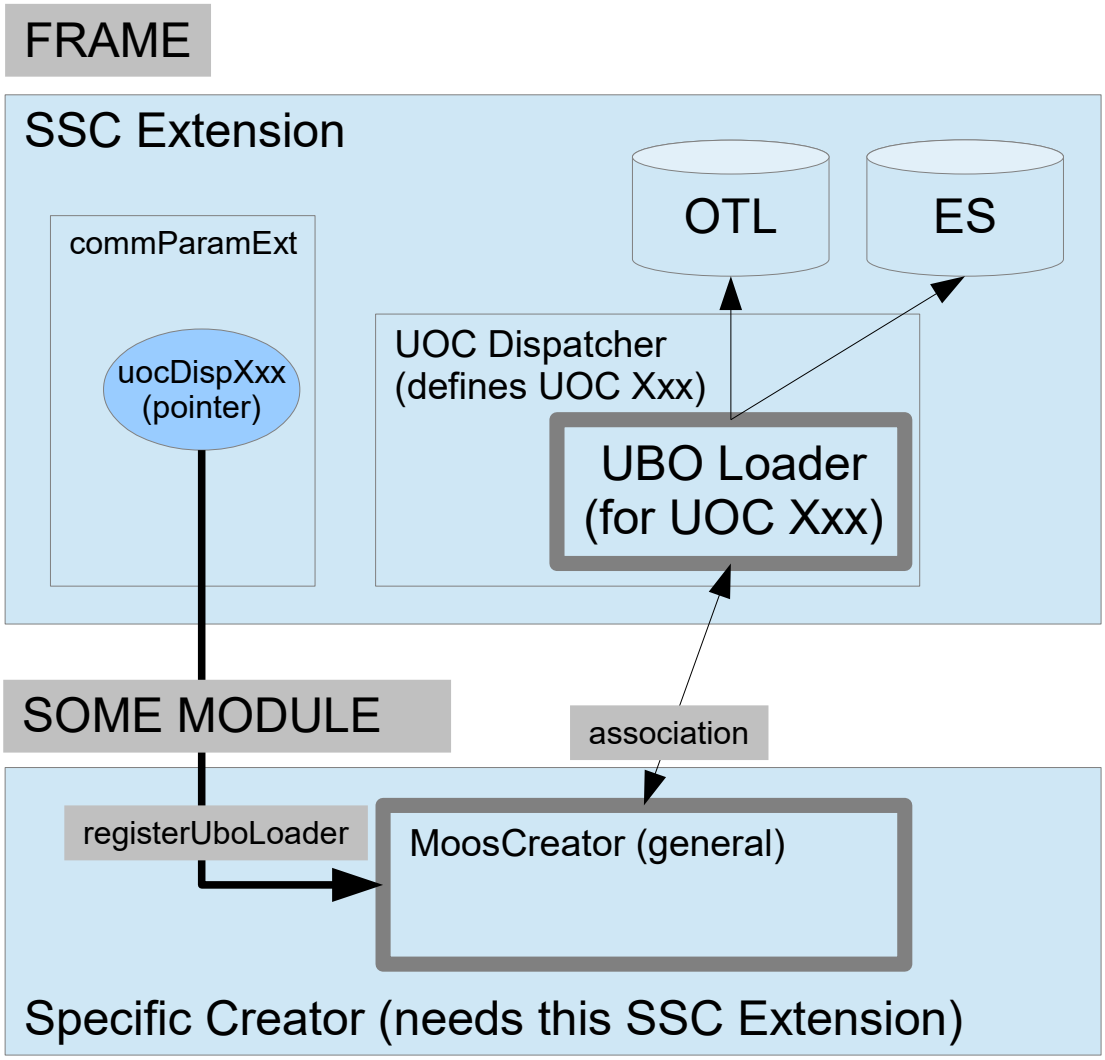
An Object Type is identified within an SMS by "UOC Name + Object Type ID"

- The UOC Name is an SMS wide unique identifier of a UOC
- Object Type IDs are unique within each UOC within an SMS
- Object Type IDs are globally registered in the global state of the UBO Loader (Object Type List – OTL) and they are identical in all scene instances of a multiuser session
- When an OT is deregistered, then all UBOs that had been created from this OT, are deleted
- Categories and URLs are defined in the DED (Dynamic Element Definition), which might be different in each scene instance. The categories and URLs are stored locally in the UBO Loader of each scene instance.
- If two scene instances use different URLs or different categories for their UBOs, then we say the scene instances provide different "views" to their users.

4.2 Architecture for Unbound Objects (UBOs)

Pre-Conditions and Basic Assumptions

- The MIB supports the **Modes Of Operation** (MOOs) I, II, III, IV and V, where the MOOs III, IV and V apply to unbound objects
    - MOO "LOADED".....the object has been freshly loaded
    - MOO III "initialized".....the object has been initialized or it has been deAttached
    - MOO IV "attached".....the object has been attached to a module
    - MOO V "disabled".....the object has been deAttached and deInitialized (if it was "attached") or it has been deInitialized (if it was "initialized") or it came directly from MOO "LOADED"
- The SSC Core and the SSC Base support the extension of the SSC by **SSC Extensions**
- Each SSC Extension can define **Universal Object Classes** (UOCs) by instantiating UOC Related SSC Dispatchers
    - UOCs are rough classifications of objects, e.g. "RailVehicles", "MilitaryHelicopters" or "Rockets"
- If an SSC Extension wants to make a UOC support UBOs, then it must instantiate one **UBO Loader** as a part of the UOC Related SSC Dispatcher
- Each UBO is created based on an **Object Type** (OT)
- Where Universal Object Classes (UOCs) are rough classifications of objects, an Object Type is a clear definition of a type of an object
    - OTs are e.g. "OeBB1044", "Apache" or "SaturnV", sticking to the above example
- The **Basic MIDAS Object MoosCreator** will support the lifecycle of all UBOs. Each instance of the MoosCreator supports one and only one UOC, it provides the current list of Object Types and Categories (see below) and enables the user to influence the lifecycle of UBOs
- **Is it possible to have a MoosCreator as "#loader" object in an SSC Extension ????**
- The Object Types, the Categories and the Object IDs of the UBOs will be unique within each UOC, e.g.
  <objTypeId> = "Uoc.Base.RailVehicles.OeBB1044",
  <category> = "Uoc.Base.RailVehicles.1435mm",
  <extObjId> = "Uoc.Base.RailVehicles-1044_212"



SSC.....Simple Scene Controller (central part of the Framework, very general)

SSC Extension.....specific 3rd party extension of the SSC

OTL (per UOC).....Object Type List (a global list of Object Type IDs)

ES (per UOC).....Existence State (a global list of <objId> + <objectTypeIx> + <objectState>)

commParamExt.....extension of the Common Parameters (commParam)

UOC.....Universal Object Class – each SSC Extension can define zero or more UOCs

UBO.....Unbound Object – UOCs that create UBOs, have an UBO Loader

Duties of the UBO Loader

- The UBO Loader maintains a global list of all **Object Types** of a UOC
    - At the Layout Description File (LDF), the Object Type is addressed by **UOC Name** AND **Object Type ID**, e.g. "Uoc.Base.RailVehicles.OeBB1044"
    - At the MoosCreator, the Object Type is addressed by the **Object Type ID** only, e.g. "OeBB1044"
    - The UBO Loader defines an **objectTypeIx**, which is unique within one UOC and which is the same in all scene instances
    - An Object Type can be identified by **UOC + objectTypeIx**
    - The objectTypeIx can be used at the interface of the UBO Loader and at the interface of the MoosCreator to identify one Object Type of the given UOC
    - An object type is locally associated with zero or more **Categories**
    - Categories are used to filter lists of Object Types or lists of UBOs
    - The UBO Loader keeps all associated MoosCreators up to date about the list of Object Types. Therefore it reads the filter from the MoosCreator, filters the current list of Object Types and outputs the filtered list to the MoosCreator
- The UBO Loader together with the MoosCreator care for the lifecycle of UBOs **including their assignment to and deAssignment from modules ???? How ????**
- Each UBO Loader maintains the global **Existence State** of all UBOs of one UOC
    - List of all **<objId>**s at index **objectIx**
    - List of all **<objectTypeIx>**s at index **objectIx**
    - List of all **<objectState>**s at index **objectIx**
- The **objectIx** is a globally valid index of each UBO within the Existence State
- Each UBO can be identified within an SMS by UOC Name + <objId> or by UOC + objectIx
- The objectIx can be used at the interface of the UBO Loader and at the interface of the MoosCreator to identify one UBO of the given UOC

When CREATING a UBO (this is a global procedure)

- The MoosCreator ensures to have all data for "ObjectTypeIx + Initial State + Initial Module Assignment"
- The MoosCreator "occupies" the UBO Loader;
  "occupying" is a global process, that means: only one instance of the MoosCreator within the whole UOC – and within all scene instances – can occupy the UBO Loader at a time
- The MoosCreator may suggest an <objId> for the creation of the UBO and then it triggers the creation
- The <objId> and the <objectTypeIx> are set to their correct values globally and the <objectState> is set to CREATING globally;
  only one objectIx can be CREATING at a time (in each UOC)
- The creating instance of the MoosCreator recognizes "his" objectIx is CREATING now and hence it loads the UBO Wrapper and brute force initializes the global Wrapper State to "ObjectTypeIx + Initial State + Initial Module Assignment + <firstInstanceFlag=true>"
- Now this UBO Wrapper can be accessed by the SSC Extension with the help of the objectIx
- Now the creating instance requests to set the <objectState> to CREATED and to FREE the UBO Loader
- All other instances of the UBO Loader recognize the state CREATED and hence they load the UBO Wrappers, too
- Now all instances of the UBO Wrapper can be accessed by the instances of the SSC Extension with the help of the objectIx

When LOADING a UBO (this is a local procedure)

- If a UBO Wrapper has been loaded and if the <objectState> at the objectIx is CREATED, then the SSC Extension may decide to actually load the local instance of the UBO
- The first instance of the UBO that is actually loaded (in whatever scene instance) recognizes that it is the first instance by the <firstInstance=true> flag in the UBO Wrapper and hence brute force initializes the state of the UBO by "Initial State + Initial Module Assignment". It deletes the <firstInstance=true> flag in the UBO Wrapper

When UNLOADING a UBO (this is a local procedure)

- If the <objectState> of a UBO is CREATED and if the UBO has been loaded locally, then the SSC Extension can decide to unload the UBO locally (without influencing the Existence State)
- The UBO Wrapper is told that it should unload the UBO and it is done
- Loading the same UBO again locally is possible, as long as the <objectState> is CREATED

When DELETING a UBO (this is a global procedure)

- The SSC Extension may decide to delete a UBO, when the <objectState> is CREATING or CREATED
- The UBO Loader will set the <objectState> to DELETED and keep the <objId> and the objectIx reserved for 20 seconds
- All Instances of the UBO Loader detect that the state changed to DELETED and hence the UBO Wrappers are disabled and unloaded – so that they unload the UBOs

When Deregistering a Module (this is a global procedure)

- The central server of SSC Base detects, when he deregisters a module in the commState. Then he informs all UBO Loaders to delete all UBOs of that module