ZSQtLib

Object Pool Model

Benutzerhandbuch

Copyright

©2008 ZeusSoft, Ing. Büro Bauer. Alle Rechte vorbehalten.

Dieses Handbuch sowie die darin beschriebene Software unterliegen lizenzrechtlichen Bestimmungen und dürfen nur in Übereinstimmung mit dieser Lizenzvereinbarung verwendet oder kopiert werden. Die Angaben und Daten in diesem Handbuch dienen ausschließlich Informationszwecken und gelten unter Vorbehalt. ZeusSoft, Ing. Büro Bauer übernimmt dafür keinerlei Haftung oder Gewährleistung und auch keine Verantwortung für Folgeschäden auf Grund von Fehlern oder Ungenauigkeiten dieses Handbuchs.

Außerhalb der Lizenzeinräumung darf ohne ausdrückliche, schriftliche Genehmigung von ZeusSoft, Ing. Büro Bauer kein Teil dieser Publikation auf irgendeine Weise reproduziert oder auf einem Medium gespeichert oder übertragen werden, weder elektronisch noch mechanisch, auf Band oder auf andere Weise.

Marken

Qt Software ist ein Plattform übergreifender Framework und eingetragenes Markenzeichen der Fa. Trolltech in Norwegen. Alle anderen Marken- und Produktnamen sind Marken oder eingetragene Marken ihrer jeweiligen Besitzer.

Inhaltsverzeichnis

1 Einführung ... 5
2 Methoden und Klassen-Referenz ... 7
2.1 Erläuterungen zum Klassen-Modell ... 7
2.2 Type Definitionen und Globale Methoden ... 10
2.2.1 Datentype (enum) EObjPoolEntryType ... 10
2.3 Klasse *CObjPoolTreeEntry* ... 11
2.3.1 Attribute ... 11
2.3.2 Konstruktoren und Destruktor ... 13
2.3.2.1 *CObjPoolTreeEntry*() ... 13
2.3.2.2 *CObjPoolTreeEntry*(EEntryType*, const QString&, const QString&*) ... 13
2.3.2.3 *~CObjPoolTreeEntry*() ... 13
2.3.3 Operationen ... 14
2.4 Klasse *CObjPoolListEntry* ... 14
2.4.1 Attribute ... 14
2.4.2 Konstruktoren und Destruktor ... 15
2.4.2.1 *CObjPoolListEntry*() ... 15
2.4.2.2 *CObjPoolListEntry*(*const QString&, const QString&, const QString&, const QString&*) ... 15
2.4.2.3 *~CObjPoolListEntry*() ... 16
2.4.3 Operationen ... 16
2.5 Klasse *CModelObjPool* ... 16
2.5.1 Datentype (enum) *EColumn* ... 16
2.5.2 Attribute ... 17
2.5.3 Konstruktoren und Destruktor ... 18
2.5.3.1 CModelObjPool(const QString&, bool, QObject*) ... 18
2.5.4 Signale ... 18
2.5.4.1 clearing(QObject*) ... 18
2.5.4.2 objectInserted(QObject*, CObjPoolListEntry*) ... 18
2.5.4.3 objectChanged(QObject*, CObjPoolListEntry*) ... 19
2.5.4.4 nameSpaceInserted(QObject*, CObjPoolTreeEntry*) ... 19
2.5.4.5 nameSpaceChanged(QObject*, CObjPoolTreeEntry*) ... 19
2.5.5 Operationen ... 19
2.5.5.1 *isDescendant(const QString&, const QString&, const QString&, bool)* ... 19
2.5.5.2 *getMutex()* ... 20
2.5.5.3 lock() ... 20
2.5.5.4 unlock() ... 20
2.5.5.5 setObjName(const QString&) ... 20
2.5.5.6 getObjName() const ... 21
2.5.5.7 isTreeModel() const ... 21
2.5.5.8 getObjCount() ... 21
2.5.5.9 getRoot() ... 21
2.5.5.10 clear(bool) ... 21
2.5.5.11 addObj(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState) ... 22
2.5.5.12 addObj(const QString&, const QString&, const QString&, QString*, QObject*, EEnabled, EStateOnOff) ... 23
2.5.5.13 addObj(const QString&, const QString&, const QString&, const QString&, int, EEnabled, EStateOnOff, EObjState) ... 23
2.5.5.14 addNameSpace(const QString&, const QString&, EEnabled, EStateOnOff) ... 24
2.5.5.15 removeObj(int) ... 25
2.5.5.16 removeObj(const QString&, const QString&, const QString&, const QString&) ... 25
2.5.5.17 removeObj(QObject*) ... 26
2.5.5.18 changedObj(int) ... 26
2.5.5.19 updateObj(int, EEnabled, EStateOnOff, EObjState) ... 26
2.5.5.20 updateObj(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState) ... 27
2.5.5.21 updateObj(const QString&, const QString&, EEnabled, EStateOnOff, EObjState) ... 27

2.5.5.22 updateObj(const QString&, EEnabled, EStateOnOff, EObjState) 28
2.5.5.23 updateNameSpace(const QString&, const QString&, EEnabled, EStateOnOff, bool).......... 28
2.5.5.24 updateNameSpace(const QString&, const QString&, EEnabled, EStateOnOff, bool).......... 29
2.5.5.25 findObjId(const QString&, const QString&, const QString&, const QString&).......... 29
2.5.5.26 findObjId(const QString&, const QString&).......... 30
2.5.5.27 findObjId(const QString&).......... 30
2.5.5.28 isActive(int).......... 30
2.5.5.29 setEnabled(int, EEnabled).......... 31
2.5.5.30 setEnabled(const QString&, const QString&, EObjPoolEntryType, EEnabled, bool).......... 31
2.5.5.31 getEnabled(int).......... 32
2.5.5.32 isEnabled(int).......... 32
2.5.5.33 setState(int, EStateOnOff).......... 32
2.5.5.34 setState(const QString&, const QString&, EObjPoolEntryType, EStateOnOff, bool).......... 32
2.5.5.35 getState(int).......... 33
2.5.5.36 setObjState(int, EObjState).......... 33
2.5.5.37 getObjState(int).......... 33
2.5.5.38 setObj(int, QObject*).......... 33
2.5.5.39 getObj(int).......... 34
2.5.5.40 findObj(const QString&, const QString&, const QString&, const QString&).......... 34
2.5.5.41 findObj(const QString&, const QString&).......... 34
2.5.5.42 findObj(const QString&).......... 35
2.5.5.43 setFileName(const QString&).......... 35
2.5.5.44 getFileName().......... 35
2.5.5.45 recall(const QString&).......... 35
2.5.5.46 save(const QString&).......... 36
2.5.5.47 getListEntry(int).......... 36
2.5.5.48 findListEntry(int).......... 36
2.5.5.49 findListEntry(QObject*).......... 37
2.5.5.50 findTreeEntry(const QString&, const QString&, const QString&, const QString&).......... 37
2.5.5.51 findTreeEntry(const QString&, const QString&).......... 38
2.5.5.52 findTreeEntry(const QString&, const QString&).......... 38
2.5.5.53 addListEntry(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState).......... 38
2.5.5.54 updateListEntry(CObjPoolListEntry*, EEnabled, EStateOnOff, EObjState).......... 39
2.5.5.55 updateListEntry(CObjPoolListEntry*, EEnabled, EStateOnOff, EObjState).......... 40
2.5.5.56 addTreeEntry(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState).......... 40
2.5.5.57 addTreeEntry(const QString&, const QString&, EEnabled, EStateOnOff, EObjState).......... 41
2.5.5.58 addTreeEntry(CObjPoolTreeEntry*, EObjPoolEntryType, const QString&, EEnabled, EStateOnOff, EObjState).......... 41
2.5.5.59 updateTreeEntry(CObjPoolTreeEntry*, EEnabled, EStateOnOff, EObjState).......... 42
2.5.5.60 updateTreeEntry(CObjPoolTreeEntry*, EEnabled, EStateOnOff, bool).......... 42
2.5.5.61 clearTreeEntry(CObjPoolTreeEntry*, bool).......... 43
2.5.5.62 saveTreeEntry(QDomDocument&, QDomElement&, CObjPoolTreeEntry*).......... 43
2.5.5.63 rowCount(const QModelIndex&).......... 44
2.5.5.64 columnCount(const QModelIndex&).......... 44
2.5.5.65 data(const QModelIndex&, int).......... 44
2.5.5.66 setData(const QModelIndex&, const QVariant&, int).......... 45
2.5.5.67 index(int, int, const QModelIndex&).......... 45
2.5.5.68 parent(const QModelIndex&).......... 45
2.5.5.69 headerData(int, Qt::Orientation, int).......... 46
2.5.5.70 flags(const QModelIndex&).......... 46
3 Anwendungsbeispiele.......... 47

1 Einführung

In verteilten Multi-Threading oder Multi-Prozess Systemen werden Objekte oft zentral in einer Komponente innerhalb einer Liste verwaltet. Damit Client-Komponenten aus anderen Threads, Prozessen oder gar über eine Netzwerk-Infrastruktur hinweg Nachrichten an ein ausgewähltes Objekt der Server-Komponente schicken oder Informationen von diesem Objekt auslesen kann, muss das Objekt eindeutig innerhalb der Server-Komponente identifiziert werden. Die Objekte müssen also eine systemweit eindeutige ID (Identifikationsnummer) erhalten. Um diese systemweit eindeutige ID zu erhalten, gibt es verschiedene Lösungsansätze, von denen drei im Folgenden erwähnt werden.

Lösungsansatz 1:

Die Server-Komponente hält die Objekte in einer Liste. Der Listen-Index entspricht der ID, wird fest vorgegeben und über ein Header File in Form von Konstanten (z.B. als enum) exportiert.

Vorteil: Der Server kann sehr schnell über die ID, die dem Listen-Index entspricht, auf die Objekte zugreifen.
Nachteil: Wird ein Objekt entfernt, hinzugefügt oder wird die Reihenfolge der Objekte in der Liste geändert, muss das enum angepasst werden und alle Software-Komponenten müssen neu kompiliert werden.

Lösungsansatz 2:

Auch hier hält die Server-Komponente die Objekte in einer Liste. Allerdings werden hier die Objekte eindeutig über einen Namen gekennzeichnet.

Vorteil: Wird ein Objekt entfernt, hinzugefügt oder wird die Reihenfolge der Objekte in der Liste geändert, muss kein enum angepasst werden und alle Software-Komponenten müssen nicht neu kompiliert werden.

Nachteil: Der Server kann nicht mehr sehr schnell auf die Objekte zugreifen sondern muss diese anhand ihres Namens suchen. Auch wenn dies über Hash-Werte geschieht, ist dies in jedem Fall langsamer als der Zugriff über den Listen-Index. Außerdem muss die Anwendung damit fertig werden, dass Client Komponenten noch auf Objekte zugreifen wollen, die es gar nicht gibt, was natürlich dadurch umgangen werden kann, dass für die Namen der Objekte Konstanten vergeben werden, die ebenfalls über Header-Files exportiert und die Client-Komponenten neu kompiliert werden.

Lösungsansatz 3:

Hier hält der Server die Objekte sowohl in einem hierarchisch geordneten Baum als auch in einer Liste. Die Objekte werden eindeutig über einen Namen gekennzeichnet, sind aber auch über einen Listen-Index adressierbar. Die Namen werden hierarchisch gegliedert und Teile des Namens werden durch einen Separater getrennt. Jede Sektion des Namens entspricht einem Ast in dem Objekt-Baum. Will eine Client-Komponente auf ein Objekt zugreifen, so

könnte sie das Objekt über den Namen, als auch über den Listen-Index ansprechen, sofern ihr der Listen-Index bekannt ist. Den Listen-Index erhält die Client-Komponente durch eine Art Registrier-Mechanismus auf das gewünschte Objekt, in dem einmalig eine Anfrage an den Server gestellt wird, anhand des Objekt-Namens den Listen-Index zurückzugeben. Diesen Listen-Index verwendet die Client-Komponente im Folgenden als eine Art Handle auf das Objekt.

V*orteil:* Wird ein Objekt entfernt, hinzugefügt oder wird die Reihenfolge der Objekte in der Liste geändert, muss kein enum angepasst werden und alle Software-Komponenten müssen nicht neu kompiliert werden. Nur der erste Zugriff auf das Objekt, der über den Namen geschieht, ist „langsam". Alle weiteren Zugriffe erfolgen über den Listen-Index und sind damit so schnell wie im ersten Lösungsansatz. Da die Objekte hierarchisch gegliedert sind, können sie darüber hinaus innerhalb eines User-Interfaces sehr übersichtlich dargestellt werden.

Nachteil: Zusätzlicher Aufwand innerhalb der Client-Komponente, um den Listen-Index von der Server-Komponente anzufordern. Wird das Objekt überhaupt nur ein einziges Mal von der Client Komponente benötigt, ist die Adressierung des Objekts umständlich und langsam. Da die Objekte hierarchisch gegliedert in einem Baum gehalten werden, kann ein Hash-Wert nicht vergeben werden (außer, man würde noch eine weitere Hash-Liste einführen). Auch hier muss die Anwendung damit fertig werden, dass Client Komponenten noch auf Objekte zugreifen wollen, die es gar nicht gibt.

Das Objekt Pool Model wurde entworfen, um den dritten Lösungsansatz zu realisieren.

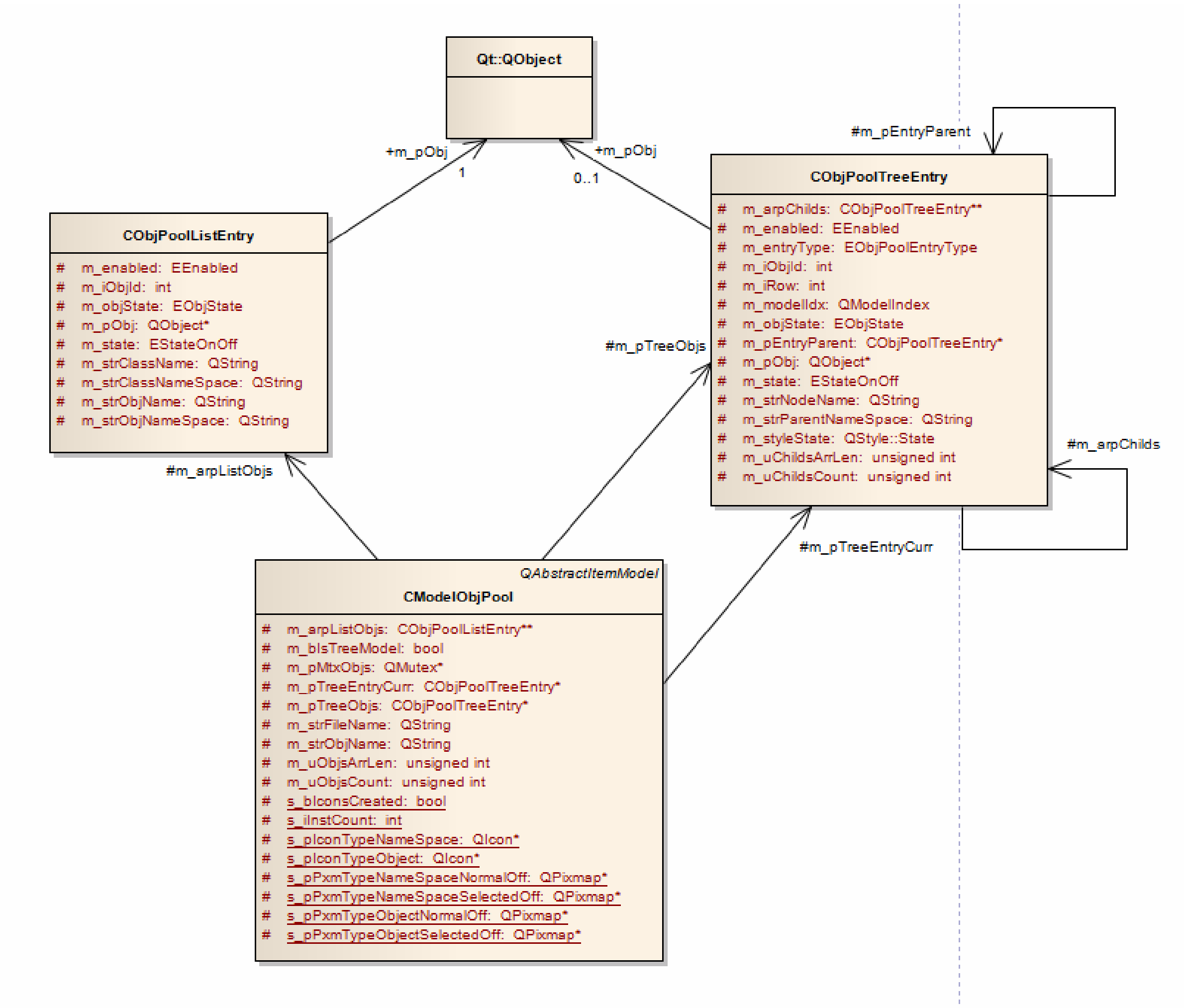
Es wurde als erstes für das Remote-Tracing eingeführt, da es hier wichtig war, die Objekte hierarchisch in einem Baum gliedern zu können, um die Übersicht über die in der zu „tracenden" Server-Anwendung angelegten Admin-Objekte zu behalten. Ferner war es wichtig, den anfallenden Datentransfer zwischen Server und Client so weit wie möglich zu reduzieren, weshalb nicht immer der Name eines Objekts sondern nur seine ID übertragen werden sollte, um die Trace-Ausgaben dem Objekt zuordnen zu können. Nebenbei erlaubte die Verwendung des Listen-Index als ID für die Objekte, dass sowohl die Trace-Clients als auch die Trace-Server die Objekte schnellstmöglich finden.

Trace-Clients fordern nach einem erfolgreichen Verbindungsaufbau zum Trace-Server einmal eine Liste aller im Server angelegten Trace-Admin-Objekte an. Dabei wird für jedes Objekt sowohl der durch Trennzeichen hierarchisch gegliederte Objekt-Name als auch die ID (der Listen-Index) des Objekts an den Client übertragen. Nachdem die Liste der Objekte einmal beim Client angekommen war, erfolgte die Kommunikation zwischen Server und Client nur noch über die ID's der Objekte.

Das Remote-Tracen stellte noch eine weitere Anforderung an das Objekt Pool Model. Und zwar können Admin-Objekte dynamisch zur Laufzeit des Programms erzeugt und zerstört werden. Beim Erzeugen des Objekts muss das Objekt mit seiner ID an die angeschlossenen Clients übertragen und in den Objekt-Baum eingetragen werden. Beim Zerstören darf das Objekt allerdings nicht aus dem Objekt-Baum entfernt werden, sondern es muss lediglich als „not alive" markiert werden. Wird das Objekt zu irgendeinem späteren Zeitpunkt wieder erzeugt, muss das Objekt wieder seine alte ID erhalten - muss also sowohl im Baum als auch in der Liste wieder an derselben Stelle eingefügt werden.

2 Methoden und Klassen-Referenz

Das nachfolgende Bild zeigt das UML Model für das Objekt Pool Model:



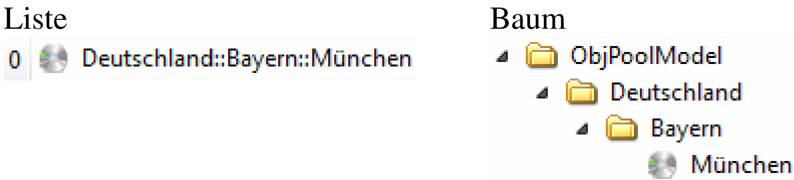
2.1 Erläuterungen zum Klassen-Modell

Die Klasse *CModelObjPool* ist die zentrale Klasse, die sowohl eine Liste als auch einen Baum enthält. Die Liste enthält Verweise auf Instanzen der Klasse *CObjPoolListEntry*. Der Baum enthält einen Verweis auf eine Instanz der Klasse *CObjPoolTreeEntry*, die als Wurzel (Root) für den Baum dient. Die Endpunkte eines Baums – also die Blätter – adressieren über die *ObjId* den zugehörigen Listeneintrag. Der Root-Eintrag sowie die Äste verweisen auf keinen Eintrag innerhalb der Liste. Ihre *ObjId* ist auf einen ungültigen Wert (-1) gesetzt. Der Root-Eintrag und die Äste, die nicht auf einen Listen-Eintrag verweisen, werden als NameSpace-Entries bezeichnet, denn sie dienen lediglich zur Organisation der Objekte im Baum. Die Blätter des Baums, die über die *ObjId* auf einen Listen-Eintrag verweisen, werden als Object-Entries bezeichnet.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

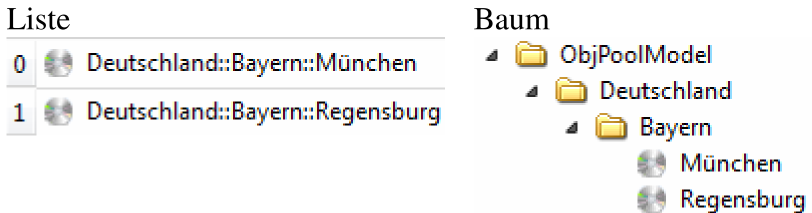
Das Object-Pool-Model wurde ursprünglich für das Remote-Tracen verwendet. Deshalb finden sich noch einige Artefakte (Datentypen, Methodenaufrufe, Schnittstellen) innerhalb der Klassendefinition wieder, die speziell für das Remote-Tracing eingeführt wurden. Da diese Artefakte nicht weiter stören, es aber einiges an Aufwand bedeuten würde, die Funktionalität in das Trace-Modul zu übertragen, wurden diese Artefakte im Object-Pool-Model beibehalten. An den entsprechenden Stellen findet sich der Hinweis „für Tracing“.

Im Nachfolgenden soll verdeutlicht werden, was geschieht, wenn Objekt-Referenzen dem Objekt Pool Model hinzugefügt werden. Hierzu nehmen wir an, dass das Model dazu verwendet werden soll, Städte nach Land und Bundesland geordnet zu speichern. Das Model enthält noch keine Einträge und das erste Objekt, das dem Model hinzugefügt wird, ist die Stadt München, die die Hauptstadt des deutschen Bundeslands Bayern ist. Um das Objekt „München“ in den Baum einzuordnen, erhält das Objekt den ParentNameSpace „Deutschland::Bayern“ und den Objekt-Namen „München“. Nachdem die Stadt München dem Model hinzugefügt wurde, haben Liste und Baum folgende Einträge:



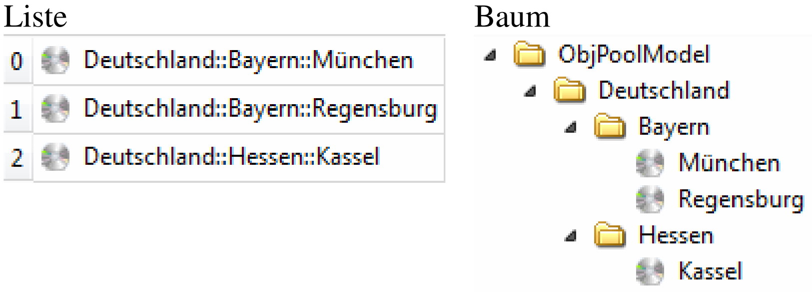
Der Root-Eintrag erhält als Voreinstellung den Objekt-Namen, der dem Model-Konstruktur beim Anlegen des Models übergeben wurde (hier „ObjPoolModel“). Zu beachten ist ferner, dass nur der Listeneintrag *CObjPoolListEntry* automatisch eine Referenz auf das Objekt München erhält. Soll dies auch für das Blatt im Baum, das die Stadt München repräsentiert geschehen, muss dies explizit durch den Methoden-Aufruf *setObj* der Klasse *CObjPoolTreeEntry* geschehen. Im Übrigen kann jedem Ast im Baum über den *setObj* Aufruf eine Referenz auf ein Objekt zugewiesen werden. Davon wird z.B. innerhalb des Subsystems ZSPhysVal der ZSQtLib Gebrauch gemacht, in der z.B. eine Instanz einer physikalischen Größenart (wie „Electricity“) ihre physikalischen Einheiten (wie „µV“, „mV“, „V“, etc.) selbst verwaltet, um ein vom Objekt Pool Model losgelöstes, abstraktes Interface anzubieten.

Wird nun die bayerische Stadt Regensburg mit dem ParentNameSpace „Deutschland::Bayern“ und dem Objekt-Namen „Regensburg“ hinzugefügt, besitzen Liste und Baum folgende Einträge:



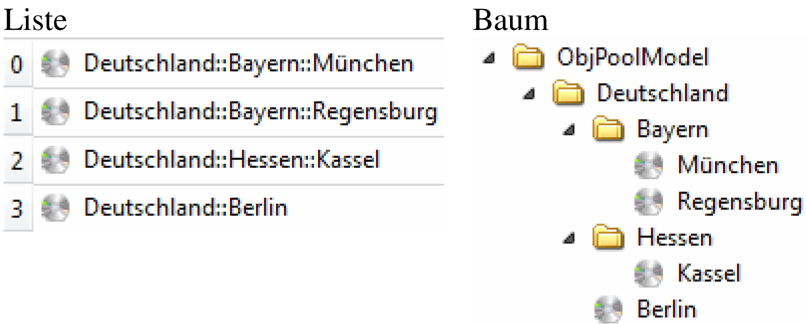
Wird nun die Stadt Kassel als Hauptstadt von Hessen mit dem ParentNameSpace „Deutschland::Hessen“ und dem Objekt-Namen „Kassel“ hinzugefügt, besitzen Liste und Baum folgende Einträge:

Liste Baum



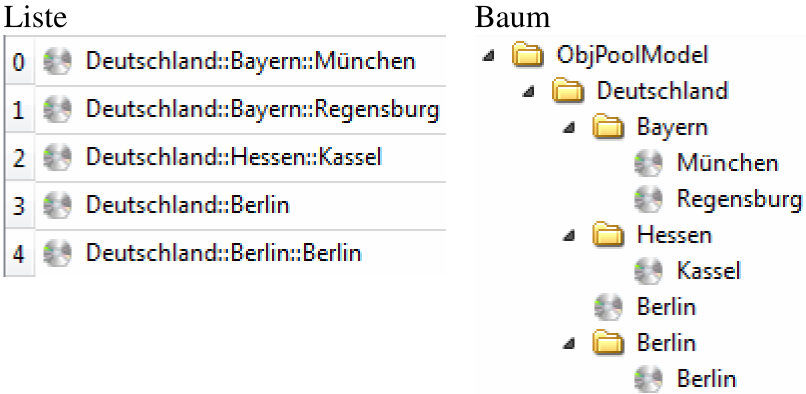
Wird nun noch die Stadt Berlin hinzugefügt – fälschlicherweise ohne Angabe des Bundeslands Berlin – also mit dem ParentNameSpace „Deutschland“ und dem Objekt-Namen „Berlin“ - besitzen Liste und Baum folgende Einträge:

Liste Baum



Zu guter Letzt wird die Stadt Berlin nochmals hinzugefügt. Dieses Mal mit Angabe des Bundeslands Berlin – also mit dem ParentNameSpace „Deutschland::Berlin“ und dem Objekt-Namen „Berlin“. Anschließend besitzen Liste und Baum folgende Einträge:

Liste Baum



Am Ende besitzt der NameSpace-Entry Deutschland drei untergeordnete Äste (Knoten), („Bayern“, „Hessen“ und „Berlin“) sowie ein untergeordnetes Blatt („Berlin“). Dabei sind „Berlin“ taucht unterhalb von Deutschland also zwei Mal auf – als Blatt (Object-Entry) als auch als Knoten (NameSpace-Entry). Ein Ast kann sowohl weitere Äste (NameSpace-Entries) als auch Blätter (Object-Entries) als „Childs“ enthalten. „Bayern“ besitzt zwei Blätter (Object-Entries) als „Childs“ – nämlich „München“ und „Regensburg“. „Hessen“ hat ein

Blatt (Object-Entry) als „Child“ – nämlich „Kassel“. Der Knoten „Berlin“ hat ebenfalls ein Blatt als „Child“ – nämlich wieder „Berlin“.

Die Liste besitzt fünf Einträge – „München, „Regensburg“, „Kassel“ und zwei mal „Berlin“. Die ID der Städte – also ihre Listenindex – wurde entsprechend der Reihenfolge zugewiesen, in denen die Objekte dem Modell hinzugefügt wurden. Wird ein Objekt dem Model hinzugefügt, dessen NameSpace sowie Objekt-Name bereits im Model vorhanden ist, wird kein neuer Eintrag erstellt und auch der Verweis auf das Duplikat wird nicht übernommen. „Deutschland::Hessen::Kassel“ kann also nur einmal sowohl im Baum als auch in der Liste auftauchen. Die Objekt-ID ist mit dem kompletten Namen des Objekts, der sich aus dem ParentNameSpace (kann auch als Pfad bezeichnet werden) sowie dem Objekt-Namen zusammensetzt.

2.2 Type Definitionen und Globale Methoden

2.2.1 Datentype (enum) EObjPoolEntryType

Wie in der Einführung zum Klassenmodell beschrieben, gibt es drei unterschiedliche Einträge innerhalb des Baums eines Object Pool Models.

EObjPoolEntryTypeRoot
: Jedes Model hat mindestens einen Eintrag. Dies ist der Wurzeleintrag unterhalb dem alle anderen Einträge liegen. Auf der Ebene des Wurzeleintrags gibt es nur diesen einen Eintrag. Alle weiteren Einträge sind „Nachfolger“ des Wurzeleintrags.

EObjPoolEntryTypeObject
: Einträge dieses Typs entsprechen den Blättern im verzweigten Baum. Hinter Einträgen dieses Typs versteckt sich ein echtes Objekt (also eine Instanz einer von QObject abgeleiten Klasse). Über die *ObjId* des Baumeintrags kann auf den Listeneintrag und über diesen auf das Objekt zugegriffen werden. Zu beachten ist jedoch, dass die Objekte zur Laufzeit zerstört werden können, der Baum- und der Listeneintrag jedoch erhalten bleiben. Bevor also wirklich auf das Objekt zugegriffen wird, muss geprüft werden, ob der Objekt-Verweis gültig (!= NULL) ist.

EObjPoolEntryTypeNameSpace
: Einträge dieses Typs entsprechen Ästen (Knoten) innerhalb des verzweigten Baums. Hinter Einträgen dieses Typs versteckt sich (normalerweise) kein echtes Objekt. Falls man jedoch eine Objekt-Klasse benötigt, um die „Kinder“ (Blätter des Astes) selbst zu verwalten, kann man explizit eine Objekt-Referenz zuordnen. Knoten verweisen niemals auf einen Listeneintrag und ihre *ObjId* ist immer ungültig (= -1).

EObjPoolEntryTypeCount
: *Count*-Enum Einträge werden dazu verwendet, um Schleifen zu programmieren oder Felder anzulegen.

EObjPoolEntryTypeUndefined
: Dieser Wert zeigt einen undefinierten Typen eines Baum- oder Listeneintrags an.

2.3 Klasse *CObjPoolTreeEntry*

Instanzen dieser Klasse repräsentieren die Äste (Knoten) und Blätter des Baums.

2.3.1 Attribute

Die Attribute Enabled, State, und ObjState werden für das Tracing benötigt. Dort werden die aktuellen Einstellungen in einem XML-File gespeichert und können aus dem XML-File wieder restauriert werden. Da jedoch die Trace-Admin-Objekte zum Zeitpunkt des Restaurierens der Einstellungen unter Umständen noch gar nicht erzeugt wurden, müssen die Einstellungen an anderer Stelle zwischengespeichert werden. Hierzu hätte „Schatten-Trace-Admin-Objekte" angelegt werden können. Es war aber wesentlich einfacher, die drei Zustände der Trace-Admin-Objekte innerhalb des Object-Pool-Models als Attribute der Baum- und Listeneinträge abzulegen.

m_entryType Datentype: EObjPoolEntryType

Legt fest, ob es sich bei dem Eintrag um die Wurzel (Root) des Baumes (EntryTypeRoot), einen Knoten (EntryTypeNameSpace) oder um ein Blatt (EntryTypeObject) handelt.

m_strParentNameSpace Datentype: QString

Falls es sich bei dem Eintrag um ein Blatt handelt (EntryTypeObject), also ein gültiger Verweis über die *ObjId* auf einen Listeneintrag besitzt, entspricht *ParentNameSpace* dem vollständigen Pfad-Namen des Objekts ohne dem Namen des Wurzel-Eintrags und ohne den Namen des Objekts selbst. Wurde das Objekt mit dem *NameSpace* „Deutschland::Bayern" und dem Objekt-Namen „München" hinzugefügt, würde *ParentNameSpace* also den Wert „Deutschland::Bayern" besitzen.
Falls es sich bei dem Eintrag um einen Knoten handelt (EntryTypeNameSpace), als nicht auf einen Listeneintrag verweist (*ObjId* = -1), entspricht *ParentNameSpace* dem vollständigen Pfad-Namen des übergeordneten Knotens ohne dem Namen des Wurzel-Eintrags und ohne den Namen des Knotens selbst. Für den Knoten „Deutschland::Bayern" würde *ParentNameSpace* also den Wert „Deutschland" besitzen.

m_strNodeName Datentype: QString

Falls es sich bei dem Eintrag um ein Blatt handelt (EntryTypeObject), also ein gültiger Verweis über die *ObjId* auf einen Listeneintrag besitzt, entspricht *NodeName* dem Namen des Objekts. Wurde das Objekt mit dem *NameSpace* „Deutschland::Bayern" und dem Objekt-Namen „München" hinzugefügt, würde *NodeName* also den Wert „München" besitzen.
Falls es sich bei dem Eintrag um einen Knoten handelt (EntryTypeNameSpace), als nicht auf einen Listeneintrag verweist (*ObjId* = -1), entspricht *NodeName* dem Namen des Knotens. Für den Knoten „Deutschland::Bayern" würde *NodeName* also den Wert „Bayern" besitzen.

m_enabled Datentype: EEnabled

(für Tracing) Objekte und NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts bzw. NameSpaces wird hier abgelegt. Für Knoten-Einträge ist der TreeEntry die einzig mögliche Stelle, an der dieser Zustand gespeichert werden kann (es verbirgt sich ja weder eine Listen-Eintrag noch ein echtes Objekt hinter dem Knoten). Für Blätter (EntryTypeObject) wird dafür gesorgt, dass auch der Listen-Eintrag als auch das Objekt selbst (wenn es denn aktuell instanziiert ist) denselben Zustand besitzen.

m_state Datentype: EStateOnOff

(für Tracing) Objekte und NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts bzw. NameSpaces wird hier abgelegt. Für Knoten-Einträge ist der TreeEntry die einzig mögliche Stelle, an der dieser Zustand gespeichert werden kann (es verbirgt sich ja weder eine Listen-Eintrag noch ein echtes Objekt hinter dem Knoten). Für Blätter (EntryTypeObject) wird dafür gesorgt, dass auch der Listen-Eintrag als auch das Objekt selbst (wenn es denn aktuell instanziiert ist) denselben Zustand besitzen.

m_objState Datentype: EObjState

(für Tracing) Dieses Attribut entspricht dem Zustand des Objekts. Wird bei Programmstart ein Model aus dem XML-File erzeugt, existieren die Objekte zu diesem Zeitpunkt unter Umständen noch gar nicht und der *ObjState* dieser Objekte ist noch Undefined. Wird das Objekt instanziiert und dem Model hinzugefügt, ist der Zustand des Objekts Created. Wird das Objekt zerstört, ist der Zustand des Objekts Destroyed.

m_iObjId Datentype: int

Falls es sich bei dem Eintrag um ein Blatt handelt (EntryTypeObject), verweist *ObjId* auf den zugehörigen Listeneintrag.

Falls es sich bei dem Eintrag um einen Knoten handelt (EntryTypeNameSpace), ist die *ObjId* ungültig und besitzt den Wert -1.

m_pObj Datentype: QObject*

Der „normale“ Weg, um auf ein Objekt zuzugreifen, das dem Model hinzugefügt wurde, geht über die *ObjId* des Objekts und damit über die Liste des Models. Der Verweis auf das Objekt wird innerhalb der Baum-Einträge nicht mitgeführt. Hier ist das Datum *pObj* also normalerweise immer NULL. Allerdings kann dieses Attribut explizit gesetzt werden, falls sich z.B. hinter den Knoten eines Baums auch echte Objekte verstecken, wie es z.B. für die Größenarten der physikalischen Einheiten der Fall ist (siehe ZSPhysVal).

m_pEntryParent Datentype: CObjPoolTreeEntry*

Verweist auf den übergeordneten Knoten des Eintrags. Für den Wurzeleintrag steht hier der Wert NULL.

m_iRow Datentype: int

Besitzt der Knoten einen übergeordneten Knoten, was für alle Knoten außer dem Wurzel-Eintrag der Fall ist, wird der Knoten als „Child“ in einer Liste des übergeordneten Knotens mitgeführt. Der Index innerhalb der Liste des Parents wird im *Row*-Attribut abgelegt und entspricht der Zeilennummer des Eintrags, wenn der übergeordnete Knoten innerhalb einer Table-View ausgegeben wird.

m_uChildsCount Datentype: unsigned int

Anzahl der Nachfolger (Chils) des Knotens.

m_uChildsArrLen Datentype: unsigned int

Länge der Liste, in der die Nachfolger (Childs) des Knotens abgelegt werden. Diese ist immer größer oder gleich der Anzahl der Nachfolger. Soll der Liste ein neues Element hinzugefügt werden, sind aber bereits alle Elemente belegt, wird die Liste vergrößert. Dabei wird die Liste nicht nur um ein Element vergrößert, sondern es wird versucht, doppelt so viele Elemente zusätzlich zu reservieren, wie die Liste aktuell besaß. Dabei ist die Anzahl der zusätzlich zu reservierenden Elemente allerdings auf eine sinnvolle, maximale Anzahl begrenzt (nicht jedoch die Gesamtlänge der Liste).

m_arpChilds Datentype: CObjPoolTreeEntry*

Liste mit Verweisen auf die dem Knoten untergeordneten Knoten (Childs).

m_modelIdx Datentype: QModelIndex
Damit der Baum entsprechend der Model/View Architektur durch die Klasse QTreeView dargestellt werden kann, wird der Model-Index des Elements mitgeführt.
m_styleState Datentype: QStyle::State
(für Tracing) Dieses Attribut wird zur korrekten Darstellung über den Delegate (siehe Qt's Model/View Architektur) benötigt.

2.3.2 Konstruktoren und Destruktor

2.3.2.1 *CObjPoolTreeEntry*()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse. Alle Elemente werden auf einen Default-Wert gestellt die anzeigen, dass die Instanz noch nicht verwendet wird (EntryType = Undefined, Parent = NULL, ChildCount = 0, ObjState = Undefined, pObj = NULL, ObjId = -1).

2.3.2.2 *CObjPoolTreeEntry*(EEntryType*, const QString&, const QString&*)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse. Es werden zwar der Type des Knotens und die Namen für das Objekt übernommen, die Instanz wird aber als noch nicht verwendet markiert (Parent = NULL, ChildCount = 0, ObjState = Undefined, pObj = NULL, ObjId = -1).
Parameter:
i_entryType (IN) Datentype: EEntryType
Legt fest, ob der Eintrag als Wurzel (EntryTypeRoot) des Baums, als Ast (Knoten, EntryTypeNameSpace) oder als Blatt (EntryTypeObject) verwendet werden soll.
i_strParentNameSpace Datentype: QString
Um das Objekt im Baum einzuordnen, kann dem Objekt ein Namensbereich seiner Klasse, der Klassenname sowie ein Namensbereich für das Objekt selbst mitgegeben werden. Diese drei Attribute werden zu einem ParentNameSpace String zusammengefügt, unter dem das Objekt im Baum einsortiert wird. Beim Einsortieren in den Baum wird der ParentNameSpace auf den Namensbereich des übergeordneten Knotens gesetzt.
i_strNodeName Datentype: QString
Entspricht dem Namen des Knotens. Handelt es sich beim dem Eintrag um ein Blatt (EntryTypeObject), entspricht der *NodeName* auch dem eigentlichen Namen (ohne übergeordnete Namensbereiche) des Objekts.

2.3.2.3 *~CObjPoolTreeEntry*()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zerstört die Instanz. Wird vom Model aufgerufen, wenn das Model selbst zerstört wird. Ein Eintrag im Baum lebt so lange, wie auch das Model lebt und kann nicht zwischenzeitlich gelöscht werden. Über die Liste referenzierte Objekte können zur Laufzeit zerstört und wieder

neu erzeugt werden. Werden sie dem Model aufs Neue hinzugefügt, werden sie wieder vom selben Eintrag verwaltet und erhalten so dieselbe ID wie zuvor.

2.3.3 Operationen

Die Operationen der Klasse beschränken sich überwiegend darauf, die Attribute der Klasse zu setzen und zu lesen sowie Child-Entries hinzuzufügen, zu finden und wieder zu entfernen. Da die Methoden trivial sind, lohnt die Mühe nicht, die Methoden einzeln aufzuführen.

2.4 Klasse ***CObjPoolListEntry***

Instanzen dieser Klasse entsprechen den Listeneinträgen des Models.

2.4.1 Attribute

Die Attribute Enabled, State, und ObjState werden für das Tracing benötigt. Dort werden die aktuellen Einstellungen in einem XML-File gespeichert und können aus dem XML-File wieder restauriert werden. Da jedoch die Trace-Admin-Objekte zum Zeitpunkt des Restaurierens der Einstellungen unter Umständen noch gar nicht erzeugt wurden, müssen die Einstellungen an anderer Stelle zwischengespeichert werden. Hierzu hätte „Schatten-Trace-Admin-Objekte“ angelegt werden können. Es war aber wesentlich einfacher, die drei Zustände der Trace-Admin-Objekte innerhalb des Object-Pool-Models als Attribute der Baum- und Listeneinträge abzulegen.

m_strClassNameSpace Datentype: QString
: (für Tracing) Klassen werden oft in Namensbereiche (NameSpaces) eingeordnet, um Doppeldeutigkeiten der Klassennamen zu vermeiden.

m_strClassName Datentype: QString
: (für Tracing) Entspricht dem Klassennamen des Trace-Admin-Objekts.

m_strObjNameSpace Datentype: QString
: (für Tracing) Häufig werden sehr viele Instanzen einer Klasse angelegt (z.B. QWidget’s). Um die Übersicht zu behalten, können diese Instanzen in Namensbereiche gegliedert werden (für Widgets könnte das z.B. der Name des übergeordneten Dialogs sein).

Anmerkung zu *ClassNameSpace*, *ClassName* und ObjNameSpace:
: Werden Objekte dem Model über die Methode *addObj* hinzugefügt, werden diese drei Attribute als Parameter verlangt. Es müssen aber nicht alle drei Attribute gesetzt werden. Leerstrings sind für alle drei Parameter erlaubt. Um das Objekt im Baum einzuordnen werden diese drei Parameter zu einem ParentNameSpace String zusammengefügt. Um eine Filterung bei den Trace-Ausgaben nach den drei Attributen zu erlauben, werden diese drei Namens-Sektionen für die Objekte mitgeführt.

m_strObjName Datentype: QString
: Eigentlicher Name des Objekts. Entspricht auch dem Namen des Knotens innerhalb des Baums, der auf diesen Listeneintrag verweist.

m_enabled Datentype: EEnabled
: (für Tracing) Objekte und NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier abgelegt. Ist das Objekt aktuell nicht instanziiert, wird der aktuelle Zustand des Objekts in diesem Attribut festgehalten. Wird das Objekt instanziiert und damit dem Model wieder hinzugefügt, wird sein Zustand entsprechend restauriert.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

m_state Datentype: EStateOnOff
(für Tracing) Objekte und NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier abgelegt. Ist das Objekt aktuell nicht instanziiert, wird der aktuelle Zustand des Objekts in diesem Attribut festgehalten. Wird das Objekt instanziiert und damit dem Model wieder hinzugefügt, wird sein Zustand entsprechend restauriert.

m_objState Datentype: EObjState
Dieses Attribut entspricht dem Zustand des Objekts. Wird bei Programmstart ein Model aus dem XML-File erzeugt, existieren die Objekte zu diesem Zeitpunkt unter Umständen noch gar nicht und der *ObjState* dieser Objekte ist noch Undefined. Wird das Objekt instanziiert und dem Model hinzugefügt, ist der Zustand des Objekts Created. Wird das Objekt zerstört, ist der Zustand des Objekts Destroyed.

m_iObjId Datentype: int
Entspricht dem Listen-Index des Eintrags innerhalb des Models.

m_pObj Datentype: QObject*
Verweis auf das Objekt, das durch den Listeneintrag repräsentiert wird. Ist das Objekt aktuell nicht instanziiert, ist *pObj* gleich NULL.

2.4.2 Konstruktoren und Destruktor

2.4.2.1 *CObjPoolListEntry*()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse. Alle Elemente werden auf einen Default-Wert gestellt die anzeigen, dass die Instanz noch nicht verwendet wird (ObjState = Undefined, pObj = NULL, ObjId = -1).

2.4.2.2 *CObjPoolListEntry(const QString&, const QString&, const QString&, const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse. Es werden zwar die Namen für das Objekt übernommen, die die Instanz wird aber als noch nicht verwendet markiert (ObjState = Undefined, pObj = NULL, ObjId = -1).

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&
(für Tracing) Klassen werden oft in Namensbereiche (NameSpaces) eingeordnet, um Doppeldeutigkeiten der Klassennamen zu vermeiden.

i_strClassName Datentype: QString
(für Tracing) Entspricht dem Klassennamen des Trace-Admin-Objekts.

i_strObjNameSpace Datentype: QString
(für Tracing) Häufig werden sehr viele Instanzen einer Klasse instanziiert (z.B. QWidget's). Um die Übersicht zu behalten, können diese Instanzen in Namensbereiche gegliedert werden (für Widgets könnte das z.B. der Name des übergeordneten Dialogs sein).

Anmerkung zu *ClassNameSpace*, *ClassName* und ObjNameSpace:

Es müssen nicht alle drei Parameter gesetzt werden. Leerstrings sind für alle drei Parameter erlaubt. Um das Objekt im Baum einzuordnen werden diese drei Parameter zu einem ParentNameSpace String zusammengefügt. Um eine Filterung bei den Trace-Ausgabe nach den drei Attributen zu erlauben, werden diese drei Namens-Sektionen getrennt für die Objekte mitgeführt.

i_strObjName Datentype: QString

Eigentlicher Name des Objekts. Entspricht auch dem Namen des Knotens innerhalb des Baums, der auf diesen Listeneintrag verweist.

2.4.2.3 *~CObjPoolListEntry*()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zerstört die Instanz. Wird vom Model aufgerufen, wenn das Model selbst zerstört wird. Ein Listeneintrag lebt so lange, wie auch das Model lebt und kann nicht zwischenzeitlich gelöscht werden. Die vom Listeneintrag referenzierten Objekte können zur Laufzeit zerstört und wieder neu erzeugt werden. Werden sie dem Model aufs Neue hinzugefügt, werden sie wieder vom selben Listeneintrag verwaltet und erhalten so dieselbe ID wie zuvor.

2.4.3 Operationen

Die Operationen der Klasse beschränken sich überwiegend darauf, die Attribute der Klasse zu setzen und zu. Da die Methoden trivial sind, lohnt die Mühe nicht, die Methoden einzeln aufzuführen.

2.5 Klasse *CModelObjPool*

Generalisierung der Klasse QAbstractItemModel.

Dies ist die zentrale Basisklasse des Object-Pool-Models. Über Methoden dieser Klasse werden Objekte dem Model hinzugefügt und können Objekte gesucht werden.

2.5.1 Datentype (enum) *EColumn*

Diese ID's entsprechen den Spalten-Indizes, wenn das Model über QTreeView oder QTableView ausgegeben wird. Die Spalten-Indizes entsprechen den entsprechenden Attributen der Baum- und Listeneinträge (siehe Klassen CObjPoolTreeEntry und CObjPoolListEntry). Folgende Daten können über die Views in Spalten ausgegeben werden:

EColumnNodeName

In dieser Spalte wird der Name des Astes (Knotens) bzw. Blatts (Objekt-Eintrag) ausgegeben.

EColumnEnabled

(für Tracing) In dieser Spalte wird das Attribute Enabled der Tree oder Listen-Einträge ausgegeben.

EColumnState

(für Tracing) In dieser Spalte wird das Attribute State der Tree oder Listen-Einträge ausgegeben.

EColumnObjId
: In dieser Spalte wird die ID des Objekts (entspricht dem Index innerhalb der Liste) ausgegeben.

EColumnObjState
: In dieser Spalte wird das Attribute ObjState der Tree oder Listen-Einträge ausgegeben.

EColumnCount
: *Count*-Enum Einträge werden dazu verwendet, um Schleifen zu programmieren oder Felder anzulegen. In diesem Fall wird der Count-Eintrag auch als Rückgabewert der ItemModel-Methode „*columnCount*“ verwendet.

2.5.2 Attribute

m_strObjName Datentype: QString
: Name des Models. Als Voreinstellung wird dem Wurzel (Root) Eintrag dieser Name zugewiesen.

m_bIsTreeModel Datentype: bool
: Das Model kann – ähnlich wie beim Windows-Explorer - zur besseren Übersicht in zwei separaten Views ausgegeben werden. Hierzu wird der Inhalt des Models teilweise kopiert – in ein Model, das nur die Knoten (NameSpace-Entries) enthält und in ein Model, das die „ersten Nachkommen“ des aktuell selektierten NameSpace-Entries beinhaltet. Das Model mit den NameSpace-Entries entspricht praktisch dem Verzeichnisbaum des Explorers, das über die Klasse QTreeView dargestellt wird. Das Model mit den Nachkommen des selektierten NameSpace-Entries wird über die Klasse QTableView ausgegeben. Je nachdem, welcher View der Model-Klasse zugeordnet ist, muss sich die Model-Klasse teilweise anders verhalten, um die etwas unterschiedlichen Interfaces der View-Klassen zu befriedigen. IsTreeModel legt fest, ob das Model über ein TreeView ausgegeben werden soll oder nicht. Nur in einem Tree-Model werden neu hinzugefügte Objekte auch hierarchisch einsortiert. In einem Nicht-Tree-Model werden neu hinzugefügte Objekte „flach“ unterhalb des Root-Eintrags erstellt.

m_pMtxObjs Datentype: QMutex
: Es bietet sich an, das Model als zentrale „Sammelstelle“ für Objekte in Multi-Threaded-Applikationen zu verwenden. Das Model muss also damit fertig werden, dass Threads asynchron auf das Model zugreifen, Objekte hinzufügen, Model Einträge suchen oder Model-Einträge ändern, wenn z.B. die Objekte zerstört werden (beim Zerstören der Objekte wird im Model nur vermerkt, dass das Objekt nicht mehr existiert). Deshalb müssen der Baum und die Liste über einen Mutex geschützt werden, der in den Methoden gelockt wird, die den Baum oder die Liste modifizieren.

m_uObjsCount Datentype: unsigned int
: Anzahl der belegten Elemente innerhalb der Liste und damit Anzahl der Objekte, die beim Model registriert wurden. Diese ist immer kleiner oder gleich der aktuellen Länge der Liste.

m_uObjsArrLen Datentype: unsigned int
: Länge der Liste. Diese ist immer größer oder gleich der Anzahl der belegten Elemente innerhalb der Liste. Soll der Liste ein neues Element hinzugefügt werden, sind aber bereits alle Elemente belegt, wird die Liste vergrößert. Dabei wird die Liste nicht nur um ein Element vergrößert, sondern es wird versucht, doppelt so viele Elemente zusätzlich zu reservieren, wie die Liste aktuell besaß. Dabei ist die Anzahl der zusätzlich zu reservierenden Elemente allerdings auf eine sinnvolle, maximale Anzahl begrenzt (nicht jedoch die Gesamtlänge der Liste).

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

m_arpListObjs Datentype: CObjPoolListEntry*
Liste mit Verweisen auf Objekte, die irgendwann einmal dem Model hinzugefügt wurden. Zu beachten ist, dass das Objekt nicht unbedingt aktuell instanziiert sein muss. Der Listeneintrag bleibt trotzdem erhalten, damit das Objekt beim Erneuten instanziieren wieder dieselbe ID (Listen-Index) erhält. Um das Debuggen zu erleichtern wurde auf die Verwendung eines Template Datencontainers verzichtet.

m_pTreeObjs Datentype: CObjPoolTreeEntry*
Verweis auf die Wurzel des Baums.

m_pTreeEntryCurr Datentype: CObjPoolTreeEntry*
Häufig kommt es vor, dass derselbe Eintrag im Baum mehrmals hintereinander angesprochen wird. Damit der Eintrag Objekt nicht immer wieder von neuem im Baum gesucht werden muss, wird der zuletzt benutzte Baum-Eintrag hier gespeichert.

m_strFileName Datentype: QString
Der Inhalt des Models kann in einem XML-File gespeichert und von dort wieder zurück gelesen werden. Der Name der Datei wird hier abgelegt.

2.5.3 Konstruktoren und Destruktor

2.5.3.1 CModelObjPool(const QString&, bool, QObject)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse und erzeugt den Wurzel-Eintrag des Baums.

Parameter:

i_strObjName (IN) Datentype: const QString&
Name des Models. Als Voreinstellung wird dem Wurzel (Root) Eintrag dieser Name zugewiesen.

i_bIsTreeModel Datentype: bool
Legt fest, ob das Model die Daten in einer Baumstruktur oder flach organisieren soll (siehe gleichnamiges Attribut der Klasse).

i_pObjParent Datentype: QObject
Übergeordnetes Objekt des Models. Darf auch NULL sein.

2.5.4 Signale

2.5.4.1 clearing(QObject)*

Beschreibung:
Dieses Signal wird gesendet, wenn das Model zerstört wird, noch bevor der erste Eintrag in der Liste bzw. dem Baum gelöscht wird.

Parameter:

i_pObjPool (IN) Datentype: QObject*
Verweis auf das Model, das das Signal sendet.

2.5.4.2 objectInserted(QObject, CObjPoolListEntry*)*

Beschreibung:
Dieses Signal wird gesendet, nachdem ein neuer Listen-Eintrag erzeugt wurde.

Parameter:

i_pObjPool (IN) Datentype: QObject*
Verweis auf das Model, das das Signal sendet.
i_pListEntry (IN) Datentype: CObjPoolListEntry*
Verweis auf den neu angelegten Listen-Eintrag.

2.5.4.3 *objectChanged(QObject*, CObjPoolListEntry*)*

Beschreibung:
Dieses Signal wird gesendet, wenn ein Listen-Eintrag geändert wurde.
Parameter:
i_pObjPool (IN) Datentype: QObject*
Verweis auf das Model, das das Signal sendet.
i_pListEntry (IN) Datentype: CObjPoolListEntry*
Verweis auf den geänderten Listen-Eintrag.

2.5.4.4 *nameSpaceInserted(QObject*, CObjPoolTreeEntry*)*

Beschreibung:
Dieses Signal wird gesendet, nachdem ein neuer Eintrag im Baum erzeugt wurde. Wird auch ein Listen-Eintrag erzeugt (bei einem Baum-Eintrag vom Typ EntryTypeObject), wird anschließend auch das Signal objectInserted gesendet.
Parameter:
i_pObjPool (IN) Datentype: QObject*
Verweis auf das Model, das das Signal sendet.
i_pTreeEntry (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den geänderten Eintrag.

2.5.4.5 *nameSpaceChanged(QObject*, CObjPoolTreeEntry*)*

Beschreibung:
Dieses Signal wird gesendet, wenn ein Eintrag im Baum geändert wurde. Wird auch ein Listen-Eintrag geändert (bei einem Baum-Eintrag vom Typ EntryTypeObject), wird anschließend auch das Signal objectChanged gesendet.
Parameter:
i_pObjPool (IN) Datentype: QObject*
Verweis auf das Model, das das Signal sendet.
i_pTreeEntry (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den geänderten Eintrag.

2.5.5 Operationen

2.5.5.1 *isDescendant(const QString&,const QString&,const QString&, bool)*

Besitzerbereich: Klasse
Sichtbarkeit: öffentlich
Beschreibung:
Prüft, ob der als ChildName übergebene Name unterhalb des als ParentName übergebenen Knotens liegt.
Parameter:
i_strRootName (IN) Datentype: const QString&
Knoten-Name des Wurzeleintrags des Baums.

i_strParentName (IN) Datentype: const QString&
Kompletter Pfad-Name des Knotens, für den geprüft werden soll, ob es den als *ChildName* übergebenen Knoten als Nachfolger besitzt.
i_strChildName (IN) Datentype: const QString&
Name des Knotens, für den geprüft werden soll, ob es ein Nachfolger des als *ParentName* übergebenen Knotens ist.
i_bRecursive (IN) Datentype: bool
Gibt an, ob nur die unmittelbaren Nachfolger oder alle darunter liegenden Äste durchsucht werden sollen.

2.5.5.2 *getMutex()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf den Mutex zurück, der vom Model benützt wird, um Zugriffe auf den Baum und die Liste zu synchronisieren. In der Regel ist es nicht nötig, den Mutex selbst zu locken oder wieder freizugeben, denn das Model übernimmt diese Aufgabe selbst.
Rückgabewert:...... Datentype: QMutex*

2.5.5.3 lock()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Lockt den Mutex, der vom Model benützt wird, um Zugriffe auf den Baum und die Liste zu synchronisieren. In der Regel ist es nicht nötig, den Mutex selbst zu locken oder wieder freizugeben, denn das Model übernimmt diese Aufgabe selbst. Wenn dies dennoch einmal erforderlich ist, den Zugriff auf das Model für andere Threads temporär zu sperren, darf nicht vergessen werden, den Mutex anschließend wieder freizugeben !!

2.5.5.4 unlock()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Mutex wieder frei, der vom Model benützt wird, um Zugriffe auf den Baum und die Liste zu synchronisieren (siehe *lock*-Methode).

2.5.5.5 setObjName(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Namen des Models.
Parameter:
i_strObjName (IN) Datentype: const QString&
Neuer Name des Models.

2.5.5.6 getObjName() const

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Namen des Models zurück.
Rückgabewert:...... Datentype: QString

2.5.5.7 isTreeModel() const

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den aktuellen Wert des Attributs *IsTreeModel* zurück.
Rückgabewert:...... Datentype: bool

2.5.5.8 getObjCount()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Anzahl der Objekte zurück, die im Model eingetragen wurden. Entspricht der Anzahl der belegten Listen-Einträge.
Rückgabewert:...... Datentype: unsigned int

2.5.5.9 getRoot()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die Wurzel des Baums zurück.
Rückgabewert:...... Datentype: CObjPoolTreeEntry*

2.5.5.10 clear(bool)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zerstört den Baum und die Liste und gibt den für die Listen- und Baumeinträge reservierten Speicher frei.
Parameter:
i_bDestroyObjects (IN) Datentype: bool

Wird dieser Parameter auf true gesetzt, werden nicht nur die Listen und Baumeinträge zerstört, sondern es wird auch der Destruktor der Objekte aufgerufen, die über die Listen- und Baumeinträge referenziert werden. Dabei wird berücksichtigt, dass ein Baum-Eintrag und der zugehörige Listen-Eintrag auf ein- und dasselbe Objekt verweisen können und der Destruktor wird für das Objekt nur einmal aufgerufen.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

2.5.5.11 addObj(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Dem Model wird ein Listen- als auch ein Blatt (Knoten vom Typ EntryTypeObject) innerhalb des Baums hinzugefügt. Falls nötig, falls also noch nicht bereits durch einen vorherigen *addObj* bzw. *addNameSpace* Aufruf geschehen, werden alle notwendigen, übergeordneten Knoten im Baum angelegt, die von der Wurzel bis zum neu angelegten Blatt führen. Falls ein Blatt mit dem angegeben Namen im angegeben Pfad bereits existiert, wird weder ein neuer Listen- noch ein neuer Baum-Eintrag erzeugt und die Methode kehrt zurück, ohne den Inhalt des Models verändert zu haben.

Diese Methode dient in erster Linie dazu, Objekte dem Model hinzuzufügen, die noch nicht erzeugt wurden. Diese Methode wird also in erster Linie zum „Restaurieren" eines Models aus dem XML-File verwendet.

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&

(für Tracing) Klassen werden oft in Namensbereiche (NameSpaces) eingeordnet, um Doppeldeutigkeiten der Klassennamen zu vermeiden.

i_strClassName (IN) Datentype: const QString&

(für Tracing) Entspricht dem Klassennamen des Trace-Admin-Objekts.

i_strObjNameSpace (IN) Datentype: const QString&

(für Tracing) Häufig werden sehr viele Instanzen einer Klasse angelegt (z.B. QWidget's). Um die Übersicht zu behalten, können diese Instanzen in Namensbereiche gegliedert werden (für Widgets könnte das z.B. der Name des übergeordneten Dialogs sein).

Anmerkung zu *ClassNameSpace*, *ClassName* und ObjNameSpace:

Werden Objekte dem Model über die Methode *addObj* hinzugefügt, werden diese drei Attribute als Parameter verlangt. Es müssen aber nicht alle drei Attribute gesetzt werden. Leerstrings sind für alle drei Parameter erlaubt. Um das Objekt im Baum einzuordnen werden diese drei Parameter zu einem ParentNameSpace String zusammengefügt. Um eine Filterung bei den Trace-Ausgaben nach den drei Attributen zu erlauben, werden diese drei Namens-Sektionen für die Objekte mitgeführt.

i_strObjName (IN) Datentype: const QString&

Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das (noch nicht erzeugte) Objekt verweisen.

i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)

(für Tracing) Objekte und NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben.

i_state (IN) Datentype: EStateOnOff (Default: EStateOff)

(für Tracing) Objekte und NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben.

i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)

Hier ist der aktuelle Zustand des Objekts zu übergeben.

Rückgabewert:.....Datentype: int

ID des Objekts. Entspricht dem Index innerhalb der Liste.

2.5.5.12 addObj(const QString&, const QString&, const QString&, QString, QObject*, EEnabled, EStateOnOff)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Methode verhält sich sehr ähnlich zur vorherigen *addObj* Methode, nur mit folgenden Unterschieden:
Diese Methode dient in erster Linie dazu, Objekte dem Model hinzuzufügen, die soeben erzeugt wurden bzw. gerade erzeugt werden. Diese Methode kann also z.B. unmittelbar im Konstruktor des Objekts aufgerufen werden. Im Listeneintrag wird ein Verweis auf das Objekt angelegt. Falls bereits ein Blatt im Baum mit demselben Namen unter demselben Pfad existiert, das auf ein anderes Objekt verweist, wird der Name durch den Methodenaufruf korrigiert (in *ObjName*+"*Copy<Nr>*") und ein neues Blatt mit dem korrigierten Namen angelegt. Diese *addObj* Methode ergänzt das Model also in jedem Fall um ein Blatt im Baum und einen zugehörigen Listen-Eintrag.
Parameter:
i_strClassNameSpace (IN) Datentype: const QString&
siehe vorherige *addObj* Methode
i_strClassName (IN) Datentype: const QString&
siehe vorherige *addObj* Methode
i_strObjNameSpace (IN) Datentype: const QString&
siehe vorherige *addObj* Methode
i_pstrObjName (IN|OUT) Datentype: QString*
Hier ist der eigentliche Name des Objekts zu übergeben. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das Objekt verweisen. Falls bereits ein Blatt im Baum mit demselben Namen unter demselben Pfad existiert, das auf ein anderes Objekt verweist, wird der Name durch den Methodenaufruf korrigiert (in *ObjName*+"*Copy<Nr>*") und ein neues Blatt mit dem korrigierten Namen angelegt. Der vom Model korrigierte Name wird über diesen Parameter zurückgeben.
i_pObj (IN) Datentype: QObject*
Verweis auf das Objekt, das dem Model hinzugefügt werden soll.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
(für Tracing) Objekte und NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben.
i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
(für Tracing) Objekte und NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben.
Rückgabewert:.....Datentype: int
ID des Objekts. Entspricht dem Index innerhalb der Liste.

2.5.5.13 addObj(const QString&, const QString&, const QString&, const QString&, int, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Methode verhält sich sehr ähnlich, wie die vorherigen *addObj* Methoden, nur mit folgenden Unterschieden:

Diese Methode dient in erster Linie dazu, Objekte dem Model hinzuzufügen, die noch nicht erzeugt wurden. Diese Methode wird also in erster Linie zum „Restaurieren" eines Models aus dem XML-File verwendet. Die Methode erlaubt es zusätzlich, eine bereits vordefinierte ID des Objekts zu verwenden. Dies setzt allerdings voraus, dass die ID eindeutig ist und nicht bereits vorher schon vergeben war. Oder mit anderen Worten, ist der Listeneintrag an der vorgegebenen ID bereits beleg bzw. ist bereits ein Blatt im Baum unter dem angegeben Pfad angelegt, wird das Objekt dem Model nicht hinzugefügt und die Methode kehrt zurück, ohne den Inhalt des Models verändert zu haben.

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&
: siehe vorherige *addObj* Methode

i_strClassName (IN) Datentype: const QString&
: siehe vorherige *addObj* Methode

i_strObjNameSpace (IN) Datentype: const QString&
: siehe vorherige *addObj* Methode

i_strObjName (IN) Datentype: const QString&
: Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das (noch nicht erzeugte) Objekt verweisen.

i_iObjId (IN) Datentype: int
: Vordefinierte ID des Objekts. Falls nicht bereits ein Objekt unter dieser ID bzw. unter demselben Pfad und Namen im Model existiert, wird das Objekt in der Liste an diesem Index eingetragen.

i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
: (für Tracing) Objekte und NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben.

i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
: (für Tracing) Objekte und NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben.

i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
: Hier ist der aktuelle Zustand des Objekts zu übergeben.

Rückgabewert:.....Datentype: CObjPoolTreeEntry*
: Verweis auf den Eintrag im Baum (Blatt vom Typ EntryTypeObject), der für das Objekt angelegt wurde. Falls das Objekt nicht hinzugefügt wurde, weil die ID bereits vergeben war oder bereits ein Blatt im Baum unter dem angegeben Pfad und Objekt-Namen angelegt war, wird NULL zurückgegeben.

2.5.5.14 addNameSpace(const QString&, const QString&, EEnabled, EStateOnOff)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Die Methode erlaubt es, explizit einen Knoten (EntryTypeNameSpace) im Baum anzulegen. Falls jedoch bereits ein Knoten unter dem angegebenen Pfad und Knoten-Namen existiert, wird der Knoten nicht angelegt und die Methode kehrt zurück, ohne den Inhalt des Models verändert zu haben.

Parameter:

i_strParentNameSpace (IN) Datentype: const QString&
: Angabe des Pfads, unter dem der Knoten anzulegen ist (z.B. „Deutschland::Bayern").

i_strNodeName (IN) Datentype: const QString&
Name des Knotens, der unter dem angegebenen Pfad angelegt werden soll.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
(für Tracing) NameSpaces können aktiviert und deaktiviert werden. Der aktuelle Zustand des Knotens wird hier übergeben.
i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
(für Tracing) NameSpaces können ein- und ausgeschaltet werden. Der aktuelle Zustand des Knotens wird hier übergeben.
Rückgabewert:.....Datentype: CObjPoolTreeEntry*
Verweis auf den Eintrag im Baum (Knoten vom Typ EntryTypeNameSpace), der für den Knoten angelegt wurde. Falls der Knoten nicht angelegt wurde, weil bereits ein Knoten im Baum unter dem angegeben Pfad und Knoten-Namen angelegt war, wird NULL zurückgegeben.

2.5.5.15 removeObj(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Entfernt den Verweis auf das Objekt mit der angegebenen ID (Listen-Index). Die Methode löscht aber weder das zugehörige Blatt im Baum noch den zugehörigen Eintrag in der Liste. Es wird lediglich vermerkt, dass das Objekt zerstört ist, indem der *ObjState* auf Destroyed und der Verweis auf das Objekt auf NULL gesetzt werden.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, das zerstört wird bzw. zerstört wurde.

2.5.5.16 removeObj(const QString&, const QString&, const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode verhält sich genauso, wie die vorherige *removeObj* Methode, nur dass zunächst der zugehörige Baum-Eintrag anhand der Pfadangabe und Angabe des Knoten-Namens gesucht werden muss, um die ID des Objekts zu erhalten. Diese Methode ist also langsamer als die *removeObj* Methode, an die die ID des Objekts übergeben wird.
Parameter:
i_strClassNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strClassName (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjName (IN) Datentype: const QString&
Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das zu entfernende Objekt verweisen.

2.5.5.17 *removeObj(QObject*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode verhält sich genauso, wie die vorherige *removeObj* Methode, nur dass zunächst der zugehörige Eintrag in der Liste anhand der übergebenen Adresse des Objekts gesucht werden muss, um die ID des Objekts zu erhalten. Diese Methode ist also langsamer als die *removeObj* Methode, an die die ID des Objekts übergeben wird.
Parameter:
i_pObj (IN) Datentype: QObject*
Verweis auf das Objekt, das zerstört wird.

2.5.5.18 *changedObj(int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode dient lediglich dazu, das Model anzuweisen, das Signal *objectChanged* zu senden. Damit werden Clients, die z.B. am Inhalt des Objekts interessiert sind, über die Änderung des Objekts informiert.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, das geändert wurde.

2.5.5.19 *updateObj(int, EEnabled, EStateOnOff, EObjState)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Aktualisiert die angegebenen Attribute des Baum- und Listeneintrags und sendet das Signal *objectChanged* aus. Attribute des Objekts selbst werden durch den Methodenaufruf nicht geändert.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, für das die Attribute im Listen- und Baum-Eintrag zu ändern sind.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
Hier ist der aktuelle Zustand des Objekts zu übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

2.5.5.20 updateObj(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Diese Methode verhält sich genauso, wie die vorherige *updateObj* Methode, nur dass zunächst der zugehörige Baum-Eintrag anhand der Pfadangabe und Angabe des Knoten-Namens gesucht werden muss, um die ID des Objekts zu erhalten. Diese Methode ist also langsamer als die *updateObj* Methode, an die die ID des Objekts übergeben wird.
Parameter:
i_strClassNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strClassName (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjName (IN) Datentype: const QString&
Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das geänderte Objekt verweisen.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
Hier ist der aktuelle Zustand des Objekts zu übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

2.5.5.21 updateObj(const QString&, const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Diese Methode verhält sich genauso, wie die vorherige *updateObj* Methode, nur dass zunächst der zugehörige Baum-Eintrag anhand der Pfadangabe und Angabe des Knoten-Namens gesucht werden muss, um die ID des Objekts zu erhalten. Diese Methode ist also langsamer als die *updateObj* Methode, an die die ID des Objekts übergeben wird.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das geänderte Objekt verweisen.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
Hier ist der aktuelle Zustand des Objekts zu übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

2.5.5.22 updateObj(const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Diese Methode verhält sich genauso, wie die vorherige *updateObj* Methode, nur dass zunächst der zugehörige Baum-Eintrag anhand der Pfadangabe und Angabe des Knoten-Namens gesucht werden muss, um die ID des Objekts zu erhalten. Diese Methode ist also langsamer als die *updateObj* Methode, an die die ID des Objekts übergeben wird.
Parameter:
i_strName (IN) Datentype: const QString&
Der Name muss den kompletten Pfad sowie den Namen des Knotens beinhalten.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
Hier ist der aktuelle Zustand des Objekts zu übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

2.5.5.23 updateNameSpace(const QString&, const QString&, EEnabled, EStateOnOff, bool)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Aktualisiert die angegebenen Attribute der Baum- und Listeneinträge innerhalb des angegebenen Knotens und sendet das Signal *nameSpaceChanged* aus. Es werden der spezifizierte Knoten sowie die Blätter des Knotens angepasst, also der spezifizierte Baum-Eintrag vom Typ EntryTypeNameSpace sowie die unmittelbaren Nachfolger vom Typ EntryTypObject. Über den *Recursive* Paramter kann die Methode angewiesen werden, zusätzlich sämtliche Baum- und Listeneinträge unterhalb des spezifizierten Knotens ebenfalls mit zu aktualisieren.
Parameter:

i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Name des Knotens.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_bRecursive (IN) Datentype: bool (Default: false)
Über diesen Parameter kann die Methode angewiesen werden, nicht nur den spezifizierten Knoten selbst sondern zusätzlich auch sämtliche Baum- und Listeneinträge unterhalb des spezifizierten Knotens zu aktualisieren.

2.5.5.24 updateNameSpace(const QString&, const QString&, EEnabled, EStateOnOff, bool)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Diese Methode verhält sich genauso, wie die vorherige *updateNameSpace* Methode.
Parameter:
i_strName (IN) Datentype: const QString&
Der Name muss den kompletten Pfad sowie den Namen des Knotens beinhalten.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledUndefined)
(für Tracing) Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_state (IN) Datentype: EStateOnOff (Default: EStateOnOffUndefined)
(für Tracing) Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_bRecursive (IN) Datentype: bool (Default: false)
Über diesen Parameter kann die Methode angewiesen werden, nicht nur den spezifizierten Knoten selbst sondern zusätzlich auch sämtliche Baum- und Listeneinträge unterhalb des spezifizierten Knotens zu aktualisieren.

2.5.5.25 findObjId(const QString&, const QString&, const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Sucht den spezifizierten Baum-Eintrag und gibt die zugehörige ID des Objekts zurück, die dem Index in der Liste entspricht.
Parameter:

i_strClassNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strClassName (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjName (IN) Datentype: const QString&
Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags des zu suchenden Objekts.
Rückgabewert: Datentype: int
ID des Objekts bzw. -1, wenn das Objekt nicht gefunden wurde.

2.5.5.26 findObjId(const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Sucht den spezifizierten Baum-Eintrag und gibt die zugehörige ID des Objekts zurück, die dem Index in der Liste entspricht.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Name des Knotens.
Rückgabewert: Datentype: int
ID des Objekts bzw. -1, wenn das Objekt nicht gefunden wurde.

2.5.5.27 findObjId(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Sucht den spezifizierten Baum-Eintrag und gibt die zugehörige ID des Objekts zurück, die dem Index in der Liste entspricht.
Parameter:
i_strName (IN) Datentype: const QString&
Der Name muss den kompletten Pfad sowie den Namen des Knotens beinhalten.
Rückgabewert: Datentype: int
ID des Objekts bzw. -1, wenn das Objekt nicht gefunden wurde.

2.5.5.28 isActive(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode gibt die zugehörigen Attribute des Listen-Eintrags zurück. Ein Objekt ist aktiv, wenn der gespeicherte Zustand in der Liste sowohl Enabled als auch On ist.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, für das die Attribute abzufragen sind.
Rückgabewert: Datentype: bool

2.5.5.29 setEnabled(int, EEnabled)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode setzt das *Enabled* Attribut des Listen-Eintrags und des zugehörigen Blatts im Baum.
Parameter:
i_iObjId (IN) Datentype: int
 ID des Objekts, für das das Attribut zu setzen ist.
i_enabled (IN) Datentype: EEnabled
 Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

2.5.5.30 setEnabled(const QString&, const QString&, EObjPoolEntryType, EEnabled, bool)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode setzt das *Enabled* Attribut im spezifizierten Knoten innerhalb des Baums. Soll das Attribut eines Knotens geändert werden (EntryTypeNameSpace), wird sowohl der Knoten als auch alle eine Ebene darunter liegenden Einträge vom Typ EntryTypeObject aktualisiert, also die Blätter des Knotens. Die Methode erlaubt es auch, durch Setzen des Parameter *Recursive* auf true, das Attribut sowohl für den spezifizierten Knoten als auch zusätzlich für alle seine direkten oder indirekten Nachfolger zu setzen. Wird das Attribut eines Blatts (EntryTypeObject) geändert (ob direkt durch Auswahl eines Blatt-Eintrags über die Parameter der Methode oder indirekt als „Child“ eines geänderten Knotens), wird immer auch der entsprechende Listen-Eintrag aktualisiert.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
 Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
 Name des Knotens.
i_entryType (IN) Datentype: EObjPoolEntryType
 Typ des zu ändernden Baum-Eintrags. Dieser muss angegeben werden, weil Pfadangabe und Name des Knotens nicht unbedingt eindeutig sind. Es kann unterhalb eines Knotens nämlich sowohl ein Blatt mit dem Namen *NodeName* als auch einen weiteren Knoten mit diesem Namen geben (bei dem Beispiel aus der Einführung ist z.B. „Berlin“ sowohl als EntryTypeObject als auch als EntryTypeNameSpace im Baum unterhalb von „Deutschland“ vorhanden).
i_enabled (IN) Datentype: EEnabled
 Objekte können aktiviert und deaktiviert werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_bRecursive (IN) Datentype: bool (Default: false)
 Über diesen Parameter kann die Methode angewiesen werden, nicht nur den spezifizierten Knoten selbst sondern zusätzlich auch sämtliche Baum- und Listeneinträge unterhalb des spezifizierten Knotens zu aktualisieren.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

2.5.5.31 getEnabled(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode gibt das *Enabled* Attribut des Listen-Eintrags zurück.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, dessen Attribut abzufragen ist.
Rückgabewert: Datentype: EEnabled

2.5.5.32 isEnabled(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode gibt true zurück, wenn das Attribut Enabled des Listen-Eintrags auf EnabledYes steht.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, dessen Attribut abzufragen ist.
Rückgabewert: Datentype: bool

2.5.5.33 setState(int, EStateOnOff)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode setzt das *StateOnOff*-Attribut des Listen-Eintrags und des zugehörigen Blatts im Baum.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, für das das Attribut zu setzen ist.
i_state (IN) Datentype: EStateOnOff
Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.

2.5.5.34 setState(const QString&, const QString&, EObjPoolEntryType, EStateOnOff, bool)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode verhält sich genauso, wie die entsprechende *setEnabled* Methode, nur wird nicht das Enabled sondern das *StateOnOff* Attribut gesetzt.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Name des Knotens.

i_entryType (IN) Datentype: EObjPoolEntryType
siehe entsprechende *setEnabled* Methode.
i_state (IN) Datentype: EStateOnOff
Objekte können ein- und ausgeschaltet werden. Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen.
i_bRecursive (IN) Datentype: bool (Default: false)
Über diesen Parameter kann die Methode angewiesen werden, nicht nur den spezifizierten Knoten selbst sondern zusätzlich auch sämtliche Baum- und Listeneinträge unterhalb des spezifizierten Knotens zu aktualisieren.

2.5.5.35 getState(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
(für Tracing) Die Methode gibt das *StateOnOff* Attribut des Listen-Eintrags zurück.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, dessen Attribut abzufragen ist.
Rückgabewert: Datentype: EStateOnOff

2.5.5.36 setObjState(int, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Methode speichert den übergebenen Zustand des Objekts im zugehörigen Listen-Eintrag.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, für das das Attribut zu setzen ist.
i_objState (IN) Datentype: EObjState
Der aktuelle Zustand des Objekts wird hier übergeben. Wird Undefined übergeben, wird der Parameter nicht übernommen. Wird Destroyed übergeben und besitzt der Listeneintrag einen gültigen Verweis (*pObj* != NULL) auf ein Objekt, wird der Verweis auf NULL zurück gesetzt.

2.5.5.37 getObjState(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Methode gibt den Zustand des Objekts zurück, so wie er im Listen-Eintrag gespeichert ist.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts, dessen Attribut abzufragen ist.
Rückgabewert: Datentype: EObjState

2.5.5.38 setObj(int, QObject*)

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich

Beschreibung:

Die Methode speichert den übergebenen Verweis auf das Objekt im zugehörigen Listen-Eintrag. Gleichzeitig wird der *ObjState* gesetzt: auf Destroyed, wenn NULL übergeben wird, auf Created, wenn ein gültiger Objekt-Verweis übergeben wird.

Parameter:

i_iObjId (IN) Datentype: int

ID des Objekts, für das das Attribut zu setzen ist.

i_pObj (IN) Datentype: QObject*

Verweis auf das Objekt, das im Listen-Eintrag zu speichern ist. Der aktuelle Zustand des Objekts wird entsprechend angepasst. Wird NULL übergeben, wird der *ObjState* auf Destroyed gesetzt. Wird ein gültiger Verweis übergeben, wird der *ObjState* auf Created gesetzt.

2.5.5.39 getObj(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Die Methode gibt den Verweis auf das Objekt zurück.

Parameter:

i_iObjId (IN) Datentype: int

ID des Objekts, dessen Attribut abzufragen ist.

Rückgabewert: Datentype: QObject*

Verweis auf das Objekt. Kann auch NULL sein.

2.5.5.40 findObj(const QString&, const QString&, const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Durchsucht den Baum nach dem spezifizierten Eintrag und gibt den Verweis auf das Objekt zurück.

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strClassName (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strObjNameSpace (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strObjName (IN) Datentype: const QString&

Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das Objekt verweisen.

Rückgabewert: Datentype: QObject*

Verweis auf das Objekt. Kann auch NULL sein.

2.5.5.41 findObj(const QString&, const QString&)

Besitzerbereich: Instanz

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

Sichtbarkeit: öffentlich
Beschreibung:
Durchsucht den Baum nach dem spezifizierten Eintrag und gibt den Verweis auf das Objekt zurück.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Name des Knotens.
Rückgabewert: Datentype: QObject*
Verweis auf das Objekt. Kann auch NULL sein.

2.5.5.42 findObj(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Durchsucht den Baum nach dem spezifizierten Eintrag und gibt den Verweis auf das Objekt zurück.
Parameter:
i_strName (IN) Datentype: const QString&
Der Name muss den kompletten Pfad sowie den Namen des Knotens beinhalten.
Rückgabewert: Datentype: QObject*
Verweis auf das Objekt. Kann auch NULL sein.

2.5.5.43 setFileName(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Das Model kann in einem XML-File gespeichert und von dort wieder zurück gelesen werden. Der Name des hierfür zu verwendenden XML-Files (incl. Pfadangabe) kann über diese Methode gesetzt werden.
Parameter:
i_strFileName (IN) Datentype: const QString&
Name (incl. Pfad) des XML-Files.

2.5.5.44 getFileName()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Das Model kann in einem XML-File gespeichert und von dort wieder zurück gelesen werden. Der Name des hierfür zu verwendenden XML-Files (incl. Pfadangabe) kann über diese Methode abgefragt werden.
Rückgabewert: Datentype: QString

2.5.5.45 recall(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Über diese Methode kann das Model aus einem XML-File geladen werden.
Parameter:
i_strFileName (IN) Datentype: const QString&
Name (incl. Pfad) des XML-Files. Wird ein Leerstring übergeben, wird der aktuell über *setFileName* gesetzte Dateiname verwendet. Wird ein gültiger Dateiname übergeben, wird aus diesem File gelesen. Wurde zuvor noch kein Dateiname für das Modell gesetzt, wird der übergebene Dateiname als aktuell zu verwendender Dateiname übernommen (es wird für diesen Fall also implizit *setFileName* aufgerufen).
Rückgabewert: Datentype: SErrResult

2.5.5.46 save(const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Über diese Methode kann das Model in einem XML-File gespeichert werden.
Parameter:
i_strFileName (IN) Datentype: const QString&
Name (incl. Pfad) des XML-Files. Wird ein Leerstring übergeben, wird der aktuell über *setFileName* gesetzte Dateiname verwendet. Wird ein gültiger Dateiname übergeben, wird in dieses File geschrieben. Wurde zuvor noch kein Dateiname für das Modell gesetzt, wird der übergebene Dateiname als aktuell zu verwendender Dateiname übernommen (es wird für diesen Fall also implizit *setFileName* aufgerufen).
Rückgabewert: Datentype: SErrResult

2.5.5.47 getListEntry(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode gibt einen Verweis auf den Listen-Eintrag am gewünschten Index zurück.
Parameter:
i_iObjId (IN) Datentype: int
ID des Objekts. Entspricht dem Index in der Liste.
Rückgabewert: Datentype: CObjPoolListEntry*
Am angegeben Index muss nicht unbedingt auch ein gültiger Listen-Eintrag vorhanden sein, weshalb auch NULL zurückgegeben werden kann.

2.5.5.48 findListEntry(int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode gibt einen Verweis auf den Listen-Eintrag am gewünschten Index zurück. Falls – was eigentlich nie der Fall sein sollte – Listen-Index und Objekt-ID nicht mehr übereinstimmen, weil der

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany

Inhalt des Models durch „Experten“ manipuliert wurde, wird die Liste nach der übergebenen Object-ID durchsucht.

Parameter:

i_iObjId (IN) Datentype: int

ID des Objekts. Diese muss bei dieser Methode nicht dem Index in der Liste entsprechen.

Rückgabewert: Datentype: CObjPoolListEntry*

Wurde kein Listen-Eintrag mit der gewünschten ID gefunden, wird NULL zurückgegeben.

2.5.5.49 findListEntry(QObject)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode durchsucht die Liste nach dem Verweis auf das spezifizierte Objekt und gibt einen Verweis auf den Listen-Eintrag zurück.

Parameter:

i_pObj (IN) Datentype: int

Adresse des Objekts, nach dem gesucht warden soll.

Rückgabewert: Datentype: CObjPoolListEntry*

Wurde kein Listen-Eintrag mit der gewünschten Objekt-Adresse gefunden, wird NULL zurückgegeben.

2.5.5.50 findTreeEntry(const QString&, const QString&, const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode durchsucht den Baum und gibt einen Verweis auf den gefundenen Eintrag im Baum zurück.

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strClassName (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strObjNameSpace (IN) Datentype: const QString&

siehe *addObj* Methoden

i_strObjName (IN) Datentype: const QString&

Eigentlicher Name des Objekts. Entspricht damit dem Namen des Knotens innerhalb des Baums sowie dem Objekt Namen des Listeneintrags, die auf das Objekt verweisen.

Rückgabewert: Datentype: CObjPoolTreeEntry*

Wurde kein Baum-Eintrag mit dem spezifizierten Namen gefunden, wird NULL zurückgegeben.

2.5.5.51 findTreeEntry(const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode durchsucht den Baum und gibt einen Verweis auf den gefundenen Eintrag im Baum zurück.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Komplette Pfadangabe (ohne den Namen des Knotens).
i_strNodeName (IN) Datentype: const QString&
Name des Knotens.
i_entryType (IN) Datentype: EObjPoolEntryType
Typ des zu suchenden Baum-Eintrags. Dieser muss angegeben werden, weil Pfadangabe und Name des Knotens nicht unbedingt eindeutig sind. Es kann unterhalb eines Knotens nämlich sowohl ein Blatt mit dem Namen *NodeName* als auch einen weiteren Knoten mit diesem Namen geben (bei dem Beispiel aus der Einführung ist z.B. „Berlin“ sowohl als EntryTypeObject als auch als EntryTypeNameSpace im Baum unterhalb von „Deutschland“ vorhanden).
Rückgabewert: Datentype: CObjPoolTreeEntry*
Wurde kein Baum-Eintrag mit dem spezifizierten Namen gefunden, wird NULL zurückgegeben.

2.5.5.52 findTreeEntry(const QString&, const QString&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Diese Methode ist „for experts only“ und wird für solche Fälle bereitgestellt, in denen es notwendig ist, Listen- und Baumeinträge unmittelbar zu manipulieren. Die Methode durchsucht den Baum und gibt einen Verweis auf den gefundenen Eintrag im Baum zurück.
Parameter:
i_strName (IN) Datentype: const QString&
Der Name muss den kompletten Pfad sowie den Namen des Knotens beinhalten.
i_entryType (IN) Datentype: EObjPoolEntryType
Typ des zu suchenden Baum-Eintrags. Dieser muss angegeben werden, weil Pfadangabe und Name des Knotens nicht unbedingt eindeutig sind. Es kann unterhalb eines Knotens nämlich sowohl ein Blatt mit dem Namen *NodeName* als auch einen weiteren Knoten mit diesem Namen geben (bei dem Beispiel aus der Einführung ist z.B. „Berlin“ sowohl als EntryTypeObject als auch als EntryTypeNameSpace im Baum unterhalb von „Deutschland“ vorhanden).
Rückgabewert: Datentype: CObjPoolTreeEntry*
Wurde kein Baum-Eintrag mit dem spezifizierten Namen gefunden, wird NULL zurückgegeben.

2.5.5.53 addListEntry(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz

Sichtbarkeit: geschützt

Beschreibung:

Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Erweitert, falls notwendig, die Liste (siehe Attribute *ObjsCount*, *ObjsArrLen* und *ListObjs*), erzeugt einen neuen Listeneintrag und trägt einen Verweis auf den Listeneintrag in die Liste ein. Die Parameter *Enabled*, *State* und *ObjState* werden in den Listeneintrag übernommen. Die *ObjId* (der Listenindex) wird jedoch noch nicht gesetzt. Dies geschieht erst in der aufrufenden Methode, die mit dem Listen-Eintrag auch einen Eintrag im Baum erzeugt und die ID bei beiden Einträgen setzt.

Parameter:

i_strClassNameSpace (IN) Datentype: const QString&
 siehe *addObj* Methoden

i_strClassName (IN) Datentype: const QString&
 siehe *addObj* Methoden

i_strObjNameSpace (IN) Datentype: const QString&
 siehe *addObj* Methoden

i_strObjName (IN) Datentype: const QString&
 siehe *addObj* Methoden

i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
 siehe *addObj* Methoden

i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
 siehe *addObj* Methoden

i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
 siehe *addObj* Methoden

Rückgabewert: Datentype: CObjPoolListEntry*
 Verweis auf den neu angelegten Listen-Eintrag.

2.5.5.54 updateListEntry(CObjPoolListEntry, EEnabled, EStateOnOff, EObjState)*

Besitzerbereich: Instanz
Sichtbarkeit: geschützt

Beschreibung:

Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Übernimmt die übergebenen Parameter, falls nicht auf Undefined gesetzt, in den übergebenen Listeneintrag und prüft gleichzeitig, ob tatsächlich ein Element des Listeeintrags geändert wurde.

Parameter:

i_pListEntry (IN) Datentype: CObjPoolListEntry*
 Verweis auf den Listen-Eintrag, dessen Attribute zu setzen sind.

i_enabled (IN) Datentype: EEnabled
 siehe *addObj* Methoden

i_state (IN) Datentype: EStateOnOff
 siehe *addObj* Methoden

i_objState (IN) Datentype: EObjState
 siehe *addObj* Methoden

Rückgabewert: Datentype: bool
 true, falls ein Attribut geändert wurde.

2.5.5.55 updateListEntry(CObjPoolListEntry, EEnabled, EStateOnOff, EObjState)*

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Übernimmt die übergebenen Parameter, falls nicht auf Undefined gesetzt, in den übergebenen Listeneintrag und prüft gleichzeitig, ob tatsächlich ein Element des Listeeintrags geändert wurde.
Parameter:
i_pListEntry (IN) Datentype: CObjPoolListEntry*
Verweis auf den Listen-Eintrag, dessen Attribute zu setzen sind.
i_enabled (IN) Datentype: EEnabled
siehe *addObj* Methoden
i_state (IN) Datentype: EStateOnOff
siehe *addObj* Methoden
i_pObj (IN) Datentype: QObject*
siehe *addObj* Methoden
Rückgabewert: Datentype: bool
true, falls ein Attribut geändert wurde.

2.5.5.56 addTreeEntry(const QString&, const QString&, const QString&, const QString&, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die übergebenen Knoten-Namen (also nicht der *ObjName*) werden hierzu aneinandergehängt. Mit dem so erzeugten *ParentNameSpace* wird die nachfolgende, geschützte *addTreeEntry*-Methode aufgerufen, um einen neues Blatt (EntryTypeObject) zu erzeugen.
Parameter:
i_strClassNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strClassName (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjNameSpace (IN) Datentype: const QString&
siehe *addObj* Methoden
i_strObjName (IN) Datentype: const QString&
siehe *addObj* Methoden
i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
siehe *addObj* Methoden
i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
siehe *addObj* Methoden
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
siehe *addObj* Methoden
Rückgabewert: Datentype: CObjPoolTreeEntry*
Verweis auf das neu angelegte Blatt.

2.5.5.57 *addTreeEntry(const QString&, const QString&, EEnabled, EStateOnOff, EObjState)*

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode erweitert den Baum um alle erforderlichen Knoten und um ein Blatt - falls notwendig. Für jeden Namensteil im übergebenen *ParentNameSpace* (Sektion im String durch „::" getrennt), für den noch kein Knoten (EntryTypeNameSpace) im Baum existiert, wird ein neuer Knoten und damit ein neuer TreeEntry Eintrag erzeugt. Der Parameter *entryType* entscheidet, welchen Type der zuletzt erzeugte Knoten mit dem *NodeName* besitzt. Auch der letzte Knoten wird nur dann angelegt, wenn noch kein Knoten mit dem Namen unter dem angegeben *ParentNameSpace* existierte. Anschließend werden die Parameter *Enabled*, *State* und *ObjState* in Tree-Entry Eintrag übernommen. Die *ObjId* (der Listenindex) wird nicht gesetzt (die Methode kennt den Listen-Index ja auch gar nicht). Die ID wird erst in der aufrufenden Methode gesetzt, die auch den zugehörigen Listen-Eintrag anlegt.
Parameter:
i_strParentNameSpace (IN) Datentype: const QString&
Pfadangabe, unter dem das neue Blatt anzulegen ist.
i_strNodeName (IN) Datentype: const QString&
Name des neu anzulegenden Knotens.
i_entryType (IN) Datentype: EObjPoolEntryType (Default: EObjPoolEntryTypeObject)
Dieser Parameter legt den Typ des Knotens fest, der am Ende der Kette unterhalb des über *ParentNameSpace* angegeben Pfades im Baum erzeugt wird.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
siehe *addObj* Methoden
i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
siehe *addObj* Methoden
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
siehe *addObj* Methoden
Rückgabewert: Datentype: CObjPoolTreeEntry*
Verweis auf den neu angelegten Knoten.

2.5.5.58 *addTreeEntry(CObjPoolTreeEntry*, EObjPoolEntryType, const QString&, EEnabled, EStateOnOff, EObjState)*

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode erzeugt einen neuen Knoten unterhalb des übergebenen Parent-TreeEntries – falls noch kein Knoten mit dem spezifizierten *NodeName* vorhanden ist. Anschließend werden die Parameter *Enabled*, *State* und *ObjState* in den Eintrag übernommen.
Parameter:
i_pTreeEntryParent (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den Parent-Tree-Entry, unter dem der Knoten anzulegen ist.
i_entryType (IN) Datentype: EObjPoolEntryType (Default: EObjPoolEntryTypeObject)
Dieser Parameter legt den Typ des Knotens fest, der am Ende der Kette unterhalb des über *ParentNameSpace* angegeben Pfades im Baum erzeugt wird.

i_strNodeName (IN) Datentype: const QString&
Name des neu anzulegenden Knotens.
i_enabled (IN) Datentype: EEnabled (Default: EEnabledYes)
siehe *addObj* Methoden
i_state (IN) Datentype: EStateOnOff (Default: EStateOff)
siehe *addObj* Methoden
i_objState (IN) Datentype: EObjState (Default: EObjStateUndefined)
siehe *addObj* Methoden
Rückgabewert: Datentype: CObjPoolTreeEntry*
Verweis auf den neu angelegten Knoten.

2.5.5.59 updateTreeEntry(CObjPoolTreeEntry*, EEnabled, EStateOnOff, EObjState)

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode überträgt die übergeben Parameter (falls nicht auf Undefined gesetzt) in den übergebenen Baum-Eintrag und prüft gleichzeitig, ob sich Attribute geändert haben.
Parameter:
i_pTreeEntryParent (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den Tree-Entry, für den die Attribute zu setzen sind.
i_enabled (IN) Datentype: EEnabled
siehe *addObj* Methoden
i_state (IN) Datentype: EStateOnOff
siehe *addObj* Methoden
i_objState (IN) Datentype: EObjState
siehe *addObj* Methoden
Rückgabewert: Datentype: bool
true, falls ein Attribut geändert wurde.

2.5.5.60 updateTreeEntry(CObjPoolTreeEntry*, EEnabled, EStateOnOff, bool)

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode überträgt die übergeben Parameter (falls nicht auf Undefined gesetzt) in den übergebenen Baum-Eintrag und – sofern der Baum-Eintrag ein NameSpace-Entry war - auch in die unmittelbaren Nachfolger vom Typ EntryTypeObject. Ist das *Recursive* Flag gesetzt, werden die Attribute für alle Nachfolger (NameSpace- und Object-Entries) übernommen. Hierzu wird die Methode rekursiv aufgerufen. Gleichzeitig prüft die Methode, ob sich ein Attribut eines Baum-Eintrags geändert hat.
Parameter:
i_pTreeEntryParent (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den Parent-Tree-Entry, unter dem der Knoten anzulegen ist.
i_enabled (IN) Datentype: EEnabled
siehe *addObj* Methoden

i_state (IN) Datentype: EStateOnOff
siehe *addObj* Methoden
i_bRecursive (IN) Datentype: bool
Gibt an, ob die Attribute nur für den spezifizierten Knoten und seine unmittelbaren Nachfolger vom Typ EntryTypeObject oder für den Knoten mit allen seinen Nachfolgern (egal welchen Typs) zu übernehmen sind.
Rückgabewert: Datentype: bool
true, falls ein Attribut geändert wurde.

2.5.5.61 clearTreeEntry(CObjPoolTreeEntry*, bool)

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode entfernt den Knoten sowie alle seine Nachfolger und wird vom Destruktor des Models bzw. aus der öffentlichen *clear*-Methode des Models aufgerufen. Über den Parameter *DestroyObjects* kann zusätzlich festgelegt werden, ob auch die referenzierten Objekte zerstört werden sollen. Da sowohl ein Tree-Entry als auch der zugehörige List-Entry auf ein- und dasselbe Objekt verweisen können, prüft die Methode diesen Umstand vor dem Löschen des Objekts und stellt sicher, dass der Destruktor des Objekts nur einmal aufgerufen wird.
Parameter:
i_pTreeEntry (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den Tree-Entry, der zusammen mit all seinen Nachfolgern zu zerstören ist.
i_bDestroyObjects (IN) Datentype: bool
Gibt an, ob auch die referenzierten Objekte zerstört werden sollen.

2.5.5.62 saveTreeEntry(QDomDocument&, QDomElement&, CObjPoolTreeEntry*)

Besitzerbereich: Instanz
Sichtbarkeit: geschützt
Beschreibung:
Interne Methode zur Kapselung wiederkehrender Code-Fragmente. Die Methode ruft sich selbst rekursiv auf, um einen Knoten mit all seinen Nachfolgern in das Domain-Document zu übernehmen, das in ein XML-File zu schreiben ist.
Parameter:
i_domdoc (IN) Datentype: QDomDocument&
Dokument, in dem der Ast mit all seinen Nachfolgern zu speichern ist.
i_domelemBody (IN) Datentype: QDomElement&
XML-Element, unter dem die Einträge des Baums abgelegt werden. Im XML-File werden die Baum-Einträge nicht hierarchisch abgelegt, also nicht eingerückt, sondern „flach“ unter dem Body-Element eingetragen. Die hierarchische Struktur wird beim Einlesen des XML-Files wieder aufgebaut, da für Elemente vom Typ EntryTypeNameSpace der komplette ParentNameSpace als auch der NodeName gespeichert wird. Für Elemente vom Typ EntryTypeObject werden alle Attribute (*ClassNameSpace*, *ClassName*, *ObjNameSpace* und *ObjName*) gespeichert. Dadurch können beim Einlesen des XML-Files die *addObj* Methoden verwendet werden, über die die Objekte auch ursprünglich dem Model hinzugefügt wurden.

i_pTreeEntry (IN) Datentype: CObjPoolTreeEntry*
Verweis auf den Tree-Entry, der zusammen mit all seinen Nachfolgern im Dokument abzulegen ist.

2.5.5.63 rowCount(const QModelIndex&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel. Werden Model-Indizes erzeugt (siehe Methode *index*), wird deren *InternalPointer* auf den Tree-Entry Eintrag gesetzt. Damit kann die *rowCount*-Methode anhand des übergebenen Parent-Model-Index den Tree-Entry ermitteln, für den die Anzahl der Zeilen (Rows) – also die Anzahl seiner Childs – abgefragt wird.
Parameter:
i_modelIdxParent (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: int
Anzahl der Zeilen (Childs) unterhalb des spezifizierten Tree-Entries.

2.5.5.64 columnCount(const QModelIndex&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel.
Parameter:
i_modelIdxParent (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: int
Die Methode gibt die Konstante EColumnCount zurück.

2.5.5.65 data(const QModelIndex&, int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel. Werden Model-Indizes erzeugt (siehe Methode *index*), wird deren *InternalPointer* auf den Tree-Entry Eintrag gesetzt. Damit kann die *data*-Methode anhand des übergebenen Model-Index den Tree-Entry ermitteln, für den Daten abgefragt werden.
Parameter:
i_modelIdx (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
i_iRole (IN) Datentype: int
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: QVariant
Wert der abgefragten Zelle.

2.5.5.66 setData(const QModelIndex&, const QVariant&, int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel. Werden Model-Indizes erzeugt (siehe Methode *index*), wird deren *InternalPointer* auf den Tree-Entry Eintrag gesetzt. Damit kann die *setData*-Methode anhand des übergebenen Model-Index den Tree-Entry ermitteln, für den Daten gesetzt werden sollen. Die Methode sendet sowohl die Signale *objectChanged* als auch nameSpaceChanged des Model aus, je nachdem, ob ein Eintrag vom Typ EntryTypeNameSpace oder EntryTypeObject geändert wurde. Natürlich wird auch das Signal dataChanged gesendet, wie von der Qt-Model/View Architektur gefordert, um etwaige an das Model angebundene Views zu aktualisieren.
Parameter:
i_modelIdx (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
i_value (IN) Datentype: const QVariant&
siehe Dokumentation der Klasse QAbstractItemModel
i_iRole (IN) Datentype: int
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: bool

2.5.5.67 index(int, int, const QModelIndex&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel. Werden Model-Indizes erzeugt, wird deren *InternalPointer* auf den Tree-Entry Eintrag gesetzt. Damit können andere Methoden anhand des übergebenen Model-Index den Tree-Entry ermitteln.
Parameter:
i_iRow (IN) Datentype: int
siehe Dokumentation der Klasse QAbstractItemModel
i_iCol (IN) Datentype: int
siehe Dokumentation der Klasse QAbstractItemModel
i_modelIdxParent (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: QModelIndex

2.5.5.68 parent(const QModelIndex&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende Schnittstellen-Methode (*must overridable*) der Klasse QAbstractItemModel. Werden Model-Indizes erzeugt (siehe Methode *index*), wird deren *InternalPointer* auf den Tree-Entry Eintrag gesetzt. Damit kann die *parent*-Methode anhand

des übergebenen Model-Index den Tree-Entry ermitteln, für dessen *Parent* der ModelIndex erzeugt werden soll.
Parameter:
i_modelIdx (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: QModelIndex

2.5.5.69 headerData(int, Qt::Orientation, int)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende virtuelle Methode der Klasse QAbstractItemModel, um die Spalten des an das Model angebundenen Views zu beschriften.
Parameter:
i_iSection (IN) Datentype: int
Je nach *Orientation* entspricht die Section entweder der Zeilen- oder der Spalten-Nummer (siehe Dokumentation der Klasse QAbstractItemModel).
i_iSection (IN) Datentype: Qt::Orientation
siehe Dokumentation der Klasse QAbstractItemModel
i_iRole (IN) Datentype: int
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: QVariant

2.5.5.70 flags(const QModelIndex&)

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende virtuelle Methode der Klasse QAbstractItemModel, um festzulegen, welche Spalten editierbar sind. In diesem Model sind keine Spalten editierbar.
Parameter:
i_modelIdx (IN) Datentype: const QModelIndex&
siehe Dokumentation der Klasse QAbstractItemModel
Rückgabewert: Datentype: Qt::ItemFlags

3 Anwendungsbeispiele

Ein einfaches Kodierungs-Beispiel zur Verwendung des Object-Pool-Models können Sie in der Beispiel-Applikation ZSAppObjPoolModel erhalten. Konkrete und ausführlichere Anwendungsfälle finden sich innerhalb der Pakete ZSTrace und ZSPhysVal wieder.

Im Folgenden soll aber auch noch kurz erläutert werden, wie das Object-Pool-Model in einer Multi-Threading Applikationen eingesetzt werden könnte.

Das Model wird zentral angelegt und mehrere Threads können asynchron (quasi gleichzeitig) auf das Model zugreifen. Server-Komponenten instanziieren Objekte, weisen den Objekten systemweit eindeutige Namen zu und fügen die Objekte dem Model hinzu. Clients wollen mit den Objekten innerhalb der Server-Komponente via Messages kommunizieren. Damit der Server die empfangenen Nachrichten schnell den adressierten Objekten zuordnen kann (und um den Speicherbedarf durch die Messages gering zu halten), sollen die Clients nicht den eindeutigen Namen zum Adressieren der Objekte, sondern deren ID verwenden. Um die ID zu erhalten, lässt sich der Client die ID vom Object-Model anhand des eindeutigen Objekt-Namens geben. Das Auffinden der Objekt ID muss nur einmalig durch den Client geschehen (sofern der Client nicht zwischenzeitlich zerstört wird).

Beispiel:

Irgendwo an zentraler Stelle (z.B. als Member der von QApplication abgeleiteten Applikations-Klasse) wird eine Instanz des Object-Pool-Models angelegt und die Referenz auf das Model wird über eine entsprechende Methode allen anderen Komponenten zugänglich gemacht.

Die Server-Komponente erhält erzeugt ein Objekt mit einem eindeutigen Namen und fügt das Objekt dem Model hinzu:

```
CWdgtDiagram* pWdgtDiagram = new CWdgtDiagram();

int m_iObjIdDiagram = pModelObjPool->addObj(
    /* strClassNameSpace */ "ZS::Diagram",
    /* strClassName      */ "CWdgtDiagram",
    /* strObjNameSpace   */ "Rf::NWA",
    /* strObjName        */ pWdgtDiagram->getObjName(),
    /* pObj              */ pWdgtDiagram );
```

Wird das Objekt zerstört, muss der Server das Objekt auch wieder vom Model entfernen. Dies sollte über die ID des Objekts geschehen:

```
pModelObjPool->removeObj(m_iObjIdDiagram);
```

Dabei wird das Objekt aber nicht wirklich aus dem Model entfernt sondern wird lediglich in den entsprechenden Baum- und Listen-Einträgen als Destroyed gekennzeichnet.

Ein Client, der mit dem Objekt des Servers kommunizieren möchte, kann sich die ID des Objekts vom Model anhand des Namens des Objekts geben lassen:

```
if( m_iObjId == -1 )
{
    m_iObjId = pModelObjPool->getObjId(
        /* strClassNameSpace */ "ZS::Diagram",
        /* strClassName      */ "CWdgtDiagram",
        /* strObjNameSpace   */ "Rf::NWA",
        /* strObjName        */ "Power Versus Time" );
}
if( m_iObjId != -1 )
{
    QObject* pObj = pModelObjPool->getObj(m_iObjId);

    if( pObj != NULL )
    {
        CMsg* pMsg = new CMsg(...);
        QApplication::postEvent( pObj, pMsg );
    }
}
```

Im obigen Code-Fragment prüft der Client zunächst, ob er sich die ID nicht schon besorgt hat. Wenn nicht, lässt er sie sich vom Model geben. Ist das Objekt im Model eingetragen (ID != -1), lässt sich der Client eine Referenz auf das Objekt geben lassen, um die Message direkt an das Objekt zu schicken. Vorher wird aber noch geprüft, ob das Objekt aktuell instanziiert ist.

Anmerkung:

Das Beispiel ist mit Vorsicht zu genießen, denn es muss dafür Sorge getragen werden, dass das Objekt nicht just zwischen den beiden Zeitpunkten zerstört wird, nachdem der Client die Message an das Objekt verschickt hat und Qt die Message an das Objekt über dessen event-Methode weiterleitet. Objekte, die Messages aus anderen Threads entgegennehmen sollen, sollten dementsprechend über die QObject-Methode *deleteLater* zerstört werden.

©2009 ZeusSoft, Ing. Büro Bauer, Bad Heilbrunn, Germany