# Hibernation of Advanced Railroad Trains (ArrT)

## Step Three – ..... and Communicate them (aCt)

Pieta / Rebase to MIB Core

This hibernation report describes the things that **have been** implemented for release "Pieta".

Chapter 3 contains the essential parts.

Version 3.x is indicating the version at the beginning of the SPARK project.

# 1 Introduction

## *1.1 In the Beginning was the "Intermediate Layer"*

When you develop a thing, most of the time you do not want to start with Adam and Eve (although there is software that actually did that).

For example, in the area of 3D graphics and virtual worlds, there is the ISO standard X3D, which already provides many functions in a prefabricated form.

You just have to put these functions together like in a LEGO kit into a so-called "scene". You do that for example using an XML syntax.

Now it is often the case that many or at least several scenes offer similar functions that are not already included in the X3D standard, but in all of these scenes are similarly composed of the functions of the X3D standard.

One wonders if it really is the last word of wisdom, if each scene author takes from the other with Copy & Paste and clones the corresponding parts more and more, or whether it would not be better to provide these sub-functions in a central location.

In an architectural way of speaking, we would say we want to put an "intermediate layer" between the scene and the Web3D browser that can be used by everyone, or at least by many scene authors.

That's exactly what we're doing in the ArrT project, and that's what we do by using the mechanism of the "X3D Prototypes".

Without going into the detail of the mechanism now: The SRR Framework is an intermediate layer that lies between Web3D browser and scene and offers general functions that are not so general that they are housed in a Web3D browser, but at least as general that they are used again and again in connection with railways.

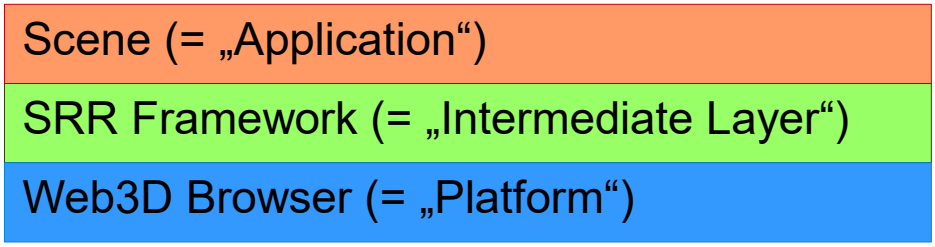


*Figure 1: Simulated Railroad Framework as Intermediate Layer*

## 1.2 The SRR Framework is Demanding

You can not put the SRR Framework under any scene.

The SRR Framework requires the scene to follow certain rules for it to work.

To be sure, the SRR Framework not only expands, but also restricts the X3D standard.

The scene author must adhere to the following rules:

- The MMF paradigm and naming rules must be followed
- The interface definition of the SRR prototypes must be considered

As a picture, you can thus decompose a SrrTrains scene into the following parts:

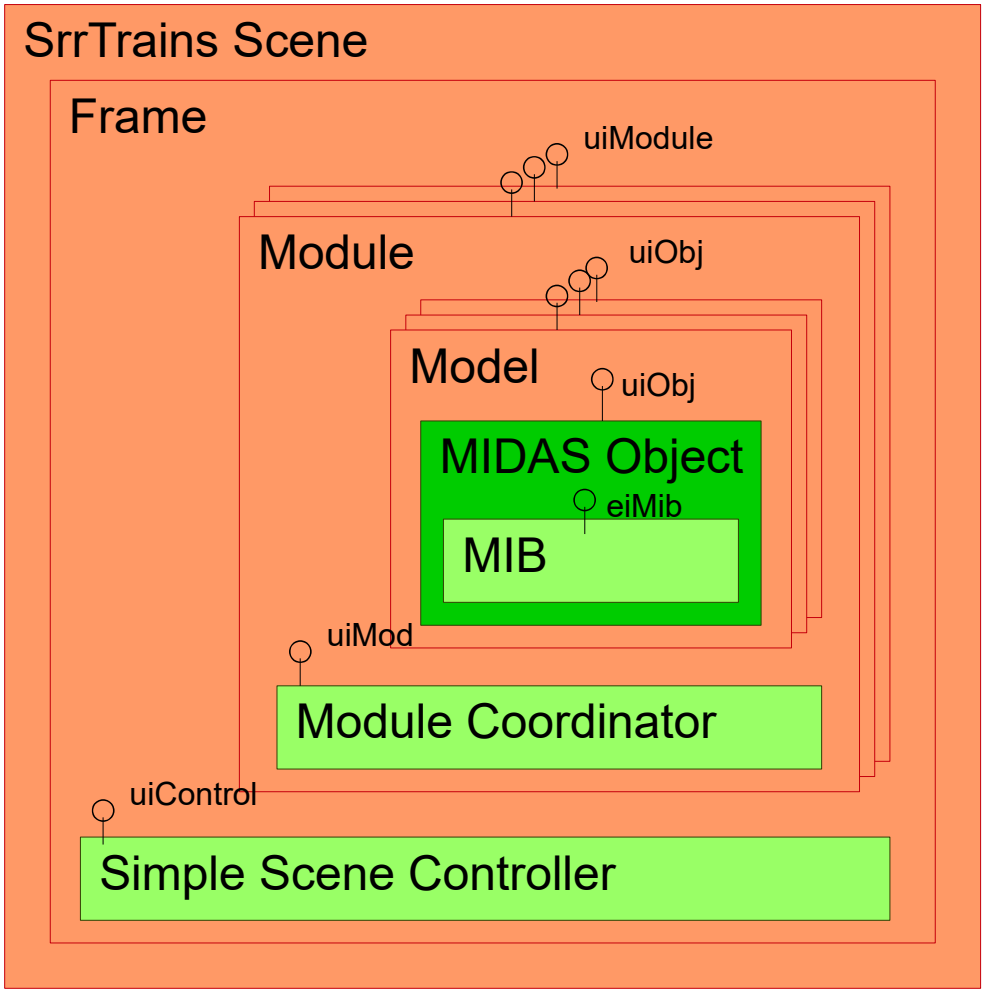


*Figure 2: Structure of an SrrTrains Scene*

Note: you can see in the color scheme that the MIDAS objects are actually not part of the SRR Framework, this only provides the so-called MIDAS Base (MIB), with the help of which the MIDAS objects (MOBs) can be developedönnen.

The MMF paradigm (model / modules / frame paradigm) means that the scene is broken down into modules and models.

Each scene contains at least one module, where modules can come from different authors, and each module contains any number of models, which can also come from different authors.

The SRR Framework provides the necessary standardization by specifying the interfaces of the modules and models, and by providing the light green prototypes whose interfaces are also precisely specified.

In detail, this means for the scene authors:

- A module must adhere to the specification of the uiModule interface so that it can be used in all SrrTrains scenes
- A model must adhere to the specification of the uiObj interface so that it can be used in all SrrTrains modules
- A MIDAS object must adhere to the specification of the uiObj interface so that it can be used in all SrrTrains models
- The Simple Scene Controller (SSC) must be instantiated exactly once per scene
- Module Coordinators (MC) need to be instantiated exactly once per module
- MIDAS objects are always contained in exactly one model
- Models can contain more than one MIDAS object
- The Simple Scene Controller must be initialized as the first of all SRR prototypes
- The modules must be given unique names
- Module Coordinators are initialized after the SSC
- By initializing, the MCs attach to the SSC
- Dynamic modules can be detained and deinitialized exactly once. If you want to show it again, you have to load it again
- Models must have unique names within their parent module
- Within a model, all MIDAS objects must have unique names
- Bound models are initialized after initialization of the MC and attachted to the MC. They will be de-initialized together with their parent module at the latest
- Unbound models can be initialized after the SSC and attached to each initialized MC.
- Unbound models can always be detached and re-attached, even to different Mcs than the first one, but they can only be deinitialisiert once. If you want to use it again after de-initialization, you have to load it again

### 1.3 The SRR Framework ist not yet finished

There are three concepts that are currently not considered in the functionality of the SRR framework (they are not shown in Figure 2 yet):

- Unbound models
- Handover
- Moving Modules

Unbound models are already partially included in this paper, but can only be tested after the completion of Step 0033.

Handover should be prepared in principle, but will be finally implemented and tested after Step 0033.

Moving Modules are currently not considered at all.

## 2 Rebase to MIB Core – Planning

The chapters 2.1 , 2.2 and 2.3 really show only historic considerations about how we planned to rebase the SRR Framework to some Core Prototypes. Therefore we did not translate those chapters from German to English. So sorry.

### 2.1 Die MIDAS Base (MIB)

Im Zuge der Entwicklung hat sich herausgestellt, dass man MIDAS Objekte (MOBs) auf drei Arten von Objekten zurückführen kann:

1. No-State MOBs: das sind MOBs, die überhaupt keinen globalen State verwalten
2. Standard MOBs: das sind MOBs, die einen diskreten globalen State verwalten, also einen State, der sich nicht allzu oft ändert
3. Animated MOBs: das sind MOBs, die einen kontinuierlichen globalen State verwalten, also einen State, der durch eine (quasi-)kontinuierliche Zeitfunktion state(t) dargestellt werden kann

Dementsprechend bestand die MIB vor dem "Rebase to MIB Core" aus folgenden Prototypen:

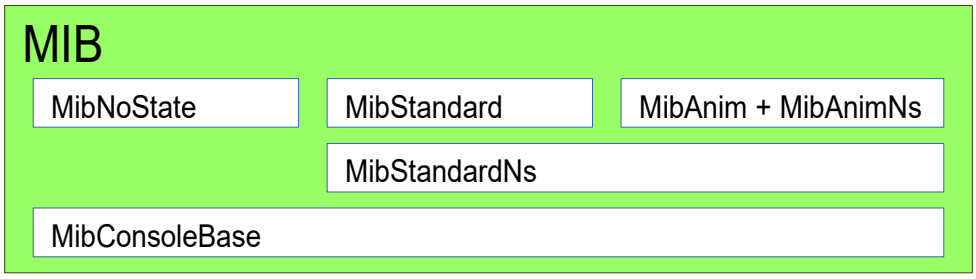


*Abbildung 3: Alter Aufbau der MIB*

## 2.2 Die geplanten Neuerungen in der MIDAS Base (MIB)

In Abbildung 3 sieht man, dass ausser der MibConsoleBase (das sind die Funktionen für das Konsoleninterface, die für alle MOBs gleich sind) ein gemeinsamer Netzwerksensor MibStandardNs definiert ist, der sowohl von standard MOBs als auch von animierten MOBs verwendet wird, und ein Netzwerksensor MibAnimNs, der nur von animierten MOBs verwendet wird.

Nun hat sich herausgestellt, dass es einige Funktionen gibt, die in allen drei Prototypen MibNostate, MibStandard und MibAnim implementiert sind, die also sinnvollerweise in einen gemeinsamen Prototypen auf der Ebene der MibConsoleBase ausgelagert werden sollten.

Es sind das

- Funktionen bezüglich Initialisierung und Attachment

Weiters hat sich gezeigt, dass einige Funktionen in MibStandard und MibAnim parallel implementiert wurden, die man besser auf die Ebene von MibStandardNs auslagern sollte.

Es sind das

- Funktionen der OBCO State Machine

Somit müsste man die MIDAS Base derart ändern, dass man die generellen Anteile in einen "gemeinsamen Keller" auslagert, in eine "Unterschicht".

Diese "Unterschicht" wollen wir als "MIB Core" bezeichnen.

Damit hätte das SRR Framework nach der "Rebase to MIB Core" folgenden Aufbau:

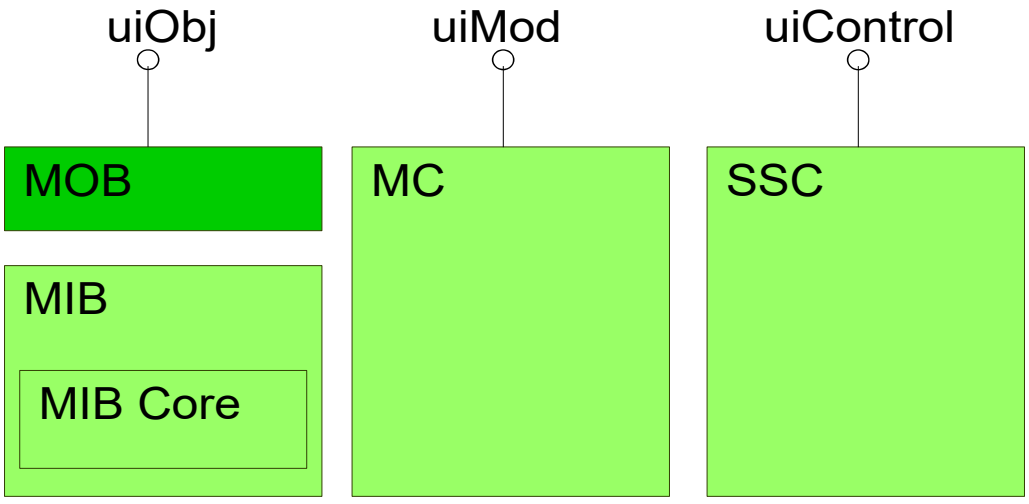


*Abbildung 4: Neuer Aufbau des SRR Framework*

## 2.3 Die Model Prototypes (MOPs)

Es gibt einige Regeln betreffend der Namensgebung von MIDAS Objekten und der Behandlung von verschachtelten MIDAS Objekten sowie bezüglich der Behandlung von Netzwerksensoren in Modellen, die nur als Anleitung für den Modellautor vorhanden sind.

Um den Modellautor bei der Befolgung dieser Regeln zu unterstützen, ist es geplant, sogenannte "Model Prototypes" zur Verfügung zu stellen, die diese Arbeiten übernehmen, die also den Modellautor bei der

- Koordinierung der MIDAS Objekte und Netzwerksensoren

unterstützen.

Damit hätte dann jedes SrrTrains Modell, wenn der Modellautor die MOPs verwendet, folgenden Aufbau:

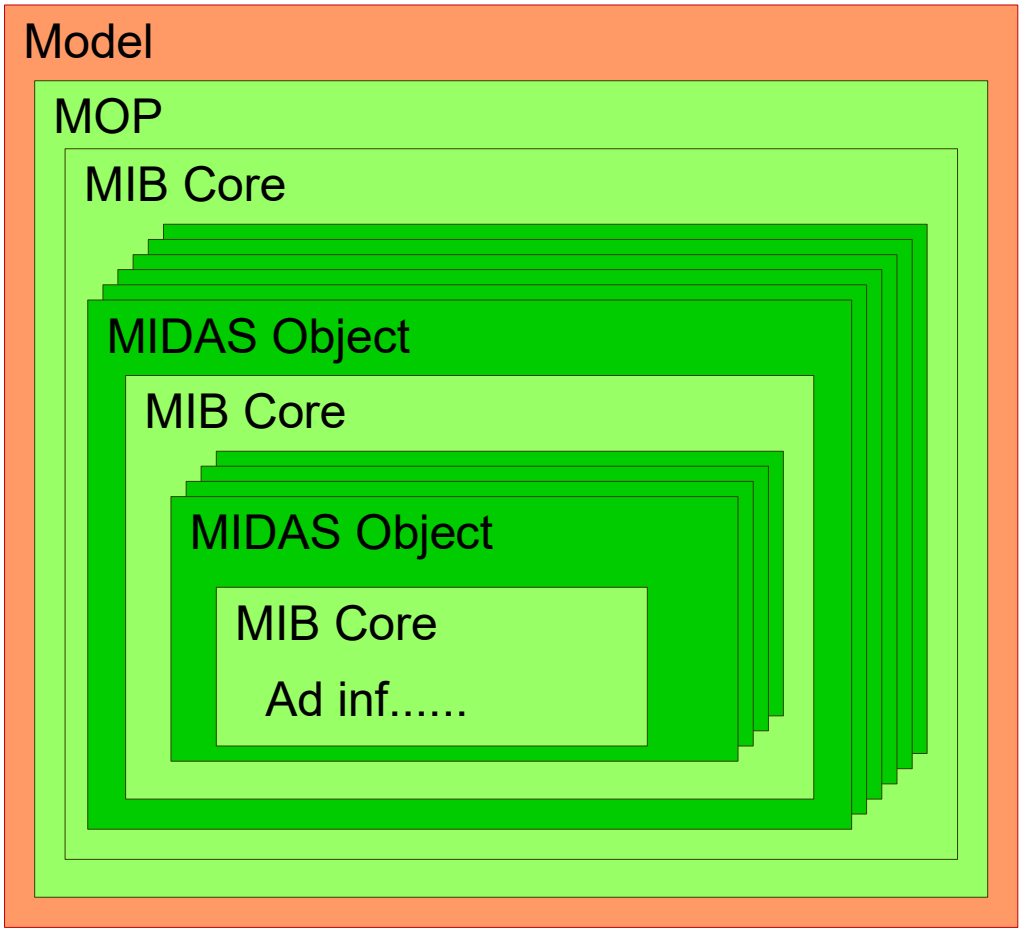


*Abbildung 5: Neuer Aufbau eines SrrTrains Modells*

# 3 Hibernation Report 001 "Pieta / MIB Core"

This chapter is the main chapter of the hibernation report 001 "Pieta / Mib Core". All previous chapters were just introduction and additional explanation of the background.

## Table of Content

The previous chapter has explained in German language, which changes were **planned** for the MIDAS Base (MIB) for step 0033.10.

Now the present chapter tells in English language, which are the **actual results** of step 0033.10.

Hence chapters 1 and 2 are of historical interest only for those, who like to deal with the backgrounds and the history of the implementation.

## 3.1 Simple Multiuser Scenes (SMS) – An Introduction

At the very beginning of the SrrTrains v0.01 project we decided to use Web3D Browsers that comply to the X3D standard.

The X3D standard is a very general ISO standard that can be used for virtually any 3D scene in the World Wide Web.

However, our aim was something more special: we were aiming at what we called "Simple Multiuser Scenes (SMS)".

Any author should be able to easily implement an SMS, based on Web3D Browsers.

Now it turned out that much of the work that had to be done to implement an SMS, would always be the same, at least very similar, for each and every SMS again and again.

This led to the goal to implement a general framework – as general as to be useful for any SMS – but not as general as a Web3D Browser.

The extensibility of X3D turned out to be a very useful concept for this purpose.

This can be shown in a layered view.

The lower a layer the more general it is and the higher a layer the more specialized.

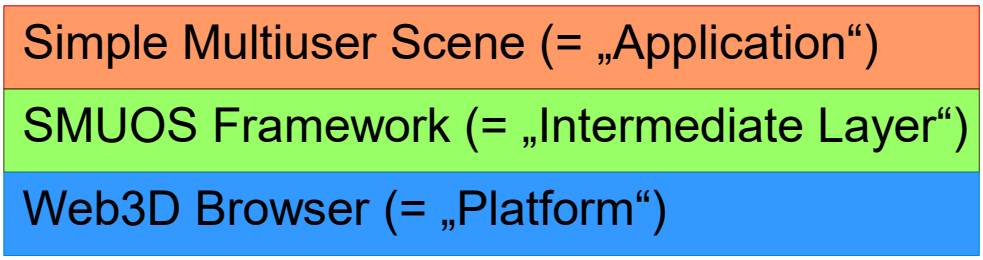


*Figure 1: A layered view of a Simple Multiuser Scene (SMS)*

## 3.2 The SMUOS Framework – Overview

The SMUOS Framework (Simple Multiuser Online Scenes Framework), which is actually a part of the SRR Framework, requires from the scene to follow **what we call the "MMF Paradigm"**.

This just means the authors have to decompose the scenes into ***m***odels, ***m***odules and ***f***rames.

Figure 2 tries to explain the relations among the parts of the SMUOS Framework and the models, modules and frame of a scene.

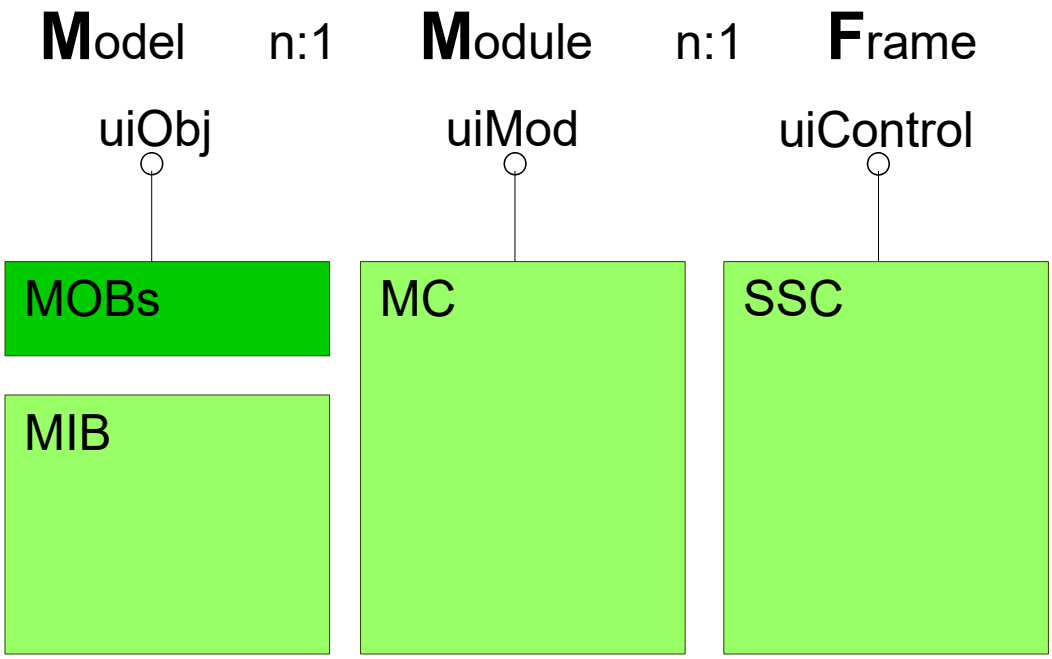


*Figure 2: Parts of the SMUOS Framework (SSC, MC, MIB) and MIDAS Objects (MOBs)*

The light green parts, MIB, MC and SSC, are actually parts of the SMUOS Framework, where the dark green part (the MOBs) are not actually parts of the SMUOS Framework, but they are closely related to the SMUOS Framework.

Each of the three light green parts has got an own core prototype during step 0033.10 – as described in chapter 3.2.3 - but first let's dig a little bit into the basics:

### 3.2.1 Are You an Author or Are You A Programmer

Well, when we talk about computer graphics, then the walls between programmers and authors will probably melt down.

Authors can only build sophisticated features of their scenes, when they are able to do this or that coding gimmick.

Programmers can only test their software, when they are able to build scenes specific for this or that purpose.

On the other hand, we make this difference between "authors" and "programmers" to indicate the parts of the software that are more graphically involved (those parts are created by what we call "authors") and the parts of the software that are more programmatically involved (those parts are created by what we call "programmers").

> *Definition: We arbitrarily set a border between authors and programmers, which we call the **user interfaces**.*

1. The **user interface uiObj** is the interface between MIDAS Object and Model, where a programmer provides a MIDAS Object and an author uses the MIDAS Object to instrument a model.
2. The **user interface uiMod** is the interface between Module Coordinator and Module, where a programmer provides the module coordinator and an author uses the Module Coordinator to instrument a module.
3. The **user interface uiControl** is the interface between Simple Scene Controller and Frame, where a programmer provides the Simple Scene Controller and an author uses the Simple Scene Controller to instrument a frame.

### 3.2.2 External Interfaces

In chapter 3.2.1 we explained the user interfaces.

Well, every user interface is an **external interface**, but not every external interface is a user interface.

> *Definition: External interfaces are well defined interfaces of the SMUOS Framework that can be used by programmers or by authors to extend or to use the SMUOS Framework.*

Interfaces that can only be used internally to the SMUOS Framework or interfaces that can be used externally but are not well defined, are not denoted "external" interfaces.

So the MIDAS Base provides an external interface to the programmer, who wants to build a MIDAS Object. **This interface is called eiMib**.

Actually, the MIDAS Base was implemented to ease the implementation of MIDAS Objects.

A programmer of a MIDAS Object could omit the usage of the eiMib interface and directly use the MC and SSC via the **external interfaces eiMod and eiControl**, respectively.

These considerations are shown in Figure 3.

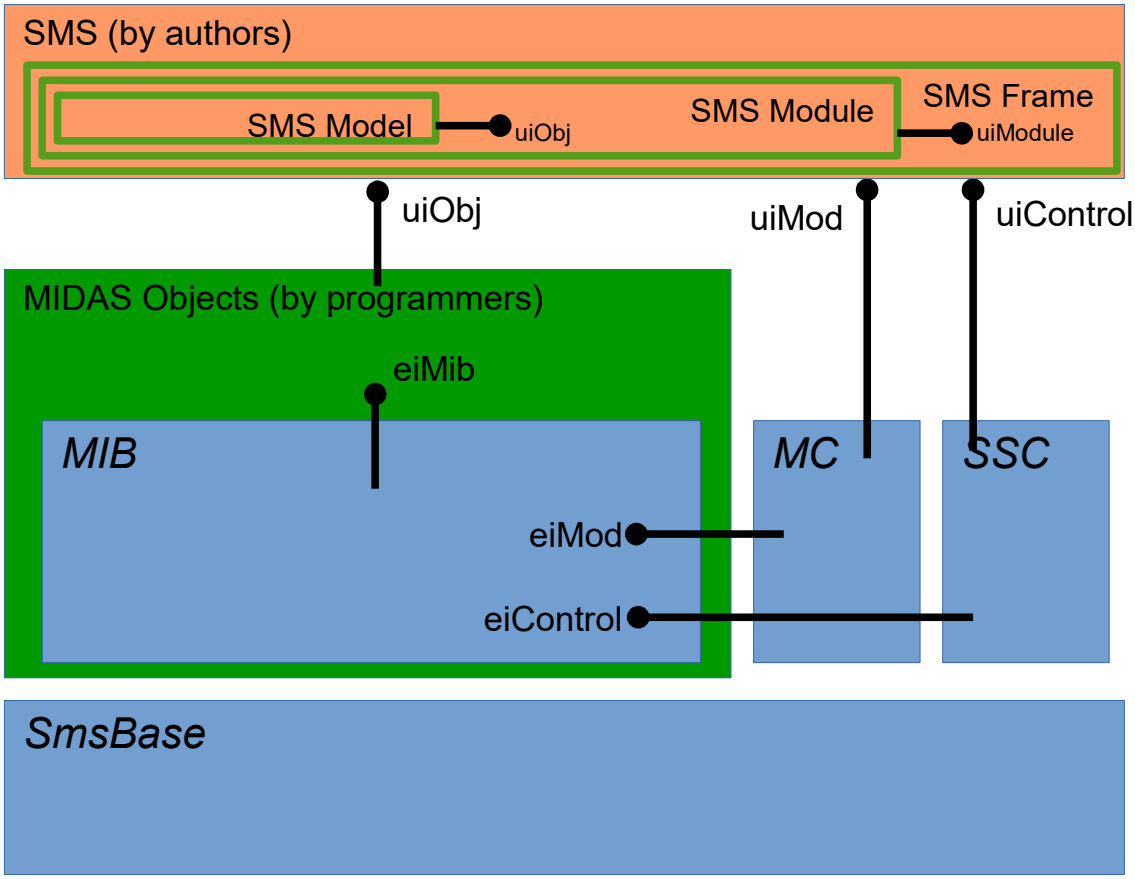


*Figure 3: External interfaces of the SMUOS Framework*

In Figure 3 we do not only see the external interfaces of the SMUOS Framework, but we also see what we call **minimum interfaces**.

Some parameters must be transported from the SSC to the MC via the frame and the module, or from the MC to the MIDAS Objects via the module and the model.

Hence the SMUOS Framework requires minimum sets of fields that must be present at the external interface of any SMS Module (**uiModule**) and of any SMS Model (**uiObj**), respectively.

### 3.2.3 Three Parts of the SMUOS Framework – Decomposition

#### *3.2.3.1 MIDAS Objects (MOBs) and MIB – What's Their Purpose?*

The magic of **MIDAS objects (MOBs)** is the "magic of the intermediate layer".

What does this mean? Well, when we are using the Network Sensor, then we can build virtually anything. Anything that needs a shared state in a multiuser scene can be built on the Network Sensor. So why do we need MOBs?

Well, if a scene author wants to build a car on the Network Sensor, than he needs to implement a steering, a motor, some shared state for the doors to get open or closed and so on, that's all.

Unfortunately **each and every scene author**, who uses the Network Sensor to build a car, **has to do the same work again and again**: building steerings, motors and other specialized mechanisms from the general Network Sensor.

The concept of the MOBs allows to **build specialized mechanisms that can be re-used again and again**, thus saving tons of effort.

Hence it must be clear that the MOBs cannot be parts of the SMUOS Framework, because the SMUOS Framework is a rather general appliance and the MOBs are rather specialized appliances.

Nevertheless, we have identified some common properties that apply to all MOBs. We grouped these properties according to three types of MOBs and built the base software for MOBs – i.e. the **MIDAS Base (MIB)**. This is actually a part of the SMUOS Framework.

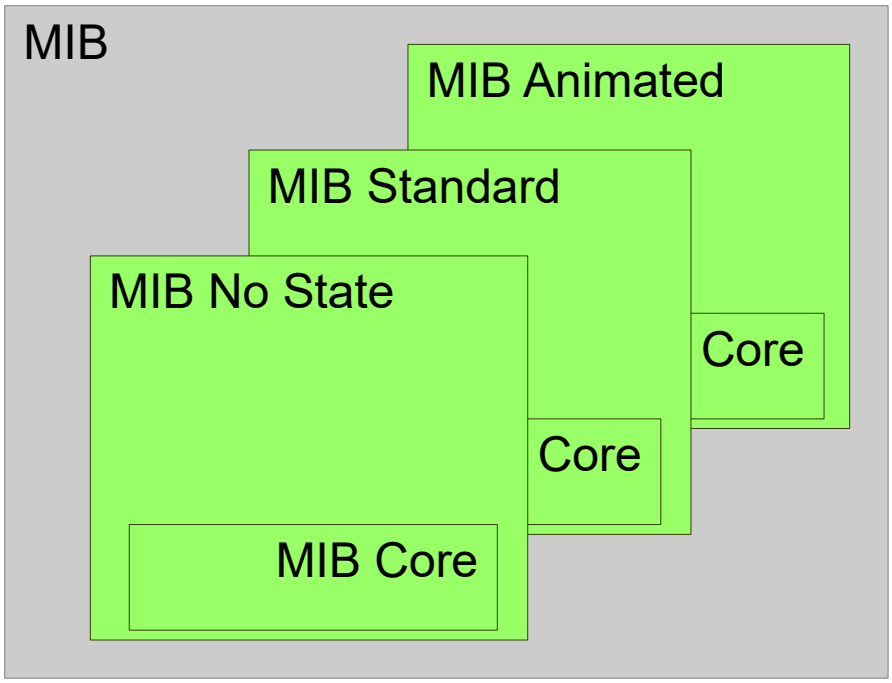


*Figure 4: MIDAS Base (MIB) for three types of MIDAS Objects (MOBs)*

#### 3.2.3.2 Module Coordinator (MC) – What's its Purpose?

One basic concept of the SrrTrains v0.01 project is to decompose the terrain and the objects like houses and trees into what we call "modules".

The first idea was to give more than one scene author the possibility to work together on one landscape (on what we call "layout").

Additionally, all MOBs within one module might share common properties, e.g. we could "deactivate" a module, when the user would not be interested in this module (e.g. he would be roaming far away from the module and he would not look at the module).

So the basic duty of the **Module Coordinator (MC)** is to coordinate all MOBs of one module.

Later we found that some specialized MIDAS Objects could require more functionality from the MC than that, which is present in the basic MC (**MC Base**).

This led to the possibility to extend the MC by your own extension (**MC Extension**).

Some basic properties that are of common interest for all MC Extensions have been outsourced to the **MC Core** in step 0033.10 (Release "Pieta").

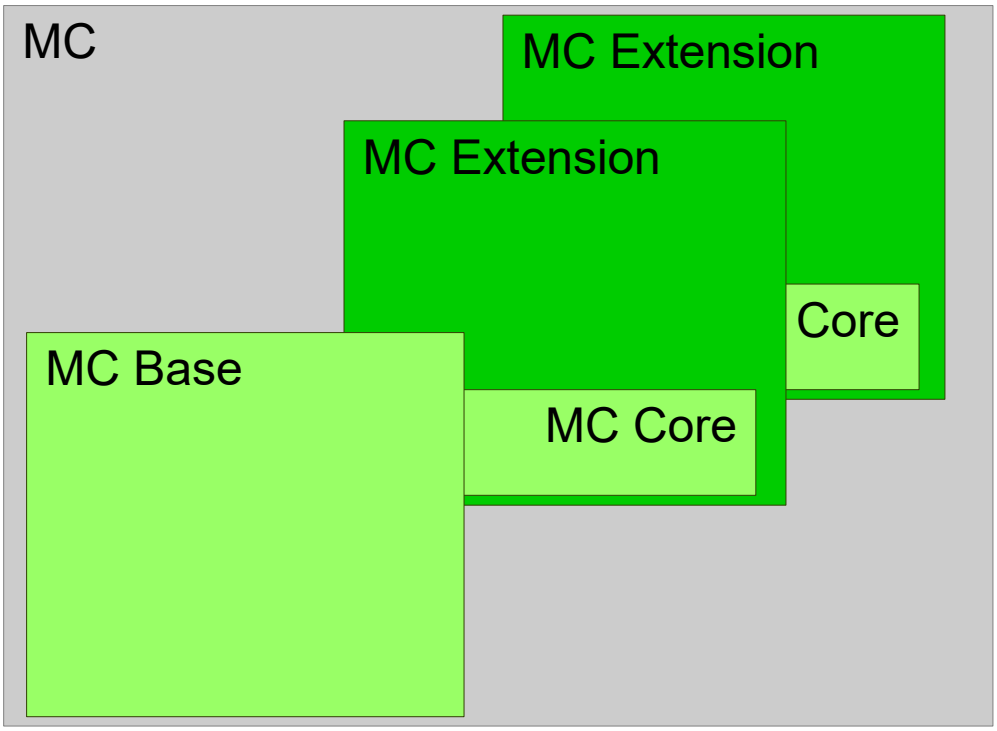


*Figure 5: MC Base, MC Core and 3rd Party MC Extensions*

The dark green parts are provided by external programmers / projects (3rd parties), where the light green parts are actually parts of the SMUOS Framework.

#### 3.2.3.3 Simple Scene Controller (SSC) – What's its Purpose?

After discussing the smallest parts of the scene – (M)odels using MIDAS Objects using the MIB – and the second-smallest parts of the scene – (M)odules using the Module Coordinator – it is now time to discuss the (F)rame.

The (F)rame is the part of the scene that provides all common and basic functionalities. It supports (M)odels and (M)odules in presenting a scene to the user.

Hence we say the user can **inhabit** a virtual scene (SMS).

The frame needs a **central access point** to access the functions of the SMUOS Framework and the Module Coordinators and the MIDAS Objects need some **central intelligence**.

This central access point and intelligence are provided by the **Simple Scene Controller (SSC)**.

Some MIDAS Objects and some MC Extensions might need additional functions that are not provided by the **SSC Base**.

Hence we added the possibility to implement your own **SSC Extensions**.

In step 0033.10 (Release "Pieta") we outsourced some common functions that are common to all SSC Extensions into the **SSC Core**.

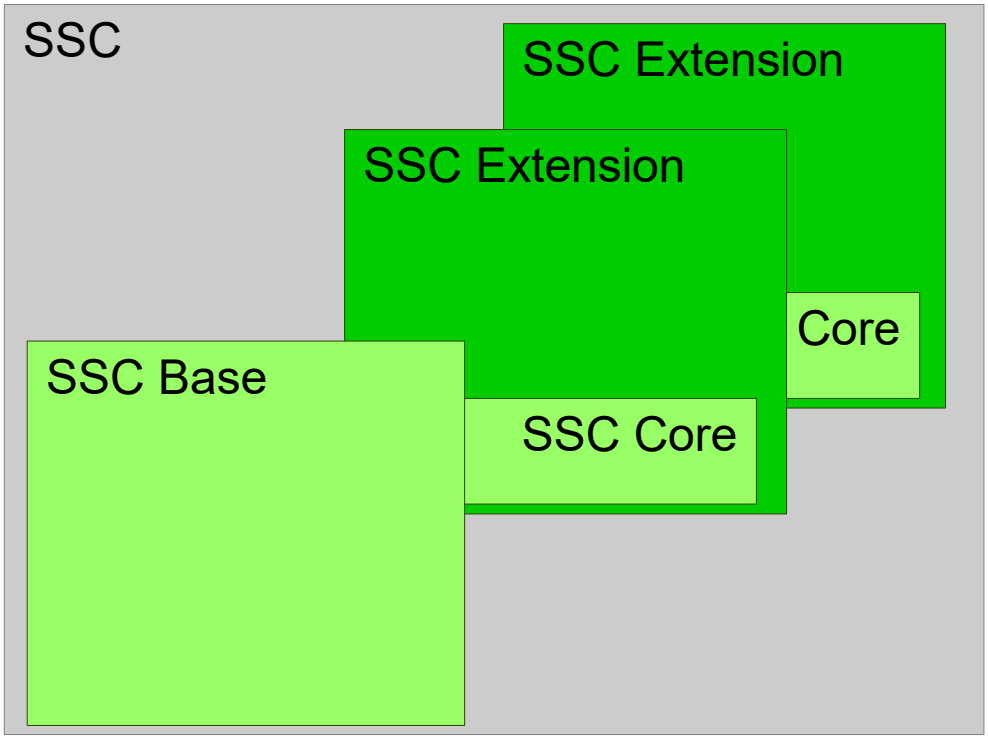


*Figure 6: SSC Base, SSC Core and 3rd Party SSC Extensions*

The dark green parts are provided by external programmers / projects (3rd parties), where the light green parts are actually parts of the SMUOS Framework.

## 3.3 The Simple Scene Controller

Chapter 3.2.3.3 introduced the Simple Scene Controller (SSC) in a short and concise way.

The present chapter describes the SSC in a more detailed way, in particular we focus on the reader, who wants to implement an SSC Extension.

### 3.3.1 Purpose of the Simple Scene Controller

The SSC tries to be a **central access point within each scene instance (SI 1, SI 2, ...)**.

That means, all aspects of the SMUOS Framework that are relevant to the whole simulation, can be accessed via the **user interface uiControl**.

The SSC needs to maintain a global state (i.e. a set of values that are identical to all scene instances). Regarding this global state we distinguish the common **Communication State (commState)**, which is maintained by the SSC Base, and the specific **global state of each SSC Extension**.

Please find an overview depicted in Figure 7.

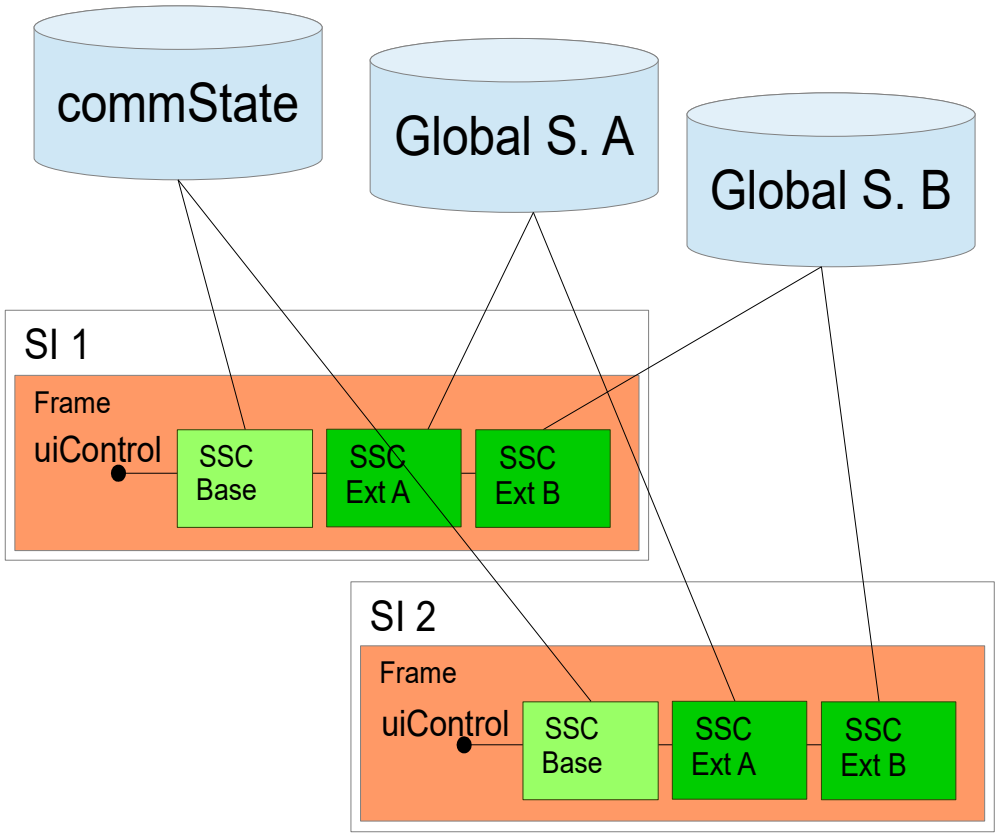


*Figure 7: Global commState and global states of all SSC Extensions*

The **initialization of the SSC within one scene instance (SI)** is started by the user of the SSC (i.e. by the frame author), when he sends the event "init" via the uiControl Interface to the SSC Base.

Then the SSC Base and all dependent SSC Extensions are initialized, i.e. they change from Mode of Operation (MOO) "LOADED" to Mode of Operation (MOO) I "initialized".

This is not enough to issue the Common Parameters (commParam), because the global state "Communication State (commState)" must be handled in advance. This initial handling of the global state is called "activation".

After the commParam have been issued, then the SSC Extensions are activated, too.

Please find an illustration in Figure 8 and in the following listing.

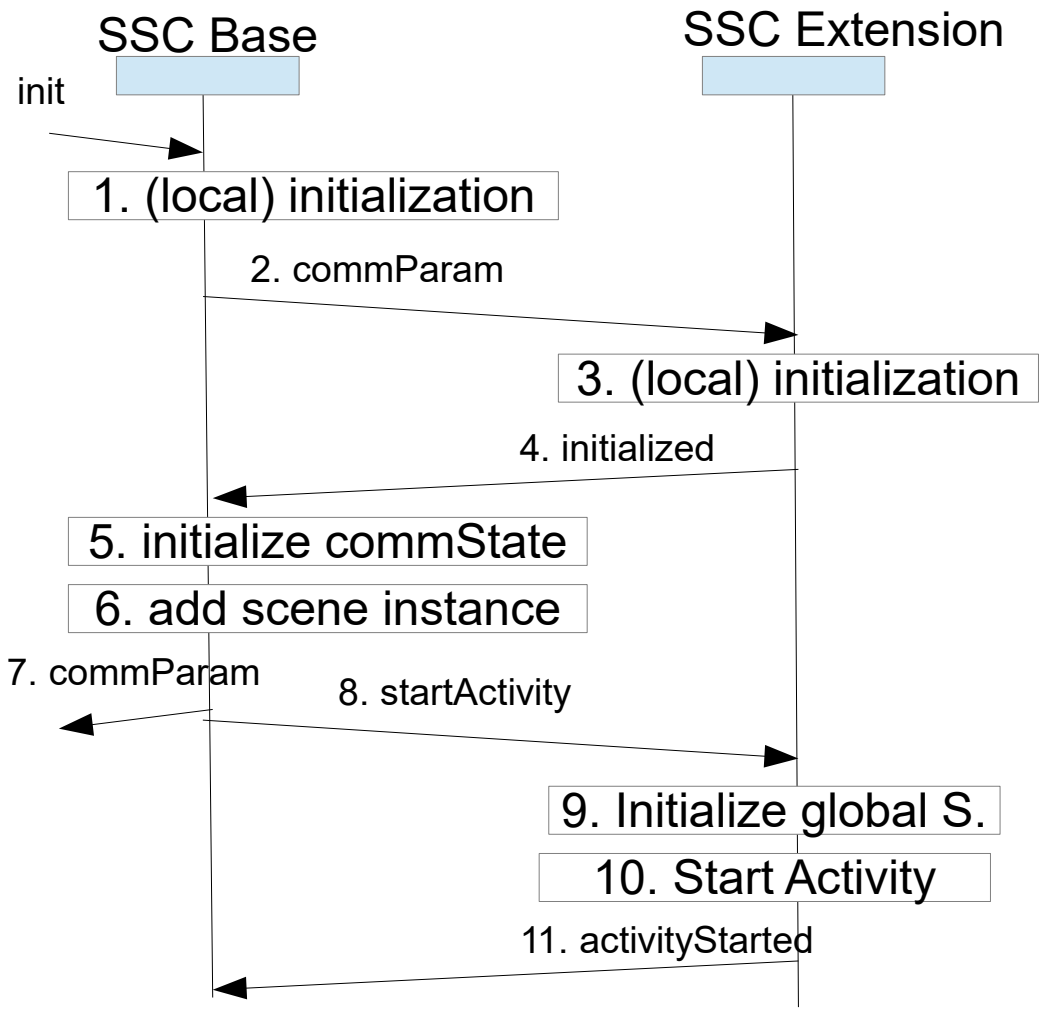


*Figure 8: SSC initialization and start activity – information flow*

1. Local Initialization of the SSC Base
   The frame of the scene triggers the initialization of the SSC with the help of the field "init" at the uiControl interface.
   Only the most important fields of the Common Parameters (commParam) are initialized now, just to allow all SSC Extensions to be initialized locally
2. Forwarding of commParam to all dependent SSC Extensions
   All dependent SSC Extensions are triggered to get initialized by the commParam
3. Local Initialization of all dependent SSC Extensions
   Each SSC Extension (exactly the SSC Core in each SSC Extension)
   - triggers and waits the local initialization of all mandatory dependent SSC Extensions
   - triggers and waits the local initialization of all optional dependent SSC Extensions
   - triggers and waits the initialization of all SSC Dispatchers of the SSC Extension
   - triggers and waits the call back procedure "initialization" (here the author of the SSC Extension can include self-written code)
   - reports "initialized" in case of successful local initialization (see step 4.)
   - triggers the initialization of all network sensors of the SSC Extension
4. Local Initialization of all dependent SSC Extensions is finished
5. Initializing the commState
   If the scene instance is the first instance of the multiuser session, then the commState will be explicitely initialized to "empty"
6. Add Scene Instance to commState
   The scene instance sends an "Access Request" to the server part of the SSC and hence the scene instance will be added to the commState
7. Now the Common Parameters are valid and the flag "iAmActive" in the commParam is set to TRUE. The flag "iAmActive" of the Common Parameter **Extensions** is still FALSE. However, the commParam can be used by the scene now and hence are published by the SSC via the uiControl interface.
8. If the scene instance is the first of the multiuser session, then the SSC Extensions are triggered with the field "initializeStateAndStartActivity", otherwise they are triggered with "startActivity"
9. Each SSC Extension initializes its global state (if it was triggered with "initializeStateAndStartActivity" – otherwise the state is already valid)
10. Now the local copy of the global state is valid and the SSC Extension can "start Activity". The meaning of "start Activity" may vary from SSC Extension to SSC Extension
11. The SSC Extension reports that activity has been started, the "iAmActive" flag in the Common Parameter Extension is set to TRUE.

### 3.3.2 The Common Parameters (commParam)

The basic idea of the Common Parameters is to have a <Script> node within the SSC that holds parameters for the whole scene instance, i.e. parameters that are "common" to the whole scene instance.

Now when we defined that the SSC Base may be extended by SSC Extensions, then the idea was obvious to add "Common Parameter Extensions" to the Common Parameters, one for each SSC Extension.

The Common Parameters got a field "extensions (MFNode)", which pointed to the Common Parameter Extensions.

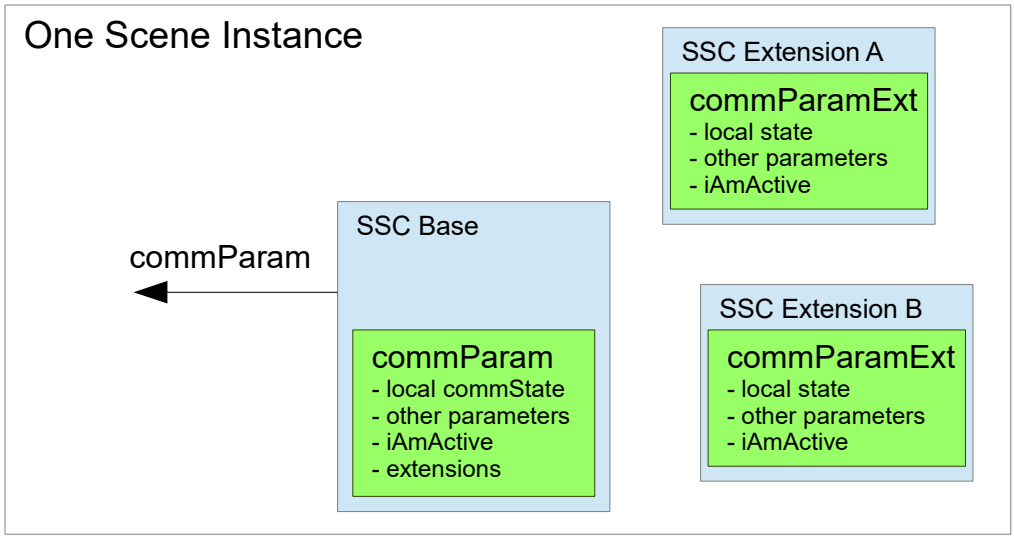


*Figure 9: The concept of commParam and commParamExt*

Each of those script nodes would hold a local copy of the global state of the part of the SSC (i.e. of the commState for SSC Base and of the global state for each SSC Extension).

The "iAmActive" flag, which we talked about in the above chapter, would also be located within those script nodes.

Last but not least, the commParam would point to all commParamExt and each commParamExt would point to the commParam.

When SSC Base publishes the "commParam" reference, then it guarantees that all mandatory dependent SSC Extensions are successfully initialized, that all optional dependent SSC Extensions are successfully or unsuccessfully initialized and that the field "extensions" points to all successfully initialized dependent SSC Extensions, but not to any unsuccessfully initialized dependent SSC Extension.

### 3.3.3 Interfaces of the Simple Scene Controller

This chapter tries to give some hints for those, who like to implement their own SSC Extension.

Following figure should give a short and concise introduction:

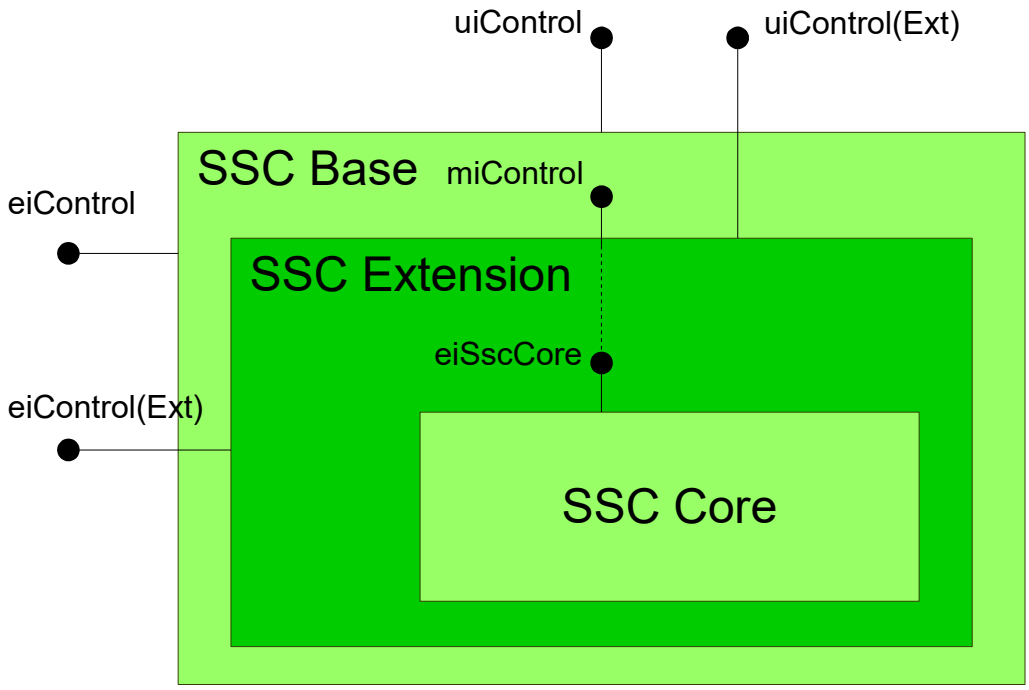


*Figure 10: Considerations for the implementation of an SSC Extension*

An SSC Extension

- **must** provide the **minimum interface miControl** to the SSC Base (or to the parent)
- **may** use the **external interface eiSscCore** of the SSC Core to ease the implementation
- **may** provide a **user interface uiControl(Ext)** to the frame.
- **may** provide an **external interface eiControl(Ext)** to its MC Extension(s) and/or to its MIDAS Objects

## *3.4 The Module Coordinator*

Chapter 3.2.3.2 introduced the Module Coordinator (MC) in a short and concise way.

The present chapter describes the MC in a more detailed way, in particular we focus on the reader, who wants to implement an MC Extension.

### 3.4.1 Purpose of the Module Coordinator

The module coordinator

- coordinates all MIDAS Objects within one module within one scene instance (SI)
- enables access to the global state(s) of the SSC (Extensions) by MIDAS Objects

The first task will become more understandable in the next chapters, the second task is depicted in Figure 11 at an example, where SMUOS Extension A consists of an SSC Extension A and SMUOS Extension B consists of an SSC Extension B and an MC Extension B.

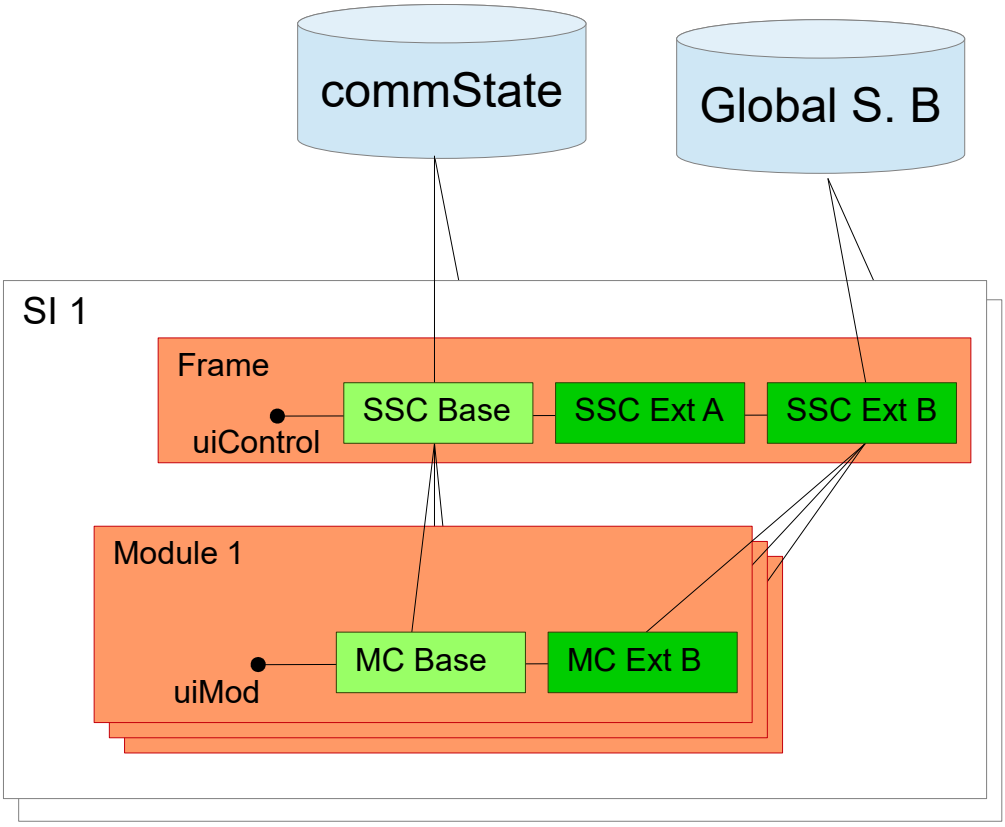


*Figure 11: Access to SSC global state(s) via the SSC (Extensions)*

The **initialization of the Module Coordinator (MC) within one instance of a module** is started by the user of the MC (i.e. by the module author), when he sends the event "commParam" via the uiMod interface to the MC Base.

Then the MC Base and all MC Extensions change their Mode of Operation (MOO) from MOO "LOADED" to MOO I "initialized".

This is not enough to issue the module parameters (modParam), because the MC Base must be **attached to the SSC Base** in advance, in order to **access the Module Activity Matrix (MAM)**, which is a part of the commState. Now the MC Base changes to MOO II "attached".

After the modParam have been issued, the MC Extensions are attached to their SSC Extensions, too.

Please find a detailed explanation in Figure 12 and in the following listing:

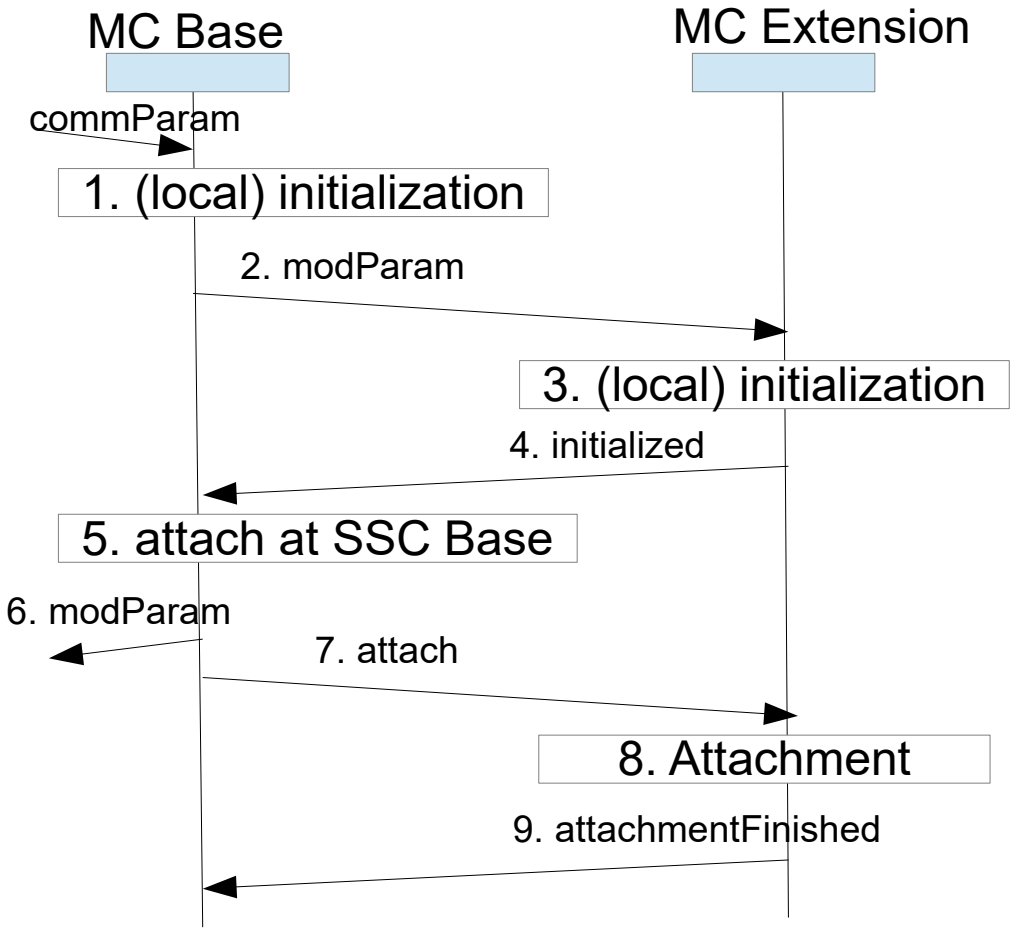


*Figure 12: MC initialization and attachment - information flow*

1. Local Initialization of the MC Base
   The instance of the module triggers the initialization of the MC with the help of the field "commParam" at the uiMod interface.
   Only the most important fields of the Module Parameters (modParam) are initialized now, just to allow all MC Extensions to be initialized locally
2. Forwarding of modParam to all dependent MC Extensions
   All dependent MC Extensions are triggered to get initialized by the modParam
3. Local Initialization of all dependent MC Extensions
   Each MC Extension (exactly the MC Core in each MC Extension)
   - triggers and waits the local initialization of all mandatory dependent MC Extensions
   - triggers and waits the local initialization of all optional dependent MC Extensions
   - triggers and waits the call back procedure "initialization" (here the author of the MC Extension can include self-written code)
   - reports "initialized" in case of successful local initialization (see step 4.)
4. Local Initialization of all dependent MC Extensions is finished
5. Attach the MC Base to the MAM (SSC Base)
   The MC Base sends an "Announcement Request" to the SSC Base and requests to be attached to the commState. Hence the module gets a "moduleIx", and if it is not yet registered, then it will be implicitly registered
6. Now the Module Parameters are valid and the index "moduleIx" in the modParam is set to the valid value >= 0. The index "moduleIx" of the Module Parameter Extensions is still < 0. However, the modParam can be used by the scene now and hence are published by the MC via the uiMod interface.
7. The MC Extensions are triggered with the field "attach"
8. Now each MC Extension can perform the "Attachment". The meaning of "Attachment" may vary from MC Extension to MC Extension
9. The MC Extension reports that attachment has been finished, the "moduleIx" index in the Module Parameter Extension is set to the correct value >= 0.

### 3.4.2 The Module Parameters (modParam)

The basic idea of the Module Parameters is to have a <Script> node within the MC that holds parameters for the whole instance of the module, i.e. parameters that are "common" to the whole instance of the module and its MIDAS Objects.

Now when we defined that the MC Base may be extended by MC Extensions, then the idea was obvious to add "Module Parameter Extensions" to the Module Parameters, one for each MC Extension.

The Module Parameters got a field "extensions (MFNode)", which pointed to the Module Parameter Extensions.

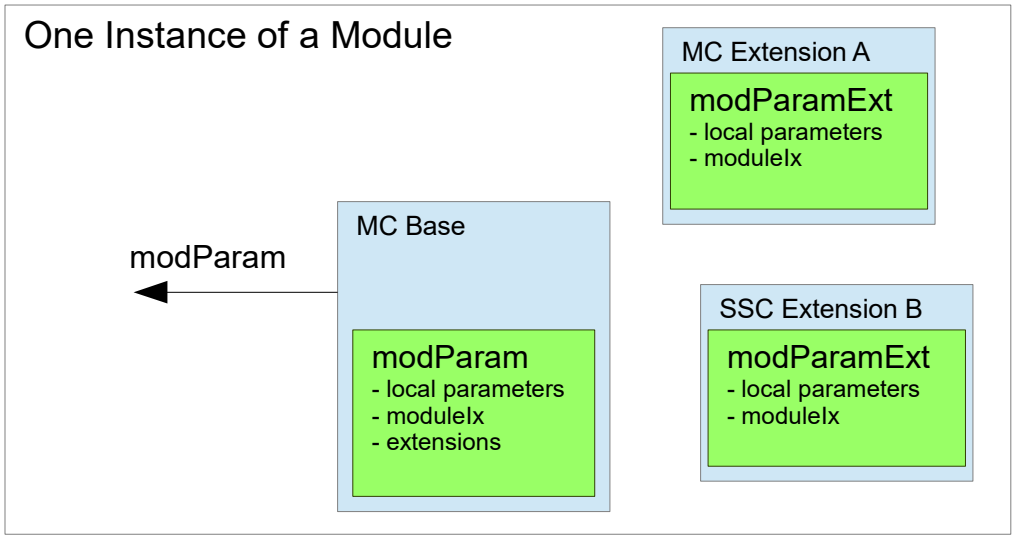


*Figure 13: The concept of commParam and commParamExt*

The "moduleIx" index, which we talked about in the above chapter, would also be located within those script nodes.

Last but not least, the modParam would point to all modParamExt and each modParamExt would point to the modParam.

When MC Base publishes the "modParam" reference, then it guarantees that all mandatory dependent MC Extensions are successfully initialized, that all optional dependent MC Extensions are successfully or unsuccessfully initialized and that the field "extensions" points to all successfully initialized dependent MC Extensions, but not to any unsuccessfully initialized dependent MC Extension.

### 3.4.3 Interfaces of the Module Coordinator

This chapter tries to give some hints for those, who like to implement their own MC Extension.

Following figure should give a short and concise introduction:

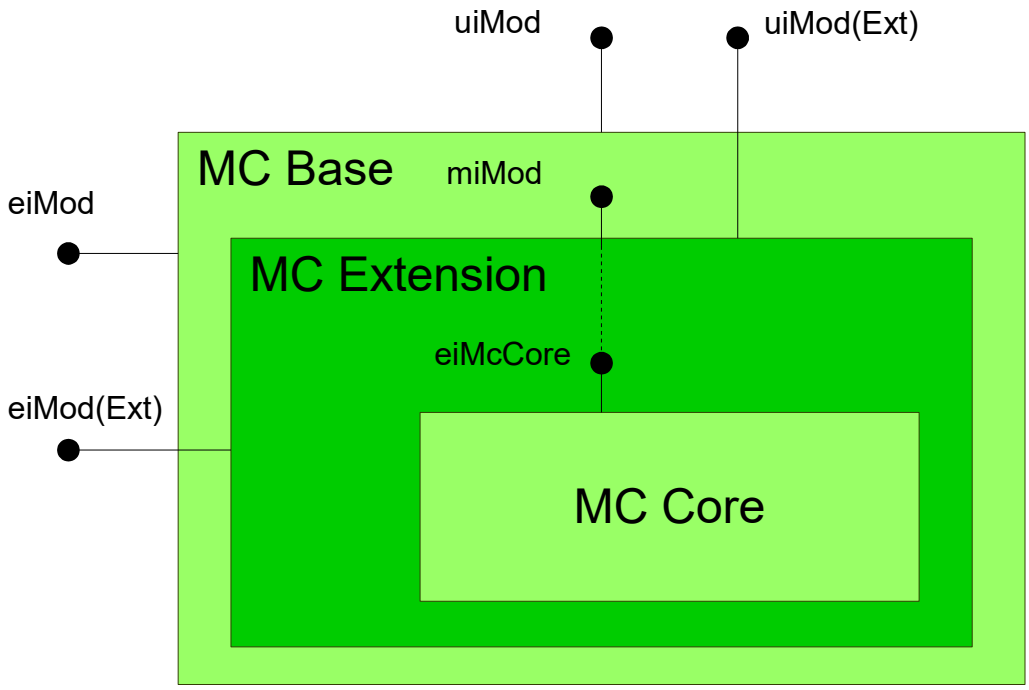


*Figure 14: Considerations for the implementation of an MC Extension*

An MC Extension

- **must** provide the **minimum interface miMod** to the MC Base (or to the parent)
- **may** use the **external interface eiMcCore** of the MC Core to ease the implementation
- **may** provide a **user interface uiMod(Ext)** to the module.
- **may** provide an **external interface eiMod(Ext)** to its MIDAS Objects

## *3.5 MIDAS Objects and the MIDAS Base*

Chapter 3.2.3.1 introduced the MIDAS Objects (MOBs) and the MIDAS Base (MIB) in a short and concise way.

The present chapter describes these concepts in a more detailed way, in particular we focus on the reader, who wants to implement an own MIDAS Object.

### 3.5.1 Purpose of MIDAS Objects (MOBs)

Short and sweet, it's the purpose of MOBs to **instrument models**.

Let's have a look at an example in the official demo layout of the SIMULRR project:

The station house is a model, which is located in the module "City". The extended object ID (extObjId) of the station house is **City-StationHouse**.

This is a so-called **bound model** (this term will be explained later).

Now the station house features a door, which can be open, closed, locked or unlocked. Therefore the station house contains two MIDAS Objects, the MoosSwitchA object and the MoosLockB object. The lock is contained in the switch to be able to lock or unlock the switch.

Hence the switch and the lock got following extended object IDs:

1. door switch: extObjId = **City-StationHouse.DoorSwitch**
2. door lock: extObjId = **City-StationHouse.DoorSwitch.Lock**

We see:

a) It's the **purpose of MIDAS Objects to instrument models**

b) **More than one MIDAS Object can be orchestrated to instrument one model**

Now the SMUOS Framework can help in several ways:

1. The SMUOS Framework features a **Model Prototype (MOP "MbBoundModel")**, which helps the author of a bound model to **orchestrate the MIDAS Objects (MOBs) of the model**
2. The SMUOS Framework features **MIBs (MIDAS Bases) for three types of MOBs**, to **ease the implementation of MOBs** by programmers
3. Both (the MOPs and the MIBs) are based on the "MibCore" prototype internally, however this is not visible to the author of the model nor to the programmer of the MOB

All this is depicted in Figure 15, as follows.

The light green parts are actually parts of the SMUOS Framework, the dark green parts are provided by 3rd party programmers and the red part is provided by a 3rd party author.

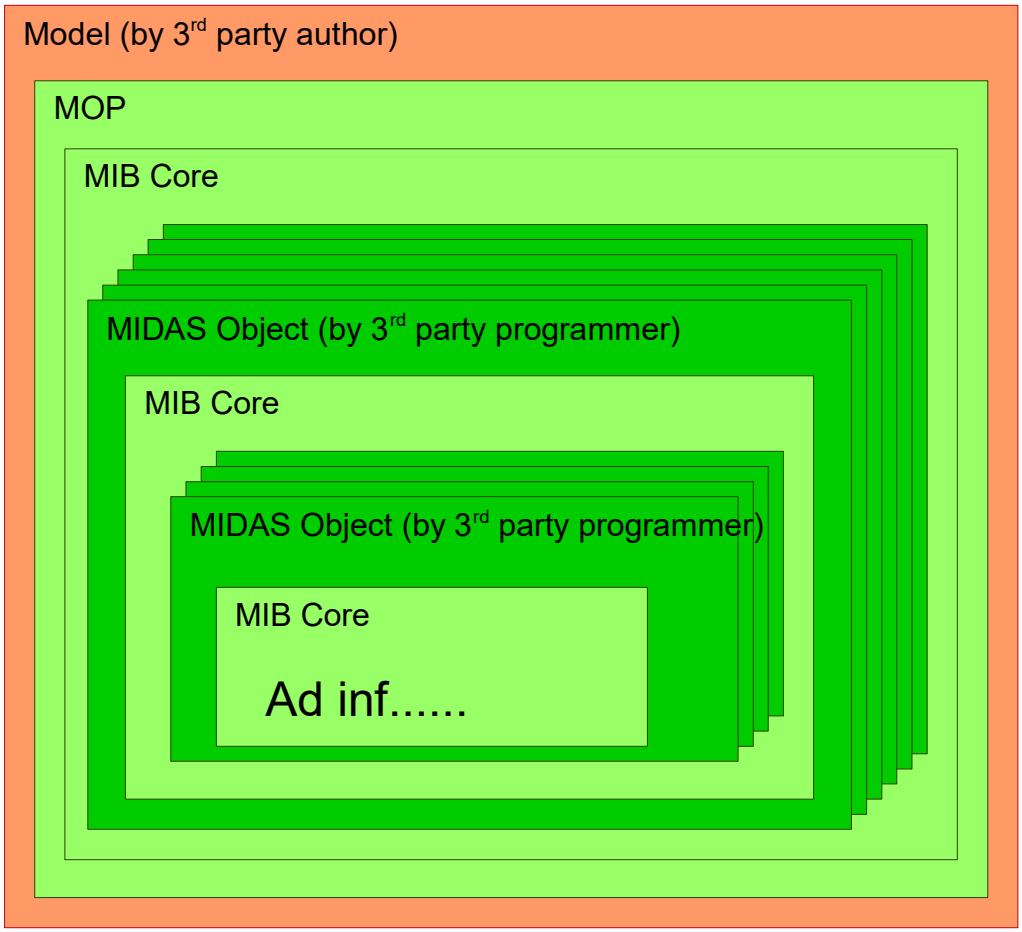


*Figure 15: Orchestrating more than one MOB to instrument a model*

The orchestration is supported by the MibCore prototype, which cares that all MOBs of a model have got the same mode of operation (MOO) at any time.

Let's have a closer look to the modes of operation (MOOs) in the next chapter.

### 3.5.2 Modes of Operation (MOOs) of Objects

On the one hand every object (i.e. every model and every MIDAS Object) needs to be attached to a module in order to get access to the information from the Module Activity Matrix (MAM).

Well, not really every object, but some objects may be "astral" objects, which exist outside of all modules.

On the other hand, we are currently planning to implement "unbound" objects in one of the next steps of the SrrTrains v0.01 project. Unbound objects will be able to exist outside of all modules and will be able to change their module (what we will call "handover").

All this led to the definition of four "Modes of Operation (MOOs)" for ongoing operation and one "Mode of Operation (MOO)" for disabled objects.

- MOO I "initialized" astral object
- MOO II "attached" bound object
- MOO III "initialized" unbound object (aka "detached")
- MOO IV "attached" unbound object
- MOO V "disabled"

Immediately after the object has been loaded, it exists in the MOO "LOADED", to be complete.

The possible MOO Changes (changes of MOO) that can be undergone by objects, are explained in more detail in chapter 3.6.4 .

At a first glance, we can settle following statements:

- "Astral" objects are never attached to a module (but they have access to the commParam)
- "Bound" objects are always attached to the their module (they have access to the modParam of that module) and they cannot change the module they are attached to
- "Unbound" objects may be attached to a module or not and they may change the module they are attached to (they may perform a "handover")

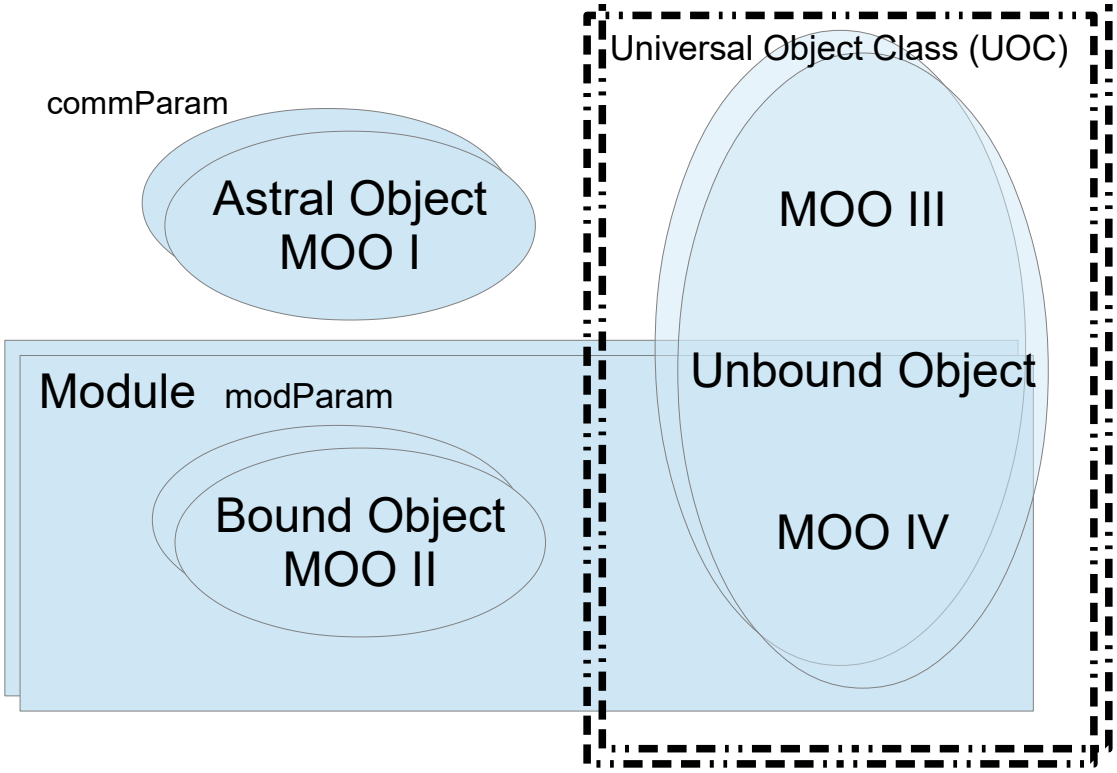


*Figure 16: Objects and their modes of operation (MOOs)*

### 3.5.3 Types of MIDAS Objects

The MOO of an object describes, how this object is related to the modules and to the frame of the scene (SMS).

The "type" of a MIDAS Object is a rough classification about how the object handles its global state (this is the set of values that are stored on the central server to ensure they are synchronized among all scene instances, i.e. among all instances of the object).

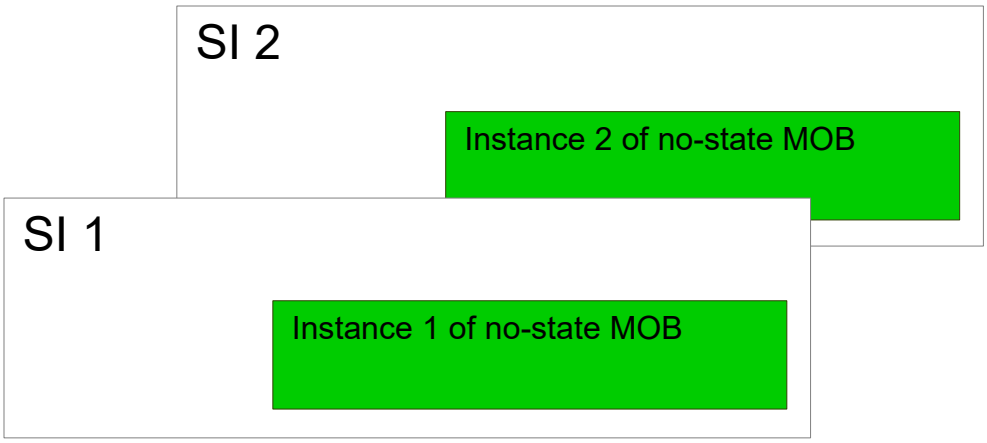


*Figure 17: Instances of a no-state MOB don't share a global state*

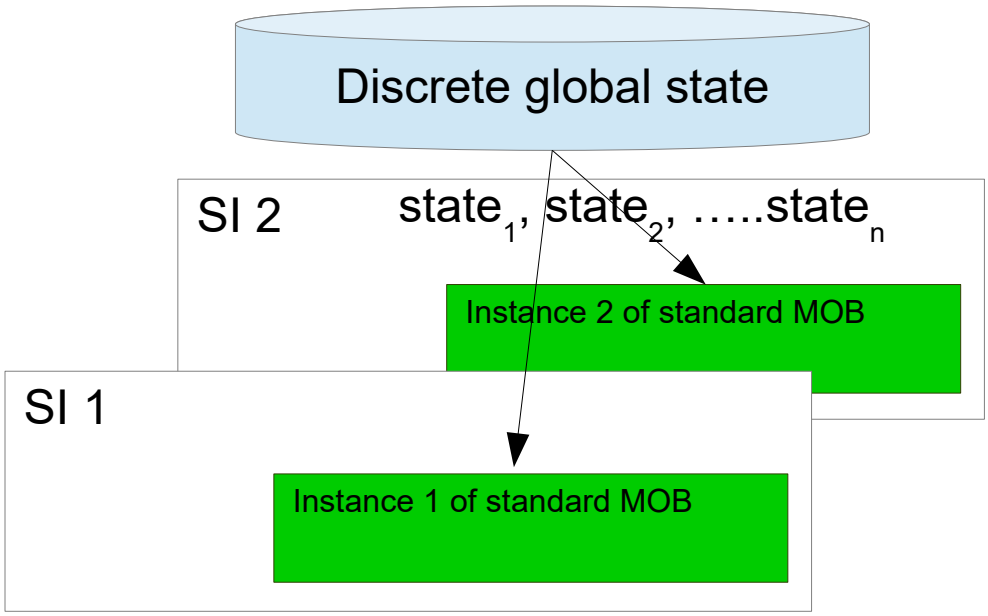


*Figure 18: Instances of a standard MOB share a discrete global state*

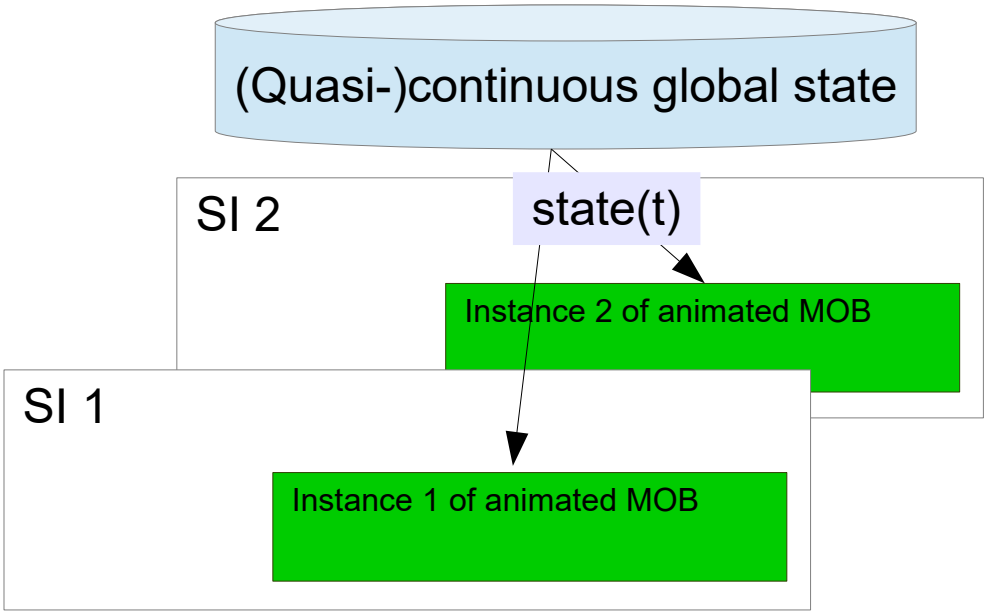


*Figure 19: Instances of an animated MOB share a (quasi-)continuous global state*

### 3.5.4 Purpose of the MIDAS Base (MIB)

The purpose of the MIBs is to ease the implementation of MOBs.

We implemented three MIBs, one for each type of MOB:

- MIB for no-state MOBs
- MIB for standard MOBs
- MIB for animated MOBs

Some core functions – which are equal to all MIBs – are implemented in the MIB Core and in the MIB OSM. However, this is of interest only for the developer(s) of the SMUOS Framework.

The next figure depicts all interfaces that you are involved with, when you implement a MIDAS Object (MOB).

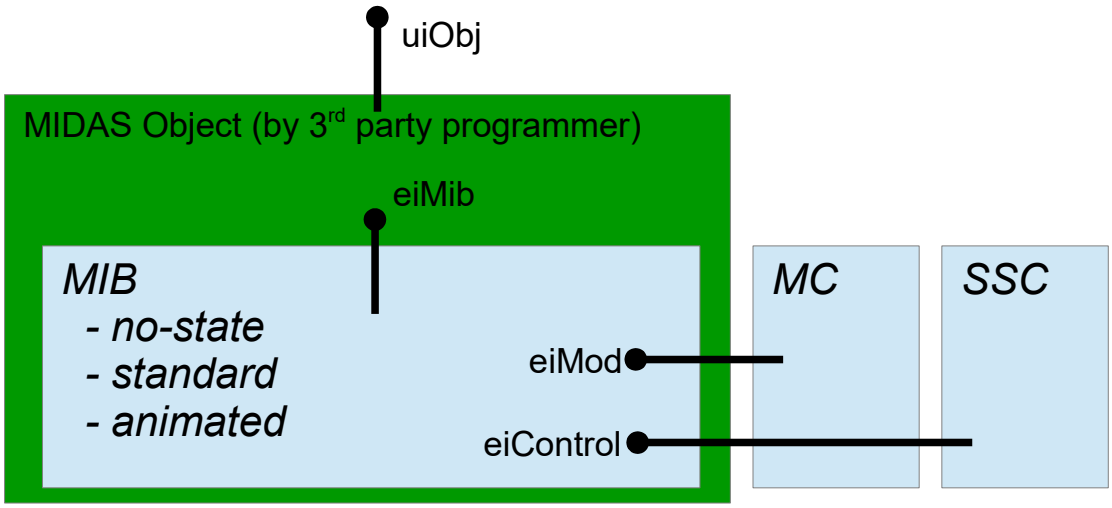


*Figure 20: Relevant interfaces for the implementation of a MOB*

Each MIDAS Object

- **must** provide the uiObj Interface to its users (model authors, module authors, frame authors)
- **may** use the eiMib Interface of one of the MIBs to ease the implementation of a "no-state", of a "standard" or of an "animated" MOB.
- **may** use the eiMod interface to directly access the MC (either MC Base or MC Extensions)
- **may** use the eiControl interface to directly access the SSC (either SSC Base or SSC Extensions)

## 3.6 Settled Properties of the Core Prototypes

SscCore

- organizes the **Access to SSC Extensions** by SSC Base and by any SSC Extension
- handles the **MOO Changes** of any SSC Extension
- orchestrates the **MOO Changes** of all dependent SSC Extensions of SSC Base and of any SSC Extension
- ensures the **MOO Changes** of all UOC Related SSC Dispatchers of SSC Base and of any SSC Extension (this includes the initialization of dispatcher stubs for UOC parameters)
- requests the handling of all **Procedures** from its user (i.e. programmer of the SSC Extension) via a call-back mechanism
- triggers the initialization of all **SSC Related Network Sensors** and of all **UOC Related SSC Dispatchers** of SSC Base and of any SSC Extension
- handles initialization failure of any **SSC Related Network Sensor** and of any **UOC Related SSC Dispatcher** of SSC Base or of any SSC Extension

McCore

- organizes the **Access to MC Extensions** by MC Base and by any MC Extension
- handles the **MOO Changes** of any MC Extension
- orchestrates the **MOO Changes** of all dependent MC Extensions of MC Base and of any MC Extension
- requests the handling of all **Procedures** from its user (i.e. programmer of the MC Extension) via a call-back mechanism
- organizes the access to every **required SSC Extension**, as required by any MC Extension

MibCore

- organizes **Nested Objects**
- handles the **MOO Changes** of any object (i.e. of any model or of any MIDAS Object)
- orchestrates the **MOO Changes** of all dependent objects of any object
- requests the handling of all **Procedures** from its user (i.e. programmer of the MIB) via a call-back mechanism
- organizes the access to every **required SSC Extension**, as required by any object
- organizes the access to every **required MC Extension**, as required by any object
- triggers the initialization of all **Object Related Network Sensors** and of all **MIB Related Network Sensors** of any object
- handles initialization failure of any **Object Related Network Sensor** and of any **MIB Related Network Sensor** of any object

### 3.6.1 Access to SSC Extensions

The following table shows some information, which is **relevant for SSC Base and for any SSC Extension**. It tries to explain, how SSC Base or any SSC Extension can access other SSC Extensions.

**Short and sweet**, each part of the SSC can access all of its dependent SSC Extensions and each SSC Extension can access the SSC Base via

- some fields that each SSC Extension **must** provide at the **minimum Interface miControl,**
- some fields that **should** be used at the **external interface eiSscCore** and
- some fields of the **commParamExt** that **will** be handled by the SSC Core prototype

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

| Interface miControl | Interface eiSscCore | Common Parameter Extension |
|---|---|---|
| **instanceIdClient**<br>**SFString** | **instanceIdClient**<br>**SFString** | |
| **instanceIdServer**<br>**SFString** | **instanceIdServer**<br>**SFString** | |
| | tracerInstanceIdClient<br>SFString | |
| | tracerInstanceIdServer<br>SFString | |
| traceLevelControl<br>SFInt32 | traceLevelControl<br>SFInt32 | |
| traceLevelComm<br>SFInt32 | traceLevelComm<br>SFInt32 | |
| **mandatorySscExtensions**<br>**MFNode** | **mandatorySscExtensionsIn**<br>**MFNode** | |
| **optionalSscExtensions**<br>**MFNode** | **optionalSscExtensionsIn**<br>**MFNode** | |
| | assertedOptionalSscExtensions<br>MFNode | |
| | commParamExtIn<br>SFNode | |
| | | commParam<br>SFNode |
| | | sscBaseExt<br>SFNode |
| | | extensions<br>MFNode |

### 3.6.2 Access to MC Extensions

The following table shows some information, which is **relevant for MC Base and for any MC Extension**. It tries to explain, how MC Base or any MC Extension can access other MC Extensions.

Each part of the MC can access all its dependent MC Extensions and each Extension can access the MC Base.

It's recommended to <connect> **bold written fields** directly from miMod to eiMcCore.

| Interface miMod | Interface eiMcCore | Module Parameter Extension |
|---|---|---|
| **instanceIdMod**<br>**SFString** | **instanceIdMod**<br>**SFString** | |
| | setModuleName<br>SFString | |
| | tracerInstanceId<br>SFString | |
| | tracerInstanceIdClient<br>SFString | |
| localTraceLevel<br>SFInt32 | traceLevelControl<br>SFInt32 | |
| **mandatoryMcExtensions**<br>**MFNode** | **mandatoryMcExtensionsIn**<br>**MFNode** | |
| **optionalMcExtensions**<br>**MFNode** | **optionalMcExtensionsIn**<br>**MFNode** | |
| | assertedOptionalMcExtensions<br>MFNode | |
| | modParamExtIn<br>SFNode | |
| | | modParam<br>SFNode |
| | | smsModCoordExt<br>SFNode |
| | | extensions<br>MFNode |

### 3.6.3 Nested Objects

The following table shows some information, which is **relevant for any MIDAS Object**. It tries to explain, how any MIDAS Object can orchestrate all its dependent MIDAS Objects.

Each MIDAS Object can access all its dependent MIDAS Objects.

It's recommended to <connect> **bold written fields** directly from uiObj to eiMib.

| Interface uiObj | Interface eiMib |
|---|---|
| **objId**<br>**SFString** | **objId**<br>**SFString** |
| **universalObjectClass**<br>**SFNode** | **universalObjectClass**<br>**SFNode** |
| **parentObj**<br>**SFNode** | **parentObj**<br>**SFNode** |
| **dependentMobs**<br>**MFNode** | **dependentMobs**<br>**MFNode** |
| | addDependentMobs<br>MFNode |
| | removeDependentMobs<br>MFNode |
| | assertedDependentMobs<br>MFNode |

### 3.6.4 MOO Changes and Procedures

The following table shows some information, which is **relevant for any SSC Extension**. It tries to explain, how the MOO Changes and Procedures of any SSC Extension are organized.

**Short and sweet**, SSC Base or other parents of any SSC Extension use the miControl interface to initialize or disable the SSC Extensions, the SSC Extensions use the eiSscCore interface to ease the implementation, because it is defined:

- some fields that each SSC Extension **must** provide at the **minimum Interface miControl,**
- some fields that **should** be used at the **external interface eiSscCore** and
- some fields of the **commParamExt** that **will** be handled by the SSC Core prototype.

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

| Interface miControl | Interface eiSscCore | Common Parameter Extension |
|---|---|---|
| basicallyInitialized<br>SFBool | | |
| **commParam**<br>**SFNode** | **commParamIn**<br>**SFNode** | |
| initialized<br>SFNode | initialized<br>SFNode | initialized<br>SFBool |
| **disable**<br>**SFTime** | **disableIn**<br>**SFTime** | |
| | disabled<br>SFBool | |
| **enabledOut**<br>**SFBool** | **enabledOut**<br>**SFBool** | enabled<br>SFBool |
| | currentProcedure<br>SFString | |
| | startProcedure<br>SFBool | |
| | procedureFinished<br>SFBool | |

The following table shows some information, which is **relevant for any MC Extension**. It tries to explain, how the MOO Changes and Procedures of any MC Extension are organized.

**Short and sweet**, MC Base or any other parent of MC Extensions use the miMod interface to initialize or disable MC Extensions, the MC Extensions use the eiMcCore interface to ease the implementation, because it is defined:

- some fields that each MC Extension **must** provide at the **minimum Interface miMod,**
- some fields that **should** be used at the **external interface eiMcCore** and
- some fields of the **modParamExt** that **will** be handled by the MC Core prototype.

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

| Interface miMod | Interface eiMcCore | Module Parameter Extension |
|---|---|---|
| basicallyInitialized<br>SFBool | | |
| **modParam**<br>**SFNode** | **modParamIn**<br>**SFNode** | |
| initialized<br>SFNode | initialized<br>SFNode | initialized<br>SFBool |
| startAttachment<br>SFInt32 | startAttachment<br>SFInt32 | |
| attachmentFinished<br>SFNode | attachmentFinished<br>SFNode | moduleIx<br>SFInt32 |
| **disable**<br>**SFTime** | **disableIn**<br>**SFTime** | |
| | disabled<br>SFBool | |
| **enabledOut**<br>**SFBool** | **enabledOut**<br>**SFBool** | enabled<br>SFBool |
| | currentProcedure<br>SFString | |
| | startProcedure<br>SFBool | |
| | procedureFinished<br>SFBool | |

The following table shows some information, which is **relevant for any MIDAS Object**. It tries to explain, how the MOO Changes and Procedures of any MIDAS Object are organized.

**Short and sweet**, the parent uses the uiObj interface to initialize and/or to attach or disable the MIDAS Object, the MIDAS Object uses the eiMib interface to ease the implementation, because it is defined:

- some fields that each MIDAS Object **must** provide at the **user Interface uiObj** and
- some fields that **should** be used at the **external interface eiMib.**

It's recommended to <connect> **bold written fields** directly from uiObj to eiMib.

| Interface uiObj | Interface eiMib |
|---|---|
| **commParam**<br>**SFNode** | **commParam**<br>**SFNode** |
| initialized<br>SFNode | initialized<br>SFNode |
| **modParam**<br>**SFNode** | **modParam**<br>**SFNode** |
| attached<br>SFNode | attached<br>SFNode |
| **disable**<br>**SFTime** | **disable**<br>**SFTime** |
| | disabled<br>SFBool |
| **enabledOut**<br>**SFBool** | **enabledOut**<br>**SFBool** |
| | currentProcedure<br>SFString |
| | startProcedure<br>SFBool |
| | procedureFinished<br>SFBool |

### 3.6.5 Required SSC Extensions

Each SSC Extension **must** define a "well known ID", which has to be used to access the SSC Extension by any MC Extension or by any MIDAS Object:

| Interface miControl | Interface eiSscCore | Common Parameter Extension |
|---|---|---|
| | | wellKnownId<br>SFString |

Each MC Extension that **requires** an SSC Extension can require and access the SSC Extension with the help of the eiMcCore interface:

| Interface miMod | Interface eiMcCore | Module Parameter Extension |
|---|---|---|
| | requiredSscExtensionsIn<br>MFString | |
| | assertedSscExtensions<br>MFNode | |

Each MIDAS Object that **requires** an SSC Extension can require and access the SSC Extension with the help of the eiMib interface:

| Interface uiObj | Interface eiMib |
|---|---|
| | requiredSscExtensions<br>MFString |
| | assertedSscExtensions<br>MFNode |

### 3.6.6 Required MC Extensions

Each MC Extension **must** define a "well known ID", which has to be used to access the MC Extension by any MIDAS Object:

| Interface miMod | Interface eiMcCore | Module Parameter Extension |
|---|---|---|
| | | wellKnownId<br>SFString |

Each MIDAS Object that **requires** an MC Extension can require and access the SSC Extension with the help of the eiMib interface:

| **Interface uiObj** | **Interface eiMib** |
|---|---|
| | requiredMcExtensions<br>MFString |
| | assertedMcExtensions<br>MFNode |

### 3.6.7 SSC Related Network Sensors and UOC Related SSC Dispatchers

Each SSC Extension may use SSC Related Network Sensors and/or UOC Related SSC Dispatchers.

Both should be made known to the SSC Core Prototype via the eiSscCore Interface. The SSC Core prototype will care for the MOO Changes of those nodes:

| Interface miControl | Interface eiSscCore | Common Parameter Extension |
|---|---|---|
| | networkSensorsIn<br>MFNode | |
| | sscDispatchersIn<br>MFNode | |

### 3.6.8 Object Related and MIB Related Network Sensors

Each MIB may contain MIB Related Network Sensors, however this is something internal to the SMUOS Framework.

Each MIDAS Object may use Object Related Network Sensors. Those network sensors should be made available to the MIB via the eiMib interface. The MIB Core prototype will care for the MOO changes of those nodes.

| **Interface uiObj** | **Interface eiMib** |
|---|---|
| | networkSensors<br>MFNode |

The same is true for bound models, which may use Object Related Network Sensors, too.

Those network sensors should be made available to the MIB Core prototype via the external interface of the Model Prototype (MOP). The MIB Core prototype will care for the MOO changes of those nodes.

## *3.7 A Possible Evolution Path for the SMUOS Framework*

### 3.7.1 Rebase to Core Prototypes ("Pieta") – done

| Software Part | | Features |
|---|---|---|
| **(F)rame** | **Simple Scene Controller** | • **init**<br>• **Support of single-user/multi-user scenes** |
| | **SSC Core** | • **Support of no-state/stateful SSC Extensions**<br>• **Support of synchronous controllers** |
| **(M)odule** | **Module Coordinator** | • **commParam + disable**<br>• **Support of static/dynamic modules** |
| | **MC Core** | • **Support of attacheable/non-attacheable MC Extensions** |
| **(M)odel** | **Model Base** | • **modParam + disable**<br>• **Support of bound models** |
| | **MIDAS Base** | • **commParam + modParam + disable**<br>• **Support of astral/bound MIDAS objects**<br>• **Support of no-state/standard/animated MIDAS objects** |

### 3.7.2 Unbound Models ("Arimathea") – ongoing

| Software Part | | New Features |
|---|---|---|
| **(M)odel** | **Model Base** | • **Support of model containers**<br>• **Support of unbound models** |
| | **MIDAS Base** | • **Support of unbound MIDAS objects** |

### 3.7.3 Handover and Moving Modules – rough ideas

| Software Part | | New Features |
|---|---|---|
| **(F)rame** | **Simple Scene Controller** | • **disable** |
| **(M)odule** | **Module Coordinator** | • **Support of moving modules** |
| **(M)odel** | **MIDAS Base** | • **Support of Handover** |