

## Konzeptpapier - Basismodul

vorläufiger Entwurf (Schritt 0021c)

# 1 Inhalts- und Abbildungsverzeichnis, Tabellen

## Inhaltsverzeichnis

1 Inhalts- und Abbildungsverzeichnis, Tabellen.....	1
2 Das Gesamtkonzept und die Architektur.....	2
3 Begriffe.....	4
4 SRR v0.01.....	6
5 Grundlegende Konzepte.....	7
5.1 Chat, Kommunikation.....	7
5.2 Konsolen-Interface, Rollen und Schlüssel.....	7
5.3 Module und Modelle.....	8
6 Das SRR Framework im Überblick.....	9
7 Abgeleitete Konzepte.....	11
7.1 Namen.....	11
7.1.1 StreamNames.....	11
7.1.2 Modulnamen.....	12
7.1.3 Objekt-IDs.....	12
7.1.4 Key-IDs.....	12
7.1.5 Parameternamen.....	12
7.2 Tracer.....	13
7.2.1 Tracer-Subsysteme und -Instanzen.....	13
7.2.2 Die Tracelevel.....	16
7.3 Communication State.....	17
7.3.1 Der zentrale Controller.....	17
7.3.2 Rollen.....	18
7.3.3 Modulaktivierung und MOC-Rolle – Module Activity.....	19
7.3.4 Empfangen eines Communication State.....	21
7.4 Modul-Management.....	21
7.4.1 Anmeldung/Initialisierung.....	21
7.4.2 Modulregistrierung.....	22
7.5 SRR-Objekte.....	23
7.5.1 Historische Ideensammlung zu SRR-Objekten.....	23
7.6 Das Konsolen-Interface.....	24
7.7 Animierte Objekte.....	25
7.7.1 Korrektur der targetDuration für jede Szeneninstanz.....	27
7.8 Schlüsselbehälter und getragene Schlüssel, Schlösser.....	28
7.9 Avatare und bewegte Objekte.....	29
8 Anhang.....	30
8.1 Nähere Informationen über die SRR-Objekte von SRR v0.01 - Basismodul.....	30
8.1.1 SrrSwitchA - Binary Switch.....	30
8.1.2 SrrDriveA – Carousel Drive.....	30
8.1.3 SrrKeyContainer – Key Container.....	30
8.1.4 SrrLock - Lock.....	31

8.1.5 SrrAvatarContainer – Avatar Container.....	31
8.1.6 SrrMasterAvatarContainer – Master Avatar Container.....	31
8.2 Die Tracer-Subsysteme und -Instanzen der Beispielanlage.....	31
8.3 Tracer-Subsysteme und -Instanzen von SRR v0.01 aufgelistet.....	36
8.4 Setzen des/der Tracelevel(s).....	38
8.5 Beispiele für Trace-Ausgaben.....	40
8.5.1 Entnahme eines Schlüssels aus einem SrrKeyContainer.....	40

## Abbildungsverzeichnis

Abbildung 1: Architektur einer SrrTrains Modellbahnanlage.....	3
Abbildung 2: SRR Framework im Überblick.....	10
Abbildung 3: Tracer-Subsysteme (weiss) und -Instanzen (blau) des SRR Frameworks.....	14
Abbildung 4: Tracer-Subsystem (weiss) und -Instanzen (blau) eines SRR-Objektes.....	15
Abbildung 5: Zuweisung von Rollen (CON und/oder AVA) durch den zentralen Controller.....	18
Abbildung 6: Aktivierung, Deaktivierung von Modulen, Zuordnung der MOC-Rolle.....	20
Abbildung 7: Initialisierung eines Moduls.....	22
Abbildung 8: Modulregistrierung.....	22
Abbildung 9: Extra- und Interpolation des Zustandes von animierten Objekten.....	27
Abbildung 10: Korrektur der targetDuration durch Messen der Round Trip Time.....	28
Abbildung 11: Schlüsselbehälter und getragene Schlüssel.....	29
Abbildung 12: Tracer-Instanzen im Tracer-Subsystem "MyFrame".....	32
Abbildung 13: Tracer-Instanzen im Tracer-Subsystem "MyFirstModule".....	33
Abbildung 14: Tracer-Instanzen im Tracer-Subsystem "MySecondModule".....	34

## Tabellenverzeichnis

Tabelle 1: Tracer-Subsysteme und -Instanzen in den Files von SRR v0.01.....	37
Tabelle 2: Unterschiedliche Tracelevel für unterschiedliche Instanzen des SRR Framework.....	38

## 2 Das Gesamtkonzept und die Architektur

Das Gesamtkonzept *SrrTrains* ist auf der Konzept-Seite (Blog Page)

<http://simulrr.wordpress.com/simul-rr-german> beschrieben und wird in diesem Paper weiter ausgeführt.

Kurz gesagt, es geht darum, mit Hilfe des X3D/VRML-Standards und entsprechender kommerziell verfügbarer Web3D Browser und Collaboration Server, sowie den open source Editoren *X3D-Edit* und *Blender*, virtuelle Multiplayer-Modellbahnanlagen zu bauen und zu betreiben.

Wenn man die Software-Anteile einer solchen Modellbahnanlage nach den Quellen (Autoren) der einzelnen Teile anordnet, ergibt sich folgendes Architekturbild.

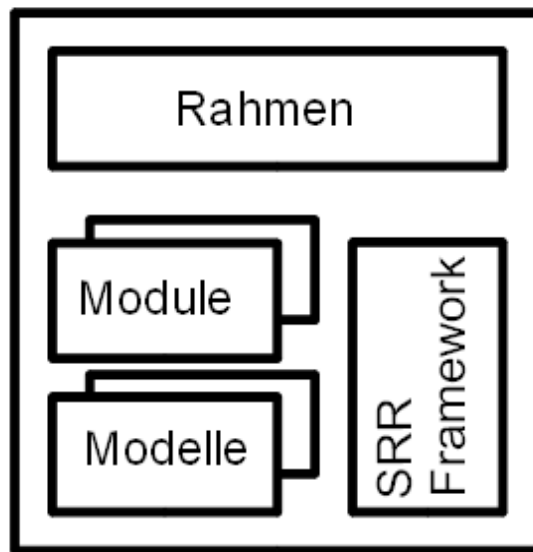


Abbildung 1: Architektur einer SrrTrains Modellbahnanlage

Der **Rahmen** wird vom **Anlagenbetreiber** in einer geeigneten Art und Weise zur Verfügung gestellt. Er enthält das **Hauptfile**, welches geöffnet werden muss, um das Spiel auf der Anlage zu beginnen. Der Anlagenbetreiber muss den Rahmen nicht selbst erstellen, er kann dafür die Dienste eines **Rahmenautors** in Anspruch nehmen.

Eine Anlage enthält mindestens ein **Modul**, kann aber auch mehrere oder viele Module enthalten. Ein Modul errichtet innerhalb des Rahmens ein lokales Koordinatensystem. Die Gleise und Weichen werden innerhalb der Module errichtet und auch die Schienenfahrzeuge werden relativ zu den Modulkoordinatensystemen dargestellt (gerendert).

Anmerkung: In SrrTrains v0.01 wird es noch nicht möglich sein, modulübergreifende Gleise zu errichten.

Ein **Modell** wird z.B. zur Laufzeit nachgeladen, wenn ein Spieler ein Fahrzeug auf die Gleise stellt.

Sowohl **Rahmen-** und **Modulautoren** als auch **Modellautoren** bedienen sich des **SRR Frameworks**, um **Interoperabilität zwischen den Modulen, Modellen und Rahmen** sicher zu stellen. Dadurch können alle SrrTrains Modelle auf allen SrrTrains Modulen betrieben werden und alle SrrTrains Module können von einem SrrTrains Rahmen zu einer Anlage zusammengestellt werden.

Das SRR Framework wird vom **SRR Core Team** zur Verfügung gestellt, welches über das SrrTrains's Blog erreichbar ist ( <http://simulrr.wordpress.com/about> ).

### 3 Begriffe

- SrrTrains** SrrTrains (Simulated **R**ailroad **T**rains) ist ein Gesamtkonzept, welches dazu geeignet ist, ein Biotop von Hobby-Autoren, Hobby-Spielern, aber auch kommerziellen Dienstleistern hervorzubringen.
- Szeneninstanz** In der X3D-Nomenklatur ist eine **SrrTrains Anlage** eine **Szene**. Diese Szene kann von einem Web3D-Browser "geöffnet" werden (d.h. die Szene wird auf einen konkreten Computer "heruntergeladen", als "Szenengraph" in den Speicher geschrieben und dann vom Web3D-Browser "interpretiert", um dem Benutzer des Browsers ein "virtuelles Betreten der Szene" zu ermöglichen).  
Diese konkrete Existenz der Szene in einem konkreten Computer wird als Szeneninstanz bezeichnet.  
Mit Hilfe eines Collaboration Servers können verschiedene Instanzen einer Szene miteinander kommunizieren und sich gegenseitig synchronisieren, um mehreren Benutzern/Spielern eine gemeinsame "virtuelle Multiplayer-Realität" zu bieten – siehe **Network Sensors**.  
Bei der **Initialisierung** muss der **Rahmen** den **SRR Controller** mit einer eindeutigen **sessionId** (einer Ganzzahl) versorgen, die vom SRR Framework verwendet wird, um die Szeneninstanzen voneinander zu unterscheiden.
- Initialisierung** Bevor eine Software ihren Benutzern Dienste fehlerfrei anbieten kann, muss sie initialisiert werden. Dabei werden z.B. Anfangswerte für bestimmte Parameter gesetzt, oder eine Instanz, die soeben erzeugt worden ist, muss sich erst mit anderen Instanzen abgleichen, bevor sie fehlerfrei funktionieren kann.  
Die **SrrTrains Initialisierung** erfolgt stufenweise:  
- Rahmen  
- SRR Controller  
- Module  
- SRR-Objekte
- Anlage** Eine SrrTrains Modellbahnanlage wird vom **Anlagenbetreiber** zur Verfügung gestellt. Das kann z.B. in der Form passieren, dass alle Files des **Rahmens** in einem Webpace veröffentlicht werden. Der Rahmen würde in diesem Falle auf **Module** verweisen, die durch URLs angesprochen werden (sie wurden vorher von ihren **Modulautoren** veröffentlicht). Zur Laufzeit werden **Modelle** (also Lokomotiven und Waggons) nachgeladen, die ebenfalls durch URLs angesprochen werden.  
In der **SRR v0.01 Beispielanlage** bestehen die URLs aus relativen Filenamen und die Beispielanlage wird in Form eines .zip-Files zur lokalen Installation veröffentlicht.
- Rahmen** der Rahmen ist der erste Teil einer SrrTrains Modellbahnanlage, der geladen wird.  
Er enthält die Instanz des **SRR Controllers**, die die gesamte Simulation koordiniert.  
In der Beispielanlage, die mit SRR v0.01 veröffentlicht wird, enthält der Rahmen den Boden, den Hintergrund, die Aussenwand und den Tisch, auf

dem die Module "montiert" sind. Er ist im **Verzeichnis *ExampleFrame*** hinterlegt.

Weiters gehören zum Rahmen die Teile im **Verzeichnis *FrameMain***. Diese Teile könnten von anderen Anlagen fast unverändert übernommen werden, gesetzt den Fall sie möchten die HUDs und die Schlüsselbretter von SRR v0.01 übernehmen.

### ***SRR Controller***

diese zentrale Einrichtung des SRR Framework tritt in jeder Szeneninstanz genau einmal auf. Sie ist zuständig für den ***Kommunikationsstatus (commState)*** der Anlage, d.i. eine Liste der teilnehmenden Szeneninstanzen (sessionIds) mit ihren Eigenschaften sowie eine Liste der registrierten Module.

Der SRR Controller ist in jeder Szeneninstanz für die ***Common Parameter (commParam)*** zuständig. Diese sind allen Modulkoordinatoren und SRR-Objekten zugänglich.

Der SRR Controller ist in den **Dateien *SrrControl.x3d* und *SrrControlNs.x3d*** enthalten.

### ***Network Sensors***

Network Sensors sind ein Konzept des Web3D Consortiums, mit dessen Hilfe unterschiedliche Instanzen ein und derselben Szene miteinander in Kontakt treten können, um den Usern das Erlebnis einer "Multiplayer-Szene" zu bieten.

Die User können einander in Form von Avataren begegnen, es werden aber auch z.B. Animationen zwischen den Szeneninstanzen synchronisiert, um wirklich "eine einzige Realität für alle User" zu bieten.

Sowohl der SRR Controller als auch die SRR-Objekte bedienen sich des Network Sensor Konzeptes.

Die Network Sensors des SRR Frameworks sind in den **Dateien *Srr\*Ns.x3d*** enthalten.

### ***Modul***

ein Modul wird von einem ***Modulautor*** beigesteuert. Im Prinzip enthält ein Modul die Landschaft und ***implizite Modelle*** (die als Teil des Moduls zu betrachten sind) und verweist auf ***statische Modelle*** (die getrennt vom Modul zur Verfügung gestellt, aber zum Ladezeitpunkt des Moduls mitgeladen werden).

Jedes SrrTrains Modul enthält eine Instanz des ***Modulkoordinators*** *SrrModCoord*.

### ***Modulkoordinator***

Jedes Modul und jeder ***Train Mover*** enthalten einen Modulkoordinator. Dieser stellt unter anderem das Interface zwischen SRR Controller und SRR-Objekten her und ist für die ***Modulparameter (modParam)*** des Moduls zuständig. Diese sind allen SRR-Objekten dieses Moduls zugänglich.

Der Modulkoordinator ist in der **Datei *SrrModCoord.x3d*** enthalten.

### ***Modell***

Ein Modell ist ein Objekt, welches ausserhalb eines Moduls zur Verfügung gestellt wird, und welches geladen und einem Modul zugeordnet werden kann.

Modelle sind z.B. Häuser, Gleise und Weichen, Lokomotiven und Waggonen.

### ***Implizite Modelle***

sind als Teil eines Moduls zu betrachten und können nicht getrennt von diesem verwendet werden.

- Statische Modelle** werden als eigenständiges Modell zur Verfügung gestellt, das Modul verweist jedoch in einer statischen Art und Weise auf dieses Modell, sodass es in allen Szeneninstanzen zum Ladezeitpunkt des Moduls zur Verfügung gestellt und mit dem Modul initialisiert wird.
- Dynamische Modelle** (Idee) sind Modelle, die zur Laufzeit dynamisch nachgeladen und wieder entladen werden können, wobei eine gewisse Synchronisierung zwischen allen Szeneninstanzen stattfindet.
- Globale Modelle** (Idee) sind dynamische Modelle, die überdies die Fähigkeit besitzen, während ihrer Lebensdauer die Zuordnung zu einem anderen Modul zu wechseln (z.B. Fahrzeuge).
- SRR-Objekte** stellen die **"eigentliche" SrrTrains Funktionalität** zur Verfügung. Sie helfen dem Modul- und Modellautor, multiplayer-fähige Module und Modelle zu entwickeln. Im allgemeinen besitzt jedes SRR-Objekt einen eigenen Network Sensor, um sich unter den verschiedenen Szeneninstanzen zu synchronisieren. SRR-Objekte erzeugen **keine graphische Repräsentation und keine akustische Repräsentation** des "wirklichen" Objektes, das wird dem Modell- bzw. Modulautor überlassen. Die Beispielanlage liefert allerdings mögliche Vorlagen dafür.
- Train Mover** (Idee) ein Train Mover ist insofern ähnlich einem Modul, als dass er einen Modulkoordinator sowie Gleise und Weichen enthält. Andererseits ist ein Train Mover auch ähnlich einem SRR-Objekt, da er einem Modul zugeordnet ist (sein lokales Koordinatensystem berechnet sich relativ zu einem bestimmten Modulkoordinatensystem). **Train Mover sind noch nicht für SRR v0.01 vorgesehen. Sie sind dazu gedacht, um Drehscheiben, Schiebebühnen, aber z.B. auch Fährschiffe zu realisieren.**

## 4 SRR v0.01

Die erste Version des SRR Framework, mit der auch eine Beispielanlage und einiges an Dokumentation veröffentlicht wird, entsteht aus folgender Zielsetzung:

*Eine Lok (z.B. die „Rocket“) fährt im Kreis, man kann ein und aussteigen und einen Geschwindigkeitsregler betätigen der die Soll-Geschwindigkeit angibt. Es ist ein Multiplayer-Spiel. Man muss den Schlüssel für die Lok erst finden, bevor man den Geschwindigkeitsregler bedienen kann.*

*Es gibt eine Ausweiche. Die Landschaft besteht aus einem hügeligen Terrain. Es gibt ein Haus, dessen Türen man auf- und zumachen kann.*

Diese – erste – Version des SRR Framework wird auf der Blog-Page <http://simulrr.wordpress.com/download-area> veröffentlicht (angestrebter Endtermin ist irgendwann im Jahr 2010, der jeweils letzte Zwischenschritt ist dort bereits verfügbar).

## 5 Grundlegende Konzepte

Die grundlegende Idee des SrrTrains Konzeptes lässt sich durch folgende Wortschöpfung greifbar machen:

### ***Do-it-yourself-multiplayer-virtuelle-Modelleisenbahn***

Alle Konzepte drehen sich also darum, ein Biotop zu entwickeln, in dessen Biosphäre Hobby-Designer, Hobby-Programmierer und Hobby-Eisenbahner in der Lage sind, eigene Pläne zu verwirklichen und ***gemeinsam*** auf einer Anlage zu spielen, auch wenn sie ***geographisch getrennt*** sind.

Ein Nebenziel ist die Verwendung von ***standardisierten*** Sprachen und ***open source*** Tools, um die Breite der Anwendbarkeit zu steigern (X3D/VRML, X3D-Edit, Blender, .....).

Zur Erreichung der ***Multiplayer-Fähigkeit*** sollen Collaboration Server nach dem ***NetworkSensor / EventStreamSensor*** – Konzept des Web3D Konsortiums verwendet werden.

### **5.1 Chat, Kommunikation**

Einen ganz wesentlichen Anteil wird die Kommunikation zwischen den Mitspielern ausmachen, wenn sie mit einer Anlage spielen (oder – ernsthafter – den Eisenbahnbetrieb simulieren).

3D-Chat kann hier nur eine Notlösung sein, selbstverständlich ist die Voice-Konferenz mit Headset oberstes Ziel.

Aber auch Avatare werden eine wichtige Rolle spielen.

### **5.2 Konsolen-Interface, Rollen und Schlüssel**

Eine SrrTrains Anlage soll einen 3-dimensionalen audio-visuellen Eindruck ermöglichen (wozu X3D bestens geeignet ist), es soll aber das ***"gute alte Kommandozeileninterface"*** nicht vernachlässigt werden.

Das führt zu der Forderung, dass jede Szeneninstanz sogenannte ***Rollen*** annehmen kann, wobei es einer Szeneninstanz auch möglich ist, mehrere Rollen gleichzeitig anzunehmen.

Es gibt die

Beobachter – Rolle.....diese Szeneninstanz ermöglicht es dem User, die Anlage zu "betreten".

Konsolen – Rolle.....bietet dem User ein Kommandozeileninterface, mit dem die Anlage beeinflusst werden kann.

Anmerkung: In Kapitel 7.6 erfährt man mehr über das ***Konsolen-Interface***, die Zuteilung von Rollen zu Szeneninstanzen wird in Kapitel 7.3 als Teil der Beschreibung des ***Communication State*** besprochen.

Eine weitere Möglichkeit, einem User Eigenschaften zuzuweisen, sind ***Schlüssel***.

Wenn ein User einen Schlüssel trägt oder ihn in ein Schloss steckt, kann er Schlösser entsperren und somit z.B. die Rolle eines Lokführers oder Weichenstellers einnehmen.

Anmerkung: Mehr über ***Schlüsselbehälter, Schlösser und getragene Schlüssel*** erfährt man in Kapitel 7.8.

### 5.3 Module und Modelle

Es soll möglich sein, Module und Modelle unterschiedlicher Autoren in einer SrrTrains Anlage zu verwenden. Das führt uns zur Notwendigkeit des **SRR Frameworks** und insbesondere der **SRR-Objekte**, mit deren Hilfe Module und Modelle erstellt werden können. SRR-Objekte werden durch eine innerhalb ihres Moduls eindeutige Objekt-ID (eine Zeichenkette) beschrieben.

Anmerkung: das SRR Framework ist überblicksmäßig in Kapitel 6 beschrieben, Kapitel 7 beschäftigt sich dann mit den Konzepten im Detail:

- mehr über SRR-Objekte im allgemeinen erfährt man in Kapitel 7.5 sowie im Kapitel 7.1 über die Objekt-IDs, mit denen SRR-Objekte identifiziert werden.
- auf animierte Objekte (Modelle) geht Kapitel 7.7 näher ein und um Gleise und Weichen kümmert sich Kapitel Fehler: Referenz nicht gefunden.
- ein spezieller Objekt-Typ ist der sog. Avatar-Container, der in Kapitel 7.9 behandelt wird.
- um Module und Modelle auch zur Laufzeit nachladen zu können, benötigt man ein *Modul-Management* (siehe Kapitel 7.4 ), bzw. ein *Modell-Management* (siehe Kapitel Fehler: Referenz nicht gefunden).
- auch Module und Modelle benötigen einen eindeutigen Namen bzw. eine Objekt-ID, worüber man ebenfalls in Kapitel 7.1 lesen kann.

Wie man Fehler in Modulen und Modellen sucht, erfährt man in Kapitel 7.2, welches sich mit dem **Tracer** beschäftigt.

## 6 Das SRR Framework im Überblick

Abbildung 2 zeigt, wie Rahmen- und Modulautoren die Elemente des SRR Framework nutzen können, um mit Hilfe des Network Sensor Konzeptes multiplayer-fähige Anlagen zu entwickeln.

Das SRR Framework besteht aus einer Sammlung von X3D Protoypen, welche normalerweise keine renderbare Graphik und auch keine Töne/Geräusche erzeugen, sondern dem Modul- und dem Modellautor dabei helfen, multiplayer-fähige Objekte zu erstellen.

Natürlich ist das SRR Framework auf den Bau von Modellbahnanlagen spezialisiert, man könnte aber mit ähnlichen Konzepten auch andere Anwendungen versorgen.

Das SRR Framework stellt einige zentrale Dienste zur Verfügung, die mehr oder weniger im **SRR Controller** zusammengefasst sind.

Jedes SrrTrains Modul benötigt einen **Modulkoordinator**, welcher im wesentlichen die Anknüpfung an den SRR Controller und die Koordination der SRR-Objekte innerhalb des Moduls übernimmt.

Die **SRR-Objekte** realisieren die eigentliche Funktion, die dann auf der Anlage für den Spieler sichtbar wird (z.B. die multiplayer-fähige Reaktion eines Schalters auf einen Mausclick).

Es ist möglich, eigene SRR-Objekte zu erstellen, sogenannte **3<sup>rd</sup> party SRR-Objekte**.

Man sieht in Abbildung 2, wie der Rahmen, nachdem er den SRR Controller initialisiert hat, die **Common Parameters (commParam)** entgegennimmt, um diese an die Module weiterzuleiten.

Die Module senden die commParam an ihre Modulkoordinatoren, um diese zu initialisieren (dabei erhält das Modul endgültig seinen Namen).

Die Module müssen die **Modulparameter (modParam)** vom Modulkoordinator entgegennehmen und diese an alle statisch eingebundenen SRR-Objekte (also an die impliziten und statischen Modelle) weitergeben, um sie zu initialisieren.

Mit Hilfe der commParam und der modParam sind alle Elemente des SRR Framework in der Lage, innerhalb einer Szeneninstanz miteinander zu kommunizieren. Der SRR Controller und die SRR-Objekte besitzen überdies Network Sensors, mit denen sich die Instanzen in allen Szeneninstanzen synchronisieren können.

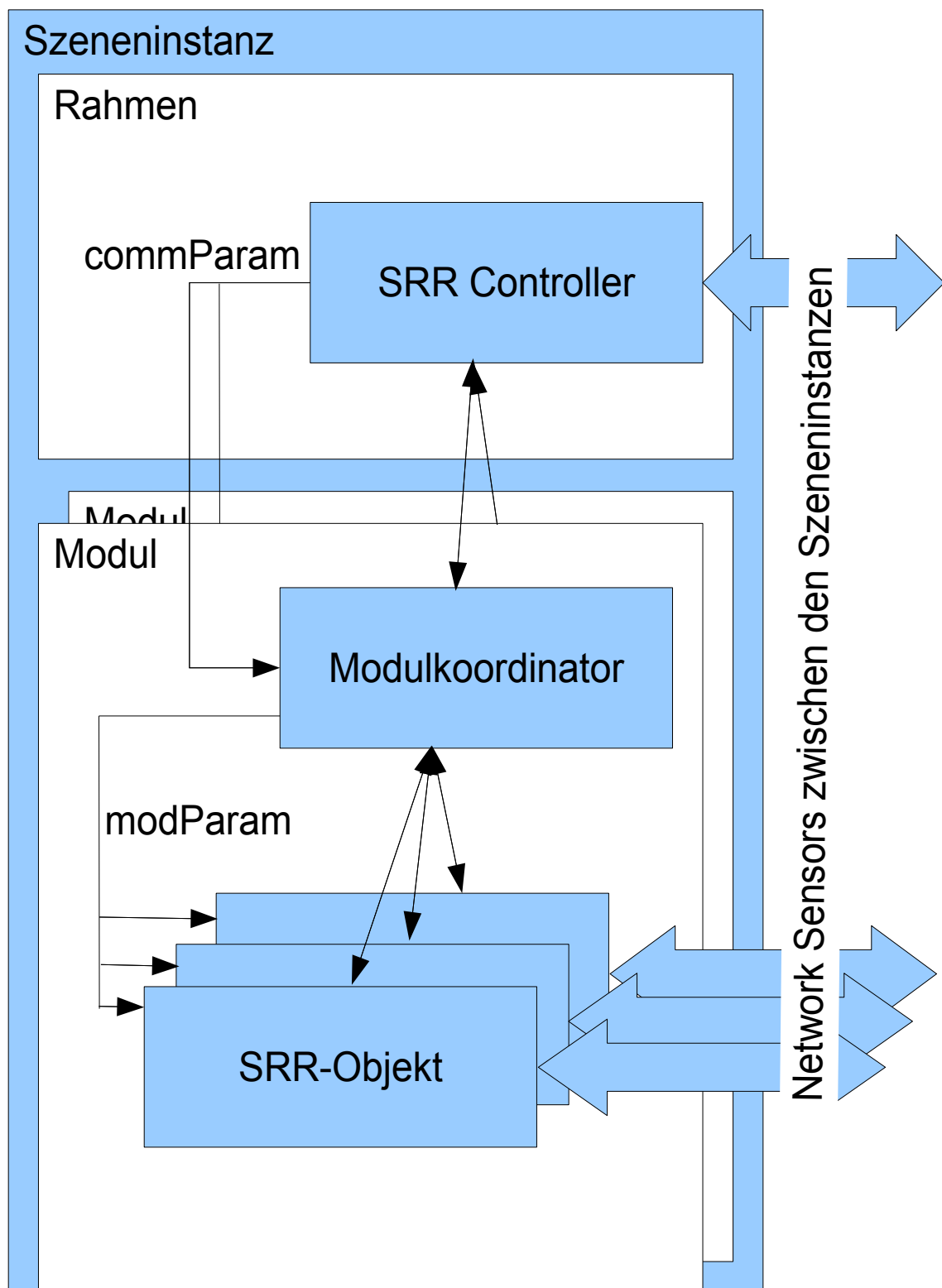


Abbildung 2: SRR Framework im Überblick

## 7 Abgeleitete Konzepte

### 7.1 Namen

Und der Herr führte Adam die Tiere zu, auf dass er ihnen Namen gebe.

So einfach ist das bei uns Programmierern nicht! Da muss ein System her, ein Schema.

Na gut, ein bisschen System steckt auch hinter SrrTrains, obwohl es sich bisher sehr chaotisch entwickelt hat.

Also, zuerst waren da einmal die Module, die hatten Namen.

Dann kamen SRR-Objekte, die man direkt in die Module einbaute, die bekamen eine Objekt-ID, sodass "moduleName+objId" in jeder Szene eine eindeutige Objekt-Bezeichnung sein sollte.

Mittendrin entwickelten sich die Schlüssel, die bekamen ein Dreigestirn als globale Key-ID, nämlich "moduleName+objId" des KeyContainers, in dem sie geboren werden plus eine lokale Key-ID.

Um SRR-Objekte über das Konsolen-Interface zu steuern, benötigt man Parameter und – richtig – auch Parameter haben Namen.

Da das "StationHouse" in eine eigene Datei ausgelagert war, also der Vorläufer der statischen Modelle sein wird, ergab es sich, diesem Haus den Namen **StationHouse** zu geben und die objIds aller im Haus enthaltenen SRR-Objekte mit **StationHouse.** einzuleiten, also **StationHouse.Door** und **StationHouse.Door.Lock**. Das Schlüsselbrett des Stationsgebäudes hat wie zum Trotz die Objekt-ID **StationKeyHooks**, obwohl es sich auch im Haus befindet.

Die Sache mit dem Punkt als "Quasi-Trennzeichen" hat mir so gefallen, dass jetzt Containment-Beziehungen generell durch diese Methode angezeigt werden:

Hinweis: Wenn man will, dass der SrrTracer und das Konsolen-Interface im Zusammenhang mit den eigenen Modellen gut funktionieren, sollte man sich an folgende Regeln halten:

- ein Modell bekommt eine Objekt-ID <objId>
- alle SRR-Objekte des Modells sollten ihre Objekt-ID von <objId> ableiten, also <objIdSrrObject>=<objIdModel>.<SrrObjectLocalName>
- Dann kann man ausserdem direkt im Modell einen SrrTracer instanziiieren und braucht ihm bloss die <objIdModel> übergeben und mit modParam zu initialisieren.

#### 7.1.1 StreamNames

SRR verwendet <EventStreamSensor>- bzw. <NetworkSensor>-Knoten, die die Kommunikation zwischen den Szeneninstanzen bewerkstelligen.

Jeder dieser Sensoren hat eine ID, einen sogenannten *streamName* (bzw. *networkSensorId*), welche ein String ist und den Sensor in der Netzwerk-Kommunikation eindeutig kennzeichnet.

Um nun dem SRR-Benützer zu ermöglichen, auch eigene <EventStreamSensor>/<NetworkSensor>-Knoten zu verwenden (z.B. für notwendige Funktionen, die in SRR – noch – nicht verfügbar sind), folgen die streamNames/networkSensorIds des SRR Framework einem gewissen Schema.

- Alle SRR streamNames/networkSensorIds beginnen mit **Srr**
  - Der SRR Controller verwendet einen Sensor mit **SrrControl**
  - Der Master Avatar Container verwendet einen mit **Srr-MasterAvatarContainer**
  - SRR-Objekte verwenden je einen Sensor mit **Srr-<moduleName>-<objId>**
  - Animierte SRR-Objekte zusätzlich einen mit **Srr-<moduleName>-<objId>-Anim**

### 7.1.2 Modulnamen

Modulnamen sind Strings, die aus den Zeichen 'A'-'Z', 'a'-'z', '0'-'9', '\_' und '.' bestehen\*).

Anmerkung: es ist für spätere Versionen von SrrTrains die Idee vorhanden, sogenannte *Train Mover* zu realisieren (Drehscheiben, Schiebepöhlen), diese sollten ihren Namen vom Namen des Elternmoduls ableiten: **<trainMoverName>=<parentModuleName>.<tmLocalName>**

\*) zur Zeit überprüft SRR NICHT, ob der User einen Namen verwendet, der dieser Konvention entspricht. Um die klaglose Funktion des Konsolen-Interfaces und des Tracers zu gewährleisten, wird jedoch strengstens empfohlen, sich an diese Konvention zu halten. Auch wird empfohlen, den Punkt '.' nicht als "normales" Zeichen zu verwenden sondern bloss, um die Containment-Beziehungen zwischen Modulen und Train Movern anzuzeigen (zur Zeit also gar nicht).

### 7.1.3 Objekt-IDs

Objekt-IDs sind Strings, die aus den Zeichen 'A'-'Z', 'a'-'z', '0'-'9', '\_' und '.' bestehen\*).

Ein Objekt, das in einem Eltern-Objekt enthalten ist, sollte einen Teil seiner <objId> vom Eltern-Objekt erben, also **<objId>=<parentObjId>.<objLocalId>**

\*) zur Zeit überprüft SRR NICHT, ob der User eine Objekt-ID verwendet, die dieser Konvention entspricht. Um die klaglose Funktion des Konsolen-Interfaces und des Tracers zu gewährleisten, wird jedoch strengstens empfohlen, sich an diese Konvention zu halten. Auch wird empfohlen, den Punkt '.' nicht als "normales" Zeichen zu verwenden sondern bloss, um die Containment-Beziehungen zwischen Modellen und SRR-Objekten anzuzeigen.

### 7.1.4 Key-IDs

Key-IDs werden beim Erzeugen von Schlüsseln automatisch generiert.

Der Modul-/Modellautor gibt im Feld *initialKeys* eine Liste von *lokalen Key-IDs* an. Bei der Initialisierung des *SrrKeyContainer*-Objektes werden Schlüssel mit der *globalen Key-ID* **<moduleName>.<objId>.<localKeyId>** erzeugt (<moduleName> und <objId> sind die Eigenschaften des SrrKeyContainer-Objektes).

Ab der Erzeugung des Schlüssels wird nur mehr die globale Key-ID verwendet, um Schlüssel zu verschieben oder den "verschlossen"-Status von Schlössern zu beeinflussen.

### 7.1.5 Parameternamen

Parameternamen sind Strings, die aus den Zeichen 'A'-'Z', 'a'-'z', '0'-'9', '\_' und '.' bestehen.

Wie erwartet, wird nicht geprüft, ob der User (Autor eines 3<sup>rd</sup> party SRR-Objektes) sich an diese

Konvention hält, es wird aber wärmstens empfohlen, sich daran zu halten.

## 7.2 Tracer

Da eine SrrTrains-Anlage typischerweise aus Elementen bestehen wird, die von vielen Quellen (Autoren) stammen, wird es nötig sein, brauchbare Hilfsmittel zur Verfügung zu stellen, um Fehler in einer Anlage aufzuspüren und dem Autor Informationen zukommen zu lassen, mit denen er den Fehler konkret korrigieren kann.

**Der SRR Tracer** ist nun ein *nützliches kleines Hilfsmittel, um*

- SrrTrains Software zu debuggen
- SrrTrains Software zu erforschen
- SrrTrains Software zu dokumentieren

Immer, wenn die Software bestimmte Punkte erreicht, sogenannte **Tracepunkte**, wird eine Trace-Information ausgegeben (unter Verwendung der Methoden `Browser.print()` bzw. `Browser.println()`).

Jeder Tracepunkt ist einem bestimmten **Trace-Level** zugeordnet. Nur wenn der aktuelle Tracelevel größer oder gleich dem Level des Tracepunktes ist, wird die Information tatsächlich ausgegeben.

Die Tracer-Ausgaben sind für Menschen lesbar (in Englischer Sprache), sie sind aber auch darauf vorbereitet, von Diagnose-Tools gelesen, interpretiert und graphisch aufbereitet zu werden.

Anmerkung: Beispiele für Tracer-Ausgaben finden sich im Anhang 8.5.

Das SRR Framework verwendet durchgehend dieses Tracer-Konzept, aber auch der Rahmen der Beispielanlage und die Module der Beispielanlage machen Gebrauch davon.

### 7.2.1 Tracer-Subsysteme und -Instanzen

Die Ausgaben der Tracepunkte des SRR Frameworks sind in **Subsysteme** und **Instanzen** gegliedert. Weiters kann jeder Trace-Ausgabe entnommen werden, in welcher **Szeneninstanz** der Tracepunkt erreicht worden ist.

#### **SRR Controller:**

Die SRR Controller Software beherbergt die Instanzen:

- **CommControl** (Communication Controller)
- **TrainControl** (Train/Vehicle Controller)
- **SrrControl** (lokaler SRR Controller)

SrrControl bietet seinem User (dem Rahmen) das **User Interface uiControl** (siehe auch Abbildung 3), welches z.B. von Interesse ist, wenn man einen Rahmen entwickelt.

Die zentralen Controller CommControl ("Communication Control") und TrainControl ("Train/Vehicle Control") haben kein direktes Interface zu selbstgeschriebener Software und sind deshalb nur von Interesse, wenn man einen Fehler im SRR Framework vermutet oder wenn man näheres über die Funktionsweise des SRR Framework in Erfahrung bringen möchte.

#### **Modulkoordinator:**

Die Modulkoordinator Software erzeugt genau eine Instanz pro geladenem und initialisierten

Modul. Der Name der Instanz ergibt sich aus *<moduleName>.Coord*.

Die Modulkoordinator-Instanzen(in diesem Beispiel *second.Coord* und *first.Coord*) bieten ihrem User (dem Modul) das **User Interface *uiMod*** (siehe auch Abbildung 3) und sind z.B. von Interesse, wenn man ein eigenes Modul entwickelt und überprüfen möchte, ob die gewünschte Information auch tatsächlich beim Modulkoordinator ankommt. Auch wenn man ein eigenes 3<sup>rd</sup> party SRR-Objekt entwickelt, kann es von Interesse sein, die Reaktionen des Modulkoordinators auf das Objekt zu tracen.

Anmerkung: eine komplette Liste aller Tracer-Subsysteme und -Instanzen von SRR v0.01 findet sich in Anhang 8.3

Abbildung 3 zeigt ein Beispiel, in dem zwei Szeneninstanzen aktiv sind, von der eine die Rolle des zentralen Controllers übernommen hat.

Anmerkung: Näheres über den zentralen Controller findet sich in Kapitel 7.3.

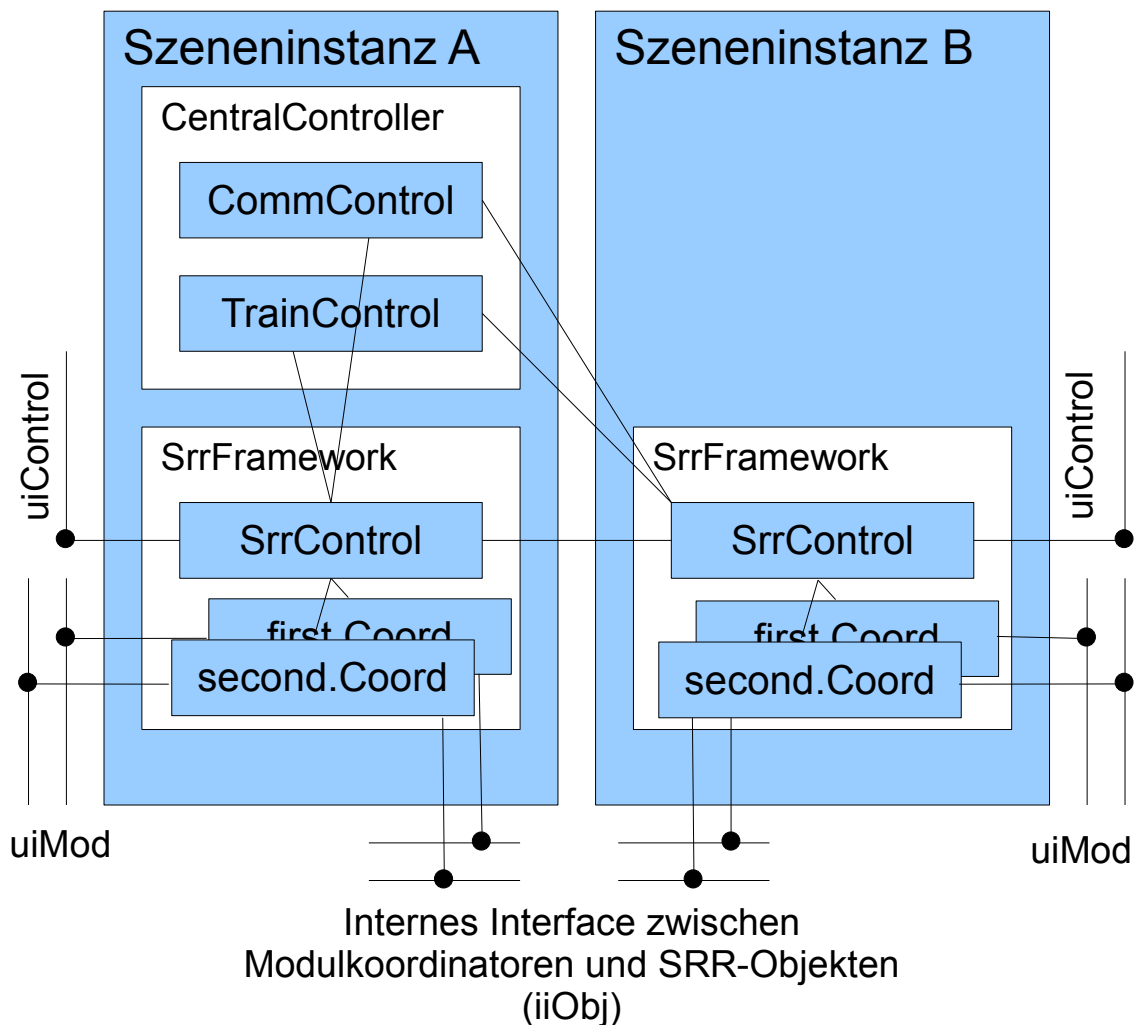


Abbildung 3: Tracer-Subsysteme (weiss) und -Instanzen (blau) des SRR Frameworks

**SRR-Objekte:**

Neben SRR Controller und Modulkoordinator verwenden auch die SRR-Objekte den Tracer.

Jeder SRR-Objekt-Typ ist ein eigenes Tracer-Subsystem und jedes daraus erzeugte Objekt bietet zwei Tracer-Instanzen Platz:

- dem Objekt-Controller und
- dem Objekt-MOC\*)

Die Objekt-MOC Instanz ist nur in der Szeneninstanz aktiv, welche die MOC Rolle für das Modul innehat, in dem sich das Objekt befindet.

Anmerkung: Näheres über die MOC Rolle findet sich in Kapitel 7.3.

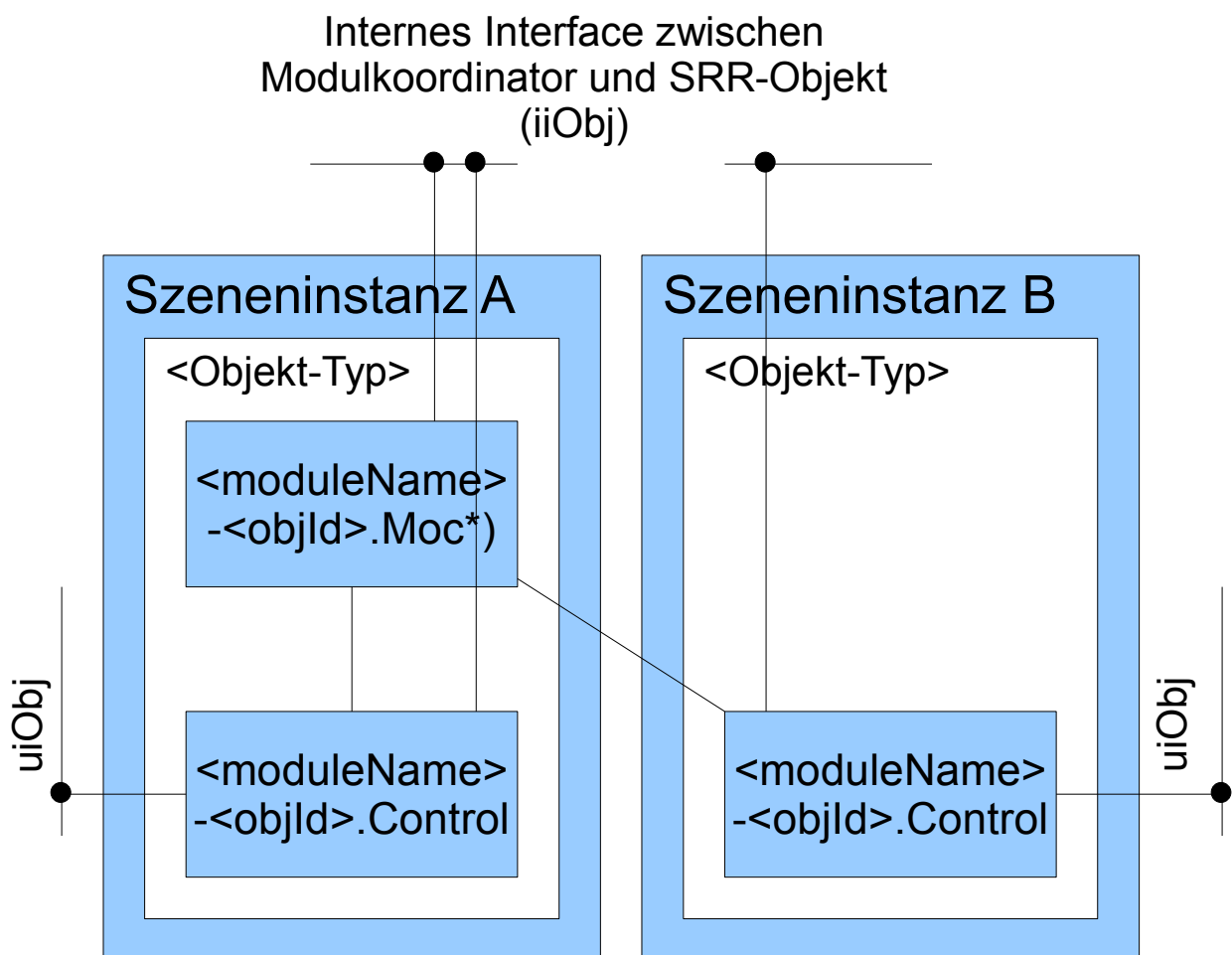


Abbildung 4: Tracer-Subsystem (weiss) und -Instanzen (blau) eines SRR-Objektes

Die Objekt-MOC Instanz\*) übernimmt zentrale Aufgaben (sie ist für den globalen Status des Objektes verantwortlich) und existiert genau einmal pro aktivem Objekt.

Die Objekt-Controller Instanz bietet ihrem User (dem Modul bzw. Modell) das **User Interface** *uiObj* und existiert in jeder Szeneninstanz, in der das Objekt initialisiert worden ist, genau einmal. Sie kann getraced werden um Module bzw. Modelle zu debuggen (zu überprüfen, ob sie die SRR-

Objekte richtig verwenden).

- \*) Avatar Container Objekte haben keine "Moc"-Instanz sondern eine "CC"-Instanz (Central Controller), die gemeinsam mit dem zentralen Controller in dessen Szeneninstanz läuft. Gleis- und Weichenobjekte haben zur Zeit überhaupt nur eine "Controller"-Instanz.

Anmerkung: eine komplette Liste aller Tracer-Subsysteme und -Instanzen findet sich in Anhang 8.3.

## 7.2.2 Die Tracelevel

Es gibt vier Tracelevel, 0 bis 3, die wie folgt definiert sind.

Im "normalen" Spielbetrieb ist generell Tracelevel 1 aktiviert (d.h. Tracepunkte mit level 0 und 1 erzeugen Ausgaben auf der Browser Konsole).

### Tracelevel 0: fataler Fehler

Wenn der Tracer nicht verfügbar ist, und dennoch ein fataler Fehler auftritt, macht die Software selbst eine Ausgabe auf die Browser-Konsole unter direkter Verwendung der Methoden `Browser.print()` bzw. `Browser.println()`(Format nicht eindeutig definiert).

Diese Ausgaben können nicht deaktiviert werden.

Es gibt auch – seltene - nicht-deaktivierbare Ausgaben, die keinen fatalen Fehler bedeuten, sondern als Information für den Spieler gedacht sind.

### Tracelevel 1: Fehler

Es sind einige wenige Fehler mit festem Text vordefiniert.

Die Software meldet dem Tracer nur eine Ganzzahl, die diesen Fehler beschreibt.

In der Trace-Ausgabe sind keine näheren Informationen über den Ort und die Zeit des Fehlers verfügbar, allein aus dem Fehlertext sollte für den Endbenutzer klar sein, was er zu tun hat um den Fehler zu beheben (z.B. einen Collaboration Server zur Verfügung stellen, wenn multiuser Betrieb gewünscht ist).

### Tracelevel 2: Info

Lokalisieren von Fehlern.

Wenn der Benutzer z.B. einen neuen Rahmen entwickelt, sollten die level 2 Tracepunkte der `SrrControl`-Instanz dabei helfen, eine eventuell falsche Verwendung des SRR Framework aufzuspüren und somit Fehler im zu entwickelnden Rahmen zu detektieren.

Es gibt folgende Arten von Infos:

- free text: beliebige Information vom Programmierer an den Benutzer
- sending/receiving messages: Übertragung von Information zwischen Szeneninstanzen
- sending/receiving events: Übertragung von Information innerhalb einer Szeneninstanz aber zwischen verschiedenen Software-Instanzen, auch über User Interfaces
- starting/stopping/expiry of timers
- setting a new state: ein interner (lokaler) State einer Instanz hat sich geändert
- starting/stopping of an instance: eine Instanz wird kreiert bzw. beendet

### Tracelevel 3: Debug

Wenn man einen Fehler in einer bestimmten Software-Instanz vermutet, wählt man Tracelevel 3 für diese Instanz und Tracelevel 2 für ausgesuchte andere Instanzen, wiederholt den Fehler und sendet das Trace-Ergebnis an den Autor der fehlerhaften Software-Instanz, mit der Bitte den Fehler zu bestätigen und gegebenenfalls zu beheben.

Es gibt folgende Arten von Debug-Infos:

- free text: beliebige Information vom Programmierer an sich selbst

## **7.3 Communication State**

Beim gemeinsamen Spiel auf einer SrrTrains Anlage müssen die einzelnen Computer den Überblick über die Kommunikation behalten.

Dazu ist es z.B. nötig, dass die Szeneninstanzen stets wissen, welche anderen Szeneninstanzen an dem Spiel momentan teilnehmen.

Alle diese Informationen sind im sogenannten *Communication State* enthalten, der stets aktuell gehalten und an alle Szeneninstanzen verteilt wird.

Der *Communication State* enthält

- eine Liste der aktuell registrierten Module (der Modul-Index *moduleIx* eines Moduls ist also in allen Szeneninstanzen identisch)
- eine Liste aller eingeloggten User (genau genommen aller Szeneninstanzen, die den SrrController initialisiert haben) mit ihren Eigenschaften
  - o sessionId
  - o Controller ja/nein
  - o aktivierte Module
  - o MOC Rollen
  - o selektierte Rollen

Die *sessionId* ist eine Ganzzahl, die der Rahmen dem SrrController bei der Initialisierung mitteilt, und die dazu dient, die einzelnen Szeneninstanzen ("Sessions") auseinander zu halten. Im Falle eines Einzelbetriebes (*single-user-mode*) ist *sessionId=-1*.

Wenn ein *Communication State* verteilt wird, ist immer genau eine Szeneninstanz der *Controller* (siehe Kapitel 7.3.1), es sei denn, es gibt überhaupt keine aktive Szeneninstanz.

Auch die *Rollen* werden zentral zugeteilt (siehe Kapitel 7.3.2). Das hat nur den Grund, dass alle Szeneninstanzen erfahren sollen, wer welche Rolle hat, der zentrale Controller *CommControl* teilt also einfach nur die Rollen zu, die der Rahmen der Szeneninstanz angefordert hat, ohne wenn und aber.

Über die *Modulaktivierung und die MOC-Rolle* kann man in Kapitel 7.3.3 mehr lesen.

### **7.3.1 Der zentrale Controller**

Für den *Communication State* muss, wie für jeden Zustand, eine einzige Instanz zuständig sein.

Diese Instanz wird in SRR der **zentrale Controller** genannt, was aber ein wenig ungenau ist, denn eigentlich gibt es zwei zentrale Controller,

- den **Communication Controller** und
- den **Train/Vehicle Controller** (dieser befindet sich noch im Versuchsstadium)

Beide zentralen Controller laufen in einer einzigen Szeneninstanz, von der man dann auch sagt, dass sie die **Controller-Rolle** innehat.

Wenn die erste Szeneninstanz eines Spiels startet (den SRR Controller initialisiert), versucht SrrControl einen sogenannten *Access Request* an den Communication Controller zu senden (sie weiss ja noch nicht, dass es keinen Controller gibt, da sie noch keinen Communication State erhalten hat).

Da sie keine Antwort erhält, macht sie sich selbst zum Controller und sendet einen ersten Communication State in die Runde (die ja noch ziemlich klein ist).

In weiterer Folge gibt es stets einen Controller, denn wenn gerade die Szeneninstanz das Spiel beendet, die die Controller-Rolle innehat, dann kann eine andere Instanz übernehmen.

Es gibt sogar die Möglichkeit, dass der Rahmen einer Szeneninstanz beim SRR Controller die Controller-Rolle anfordert (welche in weiterer Folge vom Communication Controller zugewiesen wird).

**Das alles sollte im normalen Spielbetrieb unwesentlich sein** (welche Szeneninstanz die Controller-Rolle innehat, ist für den Spieler unwesentlich, normalerweise **auch für den Modulautor und Modellautor**), kann aber eine Hilfestellung sein, wenn man den Tracer nur auf einem bestimmten Computer verwenden kann und an Traces von CommControl oder TrainControl interessiert ist.

### 7.3.2 Rollen

Rollen werden, wie gesagt, vom Communication Controller auf Antrag zentral zugewiesen, haben aber zur Zeit nur lokale Auswirkungen.

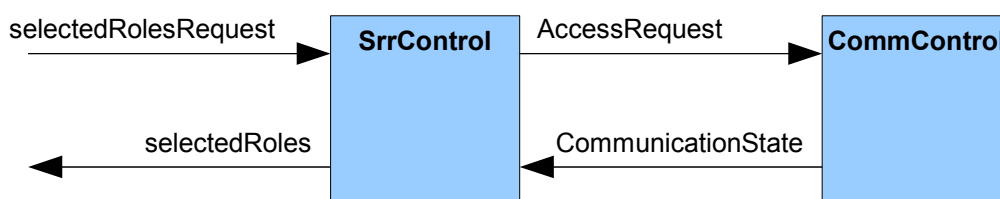


Abbildung 5: Zuweisung von Rollen (CON und/oder AVA) durch den zentralen Controller

Eine Szeneninstanz, die die **Beobachter-Rolle** (AVA-Rolle) nicht innehat, kann keine Module aktivieren.

Das hat seine Begründung darin, dass die AVA-Rolle die Voraussetzung für ein virtuelles Betreten der Module und des Rahmens sein sollte.

Anmerkung: Modulaktivierung wird in Kapitel 7.3.3 besprochen.

Eine Szeneninstanz, die die **Konsolen-Rolle** (CON-Rolle) nicht innehat, wird als Antwort auf ein Konsolen-Kommando stets eine Fehlermeldung erhalten, das Konsolen-Interface steht ihr also nicht zur Verfügung.

Anmerkung: Das Konsolen-Interface wird in Kapitel 7.6 behandelt.

### 7.3.3 Modulaktivierung und MOC-Rolle – Module Activity

Eine Überlegung, warum man eine Modellbahnanlage aus Modulen *zusammensetzt*, ist, dass man die Möglichkeit hat, Module von unterschiedlichen Autoren gemeinsam zu verwenden.

Die umgekehrte Überlegung, nämlich eine Modellbahnanlage in Module zu *zerlegen*, rührt daher, dass man sich überlegt, bei grossen Anlagen Probleme mit der Rechnerleistung zu bekommen, wenn stets alle Teile der Anlage aktiv sind.

Man wird also versuchen, möglichst viele Animationen, die man momentan nicht benötigt, nicht zu berechnen, eventuell sogar Teile des Szenengraphen wieder aus dem Speicher zu löschen oder gar nicht erst zu laden.

Eine Möglichkeit ist es, **Module zu aktivieren und zu deaktivieren**, je nachdem auf welchem Modul man sich gerade befindet. Der <ProximitySensor>-Knoten von X3D bietet hier einen Ansatz, der versuchenswert ist.

In weiterer Folge gibt es die Überlegung, dass bei Simulationen und Animationen gewisse Berechnungen sinnvollerweise nur in einer Szeneninstanz durchgeführt werden können (für einen Zustand, einen sog. "state", kann immer nur einer zuständig sein).

Da wir dem Benutzer des SRR Frameworks nicht aufbürden wollen, jedem SRR-Objekt stets mitzuteilen, in welcher Szeneninstanz es die Berechnungen durchführen soll und in welcher nicht, übernimmt SRR diese Aufgabe selbstständig.

Natürlich würde sich die Controller-Rolle anbieten, auch diese Arbeit zu übernehmen, aber wir wollen nicht einen einzigen Rechner mit der gesamten Arbeit belasten, deshalb wurde die **Rolle des Modul-Controllers, kurz MOC** erfunden.

Jedes Modul hat einen MOC. Unter allen Szeneninstanzen, in denen dieses Modul aktiv ist, ist genau eine der MOC.

Damit gibt es zumindest die Chance, dass verschiedene Szeneninstanzen verschiedene Module zuerst aktivieren (und somit die MOC-Rolle bekommen) und somit die Last ein wenig verteilt wird.

Um Module zu aktivieren und zu deaktivieren, stehen zur Zeit folgende Interfaces zur Verfügung:

- sende ein **activateRequest=true** an den Modulkoordinator, um das Modul zu aktivieren (falls es nicht schon aktiviert war)
- sende ein **activatedModulesRequest=<listOfModules>** an den SRR Controller, um genau diese Liste an Modulen zu aktivieren (falls sie nicht schon aktiv waren) und die anderen Module zu deaktivieren (falls sie noch aktiv waren)

Als Antwort darauf wird der *Communication Controller* eventuell die MOC-Rollen umverteilen und den *Communication State* aktualisieren, welcher auch die **Module Activity** enthält. Diese ist, einfach gesagt, eine Liste, die für jedes registrierte Modul

- die sessionId des MOC (wenn es einen MOC gibt),

- eine Liste aller sessionIds, in denen dieses Modul aktiv ist, enthält.

Anmerkung: Man sieht, dass auch eine Szeneninstanz der MOC sein könnte, in der das Modul gar nicht aktiv ist.

Der SRR Controller wird diese Mitteilung auswerten und dem Rahmen eine aktualisierte Liste der aktiven Module melden, sowie eine Liste der MOC-Rollen (dieser Szeneninstanz).

Über den Modulkoordinator wird eine Rückmeldung an das Modul erfolgen (*activated* ist ein Bit-Feld, welches angibt, ob das Modul frisch aktiviert oder deaktiviert worden ist oder ob es die MOC-Rolle übernommen/abgegeben hat).

Diese Information wird über die Felder takeMOC, grantMOC, activate und deactivate der Modulparameter (modParam) auch an interessierte SRR-Objekte weitergegeben.

Einige SRR-Objekte interessieren sich für die Liste der sessionIds, in denen dieses Modul momentan aktiv ist. Auch diese Information wird über ein Feld der modParam verteilt.

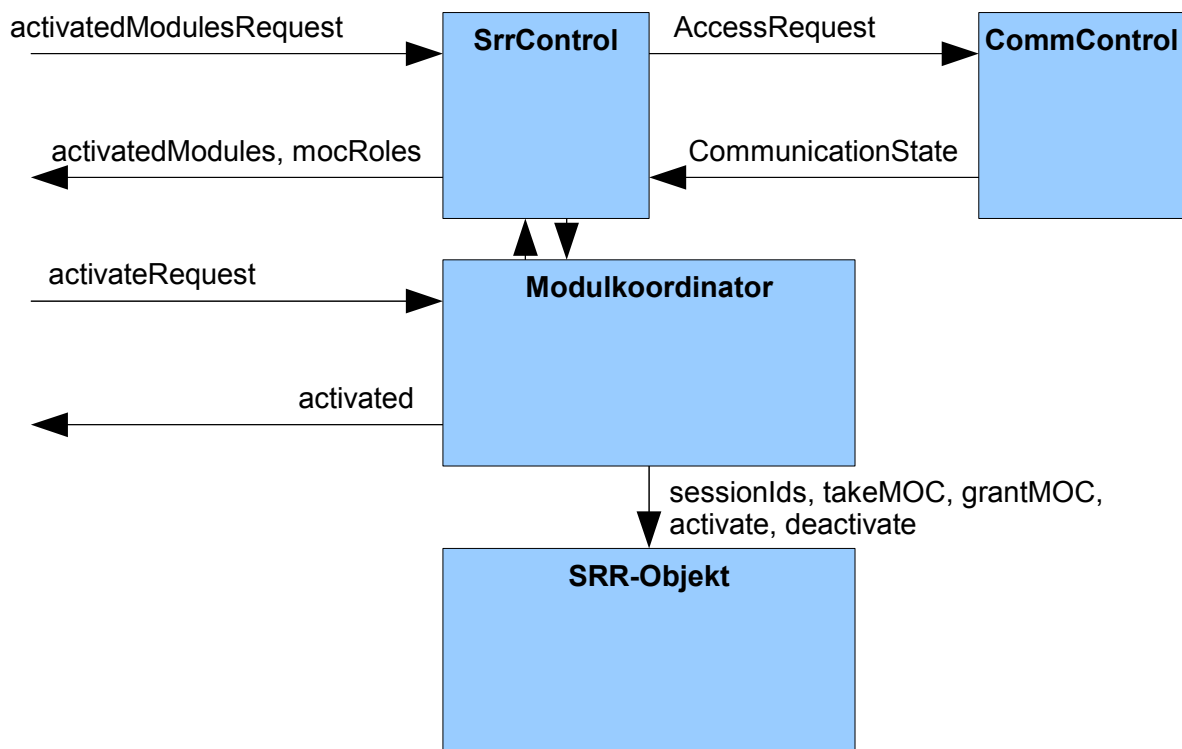


Abbildung 6: Aktivierung, Deaktivierung von Modulen, Zuordnung der MOC-Rolle

Wie schon vom Leser erahnt, stellt das SRR Framework eben nur eine Rahmenfunktionalität zur Verfügung. Was durch die Modulaktivierung/-deaktivierung und das Erhalten/Abgeben der MOC-Rolle tatsächlich ausgelöst wird, liegt im Ermessen des Modulautors und der SRR-Objekte.

Die Module der Beispielanlage haben zum Beispiel je einen `<ProximitySensor>`, der bei Betreten des Moduls einen **activateRequest** an den Modulkoordinator sendet. Der **activatedModulesRequest**

wird vom Control HUD versorgt. Die Ausgabe *activated* wird verwendet, um die Lichtquellen des Moduls ein- und auszuschalten.

### 7.3.4 Empfangen eines Communication State

Abbildung 5 und Abbildung 6 geben bereits einen guten Eindruck, was der SRR Controller tut, wenn er einen *Communication State* erhält.

Generell informiert er den Rahmen und über die Modulkoordinatoren auch alle Module und ihre SRR-Objekte über die relevanten Änderungen im *Communication State*.

Weiters speichert er die neuen Werte in den *common parameters (commParam)* ab, auf die ebenfalls alle zugreifen können.

Dort speichert er auch zwei boole'sche Arrays, die für jedes registrierte Modul angeben, ob es in dieser Szeneninstanz aktiv ist / MOC ist oder nicht. Einige SRR-Objekte verwenden diese Information.

Auch das "iAmController"-Flag, welches für einige Network Sensors wichtig ist, um zu entscheiden, ob eine empfangene Nachricht verarbeitet oder verworfen werden soll, wird dort gesetzt.

Erwähnenswert ist noch, dass nicht nur ein *Access Request* zu einem Update des *Communication State* führt, sondern auch ein *Registration Request*.

Anmerkung: Näheres über den *Registration Request* findet sich in Kapitel 7.4.

## 7.4 Modul-Management

Anmerkung: das Modul-Management befindet sich erst im Aufbau, denn zur Zeit ist es nicht möglich, Module dynamisch zu laden / entladen. Deshalb ist auch eine Deregistrierung zur Zeit nicht notwendig.

### 7.4.1 Anmeldung/Initialisierung

Um ein Modul zu initialisieren und dabei beim SRR Controller anzumelden, vergibt der User (Modulautor) einen *Namen für das Modul* (den er vom Rahmen bekommen hat) und übergibt dem Modulkoordinator eine Referenz auf die *common parameter (commParam)* – diese hat er letzten Endes über den Rahmen vom SRR Controller bekommen, nachdem dessen Initialisierung abgeschlossen war.

Der Modulkoordinator kann über die commParam den SRR Controller adressieren und meldet sich bei diesem an.

Wenn das Modul noch nicht registriert ist, wird eine Registrierung beim Communication Controller durchgeführt (siehe Kapitel 7.4.2), wodurch dem Modul ein *moduleIx* (Ganzzahl  $\geq 0$ ) zugewiesen wird. Dieser Index kann als Zugriffszahl auf alle modulbezogenen Felder der commParam verwendet werden.

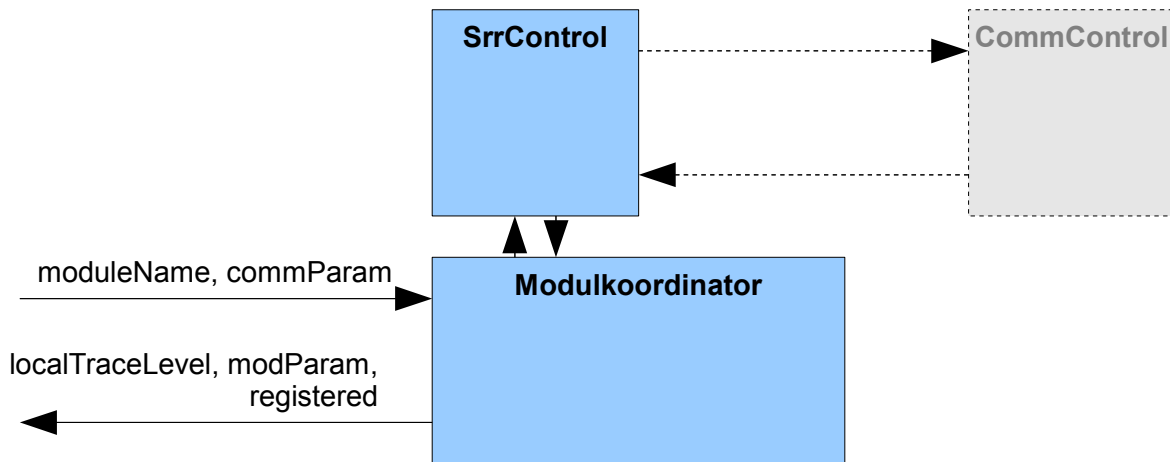


Abbildung 7: Initialisierung eines Moduls

Nachdem SrrControl den *moduleIx* im Feld **registered** übergeben hat, wird dieses und der *localTraceLevel* an den User (Modulautor) gemeldet.

Der **localTraceLevel** aller Module wird am uiControl Interface vom Rahmen gesetzt und über die *commParam* an den Modulkoordinator vermittelt. Der Modulautor kann diesen Wert benutzen, um den Tracelevel seiner eigenen Ausgaben festzulegen.

Zuletzt sendet der Modulkoordinator die Referenz auf die Modulparameter, **modParam**, an das Modul, welches diese an die statisch eingebundenen SRR-Objekte (also an die impliziten und statischen Modelle) weiterreicht, um deren Initialisierung zu starten.

### 7.4.2 Modulregistrierung

Modulregistrierung kann zur Zeit nicht aktiv vom Rahmen angestossen werden, sondern geschieht immer nur implizit durch die *Anmeldung* eines Moduls beim SRR Controller.

Wenn der Modulname noch nicht in der Liste der registrierten Module (aus dem *Communication State*) enthalten ist, wird die Registrierung durchgeführt.

Ein Registration Request enthält eine Liste von zusätzlich zu registrierenden Modulen, wobei der Communication Controller doppeltes Registrieren von Modulen verhindert.

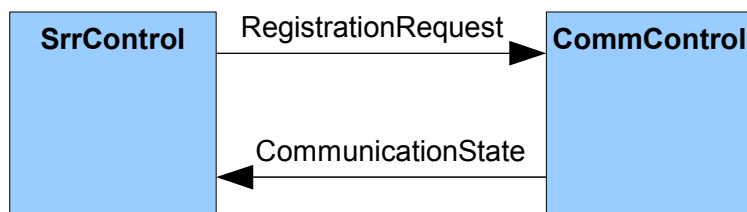


Abbildung 8: Modulregistrierung

## 7.5 SRR-Objekte

### SRR-Objekte

- sind keine renderbaren Objekte, sondern stellen Multiplayer-Funktionalität zur Verfügung
- können von Modul- und Modellautoren verwendet werden, um transportable, gegenseitig wiederverwendbare Teile von Multiplayer-Modellbahnanlagen (Module und Modelle) zu bauen.  
Hinweis: Welche Authoring Tools dafür in Frage kommen, soll noch untersucht werden.  
*Das wäre noch ein lohnendes Forschungsprojekt.*
- viele SRR-Objekte enthalten NetworkSensor/EventStreamSensor-Knoten, um mit den anderen Instanzen desselben Objektes kommunizieren zu können.
- können sowohl in Modellen als auch direkt in Modulen (als *implizite* Modelle) verwendet werden.

Der String für die Objekt-ID (objId) muss vom User (Modul- oder Modellautor) vergeben werden und für alle Szeneninstanzen identisch sein. Weiters müssen alle Objekt-IDs innerhalb eines Moduls eindeutig sein (damit Konsolen-Interface und Tracer einwandfrei funktionieren).

Konkrete Informationen über die SRR-Objekte von SRR v0.01 finden sich in Anhang 8.1.

### 7.5.1 Historische Ideensammlung zu SRR-Objekten

Zu Beginn 2009 gab es folgende Ideen, von denen viele bereits realisiert sind:

#### 7.5.1.1 Mögliche User Interfaces der SRR-Objekte:

##### Knoten:

Init: Typ des Knotens

Input:

Output: Trigger wenn Achse die Kante verlässt

##### Kante:

Init: A, B, C, normalA, normalB, Typ der Kante

Input:

Output: Graphikdaten nach Initialisierung

##### Weiche:

Init: Dauer eines Umschaltvorganges

Input: trigger toggle of state, lock, unlock

Output: softstate(real 0.0(minus) bis 1.0(plus)), locked (bool), state(plus, minus, changing)

##### Führerstand:

Init: Grenzwerte für Regler und Bremse, Anfangswerte für Regler und Bremse, ist der Führerstand relativ zur Lok vorwärts oder rückwärts angebracht?

Input: setze Reglerwert(real), setze Bremswert(real), setzevorzurück(bool), setze gültige

Schlüssel für diesen Führerstand

Output: Anzeige Reglerwert, Anzeige Bremswert

#### Achse:

Init: Trägheitsmoment der Achse(für Schleudern), Radius, Spurweite, Position im Waggon/in der Lok/im Drehgestell, vorwärts/rückwärts-flag relativ zum Waggon

Input: aktueller Reibungskoeffizient (?), Transformation nach SIMUL-RR Standard

Output: Winkelstand, Transformation nach SIMUL-RR Standard, Werte für zwei geschachtelte Transform-Knoten

#### Waggon2ax:

Init: Länge über Puffer, Masse

Input: Transformation von Achse 1, Transformation von Achse 2, Waggons-Transformation

Output: Waggons-Transformation, Werte für zwei geschachtelte <Transform>-Knoten

#### LokomotiveA1:

Init: Länge über Puffer, Masse

Input: Transformation von Achse 1, Transformation von Achse 2, Lok-Transformation

Output: Lok-Transformation, Werte für zwei geschachtelte <Transform>-Knoten

#### Antrieb:

Init: Z/v Diagramm für alle Schaltstufen, Trägheitsmoment(für Schleudern)

#### Zug:

Züge haben kein User Interface, sie werden gebildet, indem man Modelle aufs Gleis stellt.

### **7.5.1.2 Gegenseitige Zuordnung von SRR-Objekten**

Zum Beispiel ist ein Antrieb ein Teil einer Lokomotive, hat aber auch eine Assoziation mit allen Treibachsen, die von diesem Antrieb betrieben werden und mit den Führerständen dieser Lok.

## **7.6 Das Konsolen-Interface**

Der SRR Controller bietet dem Rahmen ein sog. **Konsolen-Interface**.

Der Rahmen kann dem SRR Controller ein **Konsolen-Kommando** senden, worauf er eine **Konsolen-Antwort** bekommen wird.

Welches User Interface der Rahmen verwendet, um dem Endbenutzer die Konsole zur Verfügung zu stellen, wird innerhalb SRR offen gelassen. Sei es nun ein HUD wie in der Beispielanlage oder ein Ajax3D-getriebenes Interface mit einer Server-Anwendung oder anything else.

Das Konsolen-Kommando ist ein String, die Konsolen-Antwort ein Feld von Strings (Zeilen).

Zur Zeit ist es der einzige Sinn des Konsolen-Interface, **SRR-Objekte zu beeinflussen**. Dazu stellen die SRR-Objekte **Parameter** zur Verfügung, über deren Namen man auf ihre Werte zugreifen kann.

Der SRR Controller in der Szeneninstanz des Benutzers teilt das Kommando in die Anteile, die verschiedene Module betreffen und adressiert damit ein Modul nach dem anderen.

Dazu wird jedes Teilkommando an die Szeneninstanz geschickt, die die MOC-Rolle für das Modul innehat (Parameter-Werte werden im MOC gesetzt). Dort wird es an den Modulkordinator weitergereicht.

Dieser teilt das Kommando in Teilkommandos an jedes adressierte SRR-Objekt und sammelt die Teil-Antworten von den Objekten.

Die modulspezifische Antwort wird dann zurück an den ursprünglichen SRR Controller gesendet, der sofort das nächste Modul adressiert.

So werden in geordneter Art und Weise alle Teil-Antworten zu einer Gesamt-Antwort vereinigt und in der ursprünglichen Szeneninstanz am User Interface uiControl ausgegeben.

### **Es gibt folgende Konsolen-Kommandos:**

#### **options**

gibt die Liste der Modulnamen der registrierten Module aus.

#### **options <moduleNames>**

gibt die Liste der Objekt-IDs aller SRR-Objekte aller adressierten Module aus.

#### **options <moduleNames>-<objIds>**

gibt die Liste aller Parameternamen der adressierten SRR-Objekte aus.

#### **options <moduleNames>-<objIds>-<parameterNames>**

gibt die erwarteten Datentypen aller adressierten Parameter aus.

#### **read [<moduleNames>[-<objIds>[-<parameterNames>]]]**

gibt die aktuellen Werte der adressierten Parameter aus.

#### **set [<moduleNames>[-<objIds>[-<parameterNames>]]][=<Value>]**

setzt den Wert in allen adressierten Parametern (kein Wert heisst Defaultwert für jeden Parameter) und gibt die aktuellen Werte der adressierten Parameter danach zurück.

## **7.7 Animierte Objekte**

Als Vorversuch für animierte Züge wurde ein *animiertes Karussell* implementiert. Das SRR-Objekt *SrrDriveA* liefert quasi-kontinuierlich (mit jedem Frame) einen Wert *angle* (im Intervall [0.0..6.28]), der als Eingangsgröße für einen <OrientationInterpolator>-Knoten verwendet wird.

Eigentlich liefert auch das *SrrSwitchA – Objekt* einen quasi-kontinuierlichen Wert als Ausgangsgröße, nämlich *softState*.

Beim Switch jedoch liegen die Verhältnisse einfacher, da Zustandswechsel relativ selten sind und man davon ausgehen kann, dass sich der Schalter meistens in einer Endlage befindet und somit synchronisiert ist.

Man versucht also, den Sollwert des Schalters schnell zu synchronisieren (das ist bei einem binären Schalter die Übertragung eines einzelnen Bits an alle Szeneninstanzen) und sorgt dafür, dass alle Szeneninstanzen dem Sollwert mit derselben Geschwindigkeit folgen und ihre Endlage getrennt dem User (Modell/Modul) melden.

Bei *komplizierteren Animationen*, denen eine *allgemeinere Simulationsrechnung* zugrunde liegt, ist nicht immer einfach ersichtlich, was der „Sollwert“ ist, und mit welcher Geschwindigkeit er

angepeilt werden soll. Und wie oft soll man den Sollwert übers Netz übertragen?

Anmerkung: Beim Karussell hat sich folgender Ansatz vorläufig bewährt, muss aber noch zeigen, ob er auch für Schienenfahrzeuge hält, was er verspricht.

Einigermassen allgemein gesprochen, kann man die Simulation/Animation eines Objektes durch einen zeitlich veränderlichen Zustandsvektor "state=state(t)" beschreiben (beim Karussell ist das z.B. ein Vektor aus Winkel, Winkelgeschwindigkeit und Winkelbeschleunigung).

Genau eine der Szeneninstanzen berechnet diesen Zustandsvektor quasi-kontinuierlich, das heisst mit jedem Frame neu. Diese Szeneninstanz ist der sogenannte MOC („module controller“).

Um nun eine einigermaßen exakte Animation zu erreichen, wird nicht nur einfach der „state“ zu bestimmten Zeitpunkten verteilt, sondern ***es wird ein „zukünftiger state“ vorausgesagt, state(t) wird extrapoliert.***

Je nach den Umständen und dem aktuellen state( $t_0$ ), wobei  $t_0$ =jetzt, wird der state mal weiter extrapoliert und mal weniger weit. Wenn der Sollwert der Animation eine Position ist, wird z.B. die aktuelle Beschleunigung einen guten Hinweis darüber liefern, wie weit man extrapolieren kann (je weiter, desto besser für die Netzwerk-Performance, je kürzer, desto genauer die Animation).

Der extrapolierte „state“, also „targetState“ = state( $t_{\text{target}}$ ), wird gemeinsam mit der extrapolierten Zeitspanne „targetDuration“ =  $t_{\text{target}} - t_0$  an alle aktiven Szeneninstanzen verteilt.

Die aktiven Instanzen des animierten Objektes verwenden diese Werte, also „targetState“ und „targetDuration“, um mit Hilfe eines <TimeSensors> und von Interpolatoren die gewünschten stückweise-linear interpolierten Ausgangsgrößen zu generieren.

Dadurch wird die Netzwerklast reduziert (im Vergleich zur quasi-kontinuierlichen Übertragung eines quasi-kontinuierlichen Zustandwertes bei jedem Frame), andererseits ist sichergestellt, dass stets nur eine Szeneninstanz der „Master“ über state(t) bleibt.

In Abbildung 9 wird gezeigt, wie der MOC den quasi-kontinuierlichen state berechnet, diesen immer wieder für eine Dauer "targetDuration" extrapoliert, den „targetState“ an alle aktiven Szeneninstanzen verteilt und diese dann eine lineare Interpolation durchführen.

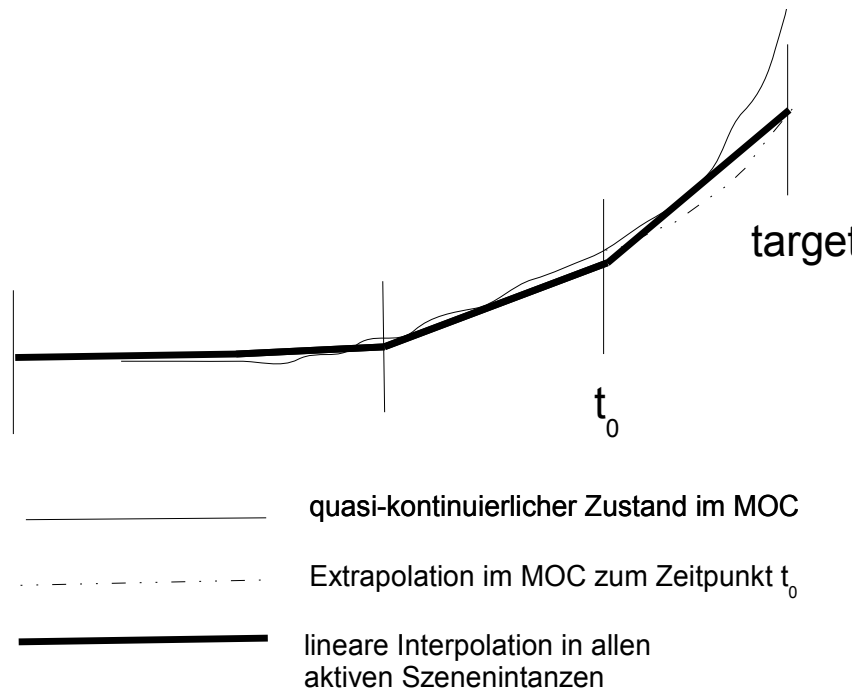


Abbildung 9: Extra- und Interpolation des Zustandes von animierten Objekten

#### Am Beispiel des Karussells:

Der MOC berechnet zu jedem Frame aufgrund des alten Wertes der Winkelgeschwindigkeit, des actualState des switch-Knotens (Karussell ein/aus) und der Antriebscharakteristik den aktuellen Wert des inneren Drehmoments  $M_i$ .

Aus  $M_i$  wird mithilfe der aktuellen Winkelgeschwindigkeit, des Reibungskoeffizienten und der Trägheit die aktuelle (neue) Winkelbeschleunigung berechnet.

Aus der Beschleunigung und der alten Geschwindigkeit kann die neue Geschwindigkeit errechnet werden und weiters auch der neue Ort.

Dieser neue „state“ (Beschleunigung, Geschwindigkeit, Winkel) wird gespeichert.

Wenn der „target timer“ abläuft, wird der state extrapoliert (parabolisch) und somit eine targetDuration und eine targetPosition erhalten, die an alle aktiven Szeneninstanzen übermittelt werden können

Diese werden daraus die „key“ und „keyValue“ Werte für den „animation interpolator“ neu berechnen sowie die „cycleTime“ des „animation timers“ setzen und den „animation timer“ neu starten.

### **7.7.1 Korrektur der targetDuration für jede Szeneninstanz**

Bei höheren Anforderungen an die Korrektheit der Animations-Interpolation (oder bei größeren Entfernungen zwischen den einzelnen Szeneninstanzen) kann folgender Mechanismus verwendet werden:

Die Szeneninstanzen antworten auf eine „target“ Message sofort mit einer „report“ Message (z.B.

der aktuellen Position). Der MOC misst die „round trip time“ vom Aussenden der „targets“ bis zum Eintreffen der „reports“ und korrigiert damit die individuelle targetDuration der nächsten „targets“ Message.

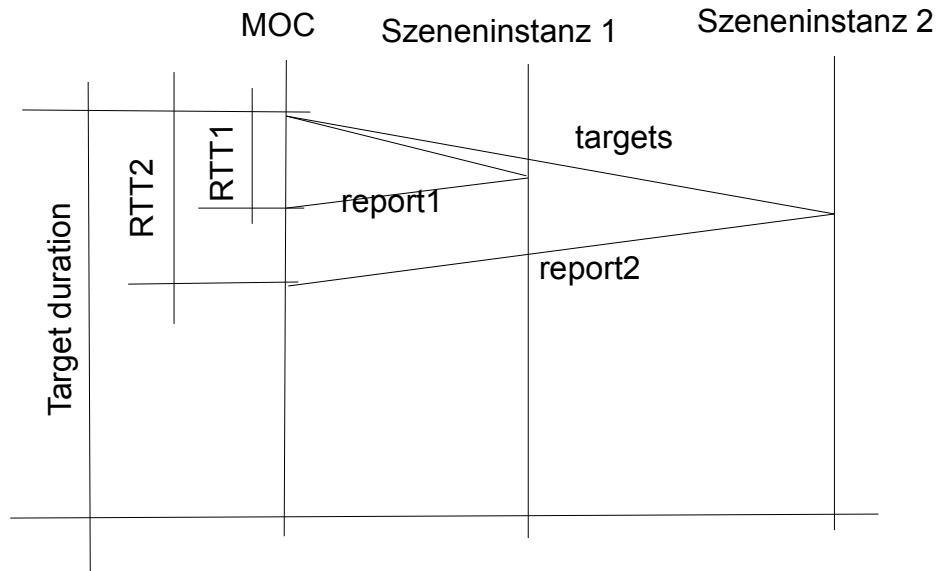


Abbildung 10: Korrektur der targetDuration durch Messen der Round Trip Time

## 7.8 Schlüsselbehälter und getragene Schlüssel, Schlösser

Um Schlüssel zu kreieren und aufzubewahren, z.B. in einem Schlüsselbrett, wurde das **SRR-Objekt** *SrrKeyContainer* entwickelt.

Gleichzeitig wurde der SRR Controller erweitert, um dem Rahmen stets eine Liste der **getragenen Schlüssel (carried keys)** zu liefern.

Dadurch wird es möglich, die getragenen Schlüssel in einer für den Rahmen spezifischen Art und Weise anzuzeigen (z.B. in einem HUD), Schlüssel einem Schlüsselbehälter zu entnehmen und Schlüssel in einen Schlüsselbehälter zu legen.

Um Schlüssel in einen Schlüsselbehälter zu legen, muss dieser zuvor mit *set\_bind* gebunden werden. Wenn kein Schlüsselbehälter gebunden ist, geht die *putKeys*-Methode ins Leere.

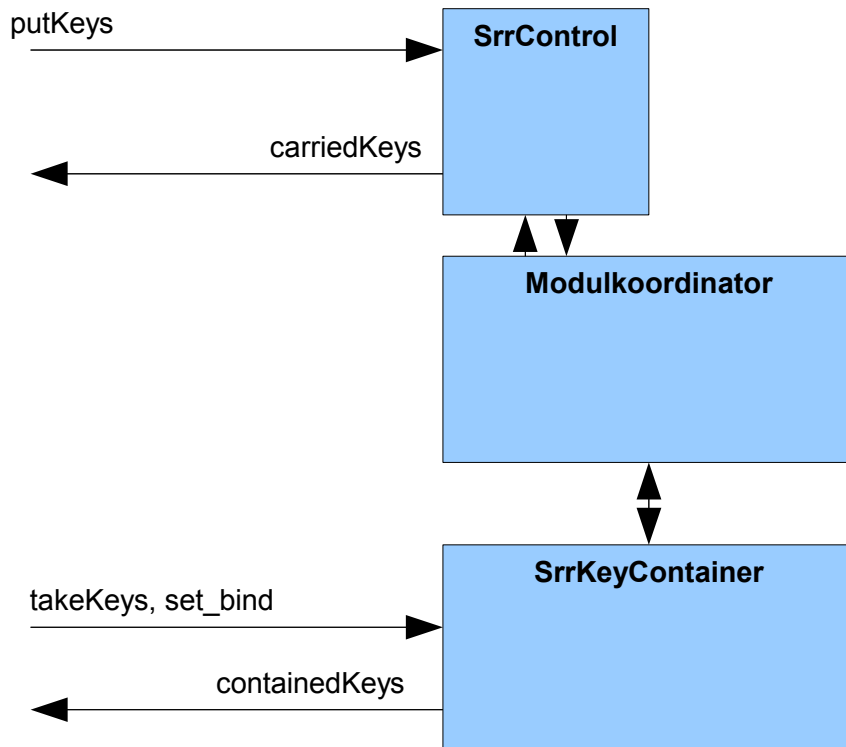


Abbildung 11: Schlüsselbehälter und getragene Schlüssel

Danach wurde das *SrrLock-Objekt* entwickelt, welches alle Funktionen eines SrrKeyContainer bietet, aber zusätzlich *fittingKeys* als Input verlangt und auf *carriedKeys* und *containedKeys* durch Ausgabe eines Feldes *locked=true/false* reagiert.

SrrLock kann auf die *carriedKeys* reagieren, weil der SRR Controller über den Modulkoordinator einen boole'schen Trigger an alle Objekte sendet, wenn sich die Liste der *carriedKeys* geändert hat (*carriedKeysChanged*). SrrLock hört auf diesen Trigger und sieht dann in einem Feld der *commParam* nach, was zur Zeit tatsächlich die *carriedKeys* sind.

## 7.9 Avatare und bewegte Objekte

Als das animierte Karussell entwickelt war, zeigte sich im multi-user-mode, dass die Avatare "ruckelten", wenn sie das Karussell betreten hatten (ich nannte das "bouncing avatars problem").

Abhilfe wurde geschaffen, indem man die Avatare beim Betreten des Karussells in das lokale Koordinatensystem des Karussells verschob und dann auch die Positionen und Orientierungen der Avatare relativ zu diesem Koordinatensystem übertrug.

Zu diesem Zweck wurden sogenannte "Avatar Container" entwickelt. Ein *Master Avatar Container* muss im Rahmen angelegt werden, um die frisch geladenen Avatare entgegenzunehmen, bzw. die Löschbefehle, wenn ein anderer User die Spielsitzung verlässt.

Der Autor eines beweglichen und betretbaren Modells muss nur einen Avatar Container in diesem Modell anlegen, initialisieren und mit *set\_bind* binden, sobald der User das Modell betreten hat.

Anmerkung: diese "Lösung" erscheint nur ein Notbehelf, solange ich nichts genaueres über

Avatartechnologien weiss. *Das wäre noch ein lohnendes Forschungsprojekt.*

## 8 Anhang

### 8.1 Nähere Informationen über die SRR-Objekte von SRR v0.01 - Basismodul

#### 8.1.1 SrrSwitchA - Binary Switch

SrrSwitchA wurde in Schritt 0002 implementiert, wo er verwendet wurde, um den Hebel zu animieren, mit dem man das Karussell ein- und ausschalten konnte.

SrrSwitchA benützt einen Network Sensor, um den „Ziel-Zustand“ (scheduled state) vom MOC an alle Szeneninstanzen zu verteilen.

Die Szeneninstanzen folgen mit der eingestellten Geschwindigkeit dem Zielzustand und melden den „tatsächlichen Zustand“ (actual state), sobald die Endlage des Schalters erreicht ist.

Jede Szeneninstanz (jeder User) kann mit einem „toggle Request“ den Zielzustand ändern. Das wird erreicht, indem die Szeneninstanz einen toggle Request an den MOC sendet – der ja den Zielzustand zentral verwaltet -, welcher den Zielzustand invertiert und an alle Szeneninstanzen verteilt. Diese starten die Animation hierauf neu.

In Schritt 0003 wurde die Möglichkeit implementiert, ein SrrLock-Objekt dem SrrSwitchA zuzuordnen, sodass dieser von dem SrrLock-Objekt auf- und zugesperrt werden kann.

#### 8.1.2 SrrDriveA – Carousel Drive

SrrDriveA wurde im Schritt 0002 realisiert, wo er verwendet wurde, um das Karussell zu simulieren und synchronisiert zu animieren.

SrrDriveA im MOC berechnet den Zustandsvektor mit jedem frame neu. Er reagiert dabei auf den Input switchState, welcher die Mi / omega Kennlinie modifiziert. Omega ist die Winkelgeschwindigkeit in Radiant / sek und Mi ist das innere Moment in kg m<sup>2</sup> / s<sup>2</sup>. Das innere Moment wird um den Wert friction\*omega reduziert und danach durch inertia dividiert, um die aktuelle Winkelbeschleunigung in rad / s<sup>2</sup> zu erhalten.

Der Zustandsvektor wird jede Sekunde an alle Szeneninstanzen verteilt, damit diese bei Gelegenheit die MOC-Rolle übernehmen können.

Wenn der “target timer” ausläuft, wird eine neue “target duration” berechnet. Diese ergibt sich aus der Formel  $\text{minDuration} * \text{accelWeakness} / \text{accel}$ . Das heisst, dass die Target-Duration genau dann die (einstellbare) „minimumDuration“ sein wird, wenn die Beschleunigung gleich gross ist wie der einstellbare Parameter „accelWeakness“.

In Schritt 0012 wurde SrrDriveA an das allgemeine Konzept angepasst. Nun gibt es keinen Input-Parameter *switchState*, sondern einen Parameter *switch*, der auf ein „SrrSwitchA“-Objekt zeigt.

#### 8.1.3 SrrKeyContainer – Key Container

SrrKeyContainer wurde im Schritt 0003 realisiert, wo er verwendet wurde, um den Türschlüssel für das Stationshaus und den Karussellschlüssel zu kreieren und aufzubewahren.

SrrKeyContainer beherbergt eine Liste von Schlüsseln. Jeder Schlüssel ist durch eine globale Key Id beschrieben, die das Format **<moduleName>.<objId>.<keyId>** hat. **<moduleName>** und **<objId>** sind die Eigenschaften des SrrKeyContainer, in denen der Schlüssel kreiert worden ist. **<keyId>** kann vom User beliebig vorgegeben werden.

Jede Szeneinstanz kann Schlüssel aus dem Key Container entnehmen (takeKeys) und sie den „carried Keys“ der Szeneninstanz hinzufügen sowie Schlüssel in den Key Container legen (putKeys), nachdem sie den Key Container „gebunden“ hat (es kann immer nur ein Key Container das Ziel von putKeys-Aktionen sein).

#### **8.1.4 SrrLock - Lock**

SrrLock wurde im Schritt 0003 realisiert, wo es verwendet wurde, um die Tür des Stationshauses zu versperren (containedKeys) und den Karussellschalter zu versperren (carriedKeys).

SrrLock enthält alle Funktionen von SrrKeyContainer plus die Eigenschaft, in Abhängigkeit von „carried Keys“ und „contained Keys“ den versperrt/aufgesperrt-Zustand zu wechseln.

#### **8.1.5 SrrAvatarContainer – Avatar Container**

Der Avatar Container dient dazu, Avatarposition und –orientierung relativ zu einem lokalen Koordinatensystem (welches animiert sein kann) zu übertragen, um das „bouncing avatars problem“ zu lösen.

Jeder Avatar Container enthält einen <Group>-Knoten, dem beliebig viele Avatare zugeordnet sein können. Dieser <Group>-Knoten ist eine Ausnahme von der Regel, dass SRR-Objekte keine renderbare Graphik produzieren.

Weiters enthält jeder Avatar Container einen <ProximitySensor>-Knoten, der die relative Avatarposition und –orientierung misst.

Der user kann mit set\_bind einen Avatar Container in seiner Szene auswählen. Die Avatarposition und –orientierung des eigenen Avatars wird darauf relativ zum lokalen Koordinatensystem dieses Avatar Containers übertragen und alle anderen Szeneninstanzen werden diesen Avatar in diesen Avatar Container „umhängen“.

#### **8.1.6 SrrMasterAvatarContainer – Master Avatar Container**

Der Master Avatar Container wird im main file verwendet.

Er bietet zusätzlich folgende Felder:

Die Felder addAvatar und removeAvatar um beim Starten und Beenden anderer Szeneninstanzen Avatare hinzuzufügen und zu löschen.

Die Felder reportOwnPosition und reportOwnOrientation, deren Output zum <BSCollaborate>-Knoten ge<ROUTE>t werden sollte. Dieser Output wird immer nur von einem der Avatar Container gespeist (nämlich vom gebundenen), dieses Routing geschieht SIMUL-RR intern.

### **8.2 Die Tracer-Subsysteme und -Instanzen der Beispielanlage**

Auch die Beispielanlage, also

- der Rahmen

- das Modul "first"
- das Modul "second"

benützt den SrrTrains Tracer.

Die Tracer-Instanzen des Subsystems "MyFrame" finden sich in Abbildung 12.

Die Repräsentation der Subsysteme "MyFirstModule" und "MySecondModule" in den Trace-Ausgaben ist aus Abbildung 13 und Abbildung 14 ersichtlich.

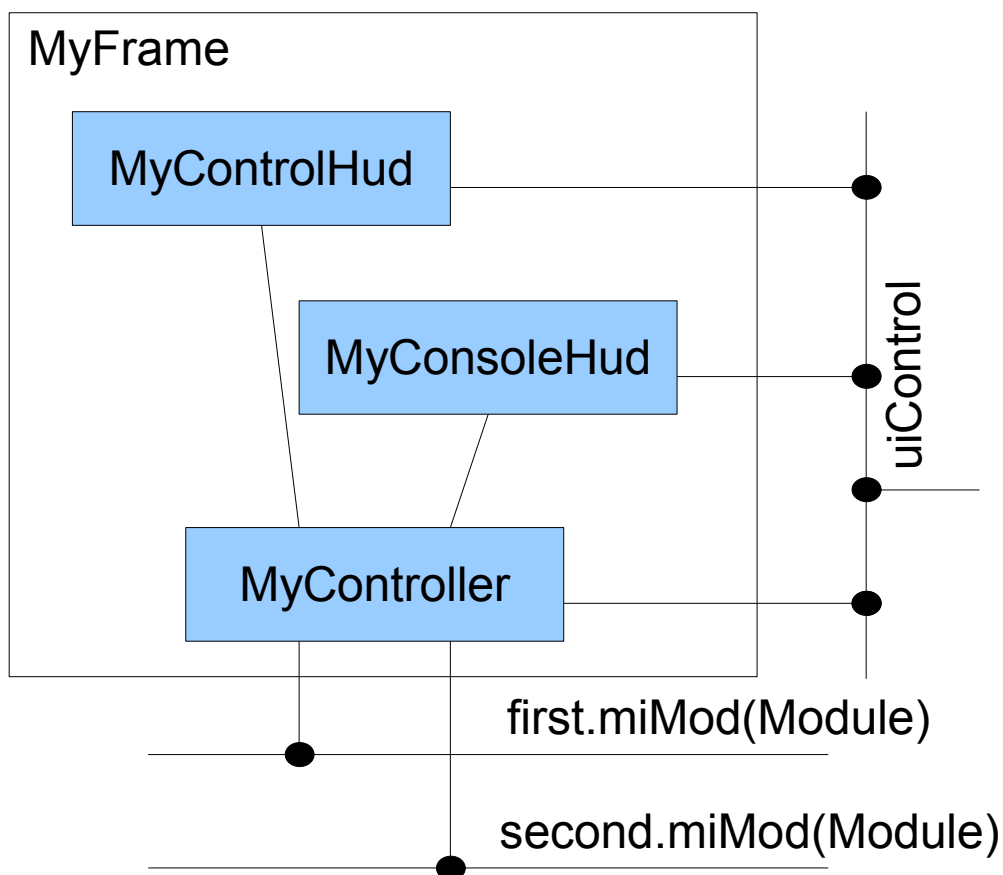


Abbildung 12: Tracer-Instanzen im Tracer-Subsystem "MyFrame"

Man sieht, dass der Rahmen der Beispielanlage hauptsächlich dazu dient, das Interface uiControl (User Interface des SRR Controllers) zu bedienen.

Zusätzlich werden die beiden Module "first" und "second" statisch geladen und über das Interface miMod(Module) initialisiert.

Anmerkung: der Rahmen der Beispielanlage besteht auch noch aus anderen Elementen, doch nur das "MyController"-Script und die beiden HUDs "MyControlHud" und "MyConsoleHud" erzeugen Trace-Ausgaben mit Hilfe des SrrTracer-Prototyps.

Anmerkung: auch andere Elemente des Rahmens der Beispielanlage beliefern das uiControl Interface (z.B. der <BSCollaborate>-Knoten "MU" und das Tracer HUD), diese erzeugen jedoch keine Tracer-Ausgaben und sind deswegen hier nicht aufgeführt.

Anmerkung: alle diese Instanzen existieren genau einmal pro Szeneninstanz.

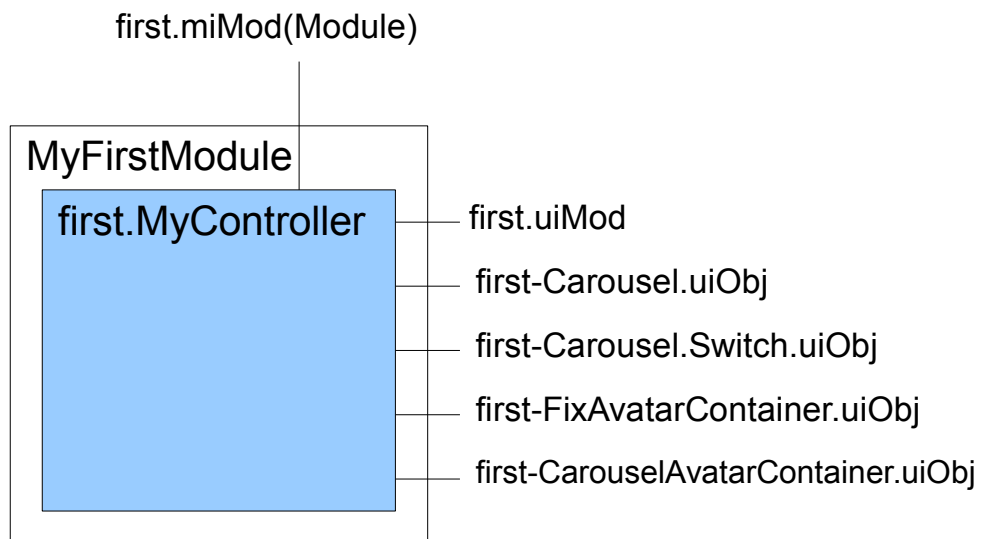


Abbildung 13: Tracer-Instanzen im Tracer-Subsystem "MyFirstModule"

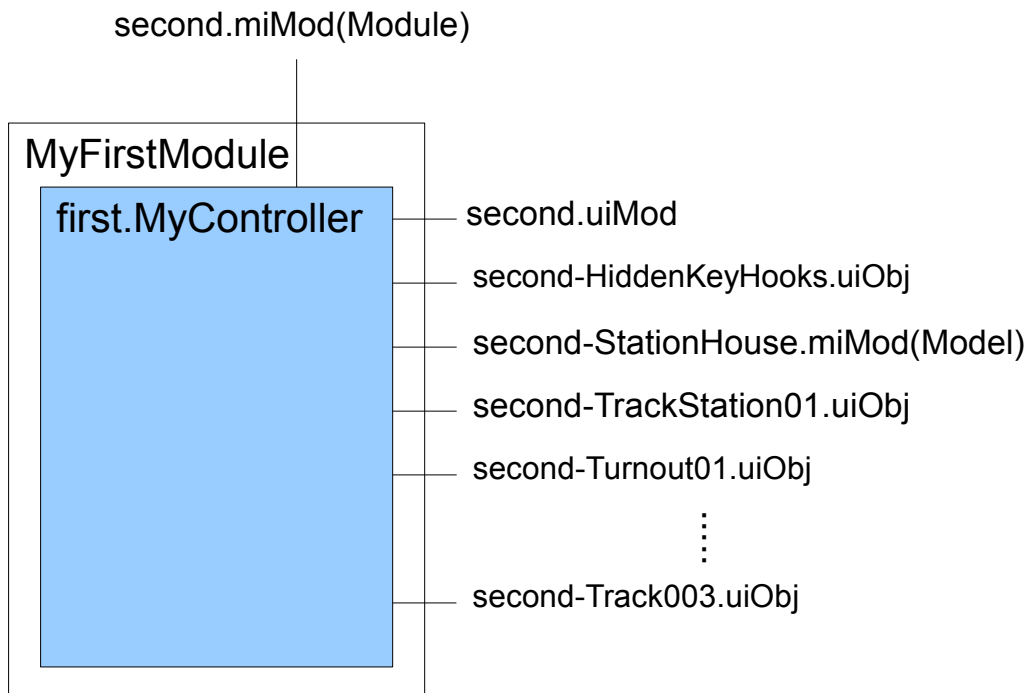


Abbildung 14: Tracer-Instanzen im Tracer-Subsystem "MySecondModule"

#### Beide Module

- bieten dem Rahmendas Interface `miMod(Module)`, um das Modul nach dem Laden zu initialisieren
- verwenden ihre Instanz des Modulkoordinators über das Interface `uiMod`

Die Tracer-Instanz "first.MyController" im Subsystem "MyFirstModule" kontrolliert das Modul und liefert Trace-Ausgaben. Sie

- initialisiert das `SrrDriveA`-Objekt "Carousel". Die **enthaltenen Objekte** "Carousel.Switch" und "Carousel.Switch.Lock" werden von "Carousel" initialisiert.
- verwendet das Feld "toggle" des `SrrSwitchA`-Objektes "Carousel.Switch", um das Karussell ein- und auszuschalten.
- hört auf die Felder "isActive" und "actualState" des `SrrSwitchA`-Objektes "Carousel.Switch" und erzeugt mit diesen Werten je eine Trace-Ausgabe zur Information.
- triggert die "set\_bind" Felder der Avatar-Container, wenn das Karussell betreten (CarouselAvatarContainer) bzw. verlassen (FixAvatarContainer) wird.
- Liefert Trace-Ausgaben und reagiert bei diversen Ereignissen innerhalb des Moduls (z.B. wird ein "activateRequest" an den Modulkoordinator gesendet, wenn ein Viewpoint des Moduls gebunden worden ist).

Die Tracer-Instanz "second.MyController" im Subsystem "MySecondModule" kontrolliert das Modul und liefert Trace-Ausgaben. Sie

- initialisiert das SrrKeyContainer-Objekt "HiddenKeyHooks" (Schlüsselbrett).
- verwendet die Felder "takeKeys" und "containedKeys" des SrrKeyContainer-Objektes "HiddenKeyHooks", um dem Schlüsselbrett einen Schlüssel zu entnehmen (wenn der User darauf geklickt hat), und um den geänderten Inhalt des Schlüsselbretts anzuzeigen.
- initialisiert das *statische Modell* "StationHouse" (das als X3D-Prototyp in einer eigenen Datei abgelegt ist) über das Interface miMod(Model).
- initialisiert alle Gleise und Weichen.
- Fordert vom Modulkordinator die "Erzeugung eines Fahrzeugs" an, wenn der User ein bestimmtes Bahnhofsgleis anclickt.
- Liefert Trace-Ausgaben und reagiert bei diversen Ereignissen innerhalb des Moduls (z.B. wird ein "activateRequest" an den Modulkordinator gesendet, wenn ein Viewpoint des Moduls gebunden worden ist).

### 8.3 Tracer-Subsysteme und -Instanzen von SRR v0.01 aufgelistet

<u>Kategorie</u>	<u>Dateien</u>	<u>Subsystem</u>	<u>Instanzen</u>
SRR Controller	SrrControl.x3d SrrControlNs.x3d	SrrFramework	SrrControl
		CentralController	CommControl TrainControl
Modulkoordinator	SrrModCoord.x3d	SrrFramework	<moduleName>.Coord
SRR-Objekte	SrrSwitchA.x3d SrrSwitchANs.x3d	SrrSwitchA	<moduleName>-<objId>.Control <moduleName>-<objId>.Moc
		SrrDriveA.x3d SrrDriveANs.x3d	SrrDriveA*)
	SrrAnimatedObjectNs.x3d	SrrAnimatedObject*)	<moduleName>-<objId>.Control <moduleName>-<objId>.Moc
	SrrKeyContainer.x3d SrrKeyContainerNs.x3d	SrrKeyContainer**)	<moduleName>-<objId>.Control <moduleName>-<objId>.Moc
	SrrLock.x3d	SrrLock**)	<moduleName>-<objId>.Control <moduleName>-<objId>.Moc
	SrrAvatarContainer.x3d SrrMasterAvatarContainer.x3d SrrAvatarContainerCore.x3d SrrAvatarContainerNs.x3d	SrrAvatarContainer	<moduleName>-<objId>.Control <moduleName>-<objId>.CC <objId>.Control <objId>.CC

\*) SrrAnimatedObject wird von SrrDriveA benützt, ein SrrDriveA-Objekt kann also Trace-Ausgaben mit subsystem=SrrAnimatedObject erzeugen

\*\*\*) SrrLock verwendet SrrKeyContainer, ein SrrLock-Objekt kann also Trace-Ausgaben mit subsystem=SrrKeyContainer erzeugen

Fortsetzung: nächste Seite

<b>Kategorie</b>	<b>Dateien</b>	<b>Subsystem</b>	<b>Instanzen</b>
SRR-Objekte (fortgesetzt)	SrrTrackEdge.x3d	SrrTrackEdge	<moduleName>-<objId>.Control
	SrrTrackNode.x3d	SrrTrackNode	<moduleName>-<objId>.Control
	SrrBasicTrackSection.x3d	SrrBasicTrackSection	<moduleName>-<objId>.Control
	SrrBasicTurnout2Way.x3d	SrrBasicTurnout2Way	<moduleName>-<objId>.Control
	SrrSwitchB.x3d SrrSwitchBNs.x3d	SrrSwitchB	<moduleName>-<objId>.Control <moduleName>-<objId>.Moc
Beispiel-Gleisgeometrie	SrrTrackGeometryABI.x3d	SrrTrackGeometryABI*)	<moduleName>-<objId>.Control
	SrrTrackSectionA.x3d	SrrTrackSectionA*)	<moduleName>-<objId>.Control
	SrrTurnoutLeftA.x3d	SrrTurnoutLeftA*)	<moduleName>-<objId>.Control
	SrrTurnoutRightA.x3d	SrrTurnoutRightA*)	<moduleName>-<objId>.Control
Beispielanlage,Rahmen	Main_bscontact.x3d Main_octaga.x3d TestMainFirst_bscontact.x3d TestMainSecond_bscontact.x3d ConsoleHud.x3d ControlHud.x3d**)	MyFrame	MyController MyConsoleHud MyControlHud
Beispielanlage, erstes Modul	FirstModule.x3d	MyFirstModule	<moduleName>.MyController
Beispielanlage, zweites Modul	SecondModule.x3d	MySecondModule	<moduleName>.MyController
Beispielanlage, statische Modelle	StationHouse.x3d	MyStationHouse	<moduleName>-<objId>.MyController

*Tabelle 1: Tracer-Subsysteme und -Instanzen in den Files von SRR v0.01*

\*) SrrTrackGeometryABI wird von allen SrrTrackEdge-Objekten benützt, die aus einem der drei statischen Modelle der Beispiel-Gleisgeometrie entstanden sind, ein SrrTrackEdge-Objekt kann also Trace-Ausgaben mit subsystem=SrrTrackGeometryABI liefern.

\*\*) Der Rahmen der Beispielanlage besteht aus mehr Dateien, doch die hier nicht angeführten liefern keine Trace-Ausgaben

## 8.4 Setzen des/der Tracelevel(s)

Die Tracer-Instanzen des SRR Frameworks können im allgemeinen unterschiedliche Tracelevel besitzen.

Es ist also z.B. möglich, den SRR Controller genau zu tracen (level 3), gleichzeitig aber die Ausgaben aller Objekte zu unterdrücken.

Die Tracelevel werden vom Rahmen beim SRR Controller angefordert und dementsprechend aktiviert.

Tracelevel gelten immer lokal, wenn man also Trace-Ausgaben von mehreren Szeneninstanzen kombinieren möchte, muss man die Tracelevel in jeder Szeneninstanz extra anfordern.

<b>Tracer-Instanz</b>	<b>Tracelevel[0]</b>	<b>Tracelevel[1]</b>
SrrControl	Tracelevel im Betrieb	Tracelevel während Initialisierung
CommControl	Tracelevel im Betrieb	Tracelevel während Initialisierung
TrainControl	Genereller Tracelevel	n/a
<moduleName>.Coord	Tracelevel im Betrieb	Tracelevel während Initialisierung
<moduleName>-<objId>.Control <moduleName>-<objId>.Moc	Genereller Tracelevel	

Tabelle 2: Unterschiedliche Tracelevel für unterschiedliche Instanzen des SRR Framework

Der Tracelevel von **SrrControl** ist ein **MFInt32** – **Wert der Länge 2**, Wert[0] ist der Tracelevel nachdem SrrControl die commParam ausgegeben hat, Wert[1] ist der Tracelevel davor.

Der Tracelevel von **CommControl** ist ein **MFInt32** – **Wert der Länge 2**, die Bedeutung von Wert[0] und Wert[1] ist analog zu SrrControl.

Der Tracelevel von **TrainControl** ist ein **MFInt32** – **Wert der Länge 1**. Wert[0] gilt generell immer.

Der Tracelevel der Module ist ein String der Syntax  
`*=<TL0>,<TL1>[;<modulePattern>=<TL0>,<TL1>]....`

Man muss also einen "Defaultwert" für alle Module angeben, kann aber zusätzlich Module selektieren (\*' als Platzhalter an letzter Stelle ist möglich) und Tracelevel für diese Module angeben.

Der Tracelevel der Objekte ist ein String der Syntax  
`*.*=<TL0>[;<modulePattern>-<objectPattern>=<TL0>]....`

Man muss also einen "Defaultwert" für alle Objekte angeben, kann aber zusätzlich Objekte selektieren (\*' als Platzhalter an letzter Stelle ist möglich) und einen Tracelevel für diese Objekte angeben.

**Doch was tun, um den Tracelevel der eigenen Software zu setzen (Rahmen, Module, Modelle)?**

Um den ***Tracelevel des Rahmens*** muss dieser sich selbst kümmern (wenn er SrrTracer.x3d verwendet). SrrTracer sollte erst benützt werden, sobald es mit den commParam initialisiert worden ist.

Die Modulkoordinatoren liefern stets den aktuellen Wert ihres Tracelevels am Interface uiMod, so dass ein Modulautor diesen Wert verwenden kann, um den ***Tracelevel des Moduls*** festzulegen. Auch hier gilt, dass SrrTracer erst dann verwendet werden sollte, nachdem der Modulkoordinator die modParam geliefert hat.

***Modelle*** haben eine eigene Objekt-ID und sie wissen immer, in welchem Modul sie sich befinden (sobald sie die modParam erhalten haben). Wenn sie den Tracer SrrTracer.x3d verwenden und diesem eine Objekt-ID angeben, die ungleich dem Leerstring ist, setzt er den eigenen lokalen Tracelevel automatisch entsprechend den Angaben, die SrrControl erhalten hat.

## 8.5 Beispiele für Trace-Ausgaben

### 8.5.1 Entnahme eines Schlüssels aus einem SrrKeyContainer

Diese Trace-Ausgaben sind in folgender Situation entstanden.

- Die Beispielanlage ist initialisiert und beide Module sind bereits aktiviert.
- Der Frame und die Module (Modulkoordinatoren) sind auf Tracelevel=2 gesetzt, alles andere auf Tracelevel=1
- Der User steht vor den "Hidden Key Hooks" und clickt auf den "DoorKey"

#### **(1)**

```
SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1
SRR: moduleName=second,subsystem=MySecondModule,fileName=SecondModule.x3d,ssVersion=0100,instanceId=second.MyController
SRR: user clicked key on key hooks -> trigger 'take' action
SRR:END
SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1
SRR: moduleName=second,subsystem=MySecondModule,fileName=SecondModule.x3d,ssVersion=0100,instanceId=second.MyController
SRR: action=sendEvent;destination=second-HiddenKeyHooks.uiObj;field=takeKeys
SRR: value=second.HiddenKeyHooks.DoorKey
SRR: trigger take action at key container
SRR:END
```

#### **(3)**

```
SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1
SRR: moduleName=second,subsystem=MySecondModule,fileName=SecondModule.x3d,ssVersion=0100,instanceId=second.MyController
SRR: action=eventReceived;origin=second-HiddenKeyHooks.uiObj;field=containedKeys
SRR: value=[]
SRR: contained keys changed in key container -> display at key hooks
SRR:END
```

#### **(4)**

```
SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1
SRR: moduleName=second,subsystem=SrrFramework,fileName=SrrModCoord.x3d,ssVersion=0100,instanceId=second.Coord
SRR: action=eventReceived;origin=second-*.Control;field=takeKeys
SRR: value=second.HiddenKeyHooks.DoorKey
SRR: received request to take keys from an SRR object
SRR:END
SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1
```

SRR: moduleName=second, subsystem=SrrFramework, fileName=SrrModCoord.x3d, ssVersion=0100, instanceId=second.Coord  
SRR: action=sendEvent;destination=SrrControl;field=takeKeys  
SRR: value=second.HiddenKeyHooks.DoorKey  
SRR: forward request to take keys to SRR controller  
SRR:END

### **(5.6)**

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: subsystem=MyFrame, fileName=Main\_bscontact.x3d, ssVersion=0100, instanceId=MyController  
SRR: action=eventReceived;origin=uiControl;field=carriedKeys  
SRR: value=second.HiddenKeyHooks.DoorKey  
SRR: received new set of carried keys from SrrControl  
SRR:END

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: subsystem=MyFrame, fileName=Main\_bscontact.x3d, ssVersion=0100, instanceId=MyController  
SRR: displaying set of carried keys in Key Hud  
SRR: value=second.HiddenKeyHooks.DoorKey  
SRR:END

### **(7)**

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: moduleName=second, subsystem=SrrFramework, fileName=SrrModCoord.x3d, ssVersion=0100, instanceId=second.Coord  
SRR: action=eventReceived;origin=SrrControl;field=carriedKeysChanged  
SRR: value=true  
SRR: SRR Controller reported change in carried keys  
SRR:END

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: moduleName=second, subsystem=SrrFramework, fileName=SrrModCoord.x3d, ssVersion=0100, instanceId=second.Coord  
SRR: action=sendEvent;destination=second-\*.Control;field=carriedKeysChanged  
SRR: tell all SRR objects that carried keys changed  
SRR:END

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: moduleName=first, subsystem=SrrFramework, fileName=SrrModCoord.x3d, ssVersion=0100, instanceId=first.Coord  
SRR: action=eventReceived;origin=SrrControl;field=carriedKeysChanged  
SRR: value=true  
SRR: SRR Controller reported change in carried keys  
SRR:END

SRR:traceLevel=2,timeStamp=2009/08/19/12:04:20,ownSessionId=-1

SRR: moduleName=first, subsystem=SrrFramework, fileName=SrrModCoord.x3d, ssVersion=0100, instanceId=first.Coord  
SRR: action=sendEvent;destination=first-\*.Control;field=carriedKeysChanged  
SRR: tell all SRR objects that carried keys changed

SRR:END

Man sieht, wie der Reihe nach folgendes passiert

1. Das Modul reagiert auf den Click des Users und sendet die "takeKeys"-Anforderung an den SrrKeyContainer.
2. Was hier folgt sieht man nicht, da der Tracelevel der Objekte auf "1" gestellt ist: der Key Container stellt sicher, dass dieser Schlüssel nur von einer Szeneninstanz entnommen wird (indem er mit der .Moc-Instanz kommuniziert) und löscht den entsprechenden Schlüssel in allen Szeneninstanzen aus der Liste der "contained keys" (in diesem Beispiel haben wir allerdings nur den single-user-mode, was man an der Ausgabe "ownSessionId=-1" in allen Tracepunkten erkennt).
3. Das Modul reagiert auf die Änderung der "contained keys" und stellt diese graphisch dar.
4. Das SrrKeyContainer-Objekt reicht die Key-ID des entnommenen Schlüssels an den Modulkoordinator weiter, welcher ihn sofort an den SrrController durchreicht.
5. Was der SrrController tut, sieht man wieder nicht, da der Tracelevel zu klein ist. Der SrrController löscht diesen Schlüssel aus der Liste der "carried keys" und gibt das Ergebnis an den Rahmen weiter.
6. Der Rahmen stellt die neue Liste der "carried keys" im "carried keys HUD" dar (wo man dann draufklicken kann, um den Schlüssel wieder wo abzulegen)
7. Weiters informiert der SrrController alle Modulkoordinatoren und somit alle SRR-Objekte darüber, dass sich die Liste der "carried keys" geändert hat. SrrLock Objekte hören auf diese Information, da sie auch auf "getragene Schlüssel" mit einem "unlock"-Ereignis reagieren können. Hier sieht man nicht, ob ein SrrLock-Objekt auf diese Information reagiert hat, da alle Objekte auf Tracelevel=1 gesetzt sind.