SMS Facility Loaders

This hobby report deals with the so-called Facility Loaders.

Facility Loaders are prototypes of the SMUOS framework that help to load dynamic elements, because dynamic elements are also referred to as "dynamic facilities".

Now, during implementation of Step 0033.11, this Hobby Report is given the 4.x version number. In particular, chapter 4 on "Unbound Objects (UBOs)" has been updated for Step 0033.11.

Version 4.3 is indicating the version at the first serious specification of the release "Arimathea" (step 0033.11), where the basic concepts of "Arimathea" seem to be stable now.

Version 4.3.1 is a small update due to the fact, that the „Module Loader“ has been extended to work either as a „Module Loader“ or as a „Model Loader“. Hence the document has been restructured.

Table of Content

1 Dynamic Elements – Dynamic Facilities....2
1.1 Introduction....2
1.2 Requirements of the SMUOS Framework....2
1.3 The SMS Loader....3
2 Module Related SSC Dispatcher....7
3 Dynamic Elements (DynMos, DynBos and UbOs)....8
3.1 Dynamic Modules (DynMos)....8
3.2 Dynamic Bound Objects (DynBos)....8
3.3 Unbound Objects (UbOs)....8
3.4 Dynamic Elements (Overview)....9
4 Dynamic Modules and Dynamic Bound Objects – TODO11....10
4.1 This Chapter is Out of Date, The „My Little Loader“ will replace the „SmsModuleLoader“ 10
4.2 Loading a dynamic module....10
4.3 Disabling and unloading a dynamic module....11
5 Unbound Objects (UBOs) – TODO11....12
5.1 Universal Object Classes (UOCs) and Object Types (OTs)....13
5.2 Architecture for Unbound Objects (UBOs)....14
5.3 Procedures for Unbound Objects (UBOs)....17
5.4 Fields of the UBO Wrapper and of the MobContainer (uiObj)....21
5.5 Fields of the UBO Loader....22
5.6 Fields of the UBO Loader Stub....22
5.7 Fields of the MoosCreator....23

1 Dynamic Elements – Dynamic Facilities

1.1 Introduction

As a "facility", we actually refer to any element of an SMS - a Simple Multiuser Scene - that somehow appears to the user, be it an avatar, a module, a model, a MIDAS object, or any aggregate of information find their expression in the scene.

As "dynamic" elements, we refer to all those elements that are not automatically loaded and initialized when loading and initializing the Scene Instance, but later "loaded" in the course of the simulation.

It should be noted that there must be some **trigger** that causes the (re-)loading of the dynamic element.

Furthermore, dynamic elements can be deleted – so to speak unloaded – long before the scene instance is destroyed. Here is the peculiarity to take into account that most X3D player, the content that is no longer needed, not immediately remove from memory, but still leave it there for a while.

That's why you have to explicitly "disable" this content before removing it from the scene so that it behaves completely "passively" and can not disturb the simulation anymore.

1.2 Requirements of the SMUOS Framework

The SMUOS Framework makes the following assumptions about all "dynamic facilities" in an SMS:

- A dynamic element can be loaded using the Browser.createVrmlFromURL () method, just the URL of the element is needed therefore
- Each dynamic element is a single X3D node that is loaded using this method (Browser.createVrmlFromURL () actually has an MFNode value as the result, but in our case only the element with index [0] is used)
- All dynamic elements of a "class of dynamic elements" can be indexed and stored in an MFNode field
- Dynamic elements can - and must - undergo "basic initialization" according to the concepts of the SMUOS framework before they can be used
- Dynamic elements can - and must - be disabled using the disable field (SFTime) before removing them from the scene

1.3 The SMS Loader

The SMS Loader is implemented in the prototype SmsLoader as a part of the „SMS Base“ subsystem and provides basic support for loading of dynamic elements.

The dynamic elements are stored in the field dynElems (MFNode), whereby unused indices are not assigned the value NULL, but with "any pointer".

The status of each element can be determined via the field dynElemStates (MFInt32), whereby it is guaranteed at all times that dynElems and dynElemStates have the same dimension.

The SMS Loader calls the method Browser.createVrmlFromURL (), then carries out the "basic initialization" of the element itself and then serves as a "pacemaker" for the initialization of the dynamic element, but the user essentially has to make himself, albeit clocked by the SMS Loader.

Only when the dynamic element is fully initialized can this SMS Loader load the next element. However, the SMS Loader maintains a queue in which the dynamic elements wait for loading, the loading of which has already been triggered.

When the SMS Loader gets the command to remove an item from the scene, it will make sure that the item is previously disabled

When the SMS Loader becomes disabled, it makes sure that all loaded elements are disabled and removed from the scene

1.3.1 "Basic Initialization" of the SMS Loader

Since the SMS Loader is one of many SMUOS prototypes, it also offers on its external interface the fields that all SMUOS prototypes offer, namely:

```
<!-- 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'/>
```

The fields **objType** and **version** are there to identify one instance of the SMS Loader as such, the other four fields are for "Basic Initialization", i. the scene waits until all external prototypes are loaded before calling the **receiveBasicInit** field.

By calling the **receiveBasicInit** field, the SMS Loader tentatively sets the size of the **dynElems** and **dynElemStates** fields, based on the field

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

and outputs to the environment the actual size in the event **dynElemSpace**:

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

Now the following two fields are initialized and can be used immediately:

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

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

1.3.2 Initialization and disabling of the SMS Loader

The SMS Loader provides the following fields for initialization and 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' />

Since the SMS Loader requires a reference to the Common Parameters (**commParam**), it must be initialized with either the **commParam** field or the **modParam** (Module Parameters) field before using it.

After use, you should put the SMS Loader back in a passive state with the field "**disable**". This also ensures that any dynamic elements that are still loaded are safely disabled and removed from the scene.

You can initialize the SMS Loader several times in succession and then disable it again. This can be used, for example, to "disassemble and unload" all the dynamic elements of a "class of dynamic elements" in one fell swoop.

If you initialize the SMS Loader with the **commParam**, then it stores this pointer for later use. If a disabling is currently in progress, then the SMS Loader first waits until disabling is completed.

If you initialize the SMS Loader with the **modParam**, then the same thing happens, but additionally the SMS Loader immediately listens to the **disable** field of the **modParam** and disables itself as soon as the module is disabled.

1.3.3 Loading and Initializing a Dynamic Element

To implement this function, the SMS Loader provides the following fields on its external 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'/>
```

In preparation for loading and initializing a dynamic element, the user must set the following fields:

numOfProcedures ..... Number of procedures into which the initialization is to be decomposed

urls ..... this value should be passed to the method Browser.createVrmlFromURL ()

In order to actually trigger the load, the user must now set the field **loadElement** to the desired value **dynElemIdx** if he already knows the desired index of the element and to a value less than 0 if he does not yet know this index.

Then the SMS Loader inserts the information about the element to be loaded into the internal queue (**numOfProcedures**, **urls** and **dynElemIdx**) and returns in the **elementQueued** = dynElemIdx field that the element is now in the queue. The final element index is already used here, and the dimensions of the dynElems and dynElemStates fields are also increased if the index does not fit into the old dimensions.

Then, when the element has its turn, ie when the previous elements in the queue have been processed, then the element is actually loaded.

First it checks if **dynElemStates** [**dynElemIdx**] is greater than or equal to zero, ie if another element is already stored at this index. In this case, the SMS Loader aborts, reports **elementOccupied** = dynElemIdx and proceeds with the next entry in the queue.

In the good case, the SMS Loader remembers the **numOfProcedures** value for this element and actually calls the Browser.createVrmlFromURL () method.

Now, when the element is actually loaded, it is stored in the **dynElems** field and undergoes basic initialization. To do this, each dynamic element must support the four fields **sendLoaded**, **receivePing**, **sendPong**, and **receiveBasicInit**.

Now that the element is actually loaded and after the "Basic Initialization" has been successfully completed, the element is initialized.

Initialization is done in **numOfProcedures** steps, counting down in **dynElemStates** [**dynElemIdx**].

If the value 0 is in **dynElemStates** [**dynElemIdx**], then the dynamic element has been successfully loaded AND initialized.

For each step (for each "procedure") performed during initialization, the following happens (if **numOfProcedures** was 0 then it never happens):

1. With the field **initializeElement** = dynElemIdx the SMS Loader requests the user to execute a procedure (ie a step of initialization)

2. The procedure is identified by the value **dynElemStates** [**dynElemIdx**], counting down to one for the steps (procedures) of **numOfProcedures**.

3. The dynamic element is already in its place in the field **dynElems** [**dynElemIdx**].

4. After completing the procedure (step), the user must use **finishProcedure** = dynElemIdx to tell the SMS Loader that the procedure is finished and that he should continue with the next procedure

After the user has performed all the procedures of the initialization, the SMS Loader reports with **elementReady** = dynElemIdx that the element can now be used.

In the event that the user wants to add the dynamic elements to the scene, for example as children of a <Group> node, because it is not enough to have the dynamic elements in the **dynElems** field, the SMS Loader offers some fields which can be routed directly to the fields of a <Group> node.

These are described in chapter 1.3.5.

1.3.4 Disabling and unloading a dynamic element

For disabling and unloading dynamic elements, the SMS Loader offers the following field:

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

Through an event **discardElement** = dynElemIdx, the user tells the SMS Loader to disable and unload a dynamic element.

1.3.5 Interaction with a <Group> node within the scene

The SMS Loader provides the following elements for interacting with a <Group> node within the scene:

<field accessType = 'outputOnly' name = '**initializingElement**' type = 'MFNode' />

<field accessType = 'outputOnly' name = '**addElement**' type = 'MFNode' />

<field accessType = 'outputOnly' name = '**removeElement**' type = 'MFNode' />

The **addElement** and **removeElement** fields can be directly linked to the **addChildren** and **removeChildren** fields of a <Group> node.

The **addElement** field has the property that dynamic elements are added to the scene only after they have been initialized.

If you use the **initializingElement** field instead of the **addElement** field, then the dynamic elements are already added to the scene before they have been initialized.

2 Module Related SSC Dispatcher

The SSC Base itself is a user of the SMS Loader, as can be easily recognized by the following excerpts from the file SscBase.x3d.

```
<ProtoDeclare name = 'SscBase'>
  <Proto Interface>
    ... …
  </ Proto Interface>
  <Proto Body>
    ... …
      <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>
```

Namely, he uses the SMS Loader to dynamically load a so-called "Module Related SSC Dispatcher" for every registered module and to save it in the **commParam.sscDispatchersGroup** field.

If the module is deregistered, then also the SSC dispatcher is disabled and unloaded.

3 Dynamic Elements (DynMos, DynBos and UbOs)

The user of the SMUOS Framework is involved with three kinds of dynamic elements, i.e. with

- Dynamic Modules (DynMos),
- Dynamic Bound Objects (DynBos) and
- Unbound Objects (UbOs).

3.1 Dynamic Modules (DynMos)

Top Level Modules[1] (TLMs) can be one of two kinds:

- Static Modules
- Dynamic Modules

Dynamic Modules MUST be wrapped by Module Wrappers. The module wrapper exists from the startup of the scene until the deletion of the scene (like a static module does), only the dynamic module itself within the module wrapper is loaded on demand by means of some **trigger** in the frame.

The frame author may use the „My Little Loader“ prototype to get supported with the handling of dynamic modules (see chapter 4).

3.2 Dynamic Bound Objects (DynBos)

MIDAS Base supports the loading and initialization as well as the attachment of bound objects, even if it happens long after the initialization of the parent module. Also the disabling and unloading of bound objects is supported by MIDAS Base, long before the parent module is disabled. If this is done, the bound object is called a „dynamic“ bound object (DynBo).

The module author may use the „My Little Loader“ prototype to get supported with the handling of dynamic bound objects (see chapter 4).

3.3 Unbound Objects (UbOs)

Unbound Objects are loaded and unloaded by means of the UBO Loader.

Each provider of an SSC Extension may decide to support a new class of Unbound Objects (a so-called Universal Object Class – UOC). Therefore he will use the services of the UBO Loader (among others).

Unbound Objects are described in chapter 5 .

1 Currently only Top Level Modules are supported. Dependent Modules (aka Moving Modules) are not supported yet

3.4 Dynamic Elements (Overview)

All dynamic elements are defined in the DED. The DED is a <Script> node that holds the definitions of all global types of the dynamic elements and their respective parameters.

A dynamic element instance is always created from a dynamic element type, as follows:

	DED	Dynamic Element Type	Dynamic Element Instance
DynMo (Dynamic Module)	[<wki>[.<wki>]...]**moduleName** + urls + moduleParameters	**moduleName**	**moduleName**
DynBo (Dynamic Bound Object)	<McClassPath>.**OTI** + categories + urls	<moduleName>-<objId>.**OTI**	Bdo.<moduleName>-<objId>.**OTIn**
UbO (Unbound Object)	<SscClassPath>.**OTI** + categories + urls	<uocName>.**OTI**	Uoc.<uocName>-**OTIn**

Dynamic Modules (DynMos) are identified by a unique <moduleName> within an SMS.

- They can be instantiated only once per scene instance, using the <moduleName>. If you want to instantiate the same module a second time, then you have to register the same URL another time with a different <moduleName>
- The optional prefix [<wki>[.<wki>]...] serves to assign the modules to different instances of the „Module Loader“ (see chapter 4) and to derive the URL of the used „Module Wrapper“ (currently the SscBase and all SscExt support only one default Module Wrapper)

Dynamic Bound Objects (DynBos) and **Unbound Objects** (UbOs) are created from <OTI>s (Object Type Identifiers), whereby

- a „running number“ is added to the <OTI> to create unique identifiers for the objects, e.g. the objects „Apache.1“, „Apache.2“ and „Apache.13“ could have been created from the <OTI> „Apache“
- The prefices <McClassPath> and <SscClassPath> serve to assign the object types to different instances of the „Model Loader“ (for DynBos) and of the „UBO Loader“ (for UbOs). Those instances are identified by <moduleName>-<objId> and <uocName>, respectively

4 Dynamic Modules and Dynamic Bound Objects – TODO11

4.1 This Chapter is Out of Date, The „My Little Loader“ will replace the „SmsModuleLoader“

The SMUOS framework provides the prototype SmsModuleLoader, so the framework has some help in loading and unloading dynamic modules.

The SMS Module Loader provides the following fields on its external interface:

<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' />

The field **dynModConf** ("Dynamic Module Configuration") must be set by the user, it contains a node whose fields contain the information about all dynamic modules.

The fields **commParam**, **registerModules** and **registeredModules** are best connected to the fields of the same name of the SSC Base:

Once the SSC is initialized, it reports the Common Parameters (**commParam**). With the **commParam** now also the SMS Module Loader is initialized, reads the contents of the "Dynamic Module Configuration" and registers the dynamic modules with **registerModules**.

Throughout the simulation, the SSC keeps the list of registered modules currently in the **registeredModules** field, which contains both dynamic and static modules.

4.2 Loading a dynamic module

If the frame decides to load a registered dynamic module, then it must pass the module name to the SMS Module Loader in the **loadModuleWrappers** field.

This will ensure that the module, if it was already loaded, initially unloaded and then freshly loaded.

As soon as the module wrapper has been loaded, the SMS Module Loader logs in with the field **moduleWrapperLoaded**. The value of this event points to the module wrapper and allows the frame to set the initial state of the module wrapper. Once that's done, the frame must "mirror" the value back to the **loadModule** field.

Now the module is actually loaded, initialized and attached to the SSC.

4.3 Disabling and unloading a dynamic module

The index of a module (the so-called **moduleIx**) is the index at which the SSC outputs the module name in the field **registeredModules**.

Now, if the frame wants to unload a dynamic module, it must pass the module's **moduleIx** to the **unloadModuleWrapper** field.

Furthermore, the SMS Module Loader automatically unloads a dynamic module, if it is loaded and when the SSC deletes the module name from the **registeredModules** field (during deregistration).

5 Unbound Objects (UBOs) – TODO11

Unbound objects – in a nutshell – are a kind of models that can move from one module to another – they are not „bound“ to modules.

This also means, they need some name in addition to the object ID that classifies the object somehow. This classification is done by the „Universal Object Class (UOC)“.

The basic ideas about „Universal Object Classes (UOCs)“ and „Object Types (OTs)“ are described in chapter 5.1 .

A UOC is always defined by an SSC Extension.

In other words: if you want to support a new class of UBOs, then you have to write an SSC Extension. The SSC Extension has to instantiate an „UOC Related SSC Dispatcher“ (aka „UOC Dispatcher“) in order to define the UOC and the UOC Dispatcher has to be extended by a UBO Loader, which is responsible for loading and unloading of UBOs.

The „Architecture for UBOs“ (including the UBO Loader) is described in chapter 5.2 .

Concrete procedures for UBOs and some further details are then elaborated in chapters 5.3 - 5.7 .

5.1 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, e.g.

- "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,Inj8,160HP", 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, too
- Categories and URLs are defined in the DED (Dynamic Element Definition), which might be different in different scene instances. 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 the UBOs.

5.2 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
- 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
- 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 filtered by Categories (see below) and enables the user to influence the lifecycle of UBOs
- 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"

Basic Elements of the Architecture for UBOs

Following elements are crucial for the support of UBOs

- The MobContainer
- The UBO Wrapper
- The UBO Loader
- The UBO Loader Stub
- The MoosCreator

Each programmer of an SSC Extension, who wants to support a new class of UBOs, provides

- The SSC Extension, which contains the UOC Dispatcher and the UBO Loader
- The „Specific Replicator“, an Extension MIDAS Object that contains the MoosCreator
- Extension MIDAS Objects for the UBOs themselves

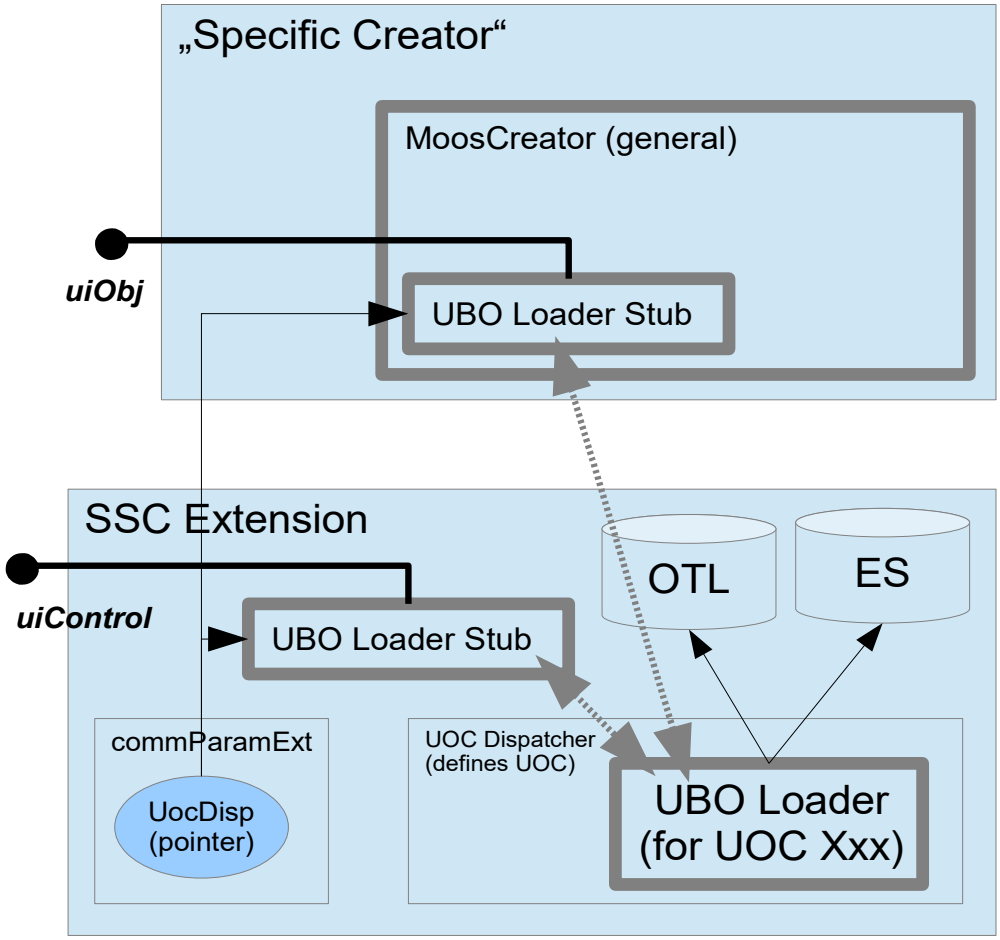

Figure 1: Architecture of SSC Extension and „Specific Creator“ for UBOs

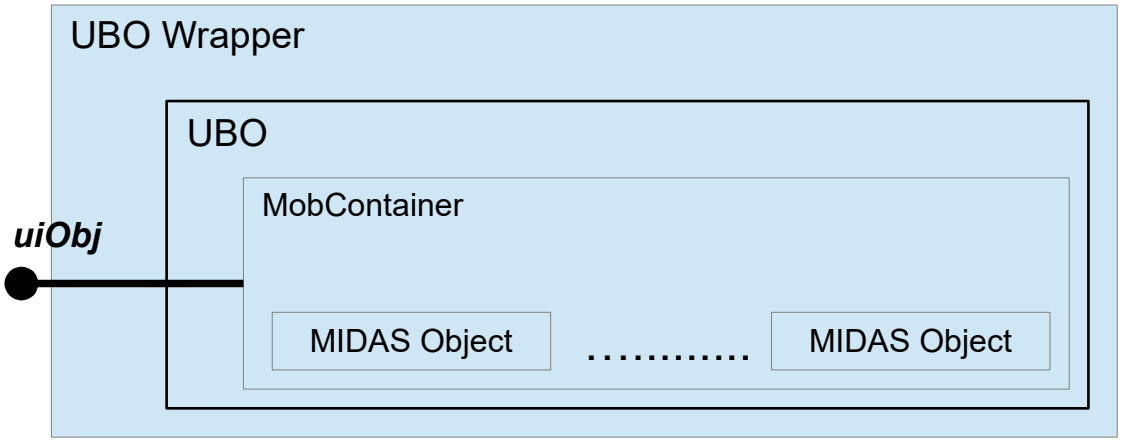

Figure 2: Architecture of an Unbound Object (UBO)

Duties of the UBO Loader, of the UBO Loader Stub and of the MoosCreator

- The UBO Loader maintains a list of all **Object Types** of one UOC (**Object Type List OTL**)
    - At the Dynamic Element Description (DED), the Object Type is addressed by **UOC Name** AND **Object Type ID**, e.g. "Uoc.Base.RailVehicles.OeBB1044"
    - At the UBO Loader Stub, the Object Type is addressed by the **Object Type ID** only, e.g. "OeBB1044"
    - An object type is locally associated with zero or more **Categories**
    - Categories are used to filter lists of Object Types
    - The UBO Loader keeps all **connected UBO Loader Stubs** up to date about the list of Object Types. Therefore it reads the filter from the UBO Loader Stub, filters the current list of Object Types and outputs the filtered list to the UBO Loader Stub.
- The UBO Loader together with the MoosCreator/UBO Loader Stub care for the lifecycle of UBOs
- Each UBO Loader maintains the global **Existence State (ES)** 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 Stub to identify one UBO of the given UOC
- The UBO Loader keeps all **connected UBO Loader Stubs** up to date about the list of UBOs

5.3 Procedures for Unbound Objects (UBOs)

When CREATING a UBO (this is a global procedure)

1. The **User of the UBO Loader Stub** selects an "objTypeIdx" and optionally sets the "preferredObjId". The User triggers the UBO Loader Stub with the "**createObjTypeIdx**" and waits for the response at "initializeObjIdx"
2. This **UBO Loader Stub** tries to "occupy" the UBO Loader with the **4-tuple of following parameters**.
   - preferredObjId
   - createObjTypeIdx
   - creatingSessionId
   - creatingExtObjId.

   It tears up a supervision timer
3. The **UBO Loader** sends the 4-tuple in a "**createRequest**" to the central server.
4. If the UBO Loader is not yet "occupied", then the **central server**
   - selects an objIdx and an objId for the UBO
   - removes the objIdx from the unregisteredUbos
   - sets the objId and the objTypeIdx at the selected objIdx, leaves the objStatus at the value "DELETED"
   - sets the triplet
     - creatingSessionId
     - creatingExtObjId
     - creatingObjIdx
   - **distributes an update of the "Existence State (ES)"**
5. The instance of the **UBO Loader at the "creatingSessionId"** detects that it is now CREATING, takes the creatingObjIdx and **loads the FIRST instance of the UBO Wrapper**.
6. As soon as the UBO Wrapper has been loaded, then the **constant values (see below) are set in the UBO Wrapper**

7. Now we have
   - the constant ID of the UBO (objId=VALID and objTypeIdx=VALID) and
     the objState=CREATING of the UBO at the correct "objIdx" in the ES
   - the constant values of the UBO Wrapper in the FIRST instance of the UW
     - the "objId",
     - the "universalObjectClass"
     - the "url",
     - the "creatingExtObjId" and
     - the "creatingSessionId".
8. The UBO Loader triggers the **UBO Loader Stub** by "**initializeObjIdx**" (with the value "creatingObjIdx"), which forwards the information to its user.
9. The **User of the UBO Loader Stub** sets the "initial state" of the UBO Wrapper
   - moduleIx (SFInt32)
   - firstInstanceFlag (SFBool)
   - initialState (MFString
10. The user of the UBO Loader Stub informs the UBO Loader Stub about successful initialization of the FIRST instance of the UBO Wrapper via "**objIdxInitialized**" (with the value "creatingObjIdx")
11. Now the UBO Loader triggers the **FIRST instance of the UBO Wrapper to distribute its shared state**
    - creatingExtObjId ..... extObjId of the creating UBO Loader Stub
    - creatingSessionId ..... sessionId of the creating PSI
    - moduleIx ..... module the UBO is assigned to initially
    - firstInstanceFlag = TRUE ... this flag will be used to detect the first instance of the UBO that is loaded
    - initialState (semantics depends on the UOC)
12. Now the UBO Loader will send the "loadRequest" to the central server
13. The central server removes the "occupation"
    - the objectState at objIdx == creatingObjIdx is set to "CREATED"
    - the "creatingSessionId", "creatingExtObjId", "creatingObjIdx" are reset
    - an update of the ES is distributed
14. All scene instances that have not yet loaded the UBO Wrapper, load the UBO Wrapper
15. All scene instances trigger the "**loadObjIdx" event at the SSC Extension**

When LOADING a UBO (this is a local procedure)

- If a UBO Wrapper has been loaded and the <objectState> at the objectIdx is CREATED or if the <objectState> at the objectIdx changes to CREATED and the UBO Wrapper is loaded, then the UBO Loader triggers the SSC Extension with the event „loadObjIdx“
- When the SSC Extension wants to immediately load the instance of the UBO, then it triggers the UBO Wrapper to actually load the UBO by means of the commParam field
- If the decision is for "delayed loading", then the progress is set to 100%, otherwise the UBO Wrapper will care for the display of the progress
- The UBO Wrapper has a look to the stored moduleIx. If it is < 0, then the UBO Wrapper will forward the commParam to the UBO (as soon as it will have been loaded), otherwise the UBO Wrapper will search for the modParam in the commParam and forward the modParam to the UBO
- Now the UBO has been loaded and it reports „attached“ or at least „initialized“ to the UBO Wrapper. The UBO Wrapper outputs „initialized“.
- The UBO Wrapper of 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 and hence initializes the state of the UBO by the field „setState“ at the uiObj interface
- If the UBO has been attached via modParam and if everything went fine, then the MobContainer in the UBO will recognize the UBO is positioned eventually. Now the MobContainer will add the UBO to the ModelGroup in the MC and finally the UBO will become visible

When a module has been loaded and initialized (this is a local procedure)

- The MC Base uses a service of the SSC Base (tbd.) to locate all UBO Wrappers that are assigned to this module
- If wrappers are found and if the contained UBOs are loaded, then the UBOs are attached to the module

When a module has been disabled

- The MC Base sends commParam to all UBOs in the <Group> node. The MobContainer removes the SELF node from the <Group> node. This is ffs.

Module Activation/Deactivation

- All UBOs that are attached to the module, receive the module activity and react accordingly.

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)

- Deleting a UBO is a service of the UBO Loader. Hence it can be called
  - by the SSC Extension
  - by any UBO of the same UOC (via „universalObjectClass“)
  - by the MoosCreator (MoosCreator has a list of all UBOs)
- The UBO Loader 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 (see above)

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 (set moduleIx = -1)

5.4 Fields of the UBO Wrapper and of the MobContainer (uiObj)

The external interface of the UBO Wrapper and of the MobContainer is nearly identical to the „standard uiObj“ interface:

Field Name	Presence at UBO Wrapper	Presence at MobContainer
objType (SFString)	yes	yes
version (SFFloat)	yes	yes
url (MFString)	yes	---
theUbo (SFNode)	yes	yes
objId (SFString)	yes	yes
universalObjectClass (SFNode)	yes	yes
commParam (SFNode)	yes	yes
modParam (SFNode)	---	yes
disable (SFTime)	yes	yes
initialized (SFNode)	yes	yes
attached (SFNode)	---	yes
enabledOut (SFBool)	yes	yes
positioningObjects (MFNode)	---	yes
setWrapperState (MFString)	yes	---
setState (MFString)	Only available at the UBO itself	

The semantics of following fields are identical to the „standard uiObj semantics“:

- objType, version, objId, universalObjectClass (always != null), initialized, attached and enabledOut

The fields commParam, modParam and disable have a slightly different semantics at the UBO Wrapper:

- commParam actually instruct the UBO Wrapper to load the UBO. If the moduleIx is greater than or equal zero, then the UBO Wrapper will search for the modParam in the commParam
- disable instructs the UBO Wrapper to actually unload the UBO

setWrapperState sets the wrapper state (moduleIx, firstInstanceFlag and initialState), positioningObjects informs the MobContainer, which objects will deliver „positioned“ events.

The UBO Wrapper uses the field „setState“ of the UBO to initialize the global state of the UBO

Loading and unloading of UBOs may happen more often than once in the lifetime of the wrapper.

5.5 Fields of the UBO Loader

Tbd.

5.6 Fields of the UBO Loader Stub

- **„universalObjectClass“ (SFNode)**
  This field takes the pointer from the SSC Extension (commParamExt) that points to the UOC Dispatcher.
  This initializes the UBO Loader Stub to connect to the UBO Loader.
  The same value will be written to the uiObj Interface of the UBOs, after they will have been loaded by the UBO Wrapper

- **„disable“ (SFTime)**
  An event „Now“ at this field will disconnect the UBO Loader Stub from the UBO Loader and it will disable the UBO Loader Stub. A new initialization with „universalObjectClass“ will be possible

- **„categories“ (MFString)**
  This field will be used as filter for the display of Object Type IDs

- **„objectTypeIds“ (MFString), objectTypeIdxs (MFInt32)**
  These fields are the filtered Object Types that can be created by this UBO Loader Stub.
  They are the „filtered OTL“ of the connected UBO Loader

- **„objects“ (MFNode), objectsStates (MFString)**
  This is a „mirror of the ES“ of the connected UBO Loader
  „objects[ix]“ contains the <objId> and the <objectTypeIx> (actually they are the UBO Wrappers)

- **„preferredObjId“ (SFString)**
  Before starting the creation of a UBO, here a specific „objId“ can be requested

- **„createObjectType“ (SFString), „createObjectTypeIdx“ (SFInt32)**
  start the creation of an UBO

- **„initializeObjectIdx“ (SFInt32)**
  This UBO Wrapper shall be initialized with „setWrapperState“ and then objectIxInitialized shall be triggered. (Value < 0 ---> an error has happened)

- **„objectIdxInitialized“ (SFInt32)**
  Initialization of the UBO Wrapper done --> de-occupy the UBO Loader again

5.7 Fields of the MoosCreator

Tbd.