GIRA

Logikbaustein SDK Dokumentation

Stand: 24.06.2019

Inhalt

1 Über diese Dokumentation

Diese Dokumentation führt den Entwickler an das Logikbaustein Software Development Kit (SDK) heran, sodass dieser in der Lage ist, Logikbausteine für den Logik Editor im Gira Projekt Assistenten (GPA) zu entwickeln. Dazu wird ein beigefügtes Visual Studio Projekt verwendet, in dem die Logikbausteine in C# programmiert werden.

1.1 Zielgruppe

Diese Dokumentation richtet sich an Personen, die bereits folgende Kenntnisse haben:

- Entwickeln in der Programmiersprache C#
- Grundlegendes Verständnis über das JSON Format
- Verwenden der Entwicklungsumgebung Visual Studio 2017
- Verwenden des Gira Projekt Assistenten
    - Funktionsweise von Datenpunkten
    - Umgang mit dem Logikeditor

2 Grundlagen

2.1 Logikbausteine

Mithilfe des im Gira Projekt Assistenten (GPA) befindlichen Logik-Editors können Logikblätter erstellt werden. Diese Logikblätter sind visuelle Darstellungen logischer Schaltungen, die nach der Inbetriebnahme des GPA Projekts auf dem Gerät (z. B. Gira X1 oder Gira L1) ausgeführt werden. Logikblätter bestehen aus Logikbausteinen und Verbindungen zwischen diesen.

Ein Logikbaustein nimmt über einen oder mehrere Eingänge Werte entgegen, verarbeitet diese gemäß seiner internen Programmierung und gibt dann neue errechnete Werte an einem oder mehreren Ausgängen aus. Die Werte an den Ausgängen werden über eine Verbindung, die im Logikblatt durch eine Linie zwischen einem Ein- und einem Ausgang dargestellt wird, an die Eingänge eines anderen Logikbausteins weitergegeben. Dadurch ist es möglich, auch aufwendige logische Schaltungen zu erstellen. Die Logik kommuniziert mit dem Rest des Projekts bisher nur über die Logikbausteine *Eingang* und *Ausgang*, deren Werte mit beliebigen Datenpunkten verknüpft sind. Andere Kommunikationswege können mit dem SDK entwickelt werden.

Der GPA enthält standardmäßig schon einige Logikbausteine z. B. das *Oder-Gatter*, den *PID-Regler* oder den *Vergleicher*. Mit dem Logikbaustein SDK bekommt der Entwickler die Möglichkeit, eigene Logikbausteine zu erstellen und in Logikblättern zu verwenden. Neue Logikbausteine sind für spezielle Anwendungsfälle sehr nützlich. In diesem Logikbaustein SDK sind bereits einige Beispiele enthalten z. B. die *Aggregation.* Mithilfe dieses Bausteins ist es unter anderem möglich, den Mittelwert der an den Eingängen anliegenden Werte zu errechnen. Für diesen Anwendungsfall ist es sehr hilfreich, einen eigenen Logikbaustein zu haben, da es sehr aufwendig ist, die Mittelwertberechnung durch Verkettung anderer Logikbausteine durchzuführen und weil der neu erstellte Logikbaustein in vielen Projekten wiederverwendet werden kann.

3 Verwendung der Beispiele

Als Teil des Logikbaustein SDK stehen in den Ordners *Example* und *Template* jeweils eine Visual Studio 2017 Solution zur Verfügung. Die Solution im Ordner *Example* beinhaltet ein Beispiel-Projekt mit dem Namen *ExampleNodes*, welches mehrere verschiedene Bausteine implementiert. Außerdem existiert im Ordner *Template* eine vorbereitete Solution mit dem Namen *LogicNodes*, welches als Vorlage für neue Logikbausteine verwendet werden kann. Des Weiteren gibt es zu beiden Projekten je ein Unit-Test-Projekt.

Die Unit-Tests verwenden das *NUnit* Framework, welches mit der in Visual Studio integrierten *NuGet* Paketverwaltung heruntergeladen wird. Die Tests werden mit dem Test Explorer von Visual Studio ausgeführt. Das *LogicNodesTest* Projekt enthält lediglich einen leeren Testfall, der dem Entwickler zur Verfügung steht, um eigene Tests für den Logikbaustein zu schreiben.

Wenn das *ExampleNodes* Projekt erfolgreich gebaut ist, liegt im Unterordner *Zip* der Solution die Datei *LogicNodesSDK.Logic.Examples-1.0.0.zip*, welche die Beispiel-Bausteine enthält. Die Logikbausteine in der .zip Datei sind noch nicht kryptografisch signiert. Das bedeutet, dass sie zwar im Gira Projekt Assistent (GPA) importiert und in der Simulation getestet werden können, allerdings können unsignierte Logikbausteine nicht auf das ausführende Gira Gerät übertragen werden.

Um die Bausteine in den GPA zu installieren, wird in der Kachel Logik-Editor die Schaltfläche *Logikbausteine hinzufügen* verwendet.

Für die Signierung muss zuerst ein Zertifikat beantragt werden. Der gesamte Prozess ist in der nachfolgenden Sektion 4 Signierung von Logikbausteinen beschrieben.

4 Signierung von Logikbausteinen

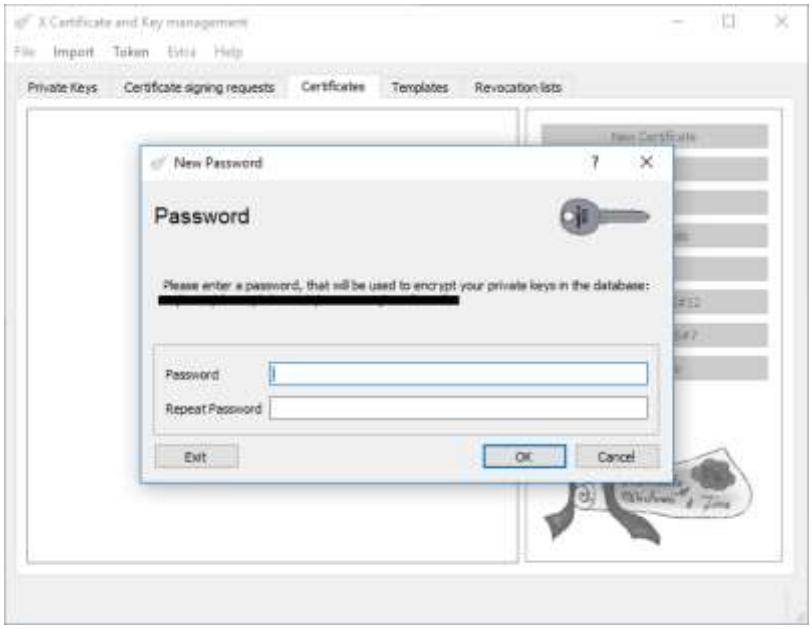
Logikbausteine werden mit dem im SDK enthaltenen *SignLogicNodes* Programm signiert. Dieses Programm benötigt dafür ein von Gira ausgestelltes PKCS#12 Zertifikat. Der Entwickler muss eine Zertifikatsanfrage stellen, um dieses zu erhalten.

4.1 Erstellen einer Zertifikatsanfrage

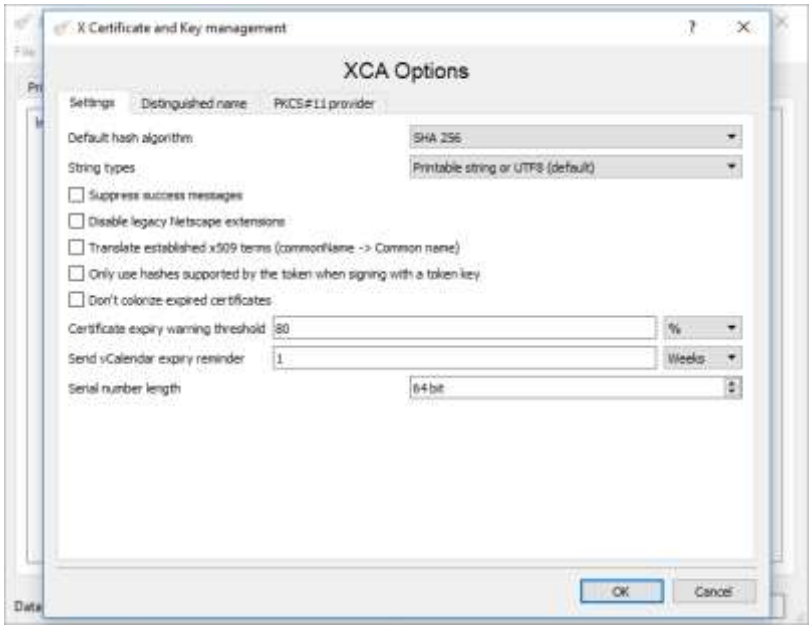
Es wird empfohlen, das *X Certificate and Key Management* Programm zu verwenden, um die Zertifikatsanfrage zu erstellen. Das Programm steht unter https://hohnstaedt.de/xca/ zur Verfügung. Es muss installiert und gestartet werden. Die Menüführung wird in diesem Dokument anhand der englischen Spracheinstellung beschrieben.

Wenn zum ersten Mal Zertifikate bzw. Schlüssel erstellt werden, muss eine Datenbank für das Programm angelegt werden. Dazu wird der Menüpunkt *File->New DataBase* ausgewählt. Alternativ kann auch eine bereits existierende Datenbank mit *File->Open DataBase* importiert werden.

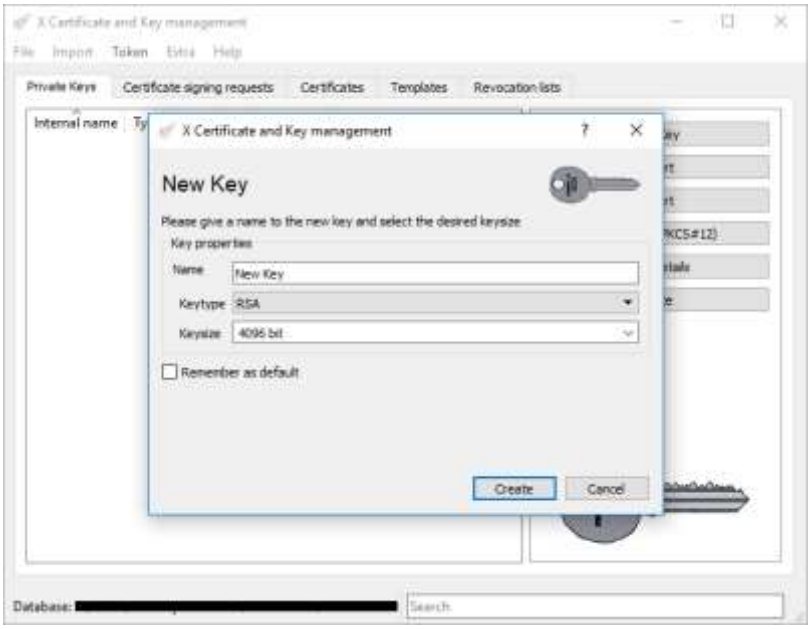
Jede Datenbank ist mit einem Passwort geschützt und kann betriebssystemunabhängig verwendet werden. Dieses Passwort wird beim Erstellen der Datenbank festgelegt.



Zunächst muss sichergestellt sein, dass der *Standard Hash Algorithmus* auf *SHA 256* eingestellt ist und diese Einstellung ggf. vorgenommen wird. Diese Einstellung ist unter dem Menüpunkt *File -> Options* zu finden.



Es muss ein privater Schlüssel erzeugt werden. Dazu wird im *Private Keys* Reiter die Schaltfläche *New Key* verwendet und ein beliebiger Name für den Schlüssel festgelegt. In den Aufklappmenüs *Keytype* und *Keysize* müssen *RSA* und *4096 Bit* ausgewählt sein.



Mit dem privaten Schlüssel wird eine Signierungsanfrage gestellt. Dazu muss der *Certificate signing requests* Reiter ausgewählt sein. Mit Klick auf die *New Request* Schaltfläche öffnet sich ein neues Fenster. In diesem Fenster müssen unter dem Reiter *Subject* personenbezogene Daten des Entwicklers eingetragen werden:

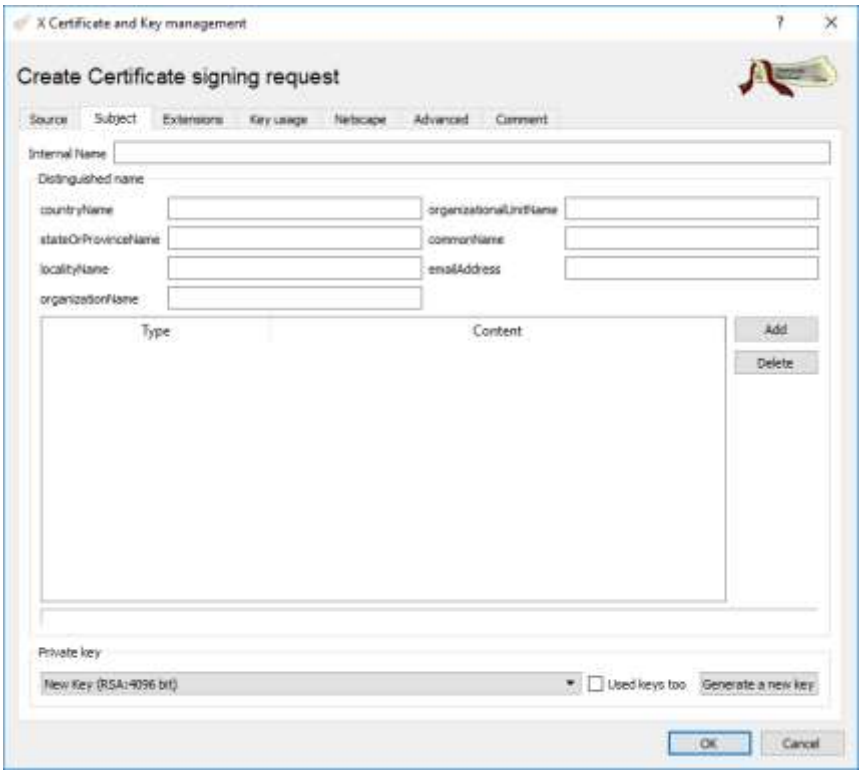
Internal Name	Anzeigename der Zertifikatsanfrage
countryName	Länderkürzel des Standorts (z. B. DE für Deutschland)
localityName	Stadtname des Standortes
organizationName	Firmenname (leer für Privatpersonen)
commonName	Name des Antragstellers
emailAddress	E-Mail Adresse des Antragstellers

Soll das Zertifikat für eine Privatperson ausgestellt werden, so muss das Feld *organizationName* leer gelassen werden.

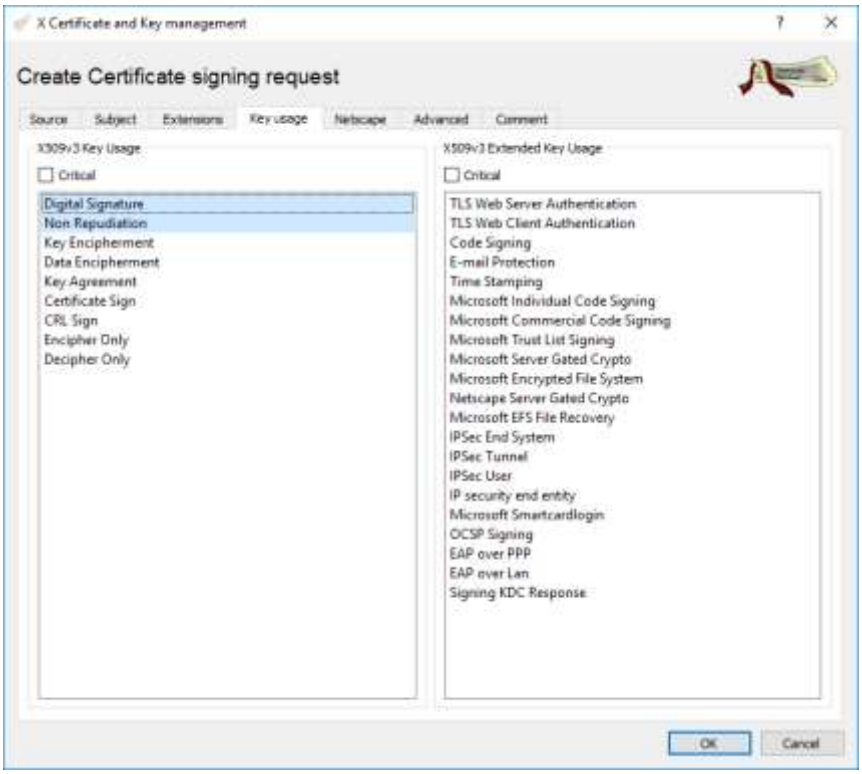
Der hier eingetragenen Daten sind für die Anzeige im GPA sowie der Gerätewebsite vorgesehen, so dass sich Nutzer bei Problemen direkt an den Herausgeber dieser Logikbausteine wenden können.

Ihre personenbezogenen Daten werden auf Grundlage der Datenschutzerklärung, die diesem Dokument beigefügt ist, erhoben und verarbeitet.

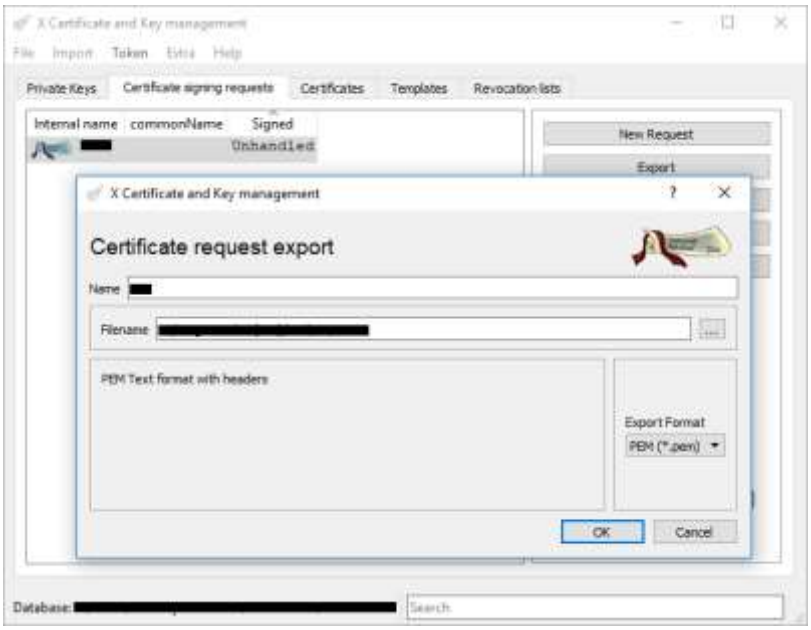
In dem Aufklappmenü *Private key* muss ein privater Schlüssel ausgewählt sein, hier kann der eben erstellte Schlüssel ausgewählt werden.



Unter dem Reiter *Key usage* müssen *Digital Signature* und *Non Repudiation* durch Anklicken ausgewählt werden. Mit diesen Angaben ist die Zertifikatsanfrage vollständig und wird mit einem Klick auf die Schaltfläche *OK* generiert.



Um diese Zertifikatsanfrage zu exportieren, wird die Schaltfläche *Export* verwendet. Unter *Export Format* muss das *PEM* Format ausgewählt werden. Diese *.pem* Datei muss per E-Mail an *developer@gira.de* gesandt werden. *Ihre personenbezogenen Daten werden auf Grundlage der Datenschutzerklärung, die diesem Dokument beigefügt ist, erhoben und verarbeitet.* Ihre Zertifikatsanfrage wird dann bearbeitet und Sie sollten nach einer gewissen Zeit eine Antwort von Gira erhalten, in der Ihr signiertes Zertifikat enthalten sein wird.



HINWEIS

Bitte senden Sie niemals Ihre X Certificate Datenbank, Ihren privaten Schlüssel oder eines Ihrer Passwörter an Gira. Geben Sie diese Dateien bzw. Informationen auch an Niemanden weiter, der nicht in der Lage sein soll, Logikbausteine in Ihrem Namen zu veröffentlichen.

4.2 Importieren des erstellten Zertifikats von Gira

Sobald die Zertifikatsanfrage bearbeitet wurde, erhält der Anfragensteller ein von Gira signiertes Zertifikat, welches in das *X Certificate and Key Management* Programm importiert wird. Dazu wird im Reiter C*ertificates* mit der Schaltfläche *Import PKCS#7* ein Dialog geöffnet, in dem das erhaltene Zertifikat ausgewählt werden muss. Der Benutzer wird in einem sich öffnenden Fenster gefragt, welche Zertifikate der Zertifikatskette importiert werden sollen. Da alle Zertifikate importiert werden müssen, wird hier die Schaltfläche *Import All* ausgewählt.

In dieser importierten Zertifikatskette muss das zuvor erstellte, und jetzt von Gira signierte, Zertifikat exportiert werden. Dazu wird die gesamte Zertifikatskette im Reiter *Zertifikate* ausgeklappt und das unterste Zertifikat wird ausgewählt. Mit einem Klick auf die Schaltfläche

Export wird das Zertifikat exportiert. In dem sich öffnenden Dateidialog muss als Dateiformat ***PKCS#12 chain*** *(*.p12)* auswählt werden.

Das von Gira unterschriebene Zertifikat ist standardmäßig 10 Jahre lang gültig. Auf den Geräten wird jedoch die Gültigkeitsdauer von den Zertifikaten der Logikbausteine nicht geprüft, so dass auch nach Ablauf der Gültigkeitsdauer die Logikbausteine weiterhin funktionsfähig sind.

Nach Ablauf dieser 10 Jahre kann ein neuer Zertifikatsantrag gestellt werden, um neue Logikbausteine mit einem aktuellen Zertifikat signieren zu können.

4.3 Verwendung des Zertifikats um Logikbausteine zu signieren

Die Logikbausteine werden mithilfe des im SDK enthaltenen Programms *SignLogicNodes* signiert.

Das Programm muss mit drei Parametern aus der Kommandozeileneingabe heraus aufgerufen werden: Dem Pfad zu einem signierten Zertifikat, dem Passwort dieses Zertifikats und dem Pfad zu der zu signierenden Logikbaustein *.zip* Datei.

SignLogicNodes.exe <p12 Zertifikat> <Passwort> <Pfad zur Logikbaustein .zip Datei>

Alternativ kann dieser Schritt auch als *Post-Build-event* im Projekt eingetragen werden.

Der signierte Baustein wird, wie im Kapitel 3 Verwendung der Beispiele beschrieben, mit der Schaltfläche *Logikbausteine hinzufügen* im GPA hinzugefügt. Dadurch können die Logikbausteine im GPA verwendet und auch als Projektbestandteil auf einen Gira X1 bzw. ein Gira L1 hochgeladen werden.

5 Beschreibung der API

5.1 Vorwort

Das Logikbaustein SDK stellt eine Programmierschnittstelle oder kurz API zur Verfügung, um Logikbausteine zu entwickeln. Die Verwendung dieser Logikbaustein API (im Weiteren nur noch API genannt) stellt sicher, dass die entwickelten Logikbausteine mit dem GPA und den anderen Logikbausteinen kompatibel sind.

5.2 Erstellen einer neuen Logikbaustein Klasse

Jeder neu entwickelte Logikbaustein wird als Klasse in C# implementiert.

```
using LogicModule.Nodes.Helpers;
using LogicModule.ObjectModel;

public class Node : ILogicNode
{
  public Node(INodeContext context)
  {
    ...
  }
  ...
}
```

Die Klasse *Node* erbt dabei von der Klasse *ILogicNode*, welche im Namespace *LogicModule.ObjectModel* existiert. Der Konstruktor der Klasse *Node* muss so implementiert werden, dass dieser eine Variable vom Typ *INodeContext* übergeben bekommt. Der Typ *INodeContext* existiert ebenfalls im Namespace *LogicModule.ObjectModel*.

ILogicNode ist eine Interfaceklasse, es müssen daher alle Methoden dieser Klasse *Execute*, *Startup*, *Localize* und *Validate* in der Klasse *Node* implementiert werden. Mehr zu diesen Methoden ist in Kapitel 5.4 zu finden. Die Klasse *LogicNodeBase* implementiert für diese Methoden sinnvolle Dummys. Daher ist es, vor allem für den Einstieg, sinnvoll, den Baustein von *LogicNodeBase* erben zu lassen. Diese befindet sich im Namespace *LogicModule.Nodes.Helpers.*

Die Variable *context* ist der Zugangspunkt zur API. Über dieses Interface fragt der Baustein spezielle Services an, um auf die verschiedenen Funktionen wie Datenpunkte, Zeiten oder Eingangs- und Ausgangs-Erzeugung zuzugreifen. Die zur Verfügung stehenden Dienste werden im nachfolgenden Kapitel 5.3 beschrieben.

5.3 Dienste der API

Die API stellt für die unterschiedlichen Funktionalitäten Dienste zur Verfügung. Instanzen dieser Dienste werden aus dem *INodeContext* Objekt im Konstruktor abgerufen. Dafür muss die Zeile

```
‘ServiceName’ typeService = context.GetService<‘ServiceName’>();
```

aufgerufen werden. Der Platzhalter *‘ServiceName’* ist dabei der Typ des gewünschten Dienstes. Die folgenden Dienste sind in der API verfügbar.

5.3.1 ITypeService

Der *ITypeService* stellt Methoden zur Verfügung, um die Eigenschaften der Ports des Logikbausteines zu definieren. Ein Port ist dabei ein Eingang, Ausgang oder Parameter. Ports werden durch *ValueObjects* repräsentiert. Der *ITypeService* stellt Methoden zur Verfügung, um konkrete Instanzen von *ValueObjects* zu erzeugen. Der folgende Abschnitt listet die Methoden auf, mit deren Hilfe Instanzen der verschiedenen Typen erzeugt werden.

```
AnyValueObject CreateAny(
  string typeName, string name, object defaultValue = null);

BoolValueObject CreateBool(
  string typeName, string name, bool? defaultValue = null);

ByteValueObject CreateByte(
  string typeName, string name, byte? defaultValue = null, string unit = null);

IntValueObject CreateInt(
  string typeName, string name, int? defaultValue = null, string unit = null);

DoubleValueObject CreateDouble(
  string typeName, string name, double? defaultValue = null, string unit = null);

UIntValueObject CreateUInt(
  string typeName, string name, uint? defaultValue = null, string unit = null);

UShortValueObject CreateUShort(
  string typeName, string name, ushort? defaultValue = null, string unit = null);

TimeSpanValueObject CreateTimeSpan(
 string typeName, string name, TimeSpan? defaultValue = null, string unit = null);

DateTimeValueObject CreateDateTime(
 string typeName, string name, DateTime? defaultValue = null, string unit = null);

StringValueObject CreateString(
  string typeName, string name, string defaultValue = null);
```

Diese Methoden sind nach einem Schema aufgebaut:

- Der erste Parameter ist der Name des zu erzeugenden Typs. Hierüber wird definiert, welchen Datentyp der Port hat. Ein Beispiel ist "NUMBER", um einen Port zu definieren, der einen Wert vom Typ *Number* entgegennimmt bzw. weitergibt.
- Der zweite Parameter ist der Name des Ports bzw. *ValueObjects*. Dieser wird auch im GPA als *Portname* angezeigt, sofern er nicht durch eine Übersetzung in eine andere Sprache überschrieben wird, siehe dazu Kapitel 5.5. Der *Portname* muss innerhalb eines Logikbausteins eindeutig sein.
- Der dritte Parameter ist immer optional und ist der Initialwert, den der Port beim Erzeugen des Bausteins, also beim Hinzufügen zum Logikblatt, hat. Bevor die Logik gestartet wird, kann der Benutzer des GPA den Wert des Ports jedoch ändern oder sogar löschen. Ports deren Werte nicht vorhanden sind oder gelöscht wurden, werden mit dem Wert *null* gelesen.
- Der vierte Parameter ist nur bei einigen Methoden vorhanden und auch immer optional. Er gibt an, in welcher physikalischen Einheit der Zahlenwert des Ports zu interpretieren ist, z. B. ob eine Länge in Metern, Zentimetern oder Fuß angegeben ist. Dies dient lediglich der Information des Benutzers des Bausteins und wird vom GPA als Porteigenschaft angezeigt.

Die Typen existieren im Namespace *LogicModule.ObjectModel.TypeSystem*.

Außer den obigen Funktionen stellt *ITypeService* noch die Methoden *CreateValueObject* und *CreateEnum* zur Verfügung.

```
IValueObject CreateValueObject(
  string typeName, string name, object defaultValue = null);

EnumValueObject CreateEnum(
  string typeName, string name, string[] allowedValues, string defaultValue=null);
```

Diese Methoden sind ähnlich zu den anderen aus der Liste der erzeugenden Funktionen. Ihre Parameter werden größten Teils so verwendet wie bereits beschrieben. Sie benötigen aber noch eine gesonderte Erläuterung.

Die Methode *CreateValueObject* erzeugt eine Instanz des Typs *IValueObject.* Das *IValueObject* ist ein Interfacetyp und kann nicht direkt verwendet werden, *IValueObject* muss durch einen Cast in einen der anderen *ValueObject* Typen umgewandelt werden. Es ist damit möglich, alle anderen Funktionen aus der Liste der zu erzeugenden Funktionen zu ersetzen. Es empfiehlt sich aber, die spezifischeren Funktionen zu verwenden, um den Cast einzusparen.

Mit der *CreateEnum* Methode wird ein neuer *ValueObject* Typ definiert. Die möglichen Werte dieses Typs sind Strings aus einer Liste. Der Parameter *typeName* legt den Namen des neuen Typs fest, sodass später weitere Instanzen dieses Enumtyps angelegt werden können. Die möglichen Werte, die der Enumtyp annehmen kann, werden mit *allowedValues* als Array aus Strings übergeben.

Falls eine weitere Instanz eines zuvor definierten Enumtyps erzeugt werden soll, bietet *CreateEnum* eine Überladung an.

```
EnumValueObject CreateEnum(
  string typeName, string name, string defaultValue = null);
```

Hier fehlt lediglich das Array mit den möglichen Werten. Der Parameter *typeName* muss der Namen eines zuvor definierten Enumtyps sein.

Der Beispiel-Logikbaustein *Farben-Konverter* verwendet einen Parameter vom Enumtyp, um die beiden möglichen Richtungen der Konvertierung auswählen zu können.

Die vollständige Portdefinition wird in Kapitel 5.3.1b behandelt. Hier wird zunächst auf das Typsystem eingegangen.

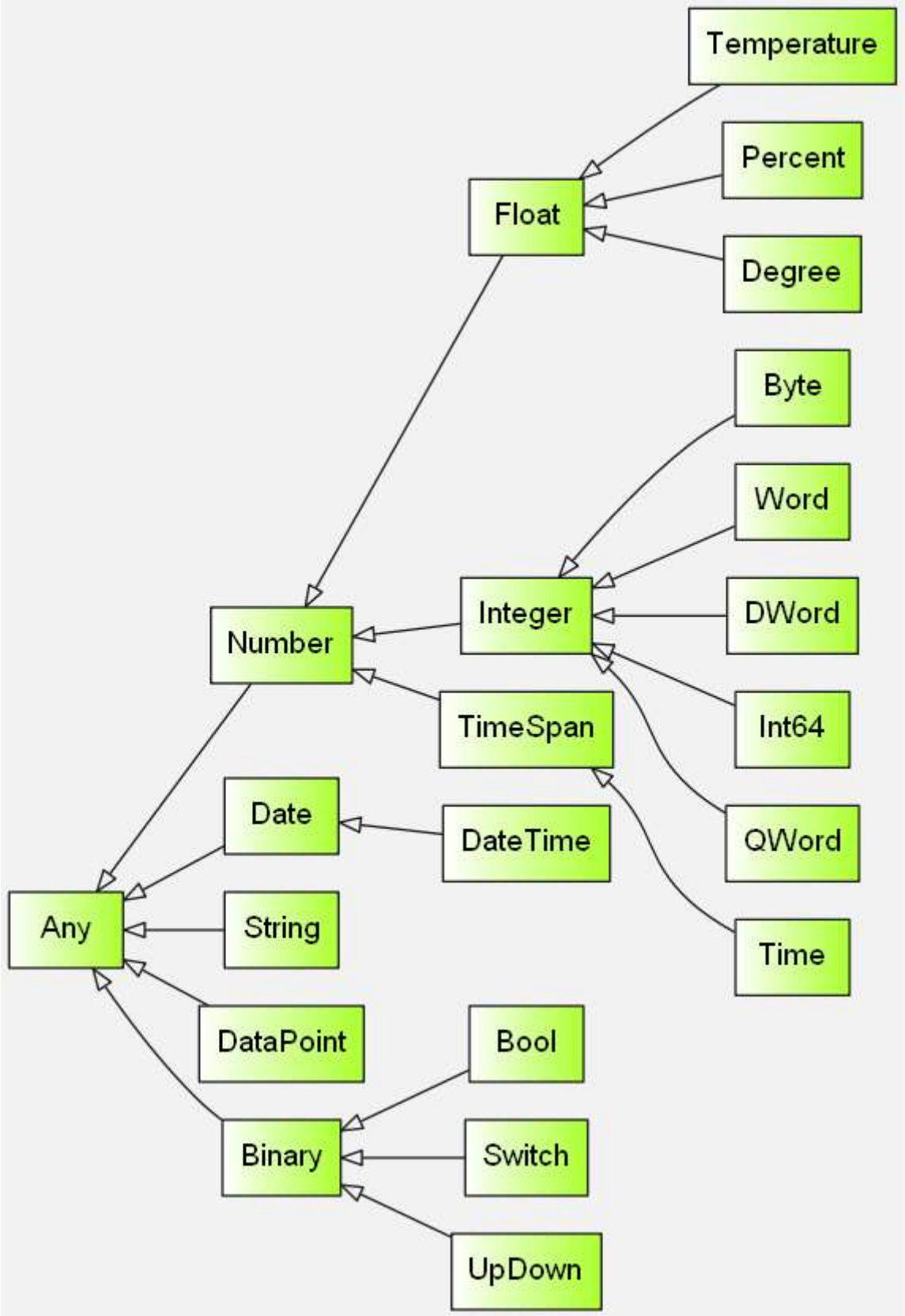
5.3.1a Das Typsystem

Das Typsystem stellt verschiedene Datentypen zur Verfügung, um Ports sinnvoll zu typisieren. Hierdurch wird verhindert, dass Ports verschiedener Bausteine verbunden werden, deren Verbindung keinen Sinn ergeben. Zum Beispiel ist eine Verbindung eines Ausgangs vom Typ "STRING" mit einem Eingang vom Typ "NUMBER" nicht sinnvoll. Auf der anderen Seite ist zu vermeiden, dass für jeden Typ ein eigener Baustein implementiert werden muss, um die Anzahl der Bausteine und die Entwicklungskosten klein zu halten. Es empfiehlt sich daher, den Typ der Ports so allgemein wie möglich, aber so spezifisch wie nötig zu wählen.

Ein Beispiel hierfür ist ein Logikbaustein, der die Werte zweier oder mehrerer Eingänge addiert und das Ergebnis am Ausgang ausgibt. Für diesen Anwendungsfall ist es sinnvoll, für die Typen der Ports "NUMBER" zu wählen. Dadurch ist sichergestellt, dass die Ports mit Ports anderer Bausteine verbunden werden können, die z. B. den Typ *Integer*, *Float* oder *Byte* tragen. Ein anderes denkbares Beispiel ist die Winkeladdition. Solch ein Logikbaustein addiert die Werte an den Eingängen, wendet eine Modulo 360 Operation auf das Ergebnis an und

schreibt das Ergebnis auf den Ausgang. Für diesen Baustein ist es sehr sinnvoll, den Ports den Typ "DEGREE" zuzuweisen. Dadurch wird verhindert, dass versehentlich die Ports mit anderen Ports verbunden werden, die inkompatibel mit dem Typ "DEGREE" sind.

In der Abbildung zum Typsystem sind alle implementierten Typen und ihre Beziehungen untereinander dargestellt.


Temperature
Percent
Float
Degree
Byte
Word
Integer
DWord
Number
TimeSpan
Int64
Date
DateTime
QWord
Any
String
Time
DataPoint
Bool
Binary
Switch
UpDown

5.3.1b Vollständige Portdefinition

Eine Portdefinition erfolgt in zwei Schritten. Als erstes wird in der Klasse des Logikbausteins eine Property angelegt.

```
[Input(DisplayOrder = 1, IsRequired = true)]
public DoubleValueObject Input { get; private set; }
```

Die Property muss *public* sein, wobei der *setter* aber auf *private* gesetzt werden kann. Der Name der Property ist vom Entwickler frei wählbar. Der Typ der Property muss zu dem gewünschten Porttyp passen. In dem Beispiel wird ein *DoubleValueObject* definiert, damit im zweiten Schritt der Porttyp "NUMBER" gewählt werden kann. An dem Attribut *Input* ist erkennbar, dass hier ein Eingang definiert wird. Insgesamt gibt es drei Attribute für Ports:

- *Input* zur Definition eines Eingangs
- *Output* zur Definition eines Ausgangs
- *Parameter* zur Definition eines Parameters

Am Beispiel Input Property ist zu sehen, dass dem Attribut Parameter übergeben werden können. Folgende Parameter stehen zur Verfügung.

Port-Parameter

DisplayOrder

DisplayOrder ist für alle Attribute (Input, Output, Parameter) verfügbar. Mit dem Wert dieses Parameters wird die Anzeigereihenfolge im GPA beeinflusst. Je kleiner der Wert von *DisplayOrder* ist, desto weiter oben wird der entsprechende Port in der visuellen Darstellung im Logikeditor des GPA angezeigt. Im unteren Beispiel wird *Output1* vor *Output2* angezeigt.

```
[Output(DisplayOrder = 2)]
public IValueObject Output2 { get; set;}


[Output(DisplayOrder = 1)]
public IValueObject Output1 { get; set;}
```

Dabei gilt es zu beachten:

- Für den Fall, dass *DisplayOrder* nicht gesetzt wurde, ist die Reihenfolge undefiniert.
- *DisplayOrder* gilt für das Attribut *Output* getrennt von dem der anderen beiden Attribute, sodass Ausgänge im GPA weiterhin immer rechts am Baustein angezeigt werden und Eingänge und Parameter immer links.
- Der Wert von *DisplayOrder* muss größer oder gleich 1 sein, alle anderen Werte werden so interpretiert, als sei der Wert nicht gesetzt.

InitOrder

Der Parameter *InitOrder* ist für die Attribute *Input* und *Parameter* verfügbar. Hierüber wird die Reihenfolge der Initialisierung beeinflusst. Im unteren Beispiel wird zuerst der Parameter *Datapoint* und dann der Eingang *Trigger* initialisiert. Ist *InitOrder* nicht gesetzt, so ist die Reihenfolge undefiniert.

```
[Input(InitOrder = 2)]
public AnyValueObject Trigger { get; set; }

[Parameter(InitOrder = 1)]
public DataPointValueObject Datapoint { get; private set; }
```

Hier ist wieder zu beachten, dass der niedrigste mögliche Wert 1 ist. Alle Werte kleiner als 1 werden als nicht gesetzt interpretiert.

AsTitle

Dieser Parameter ist nur für das Attribut Parameter verfügbar. Es handelt sich hierbei um ein Flag, daher können nur die Werte *true* oder *false* zugewiesen werden. Ist dieses Flag auf *true* gesetzt, zeigt dies an, dass der Parameter von fundamentaler Bedeutung für den Baustein ist. Pro Baustein kann höchstens bei einem Parameter dieses Flag auf *true* gesetzt sein. Wird solch ein Baustein dem Logikblatt hinzugefügt, so erwartet der GPA, dass sofort ein Wert für diesen Parameter eingetragen wird. Fehlt der Wert, wird die entsprechende Textbox vom GPA gelb hinterlegt. Der eingetragene Wert für diesen Port wird als Titel des Bausteins eingesetzt. In den Logikbausteinen "Eingang" und "Ausgang" wird diese Funktion verwendet, um den Benutzer dazu aufzufordern, einen Datenpunkt einzutragen, dessen Name dann als Titel des Bausteins angezeigt wird.

```
[Parameter(AsTitle = true)]
public DataPointValueObject Datapoint { get; private set; }
```

IsDefaultShown

Alle drei Attribute bieten den Parameter *IsDefaultShown* an. Auch hierbei handelt es sich um ein Flag. Es legt fest, ob der Port standardmäßig angezeigt wird oder ausgeblendet ist. Bei einem Parameter wird hierüber festgelegt, ob der Wert des Parameters standardmäßig am Baustein angezeigt wird. Die Voreinstellung für dieses Flag ist *true*, sodass jeder Port beim Erzeugen des Bausteins angezeigt wird, es sei denn, das Flag wurde aktiv auf *false* gesetzt.

```
[Input]
public IValueObject Input1 { get; private set; }

[Input(IsDefaultShown = true)]
public IValueObject Input2 { get; private set; }

[Input(IsDefaultShown = false)]
public IValueObject Input3 { get; private set; }
```

Die beiden Eingänge *Input1* und *Input2* werden im GPA standardmäßig am Baustein angezeigt, wohingegen *Input3* nicht standardmäßig angezeigt wird. Unabhängig von diesem Flag kann der Benutzer des GPA die Ports ein- oder ausblenden. Mit diesem Flag wird nur die Standardeinstellung, die der Port beim Hinzufügen des Logikbausteins zum Logikblatt hat, festgelegt.

IsRequired

Der Parameter *IsRequired* ist auch für alle drei Attribute verfügbar, wirkt sich auf die unterschiedlichen Attribute aber etwas anders aus. Dieser Parameter ist wieder ein Flag und kann daher nur die Werte *true* oder *false* annehmen, die Voreinstellung ist *false*. Ist der Wert des Flags auf *true* gesetzt, gelten für alle drei Attribute, dass der Port im GPA nicht mehr ausgeblendet werden kann. Falls zusätzlich *IsDefaultShown = false* angegeben ist, wird dies mit der Einstellung *IsRequired = true* überschrieben und der Port wird standardmäßig

angezeigt. Für Eingänge ist es im Logikeditor des GPA möglich, zwischen zwei Zuständen umzuschalten. Entweder der Eingang erhält seinen Wert von einem Ausgang eines anderen Bausteins oder der Benutzer muss händisch einen festen Wert eintragen. Für Eingänge mit *IsRequired = true* entfällt diese Umschaltmöglichkeit und der Eingang muss seinen Wert immer von einem Ausgang eines anderen Bausteins beziehen.

```
[Input(IsRequired = true)]
public IValueObject Input { get; set; }
```

IsInput

Dieser Parameter ist nur für das Attribut Input verfügbar, daher nur für Eingänge und auch nur für die *IsRequired = true* nicht gesetzt wurde. Es handelt sich wieder um ein Flag mit der Voreinstellung *true*. Es legt fest, wie der Eingang dem Benutzer angezeigt wird, ob standardmäßig ein fester Wert eingetragen werden muss oder ob der Eingang mit einem Ausgang verbunden werden muss. Dieses Flag betrifft nur die voreingestellte Anzeige, wenn der Baustein dem Logikblatt hinzugefügt wird. Es besteht für den Benutzer weiterhin die Möglichkeit, den Baustein zu konfigurieren und zwischen den beiden Arten der Wertezuweisung hin und her zu schalten.

```
[Input]
public IValueObject Input1 { get; private set; }

[Input(IsInput = false)]
public IValueObject Input2 { get; private set; }

[Input(IsInput = true)]
public IValueObject Input3 { get; private set; }
```

Der Eingang *Input2* wird im GPA beim Erzeugen des Bausteins so angezeigt, dass ein fester Wert durch den Benutzer eingetragen werden muss, wohingegen *Input1* und *Input3* so angezeigt werden, dass eine Verbindung zu einem Ausgang hergestellt werden muss.

Instanziierung

Wurden alle gewünschten Port-Parameter mit Werten versehen, so folgt der nächste Schritt der Portdefinition im Konstruktor des Bausteins. Die Properties der Ports müssen instanziiert werden, indem ihnen eine durch das *ITypeService* Objekt erstellte Instanz zugewiesen wird. Wie dies funktioniert, wird am besten an einem Beispiel deutlich.

```
public class Node : LogicNodeBase
{
  ...
  [Input(DisplayOrder = 2, IsDefaultShown = false, IsInput = true)]
  public BoolValueObject Reset { get; private set; }
  ...

  public Node(INodeContext context)
  : base(context)
  {
    ...
    ITypeService typeService = context.GetService<ITypeService>();
    this.Reset = typeService.CreateBool(PortTypes.Binary, "Reset", false);
    ...
  }
}
```

Im Konstruktor der Klasse *Node* wird dem Eingang *Reset* der Anzeigename *Reset* und der Initialwert *false* zugeordnet. Außerdem wird der Porttyp auf "BINARY" gesetzt. Alle verfügbaren Porttypnamen sind in der statischen Klasse *PortTypes* als konstante Stringvariablen vorhanden, sodass dem Entwickler eine Hilfestellung für die Schreibweise gegeben ist.

Bei einigen Typen des Typsystems besitzen die Instanzen Metainformationen, wie z. B. einen minimalen oder einen maximalen Wert. Diese werden nach der Instanziierung gesetzt und abgerufen. Für *ValueObjects* vom Typ "NUMBER" und den abgeleiteten Typen gibt es einen minimalen und maximalen Wert, der bei Bedarf festgelegt werden kann.

```
this.Count = typeService.CreateInt(PortTypes.Integer, "Count", 1);
this.Count.MinValue = 1;
this.Count.MaxValue = 10;
```

Die eingestellten minimalen und maximalen Werte werden vom GPA als Porteigenschaft angezeigt. Für *StringValueObjects* kann eine maximale zulässige Länge festgelegt werden, diese wird über den Parameter *MaxLength* eingestellt.

```
this.InputStr = typeService.CreateString(PortTypes.String, "Input String", "42");
this.InputStr.MaxLength = 12;
```

Auch diese Metainformation wird vom GPA als Porteigenschaft angezeigt. Einige Porttypen haben implizite minimale und maximale Werte. Beispielsweise befindet sich der Wert eines Typ "BYTE" Ports immer zwischen 0 und 255. Es kann festgelegt werden, wie sich der Logikbaustein verhält, falls auf einen Port ein Wert geschrieben wird, der sich außerhalb des zulässigen Bereichs befindet. Es gibt zwei mögliche Einstellungen:

- Saturation: Der Wert wird automatisch angepasst, um in den gültigen Wertebereich zu liegen. Wenn der Wert z. B. den gesetzten oder impliziten Maximalwert übersteigt, so wird er durch den Maximalwert ersetzt, welchen der Port dann übernimmt.
- Exception: Der falsche Wert führt zu einer Ausnahme in der Logik. Dies bedeutet, dass der Port keine Werteänderung übernimmt. Außerdem wird der Fehler aufgezeichnet und der Teil der Logik, in welchem der Baustein sich befindet, angehalten.

Das voreingestellte Verhalten ist das unter Saturation beschriebene. Gesetzt wird diese Metainformation durch die Zeile

```
this.Count.OutOfRangeBehavior = OutOfRangeBehavior.Saturation;
```

Liste von Ports

Für viele Logikbausteine ist es notwendig, dass die Anzahl der Ein- oder Ausgänge dynamisch während des Erstellens des Logikblatts eingestellt wird. Diese Funktionalität wird unter anderem in dem Beispielbaustein *Aggregation* verwendet.

Um Ports dynamisch zu erzeugen, wird eine Liste von Ein- oder Ausgängen als Property der Bausteinklasse definiert. Die Syntax ist dabei sehr ähnlich zur Definition eines einzelnen Ein- oder Ausgangs.

```
[Input]
public List<DoubleValueObject> Inputs { get; private set; }
```

Im Konstruktor der Bausteinklasse muss diese Liste initialisiert werden.

```
this.Inputs = new List<DoubleValueObject>();
```

Um die Liste mit Instanzen des entsprechenden Ports zu füllen, werden zwei Hilfsfunktionen angeboten. Diese befinden sich im Namespace *LogicModule.Nodes.Helpers* innerhalb der Klasse *ListHelpers.*

```
void UpdateListLength<T>(

IList<T> list, int newCount, Func<int, IValueObject> creator,
EventHandler<ValueChangedEventArgs> handler = null)

where T : IValueObject;


void ConnectListToCounter<T>(

IList<T> list, ValueObject<int> countParameter, Func<int, IValueObject> creator,
EventHandler<ValueChangedEventArgs> handler, Action listLengthChangedCallback =
null)

where T : IValueObject;
```

Die Funktion *UpdateListLength* füllt die Liste von Ein- oder Ausgängen mit einer festen Anzahl an Instanzen. Der erste Parameter muss dabei die Liste selbst sein. Der zweite Parameter gibt die geforderte Listenlänge an. Um die Liste zu füllen, benötigt die Funktion eine Vorschrift, um neue *IValueObject* Instanzen zu erzeugen, diese ist der dritte Parameter *creator*. Wie diese Vorschrift definiert wird, ist weiter unten beschrieben. Außerdem gibt es einen optionalen vierten Parameter, den Eventhandler *handler*. Dieser wird dem *ValueSet* Ereignis (siehe weiter unten ValueSet), das bei der Änderung der Listenlänge aufgerufen wird, den neu erstellen Ports hinzugefügt bzw. bei den gelöschten entfernt. Das hat den Effekt, dass die durch *handler* repräsentierte Methode aufgerufen wird, sobald sich einer der Werte der Ports aus der Liste ändert.

Ein gängiger Anwendungsfall ist, dass der Benutzer des GPA die erforderliche Anzahl von Ports manuell angibt. Das ist auch bei dem Beispielbaustein *Aggregation* der Fall. Dem Benutzer steht ein Parameter Port zur Verfügung, um die gewünschte Anzahl an Ports vom Benutzer einzutragen. Mit der Methode *ConnectListToCounter* wird der Wert des Parameters automatisch mit der Listenlänge verknüpft. Der erste Parameter dieser Funktion ist die dynamische Liste der Ports, der zweite ist der Parameter Port, dessen Wert die Listenlänge angibt. Man beachte, dass der Porttyp dieses Parameters *Integer* sein muss. Der kleinste zulässige Wert für die Listenlänge ist "0", daher müssen die Werte, die der Parameter annehmen kann, durch die Properties *MinValue* und *MaxValue* auf positive Werte beschränkt werden. Der dritte und der vierte Parameter der Funktion sind analog zu denen von *UpdateListLength*. Der fünfte Parameter ist eine optionale Callback-Funktion, die bei jeder Längenänderung der Portliste aufgerufen wird.

Der Parameter *creator* lässt sich mithilfe des *ITypeService* Objekts erzeugen.

```
Func<int, IValueObject> creator =
this.typeService.GetValueObjectCreator(PortTypes.Number, InputPrefix);
```

Dabei ist der erste Parameter von *GetValueObjectCreator* der Porttyp der Ports, die mit *creator* erzeugt werden, und der zweite ein String, der als Namenspräfix für die zu erzeugenden Ports verwendet wird. Ein Beispiel: Der Entwickler wählt "Eingang " für *InputPrefix* und übergibt die

so erzeugte Variable *creator* an die Funktion *UpdateListLength* oder *ConnectListToCounter*. Diese Methoden legen somit Ports mit den Namen "Eingang 1", "Eingang 2", "Eingang 3" usw. an. Man beachte, dass die Nummerierung der Namen mit "1" beginnt, der Listenindex des ersten Ports allerdings "0" ist.

5.3.2 ISchedulerService

Das *ISchedulerService* Interface stellt Methoden zur Verfügung, mit deren Hilfe Aktionen zu einem bestimmten Zeitpunkt ausgeführt werden. Außerdem kann auch die aktuelle Systemzeit abgefragt werden. Das Interface bietet dabei eine Abstraktionsschicht zur wirklichen Systemzeit. Durch diese Abstraktion ist es möglich, eine andere Zeit zu simulieren, z. B. bei der Simulation eines Logikblatts im GPA. Ein direkter Zugriff auf die Systemzeit über das *.NET-Framework* ist im Baustein zu vermeiden.

Das *ISchedulerService* Interface stellt folgende Methoden und Properties zur Verfügung.

```
SchedulerToken InvokeIn(TimeSpan delay, Action action);

SchedulerToken InvokeAt(DateTime dueTime, Action action);

bool Remove(SchedulerToken schedulerToken);

DateTime Now;
```

Über die Methode *InvokeIn* wird eine Aktion *action* eingestellt, die nach der Zeit *delay* ausgeführt wird. Analog dazu wird die Aktion, die der Methode *InvokeAt* übergeben wird, zu dem festen Zeitpunkt *dueTime* nach UTC ausgeführt. Beide Funktionen liefern einen Token vom Typ *SchedulerToken* zurück. Bei Bedarf kann dieser Token in einer Variable gespeichert werden, die der Methode "Remove" übergeben werden kann, um die entsprechende Aktion wieder aus der Warteschlange der Aktionen zu entfernen. Der Typ *SchedulerToken* ist ein nullbarer Typ. Mit *Now* kann die abstrahierte lokale Systemzeit abgefragt werden. Hinweis: Die Methode *Now* liefert die lokale Zeit, wobei *InvokeAt* eine Zeitangabe nach UTC erwartet.

Wie diese Methoden ineinandergreifen, wird in dem folgenden Beispiel gezeigt.

```
public class Enterprise : LogicNodeBase
{
  ...
  private ISchedulerService schedulerService;
  private SchedulerToken abortToken;
  ...
  public Enterprise(INodeContext context)
    : base(context)
  {
    ...
    this.schedulerService = context.GetService<ISchedulerService>();
    TimeSpan countdown = new TimeSpan(0, 0, 10, 0);  //10min
    this.abortToken = this.schedulerService.InvokeIn(countdown, this.SelfDestruction);
    //Do not lose this token or let the Klingons find it
    ...
  }
  ...
  private void SelfDestruction()
  {
    ...  //Explosion
  }
  private void Abort()
  {
    if (abortToken != null)
    {
      this.schedulerService.Remove(this.abortToken);
      this.abortToken = null;
    }
  }
  ...
}
```

5.3.3 IPersistenceService

Mit diesem Interface speichert der Baustein Werte über die Laufzeit des Gerätes hinweg. Folgende Methoden stehen zur Verfügung.

```
void SetValue(ILogicNode node, string key, string value);

string GetValue(ILogicNode node, string key);

void DeleteValue(ILogicNode node, string key);
```

Alle drei Methoden benötigen als ersten Parameter die Instanz des aufrufenden Logikbausteins, es muss also *this* übergeben werden. Der zweite Parameter ist der Schlüssel, mit dem auf den Wert zugegriffen wird. Die Methode *SetValue* speichert oder überschreibt den Wert *value* zu dem Baustein *node* und dem Schlüssel *key*. Die Methode *GetValue* ruft den von *SetValue* gespeicherten Wert wieder ab und gibt ihn als Rückgabeparameter aus. Ist zu dem Schlüssel *key* kein Wert abgespeichert, liefert *GetValue null* zurück. Mit der Methode *DeleteValue* wird der Wert wieder gelöscht. Es ist nur möglich Werte als String zu speichern und abzurufen.

Die gespeicherten Werte sind auch noch nach einem Neustart des Geräts, auf dem der Logikbaustein benutzt wird, mit *GetValue* abrufbar. Auch bei einer erneuten Inbetriebnahme des GPA-Projekts in dem derselbe (mit derselben ID) Logikbaustein verwendet wird, ist ein gespeicherter Wert weiterhin verfügbar. Die ID des Logikbausteins wird automatisch erzeugt, sobald der Baustein dem Logikblatt hinzugefügt wird. Durch die Baustein ID ist ein Wert

eindeutig einem Baustein und einem Schlüssel (*key*) zuordenbar. Damit ist es ausgeschlossen, dass ein Baustein auf die Werte eines anderen Bausteins zugreift. Das bedeutet auch, dass Werte, die über das *IPersistenceService* Interface gespeichert wurden, verloren gehen, sobald der Baustein aus dem Logikblatt entfernt wird.

Die gespeicherten Werte werden gelöscht oder sind nicht mehr verfügbar, wenn

- Der Logikbaustein aus dem Logikblatt entfernt wird.
- Das Gira Gerät einem anderen Projekt zugeordnet wird bzw. die Zuordnung entfernt wird.
- Eine Inbetriebnahme mit gesetztem Flag "Applikationsdaten löschen" durchgeführt wird.
- Das Gira Gerät auf Werkseinstellungen zurückgesetzt wird.

Die Methoden *SetValue* und *GetValue* werfen eine *CommunicationException* Ausnahme, falls der Zugriff auf die Werte fehlschlägt oder zu lange dauert.

5.3.4 IEditorService

Die Methoden dieses Interfaces beeinflussen die Editierbarkeit des Logikblattes und sind mit großer Vorsicht zu verwenden. Der Einsatz dieses Interfaces ist nur im Konstruktor und den dort eventuell registrierten Eventhandlern sinnvoll, weil es nur die im GPA angezeigten Porttypen beeinflusst und damit nur die Bedienung während der Erstellung des Logikblattes. Es gibt eine Reihe von Bausteinen, bei denen ein Wert ohne nähere Auswertung unter bestimmten Bedingungen durchgeleitet wird. Der *Send-By-Change* Baustein ist dafür ein Beispiel. Für diesen Baustein darf keine Einschränkung bezüglich der Porttypen gemacht werden, daher verwenden die entsprechenden Eingänge und Ausgänge den Porttyp "ANY". Hat der Benutzer einen der Ports mit einem Port eines anderen Bausteins verknüpft, ist klar, dass sowohl der Eingang als auch der Ausgang des *Send-By-Change* Bausteins den Porttyp dieses externen Ports übernehmen müssen. Um solche Szenarien abbilden zu können, werden vom *IEditorService* folgende Methoden angeboten.

```
ISharedTypeToken RegisterSharedType(ILogicNode node, params IValueObject[] ports);

void ClearAllSharedTypeRegistration(ILogicNode node);
```

Die Methode *RegisterSharedType* fasst eine Gruppe von Ports, die der Funktion aufgelistet übergeben werden, zu einer Gruppe zusammen. Wird einer der Ports aus der Gruppe beim Erstellen des Logikblattes mit einem anderen Port eines spezifischeren Porttyps (Im Bild zum Typenmodel heißt das, dass ein Typ der weiter "rechts" angeordnet ist als der eigentlich Porttyp.) verknüpft, so übernehmen alle anderen Ports der Gruppe auch diesen Typen. Als ersten Parameter muss der Methode die *this* Referenz auf den aufrufenden Baustein übergeben werden. Der Rückgabeparameter ist ein Token, der verwendet wird, um die Gruppe zu einem späteren Zeitpunkt wieder aufzulösen. Dafür stellt der Typ *ISharedTypeToken* die Methode *Unregister* zur Verfügung.

```
void Unregister();
```

Nach Aufruf dieser Methode ist die Weitergabe des Porttyps zwischen den Ports der Gruppe wieder aufgehoben. Mit der Methode *ClearAllSharedTypeRegistration* werden alle erstellten Gruppen von Ports wieder aufgelöst.

Das Beispiel des *Send-By-Change* Bausteins ist besonders einfach. Im Konstruktor wird lediglich eine Zeile aufgerufen, um die Porttypen des Eingangs und des Ausgangs miteinander zu verknüpfen.

```
context.GetService<IEditorService>().RegisterSharedType(
this, this.Input, this.Output);
```

5.4 Funktionalität des Bausteins Programmieren

Dieses Kapitel beschreibt, wie die Funktionalität des Bausteins implementiert wird. Die Basisklasse *ILogicNode,* von der jede Bausteinklasse direkt oder indirekt erbt, stellt zwei Methoden zur Verfügung, die von der Bausteinklasse implementiert werden müssen. Als Beispiel soll der folgende Code betrachtet werden.

```
public class Node : ILogicNode
{
 public Node(INodeContext context)
 {
  ...
 }
 public override void Startup()
 {
  ... //This code is executed once when the logic engine starts up
 }
 public override void Execute()
 {
  ... //This code is executed every time a value is written to an input,
      //and all inputs have a value
 }
 ...
}
```

Die Methode *Startup* wird beim Start des Bausteins ein einziges Mal ausgeführt. Diese Methode ist sehr praktisch, um zu testen, ob Eingängen oder Parametern des Bausteins bei der Konfigurierung Werte zugewiesen wurden, und um diese Werte auszulesen. Die Klasse *LogicNodeBase* implementiert eine leere *Startup* Methode. Zentral für die Funktion des Bausteins ist die Methode *Execute*. Diese wird jedes Mal aufgerufen, wenn ein Wert (nicht zwangsläufig ein neuer Wert) durch die Verknüpfung zu einem anderen Baustein auf einen Eingang geschrieben wird und alle Eingänge einen gültigen (nicht *null*) Wert haben. In der Methode werden die Werte abgerufen, verarbeitet und ggf. die Ausgänge aktualisiert. Auch hier implementiert die Klasse *LogicNodeBase* eine leere *Execute* Methode.

5.4.1 Properties von *IValueObject*

WasSet

WasSet ist ein Flag und nur in der *Execute* Methode sinnvoll verwendbar. Diese Eigenschaft wird auf *true* gesetzt, wenn der Wert des entsprechenden Eingangs gesetzt wird. Es ist damit bestimmbar, welcher Eingang Auslöser für den Aufruf der *Execute* Methode ist. Dort kann auf unterschiedliche Weise auf die Wertänderungen an den verschiedenen Eingängen reagiert werden. Das ist z. B. für den Fall nützlich, wenn der Baustein zwei Eingänge hat: Einen Reset-Eingang, der beim Erhalt eines Wertes den Baustein in einer definierten Form zurücksetzt und einen Eingang, der die eigentliche Funktion des Bausteins auslöst. Nach der Ausführung der *Execute* Methode werden die Werte von *WasSet* aller Eingänge automatisch wieder auf *false* gesetzt.

HasValue

HasValue ist ebenfalls ein Flag. Über diese Eigenschaft kann abgefragt werden, ob der Wert dieses Ports gültig ist, also nicht *null*. Dies kann bei Ports ohne Initialwert der Fall sein oder wenn der Initialwert durch den Benutzer entfernt wurde. Es ist daher für alle Ein- und Ausgänge erforderlich, vor dem Abrufen des Wertes zu prüfen, ob ein Wert verfügbar ist.

Value

Diese Property ist von zentraler Bedeutung für die *IValueObject* Instanz. Sie enthält den eigentlichen Wert. Der Datentyp ist durch den Porttypen festgelegt, für *BoolValueObject* ist *Value* immer vom Typ *bool*, für *UIntValueObject* ist *Value* immer vom Typ *uint* usw. Für eine nicht näher spezifizierte Instanz von *IValueObject* und für *AnyValueObject* Instanzen ist der Typ von *Value object*. *Value* ist sowohl schreibbar als auch lesbar.

ValueEquals

Mit der Methode *ValueEquals* wird geprüft, ob der in der *IValueObject* Instanz gespeicherte Wert mit dem übergebenen Wert übereinstimmt. Dies ist eine kurze Schreibweise für Wertevergleiche. *ValueEquals* überprüft intern vor dem Wertevergleich mit *HasValue*, ob der Wert an dem *IValueObject* gesetzt ist. Für den eigentlichen Wertevergleich benutzt *ValueEquals* die Methode *Object.Equals* bzw. deren Überladungen aus dem *.NET Framework*.

BlockGraph

Diese Property ist nur sinnvoll für Ausgänge und darf auch nur für diese verwendet werden. *BlockGraph* setzt den Wert des Ausgangs auf *null* und blockiert die Ausführung aller nachgeschalteten Logikbausteine. Diese ist z. B. nützlich für den Fall, wenn der Baustein eine asynchrone Operation durchführt und auf einen Rückgabewert dieser Operation wartet. Während des Wartens ist die Ausführung der nachgeschalteten Logikbausteine gestoppt. Erst wenn wieder ein Wert auf den Ausgang, für den *BlockGraph* aufgerufen wurde, geschrieben wird, wird die Ausführung der nachgeschalteten Logik mit dem neu geschrieben Wert fortgesetzt.

HINWEIS: Die Simulation eines Logikblattes im GPA gibt das Verhalten von *BlockGraph* nicht korrekt wieder.

ValueSet

Diese Property ist ein Eventhandler und für alle Ports verfügbar. Die Wertänderung eines Ports löst ein Ereignis aus, welches alle unter *ValueSet* registrierten Funktionen aufruft. Mithilfe des "+=" Operators wird eine weitere Funktion für dieses Ereignis registriert.

```
this.Direction.ValueSet += this.DirectionOnValueSet;
```

Dieses Beispiel stammt aus dem Programmcode des Beispiel Bausteins *ColorConverter*. Hier wird für den Parameter *Direction* die Funktion *DirectionOnValueSet* dem *ValueSet* Ereignis hinzugefügt. Jedes Mal, wenn der Wert des Parameters gesetzt wird, wird automatisch die Funktion *DirectionOnValueSet* aufgerufen. Nützlich ist die Property *ValueSet* vor allem, wenn, wie für den *ColorConverter* Baustein, Eingänge und Ausgänge dynamisch zu erzeugt werden.

5.5 Überprüfen der Logikbausteine

Die Interfaceklasse *ILogicNode* enthält auch die Methode *Validate,* diese muss ebenfalls in der Bausteinklasse überschrieben werden. Die Methode *Validate* ist nicht für den Aufruf durch den Baustein vorgesehen, sondern wird nur vom GPA aufgerufen. Sie dient dazu, dem GPA mitzuteilen, ob der Baustein sinnvoll initialisiert wurde. Die Validierung der Logikbausteine kann im GPA an verschiedenen Stellen aufgerufen werden, z. B. durch Starten der Simulation eines Logikblattes durch den Benutzer. Jedes Mal, wenn die Simulation des Logikblattes gestartet wird, ruft der GPA die *Validate* Methoden aller Bausteine auf. Meldet dabei ein Logikbaustein einen Fehler, wird der Start der Simulation abgebrochen und im GPA eine Fehlermeldung angezeigt. Ebenso werden die Logikblätter validiert, wenn der Benutzer die Inbetriebnahme des Projekts startet oder manuell die Prüfung des Projekts auslöst. Für diese beiden Fälle wird dem Benutzer im Infofenster angezeigt, für welchen Logikbaustein die Validierung fehlgeschlagen ist und welche Fehlermeldung der Baustein generiert. Diese Funktionalität wird z. B. im Logikbaustein *Schwellwert mit Hysterese* verwendet, um sicherzustellen, dass der untere Schwellwert kleiner ist als der obere.

```
public override ValidationResult Validate(string language)
{
  if ((this.LowerThreshold.HasValue && this.UpperThreshold.HasValue) &&
       this.LowerThreshold.Value > this.UpperThreshold.Value)
  {
    return new ValidationResult { HasError = true, Message =
          this.Localize(language,"HysteresisValidationErrorMessage") };
  }
  return base.Validate(language);
}
```

Die Methode hat ein *ValidationResult* Objekt als Rückgabeparameter. *ValidationResult* hat die Properties *HasError*, ein Flag, das angibt, ob ein Validierungsfehler vorliegt (*false* bedeutet, dass kein Fehler vorliegt) und *Message*, welche vom Typ *string* ist. Wenn das Flag *HasError true* ist, wird dem Benutzer *Message* im Infofenster angezeigt. Der Übergabeparameter *language* ist ein Sprachkürzel, mit dem der GPA die Fehlermeldung *Message* in einer bestimmten Sprache bei der Methode anfragt (Details zu Übersetzungen werden im Kapitel 5.6 beschrieben). Die Implementierung dieser Methode in *LogicNodeBase* liefert den festen *HasError=false* und eine leere *Message* zurück. Die Validierung ist für diese Klasse also immer erfolgreich.

5.6 Übersetzung der Logikbausteine

Die Logikbausteine können dem Benutzter in unterschiedlichen Sprachen zur Verfügung gestellt werden. Dafür muss die Methode *Localize* der Interfaceklasse *ILogicNode* in der

Bausteinklasse implementiert werden. Diese Methode wird vom GPA aufgerufen, um jeden im Baustein vorhandenen Portnamen zu übersetzen.

```
string Localize(string language, string key);
```

Die Struktur ist dabei sehr einfach, *key* ist der zu übersetzende String und *language* das Kürzel der Sprache, in die der Inhalt von *key* übersetzt werden soll. Der Rückgabewert von *Localize* ist das übersetzte Ergebnis, welches vom GPA als Portnamen angezeigt wird. Die *Localize* Methode wird auch in dem Beispiel aus Kapitel 5.5 bausteinintern verwendet um z. B. die Fehlermeldung *Message* zu übersetzen. Der GPA übergibt der *Localize* Methode das Sprachkürzel der im GPA eingestellten Sprache. Im GPA sind die folgenden Sprachen verfügbar.

Sprachkürzel nach ISO 639-1	Sprache
de	Deutsch
en	Englisch
es	Spanisch
nl	Niederländisch
ru	Russisch
zh	Chinesisch
(it)	(Italienisch)

Italienisch ist noch nicht (GPA v3.1) verfügbar, soll in späteren Versionen aber nutzbar werden.

Sehr komfortabel ist, dass die Klasse *LogicNodeBase* die *Localize* Methode implementiert. Im Folgenden wird beschrieben, wie die Implementierung der *Localize* Methode in den Klassen *LogicNodeBase* und *LocalizablePrefixLogicNodeBase* verwendet wird.

Die *Localize* Methode von *LogicNodeBase* verwendet den *ResourceManager* aus dem Namespace *System.Resources*. Diese Klasse ist im *.NET Framework* dokumentiert. Als Übersetzungstabellen werden *.resx* Dateien herangezogen. Mit Visual Studio lassen sich *.resx* Dateien komfortabel erstellen und editieren. Die Dateien müssen im selben Verzeichnis wie die Projektdatei des Visual Studio vorhanden sein. Pro Sprache muss eine *.resx* Datei angelegt werden und ggf. eine Standard *.resx* Datei. Wenn es nur eine Ressourcen-Datei (inkl. mehrere Sprachen) gibt, wird diese Datei zur Übersetzung benutzt. Falls mehrere Dateien gefunden werden, wird vorzugsweise die Datei *Resources.<Sprachkürzel>.resx* verwendet, andernfalls eine zufällige. <Sprachkürzel> entspricht dem Sprachenkürzel der Sprache, die durch diese Übersetzungsdatei zur Verfügung gestellt wird. Der Dateiname der Standard *.resx* Datei enthält kein Sprachkürzel, also <name>.resx.

Visual Studio stellt die *.resx* Dateien im Editor als Tabelle dar. In der ersten Spalte *Name* werden die zu übersetzenden Strings eingetragen, in der zweiten Spalte *Value,* die entsprechende Übersetzung. Für den Fall, dass die mit dem Parameter *language* angefragte Sprache keine Übersetzungsdatei hat oder der entsprechende *key* nicht übersetzt wurde, greift die *Localize* Methode auf die Standard *.resx* Datei zurück. Es empfiehlt sich solch eine Standardübersetzungsdatei anzulegen, da dadurch die angezeigten Portnamen benutzerfreundlicher gestaltet werden. Im Quellcode trägt der Entwickler unter Umständen Abkürzungen für die Portnamen ein oder vermeidet Leer- und Sonderzeichen. Mit der Standardübersetzung werden die Portnamen nachträglich angepasst. Für den Fall, dass weder ein sprachspezifische noch eine Standardübersetzung gefunden wurde, gibt diese *Localize* Methode den Eingabewert *key* zurück.

Die Übersetzung ist etwas aufwendiger, wenn der Baustein Ports in Form einer Liste definiert. Speziell für diesen Fall steht im Namespace *LogicModule.Nodes.Helpers* die Klasse *LocalizablePrefixLogicNodeBase* zur Verfügung. Diese erbt von *LogicNodeBase* und überschreibt nur die *Localize* Methode. Es bietet sich für den Entwickler ggf. an, die Bausteinklasse von *LocalizablePrefixLogicNodeBase* erben zu lassen. Der Konstruktor dieser Klasse benötigt zwei Übergabeparameter.

```
public LocalizablePrefixLogicNodeBase(INodeContext context, string inputPrefix)
```

Der Parameter *context* muss von der Bausteinklasse an diese Basisklasse übergeben werden, *inputPrefix* ist das Präfix der Portnamen aus der Portliste, wie es auch der Methode *GetValueObjectCreator* übergeben wird. Die *Localize* Methode behandelt Strings (*key*), die mit dem Präfix beginnen, separat und ersetzt nur das Präfix durch die in der *.resx*" Datei gegebene Übersetzung dieses Präfixes. Außerdem fügt die *Localize* Methode ein Leerzeichen zwischen dem Präfix und dem Rest des Strings ein, falls die Übersetzung des Präfixes nicht mit einem Leerzeichen endet. Alle anderen Strings (*key*), die nicht mit dem Präfix *inputPrefix* beginnen, behandelt diese Methode genauso wie die *Localize* Methode der Basisklasse *LogicNodeBase*. Der Entwickler muss darauf achten, dass die Portnamen und das Präfix so gewählt sind, dass nicht ein anderer Portname unabsichtlich mit dem Präfix beginnt.

Es ist auch möglich, dass ein Baustein mehr als eine Portliste enthält, z. B. eine für Eingänge und eine für Ausgänge. Diese Portlisten haben unterschiedliche Präfixe. Für diesen Fall bietet das SDK keine vorimplementierte *Localize* Methode an, sodass der Entwickler diese selbst implementieren muss.

6 Die Manifest.json Datei

Jeder Logikbaustein Library benötigt eine *Manifest.json* Datei, in der die Metadaten der enthaltenden Logikbausteine definiert sind. Diese Datei muss in dem gleichen Ordner wie die Quellen der Logikbausteine selbst vorhanden sein. In dem Logikbaustein SDK sind zwei *Manifest.json* Dateien enthalten: Eine für das *ExampleNodes* Projekt und eine für das *LogicNodes* Projekt. Die jeweiligen Visual Studio Projekte enthalten diese *Manifest.json* Dateien bereits, sodass sie einfach aus dem Visual Studio heraus editierbar sind.

6.1 Anpassen der Codierung

Damit Sonderzeichen, wie z. B. Umlaute, in den Namen und Beschreibungen der Logikbausteine im GPA korrekt angezeigt werden, muss die *Manifest.json* Datei mit der *UTF-8-BOM* Codierung gespeichert werden.

In Visual Studio ist die Codierung einstellbar unter:

File -> Save<Dateiname> As.... Im Filesave-Dialog muss im Aufklappmenü neben *Save* der Punkt *Save with Encoding...* auswählt werden. In dem sich öffnenden Fenster *Advanced Save Options* muss unter *Encoding* die Option *Unicode (UTF-8 with signature) - Codepage 650001* gewählt werden.

6.2 Beispiel Manifest.json Datei

```
{
  "PackageFormatVersion": "1.0",
  "Assembly": "LogicNodesSDK.Logic.Examples.dll",
  "PackageName": {
    "en": "Example logic nodes",
    "de": "Beispiellogikbausteine"
  },
  "DependentFiles": [
    "LogicModule.Nodes.Helpers.dll"
  ],
  "Version": "1.0.0",
  "Author": "Gira Giersiepen GmbH & Co. KG",
  "Copyright": "Gira Giersiepen GmbH & Co. KG (copyright)",
  "DeveloperId": "LogicNodesSDK",
  "License": "Free",
  "PackageId": "183FBF6C-AE53-4393-A2A5-2CBE0F1696AF",
  "Nodes": [
    {
      "Type": "LogicNodesSDK.Logic.Examples.BasicNode",
      "Name": {
        "en": "Examples: Basic logic node",
        "de": "Beispiele: Einfacher Logikbaustein"
      },
      "IsConverter": false,
      "Category": "Node",
      "DefaultIcon": "icons/GenericLogicNode.png",
      "HelpTooltip": {
        "en": "Basic logic node that sets the output to the same value as the input.",
        "de": "Einfacher Logikbaustein, der den Ausgang auf den gleichen Wert wie den
Eingang setzt."
      },
      "HelpFileReference": null
    },
    {
      "Type": "LogicNodesSDK.Logic.Examples.Aggregation",
      "Name": {
        "en": "Examples: Aggregation",
        "de": "Beispiele: Aggregation"
      },
      "IsConverter": false,
      "Category": "Node",
      "DefaultIcon": "icons/GenericLogicNode.png",
      "HelpTooltip": {
        "en": "Calls aggregate functions for multiple inputs: minimum, maximum, sum,
average.",
        "de": "Ruft Aggregatfunktionen für mehrere Eingänge auf: Minimum, Maximum,
Summe, Durchschnitt."
      },
      "HelpFileReference": null
    }
  ]
}
```

6.3 Beschreibung der Einträge in der Manifest.json Datei

Schlüssel	Beispielwert	Verwendung
PackageFormatVersion	"1.0"	Die Version des Formats der Manifest.json. Immer „*1.0*“.
Assembly	"LogicNodesSDK.Logic.Examples.dll"	Dateiname der Logikbaustein Bibliothek bzw. der Name der DLL.
PackageName	{ "en": "Example logic nodes", "de": "Beispiellogik-bausteine" }	Ein JSON Objekt, dessen Schlüssel Länderkürzel sind und dessen Werte der Name des Bausteinbibliothek in der jeweiligen Sprache sind.
DependentFiles	["LogicModule.Nodes.Helpers.dll"]	JSON Array mit Dateinamen der benötigten DLLs.
Version	"1.0.0"	Version des Logikbausteins. Im GPA ist immer nur die höchste Version eines Logikbausteins verfügbar. Wird im GPA in den Eigenschaften des Logikbausteins unter *Version* angezeigt.
Author	"Gira Giersiepen GmbH & Co. KG"	Der Entwickler des Logikbausteins.
Copyright	""	Feld wird nicht ausgewertet.
DeveloperId	"LogicNodesSDK"	Die ID des Entwicklers. Die ID wird mit dem Zertifikat vergeben.
License	"Free"	Lizenzmodell der Bibliothek. Mögliche Werte sind: „Free“ oder „Device“.
PackageId	"183FBF6C-AE53-4393-A2A5-2CBE0F1696AF"	GUID der Bibliothek. Hier muss eine eigene generiert werden. In Visual Studio unter *Tools -> Create GUID*.
Nodes	[<Siehe extra Tabelle>]	JSON Array mit einem Objekt pro Logikbaustein. Der spezifische Aufbau ist in der nachfolgenden Tabelle beschrieben.

Aufbau der Objekte im Nodes Array

Schlüssel	Beispielwert	Verwendung
Type	"LogicNodesSDK.Logic.Examples.ColorConverter"	Klassenname des Bausteines inklusive der Namespaces.

Name	<pre>{
	  "en": "Color converter",
	  "de": "Farben-Konverter"
	}</pre>
		Ein JSON Objekt, dessen Schlüssel Länderkürzel sind und dessen Werte der Name des Bausteins in der jeweiligen Sprache sind.

IsConverter	*false*	Gibt an, ob der Baustein bei einem Rechtsklick auf einen Ausgang direkt dem Ausgang nachgeschaltet werden kann.
Category	"Node"	Für diese Version des Logikbaustein SDK soll immer *Node* eingetragen werden.
DefaultIcon	"icons/GenericLogicNode.png"	Relativer Pfad zum Icon des Logikbausteins.

HelpTooltip	<pre>{
	  "en": "Converts colors between 1x 3 byte and 3x 1 byte.",
	  "de": "Wandelt Farbwerte zwischen 1x 3 Byte und 3x 1 Byte um."
	}</pre>
		Ein JSON Objekt, dessen Schlüssel Länderkürzel sind und dessen Werte eine kurze Beschreibung der Funktionsweise des Logikbausteins in der jeweiligen Sprache sind.

HelpFileReference	"BasicArithmetic"	Hilfedatei, die im GPA bei einem Klick auf "Weitere Informationen" geöffnet wird. Abhängig davon, womit der eingetragene Wert beginnt, wird der Wert anders behandelt. Details werden im Kapitel 6.4 beschrieben.

6.4 Lizenzmodell

In der *Manifest.json* Datei existiert der Eintrag *License,* dieser gibt an unter welcher Lizenz das Bausteinpaket zur Verfügung stehen soll. Der Tabelle kann bereits entnommen werden, dass es zwei mögliche Werte für diesen Eintrag gibt.

6.4.1 Free

Beim Lizenzmodell „Free“ können Bausteine aus dem Paket beliebig oft auf jedem Gerät verwendet werden. Es ist keine zusätzliche Lizenzdatei auf dem Gerät erforderlich. Außerdem ist „Free“ der Standardwert, der verwendet wird, falls der Eintrag *License* in der *Manifest.json* nicht vorhanden ist.

6.4.2 Device

Die andere Alternative ist das Lizenzmodell „Device“. Hierbei können Logikbausteinpakete gegen eine Gebühr über den Gira Appshop vertrieben werden. Zur Verwendung dieser Bausteine wird eine Lizenz-Datei mit vergeben, durch die das Bausteinpaket über die MAC-Adresse an ein festes Gerät gebunden wird. Diese Lizenzdatei kann entweder manuell im GPA importiert werden oder bei aktiver Internetverbindung automatisch vom Gerät runtergeladen werden.

Es ist möglich, Lizenzdateien mit einer begrenzten Gültigkeit an Kunden auszugeben, sodass ein Logikbaustein des Pakets nach Ablauf dieser Gültigkeit auf dem Gerät nicht mehr lauffähig ist. Dadurch können z.B. Probeversionen eines Logikbausteinpaktes vertrieben werden, die nach beispielsweise 30 Tagen ihre Gültigkeit verlieren.

In der Gira Smart Home App wird ein Benutzer 30 Tage im Voraus darüber benachrichtigt, falls eine Lizenz in naher Zukunft auslaufen wird.

6.5 Beschreibung der *HelpFileReference* in der Manifest.json Datei

Der GPA kann zu jedem Logikbaustein eine Hilfeseite im Browser anzeigen, hierfür wird der im System des Benutzers eingestellte Standardbrowser verwendet. Die Hilfeseite wird aufgerufen, wenn im Infofenster des Bausteins auf „…Weitere Informationen“ geklickt wird. Der Eintrag *HelpFileReference* in der *Manifest.json* Datei legt fest, wie der GPA die Hilfeseite lokalisiert und aufruft.

Der Platzhalter <Wert> steht hier immer für den Wert des Eintrags *HelpFileReference* in der *Manifest.json*. Dabei beginnt <Wert> immer mit einem Steuerzeichen, das festlegt, wie die Hilfeseite vom GPA gefunden wird.

Steuerzeichen „!“

Mit dem Steuerzeichen „!“ wird der GPA angewiesen, eine externe URL aufzurufen. Dafür muss nach dem „!“ direkt die vollständige URL samt Protokollnamen folgen.

```
"HelpFileReference": "!https://gira.de/",
```

Diese Möglichkeit, eine Hilfeseite anzubieten, ist sehr einfach umsetzbar und ermöglicht es, dem Entwickler die Hilfeseite anzupassen, auch ohne, dass der Baustein in einer neuen Version ausgeliefert wird. Der Benutzer benötigt allerdings eine aktive Internetverbindung.

Steuerzeichen „$“

Es ist auch möglich, dass der Baustein eine *.html* Datei mitliefert, die im Browser angezeigt wird. Dafür muss <Wert> mit dem Steuerzeichen „$“ beginnen, gefolgt von dem Dateinamen.

```
"HelpFileReference": "$help.html",
```

Die Datei *help.html* muss vor dem Bau des Bausteins durch das Visual Studio im Zielverzeichnis *$(TargetDir)* vorhanden sein. Das Programm *LogicNodeTool.exe*, welches im *Post-Build event* aufgerufen wird, erstellt die *.zip* Datei des Bausteins inklusive der *help.html* Datei.

Steuerzeichen „@“

Für einige Bausteine ist es sinnvoll, dass die Hilfeseite in der im GPA eingestellten Sprache angezeigt wird. Um dies zu realisieren, muss <Wert> mit dem Steuerzeichen „@“ beginnen. Die entsprechende *.html* Datei muss, wie zuvor auch, schon vor dem Bau im Zielverzeichnis vorhanden sein.

Der genaue Pfad lautet *$(TargetDir)\Help*\<Sprachkürzel>\<Assembly>*.html*. Diese Pfadstruktur bleibt erhalten, wenn das Programm *LogicNodeTool.exe* die *.zip* Datei des Bausteins erstellt. Dabei ist <Sprachkürzel> eines der Sprachkürzel aus der Tabelle, wobei der GPA dasjenige der intern eingestellten Sprache verwendet. <Assembly> ist der vollständige Dateiname der Logikbaustein Bibliothek, wie er in der *Manifest.json* Datei hinter dem Schlüssel *Assembly* angegeben ist (ohne die Dateiendung *.dll*). Nach dem Steuerzeichen „@“ gibt <Wert> die ID einer Sprungmarke in der *.html* Datei an.

```
"HelpFileReference": "@node",
```

Beim Aufruf der Hilfeseite öffnet der GPA die entsprechende <Assembly>*.html* und springt zu der Sprungmarke *node*. Dadurch ist es möglich, die Hilfen mehrere Logikbausteine in einer Hilfedatei unterzubringen und mit der Sprungmarke an die richtige Stelle zu navigieren.

7 Hinweise und Tipps

Bevor ein Logikbaustein veröffentlicht wird, muss das Risiko für enthaltende Fehler minimiert werden, ganz auszuschließen sind sie natürlich nie. Die folgenden Punkte stellen eine Liste mit Fehlern und Problemen dar, die oft auftreten.

Debuggen / Testen der Logikbausteine

Um die Funktionalität der erstellten Logikbausteine zu testen, bietet es sich an, das vorbereitete NUnit Test Projekt zu verwenden, indem dort ausreichend Testfälle geschrieben werden, die die Funktionalität des Bausteins sicherstellen.

Es ist ebenfalls möglich in der Simulation des GPA zu debuggen. Dazu kann im Visual Studio das Feature *An den Prozess hängen* verwendet werden. Visual Studio muss sich an den GPA Prozess hängen. Wenn nun in der Logikbaustein Simulation des GPA ein Breakpoint von ihrem Baustein getroffen wird, hält Visual Studio den GPA Prozess an und dann ist es möglich zu debuggen.

Versionierung

Die Logikbausteine werden über die Schaltfläche *Logikbausteine hinzufügen* dem GPA hinzugefügt. Der Baustein wird vom GPA allerdings nur angenommen, wenn der Baustein in der hinzugefügten Version nicht schon im GPA existiert. Das heißt, dass jedes Mal, wenn der Quellcode des Bausteins verändert wurde, vor dem Bau des Bausteins auch die Versionsnummer in der *Manifest.json* Datei erhöht werden muss, da ansonsten der Baustein nicht dem GPA hinzugefügt werden kann. Die neue Versionsnummer muss händisch in der *Manifest.json* Datei unter dem Schlüssel *Version* eingetragen werden.

Die neueste Version des Bausteins wird nur für Bausteine verwendet, die einem Logikblatt neu hinzugefügt werden. Ist der Logikbaustein in einer älteren Version schon in dem Logikblatt vorhanden so behält dieser seine ältere Version. Mit der Schaltfläche *Logikbausteine aktualisieren* wird die Version der Bausteine, die bereits in einem Logikblatt verwendet werden, auf die neueste verfügbare Version aktualisiert. Es ist in jedem Fall zu vermeiden, dass

mehrere Versionen desselben Bausteins in einem Logikblatt vorhanden sind. Wird solch ein Logikblatt auf ein Gira Gerät geladen, führt dies zu unerwartetem Verhalten der Logik.

Vor der Veröffentlichung eines Bausteins ist daher zu prüfen, ob die Versionsnummer korrekt inkrementiert ist.

Übersetzung und Hilfe

Bei der Übersetzung der Logikbausteine ist zu empfehlen, die vorhandenen *Localize* Methoden aus den Klassen *LogicNodeBase* oder *LocalizablePrefixLogicNodeBase* zu verwenden, falls dies möglich ist. Es ist auch empfehlenswert, eine Standard *.resx* Datei anzulegen, um für den Benutzer sinnvolle Portnamen im GPA anzuzeigen. Wenn eine Portliste verwendet wird, muss darauf geachtet werden, dass das Namenspräfix nicht gleichzeitig am Anfang eines anderen Portnamens verwendet wird, um fehlerhafte Übersetzungen zu vermeiden. Außerdem müssen die Portnamen, die den Ports bei der Instanziierung zugewiesen werden, innerhalb des Bausteins eindeutig sein.

Beim Erstellen sprachspezifischer Hilfedateien muss der Entwickler einiges beachten. Es gibt keine Standardhilfedatei, auf die zurückgegriffen wird, wenn die im GPA eingestellte Sprache nicht verfügbar ist. Daher sollte zu jeder im GPA verfügbaren Sprache eine Hilfedatei angelegt werden, auch wenn diese nicht in der entsprechenden Sprache geschrieben ist. Eine Hilfedatei in einer anderen Sprache ist für den Benutzer nützlicher als gar keine Hilfedatei.

Häufige Fehlverwendungen

Zugriff auf nicht vorhandene Werte:

Vor dem Zugriff auf den Wert eines Ports muss mit *HasValue* geprüft werden, ob dieser Wert vorhanden ist. Viele Operationen, die mit *null* Werten aufgerufen werden, führen zum Absturz der Logik. Die Prüfung, ob der Wert vorhanden ist, ist auch erforderlich, wenn der Port im Konstruktor einen Initialwert zugewiesen bekommen hat, da der Benutzer des GPA den Initialwert des Ports löschen kann.

Verlassen des Wertebereichs für Listenlängen:

Häufig ist es für einen Baustein erforderlich, dass mithilfe eines Parameters die Anzahl der Ein- oder Ausgänge vom Benutzer des GPA eingestellt werden. Die Hilfsfunktion *ConnectListToCounter* erwartet einen Parameter vom Porttyp "INTEGER". Es ist daher unbedingt erforderlich, dass die zulässigen Werte des Parameters mit den Properties *MinValue* und *MaxValue* auf positive und nicht "zu große" Werte beschränkt werden. Andernfalls kann der Benutzer einen Wert eingeben, der zum Absturz führt.

Unterschiedliche Zeitzonen für *ISchedulerService*:

Mit der Methode *InvokeAt* des *ISchedulerService* Interface können Aktionen eingestellt werden, die zu einem festen Zeitpunkt ausgeführt werden. Die Zeitangabe muss dabei nach dem UTC Zeitstandard erfolgen. Wird der feste Zeitpunkt aus einer Zeitspanne und der aktuellen Zeit berechnet, so muss unbedingt beachtet werden, dass die Methode *Now* die lokale Zeit liefert und nicht die Zeit nach UTC.

.NET Version

Es ist unbedingt erforderlich, dass in den Einstellungen des Projekts zum Logikbaustein das *Target framework* ".NET Framework 4.5" oder ".NET Framework 4.0" ausgewählt ist. Ansonsten kann der Logikbaustein auf dem Gerät nicht richtig ausgeführt werden. In allen mitausgelieferten Projekten ist dies bereits eingestellt.

Unterstützte Bibliotheken auf den Geräten

Die LogicEngine auf dem Gira X1 / Gira L1 wird mit mono unter Linux ausgeführt.

Bei dem Gira X1 ist dies mono 4.0.2, bei dem Gira L1 mono 3.0.1. In beiden Versionen stehen nicht alle Bibliotheken aus dem .NET Framework zur Verfügung.

Liste der verfügbaren mono-Bibliotheken:

- Mono.Security
- System
- System.Configuration
- System.Core
- System.Data
- System.Drawing (*)
- System.Numerics
- System.Runtime.Remoting (*)
- System.Runtime.Serialization
- System.Security
- System.ServiceModel
- System.ServiceModel.Web
- System.Transactions
- System.Web
- System.Xml
- System.Xml.Linq

(*) = nicht auf dem L1 verfügbar

Falls die Verwendung von anderen externe Bibliotheken erwünscht ist, müssen diese zusammen mit dem Baustein in der Zip-Datei auf das Gerät kopiert werden.