Die Sprache VPL (Visual Programming Language)

von Ulrich Grude (grude@tfh-berlin.de)
und Friedrich-Wilhelm Schröer (f.w.schroeer@first.fraunhofer.de

Inhaltsverzeichnis

Im Dezember 2006 wurde die Programm-Entwicklungsumgebung Microsoft Robotics Studio veröffentlicht, zu der auch die neue Programmiersprache VPL gehört. Diese Entwicklungsumgebung kann man sich kostenlos aus dem Netz laden (siehe Abschnitt 1. Installation).

Die Sprache VPL eignet sich zuerst einmal dazu, *Steuerungen für Roboter* zu programmieren (z.B. für Lego-Roboter, Fischertechnik-Roboter oder „größere“ Roboter). Allgemeiner kann man in VPL *Netzdienste* (web services) benutzen und zu neuen Netzdiensten verbinden. Möglicherweise wird man in naher Zukunft die „Programmierung im Kleinen“ (d.h. das Programmieren von elementaren Netzdiensten) in einer objektorientierten Sprache wie Java oder C# erledigen, und für die „Programmierung im Großen“ (d.h. für das Kombinieren von Netzdiensten zu komplexeren Netzdiensten) spezielle Sprachen wie VPL oder BPEL verwenden.

Die Sprache VPL ist Datenfluss-orientiert (und nicht Kontrollfluss-orientiert wie Java, C#, C/C++ und andere verbreitete Sprachen) und VPL-Programme kann man nur visuell erstellen (indem man auf einem Bildschirm graphische Elemente mit der Maus arrangiert und verbindet).

Zum Robotics Studio (Version 1.0, Dezember 2006) gehören Ansätze eines Hilfesystems und einige sehr hilfreiche Tutorials. Dieses Papier ist als Ergänzung zu diesen Dokumentationen gedacht.

1. Installation

Die Installationsdatei für das Robotics Studio (ca. 46 MB) kann man von der Netzadresse `http://msdn.microsoft.com/robotics` herunterladen. Während der Installation werden (falls noch nicht vorhanden) das Rahmensystem `Microsoft .NET Version 2.0` (?? MB), Teile von `.NET Version` 3.0 (ca. 32 MB), Direct-X-Komponenten (16 MB) und eine Physik-Maschine der Firma Ageia heruntergeladen und installiert. Beim Installieren von `.NET Version 3.0` wird

eventuell eine Fehlermeldung ausgegeben, die man ignorieren kann. Das Studio wird in das Verzeichnis `C:\Microsoft Robotics Studio (1.0)` installiert.

Für die Installation braucht man Administrator-Rechte. Wenn man das Studio dann als normaler Benutzer `ABC` starten und benutzen möchte, muss man vorher (als Administrator, nicht als `ABC`) mit dem Programm `HttpReserve` einen Namensraum für den Benutzer `ABC` reservieren, etwa so:

```
> C:\Microsoft Robotics Studio (1.0)\bin\HttpReserve /port:50000 /user:ABC
```

Diese Reservierung ist *dauerhaft* (d.h. sie ist auch nach weiteren Boot-Vorgängen noch wirksam). Mit dem folgenden Kommando kann man sie wieder *beseitigen*:

```
> C:\Microsoft Robotics Studio (1.0)\bin\HttpReserve /port:50000 /user:ABC /remove
```

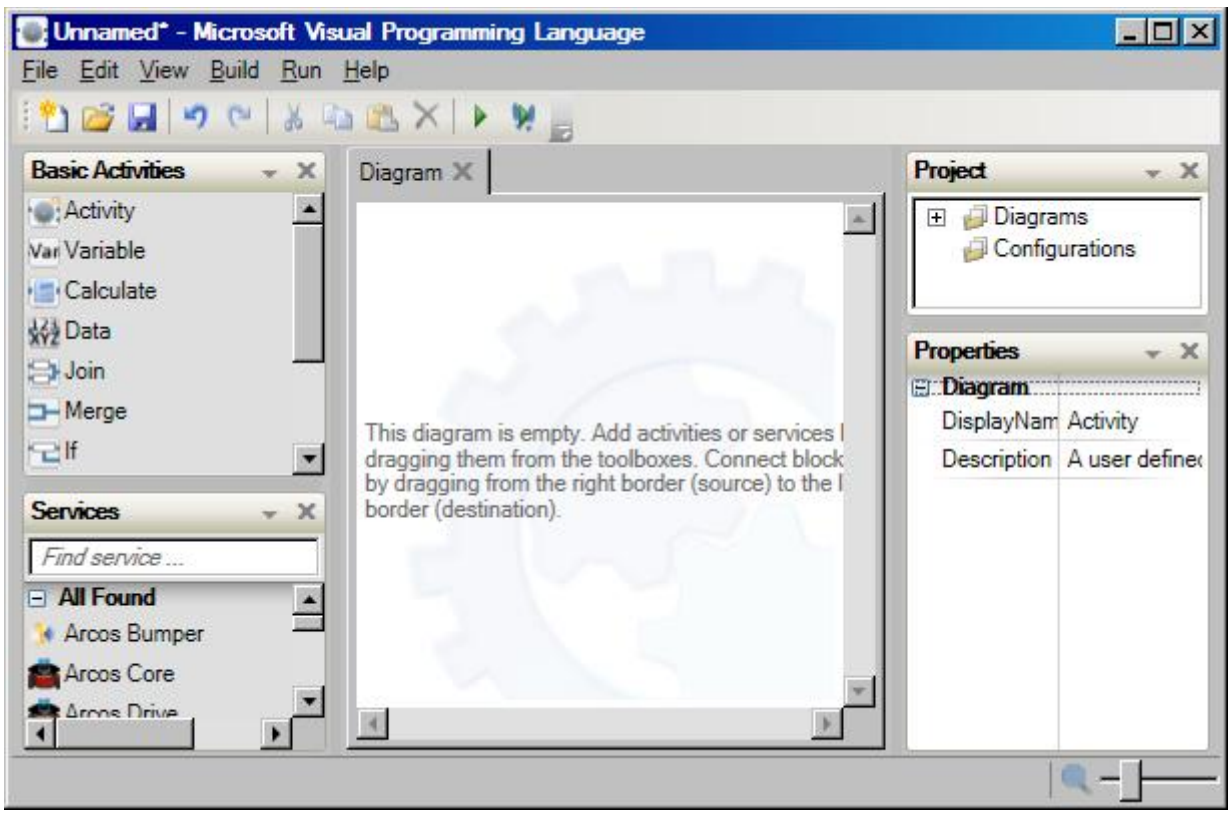
Eine solche Reservierung ist *nicht* nötig, wenn man das Studio als *Administrator* starten und benutzen will.

Auf einem PC mit 1 GB Hauptspeicher und einem 2.5 MHz Pentium-Prozessor läuft das Robotics Studio etwas zögerlich, aber es läuft. Für die fortgeschrittenen Beispiele, die auch das `Microsoft Visual Simulation Environment` (VSE) verwenden, ist unbedingt eine ausreichend schnelle Graphik-Karte erforderlich. Um den Sprachausgabe-Service `Text to Speech (TTS)` zu benutzen benötigt man einen PC mit Lautsprechern.

2. Ein Hallo-Programm in VPL

Starten Sie das Programm `C:\Microsoft Robotics Studio (1.0)\bin\vpl.exe` (das kann einige zig Sekunden dauern). Ein Fenster mit der folgenden Struktur sollte erscheinen:

Beispiel-01: Die Entwicklungsumgebung für VPL-Programme



Dieses Fenster enthält 5 Teilfenster: Links Basic Activities und Services, rechts Project und Properties und dazwischen die *Arbeitsfläche*, die anfangs ein leeres Diagramm zeigt. Unter der blauen Titel-

leiste gibt es eine Menü-Leiste (`File Edit View Build Run Help`) und darunter eine Werkzeugleiste mit 11 Werkzeugsymbole (`Neu`, `Öffnen`, `Speichern`, `Rückgängig`, `Wiederholen`, `Ausschneiden`, `Kopieren`, `Einfügen`, `Löschen`, `Start`, `Debug Start`) zum Anklicken.

Das Teilfenster Basic Activities enthält genau 11 Klassen von *Basis-Aktivitäten* (diese Klassen heissen `Activity`, `Variable`, `Calculate`, ... etc.). Das Teilfenster Services enthält zahlreiche Klassen von *Service-Aktivitäten* (mit Namen wie z.B. `Arcos Bumper`, `Lego® NXT Brick`, `Simple Dialog`, `Text to Speech (TTS)`, `Robotics Tutorial 1`, ... etc.).

Anmerkung: Ein Name wie `Robotics Tutorial 1` im Services-Fenster bezeichnet kein Tutorial, sondern eine Klasse, die in dem betreffenden Tutorial benutzt wird. Tutorials findet man im `Help`-Menü.

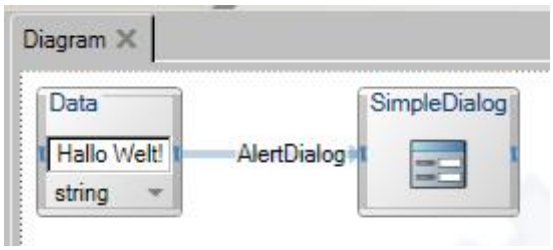
Wenn man eine der Klassen aus dem Basic Activities-Fenster oder dem Services-Fenster mit der Maus auf die Arbeitsfläche zieht und dort „losläßt“, wird ein Objekt der betreffenden Klasse erzeugt und graphisch als *Block* dargestellt. Ein Block ist ein Rechteck mit einem Namen und eventuell weiteren Beschriftungen darin.

Als Erstes sollen Sie jetzt ein Hallo-Programm in VPL erstellen. Ziehen Sie dazu mit der Maus aus dem Fenster Basic Activities einen `Data`-Block auf die Arbeitsfläche. Der `Data`-Block enthält anfangs den Wert `0` vom Typ `int`. Ändern Sie den Typ zu `string` und den Wert zu `Hallo Welt!` (diesen Text brauchen und sollten Sie hier in einem `Data`-Block *nicht* in Anführungszeichen einfassen. An allen anderen Stellen eines VPL-Programms *müssen* Sie String-Literale in doppelte Anführungszeichen einschliessen).

Ziehen Sie mit der Maus aus dem Fenster Services einen `Simple Dialog`-Block auf die Arbeitsfläche und positionieren Sie ihn rechts vom `Data`-Block.

Verbinden Sie die beiden Blöcke indem Sie mit der Maus aus dem `Data`-Block (aus dem kleinen blauen Strich im rechten Rand) einen Pfeil ziehen und zum `Simple Dialog`-Block (zum kleinen blauen Strich im linken Rand) hinüber ziehen. Dadurch sollte ein `Connections`-Fenster mit einem `From:` und einem `To:` Teilfenster darin aufgehen. Im `From:` Teilfenster steht nur `DataValue` und sie können nichts anderes auswählen. Im `To:` Teilfenster sollten Sie die Aktion `AlertDialog` wählen, und dann das `Connections`-Fenster schliessen (unten auf `OK` klicken). Die Arbeitsfläche sollte jetzt etwa so aussehen:

Beispiel-02: Das Programm `B01DataSimpleDialogA`



Anmerkung: Man beachte, dass der erste Block genau so heisst wie die Klasse, aus der er erzeugt wurde (`Data`). Dagegen heisst der aus der Klasse `Simple Dialog` erzeugte Block `SimpleDialog` (mit einem bzw. ohne ein Blank zwischen den Worten). Einige Namen von *Klassen* (in den Fenstern Basic Activities und Services) enthalten Blanks, Klammern und exotische Sonderzeichen wie ®. Die Namen von *Objekten* (mit denen die Blöcke in einem Diagramm beschriftet sind) bestehen dagegen nur aus Buchstaben und Ziffern. Hier ein paar Beispiele für Klassen- und für Objekt-Namen:

Klassen-Namen	*Objekt-Namen (in Blöcken eines Diagramms)*
`Data`	`Data`
`Simple Dialog`	`SimpleDialog, SimpleDialog0, SimpleDialog1, ...`
`Text to Speech (TTS)`	`TexttoSpeechTTS, TexttoSpeechTTS0, TexttoSpeechTTS1, ...`
`Lego® NXT Brick`	`LegoNXTBrick, LegoNXTBrick0, LegoNXTBrick1, ...`

Öffnen Sie das Kontextmenü des zuletzt eingezeichneten Pfeils (indem Sie den Pfeil mit der rechten Maustaste anklicken). Es enthält 6 Einträge, von denen die letzten beiden (`Connections` und `Data Connections`) besonders wichtig sind. Wählen Sie `Data Connections`. Im `Data Connections`-Fenster steht unter `Value` der Eintrag `null`. Ersetzen Sie `null` durch `value`. Der Name `value` bezeichnet immer den „ganzenWert“, der über einen Pfeil ankommt. In diesem Beispiel ist das der `string`-Wert `Hallo Welt!`.

Regel-01: Wenn ein VPL-Programm nicht funktioniert und einfach Nichts macht, dann liegt das häufig (oder fast immer) daran, dass im `Data Connections`-Fenster eines Pfeiles unter `Value` der Standardeintrag `null` steht anstelle eines „richtigen Eintrags“ wie z.B. `value`.

Das `Data Connections`-Fenster eines Pfeils kann man in seinem Kontextmenü oder (nachdem man den Pfeil mit der Maus gewählt hat) im `Edit`-Menü öffnen. Das Kontextmenü eines Pfeils öffnet man, indem man den Pfeil mit der rechten Maustaste anklickt.

Wenn man an einem Diagramm an einer Stelle bestimmte Änderungen vornimmt, werden dadurch manchmal an anderen Stellen die `Data Connections`-Eigenschaften einiger Pfeile verändert (indem dort „richtige Einträge“ unter `Value` automatisch wieder durch den Standardeintrag `null` ersetzt werden). Unter anderem deshalb trifft die Regel-01 erstaunlich oft zu.

Um einen Pfeil zu überprüfen genügt es, den Cursor über dem Pfeil zu positionieren (ohne zu klicken). Dadurch geht ein Infofenster (engl. tool tip) auf, in dem unter anderem die Einträge der `Data Connections` (falls vorhanden) dargestellt sind. Leider verschwindet das Infofenster nach kurzer Zeit von allein, häufig noch bevor man mit dem Lesen und Überprüfen fertig ist. Dann muss man die Maus ein bisschen fortbewegen und erneut über dem Pfeil positionieren.

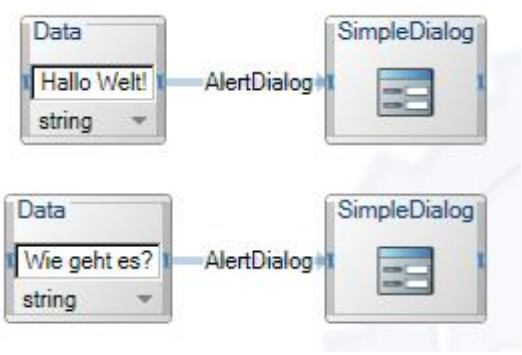
Das erste Hallo-Programm besteht nur aus 2 Blöcken und einem Pfeil dazwischen (wie oben dargestellt). Speichern Sie es ab, z.B. unter dem Namen `B01DataSimpleDialogA`. Durch einen Druck auf die Funktionstaste F5 können Sie es dann ausführen lassen. Nach ein paar Sekunden (etwas Geduld!) sollte ein `Run`-Fenster aufgehen und nach ein paar weiteren Sekunden sollte ein `Alert Dialog`-Fenster mit dem Text `Hallo Welt!` darin aufgehen. Beenden Sie die Programmausführung, indem Sie im `Run`-Fenster auf `Stop` klicken (im `Alert Dialog`-Fenster auf `OK` zu klicken genügt *nicht*).

Statt das Programm mit F5 zu starten, können Sie auch im `Run`-Menü `Start` wählen oder einfach auf das kleine grüne Dreieck in der Werkzeugleiste (das vorletzte Symbol) klicken.

3. Zwei nebenläufige Programmteile sequentialisieren

Erstellen Sie nun das folgende Programm:

Beispiel-01: Das Programm namens `B01DataSimpleDialogB`:



Wenn dieses Programm nicht gleich richtig funktioniert, sollten Sie die beiden Pfeile entsprechend der Regel-01 überprüfen und evtl. ändern.

Regel-02: Die Aktionen in den Blöcken eines VPL-Programms werden ausgeführt, sobald alle dafür benötigten Daten vorhanden sind.

Ein `Data`-Block enthält nur eine einzige Aktion, deren Name (`Create`) selten angezeigt wird, da er selbstverständlich ist. Die `Create`-Aktionen in den beiden `Data`-Blöcke im Beispiel-03 benötigen keine Daten und können sofort ausgeführt werden. Die Aktion `AlertDialog` im oberen `Simple Dialog`-Block wird erst ausführbar, wenn sie (vom `Data`-Block) Eingangsdaten bekommen hat, und entsprechendes gilt für die `AlertDialog`-Aktion im unteren `Simple Dialog`-Block.

Das Programm im Beispiel-01 gibt zwei `Alert Dialog`-Fenster aus, aber in welcher Reihenfolge diese Fenster auf dem Bildschirm erscheinen (oder ob sie gleichzeitig oder zeitlich überlappt erscheinen etc.) ist durch das Programm nicht festgelegt und kann von Ausführung zu Ausführung verschieden sein.

Erstellen Sie nun das folgende Programm (indem Sie im vorigen Programm *einen* weiteren Pfeil einzeichnen):

Beispiel-02: Das Programm `B01DataSimpleDialogC`

Wenn dieses Programm nicht gleich richtig funktioniert, sollten Sie alle drei Pfeile entsprechend der Regel-01 überprüfen und evtl. ändern.

Wenn das Programm dann richtig funktioniert, gibt es erstmal nur *ein* `Alert Dialog`-Fenster aus. Erst wenn der Benutzer in diesem Fenster auf `OK` klickt, wird auch das zweite Fenster ausgegeben.

In einer Datenfluss-orientierten Sprache wie VPL ist es besonders einfach, Aktionen so anzuordnen, dass sie nebenläufig zueinander ausgeführt werden können (wie im Beispiel-01) oder aber in einer bestimmten Reihenfolge (sequentiell) ausgeführt werden müssen (wie im Beispiel-02).

Regel-03: In einem VPL-Diagramm können Pfeile nur von der *rechten* Seite eines Blocks *ausgehen* und zur *linken* Seite eines anderen Blocks *hinführen*. Der konkrete graphische Verlauf eines Pfeils wird ausschließlich durch *die Positionen der verbundenen Blöcke* bestimmt und kann vom Programmierer nur durch das Verschieben der Blöcke beeinflußt werden.

4. Aktivitäten und Aktionen

Die Fenster Basic Activities und Services der VPL-Entwicklungsumgebung enthalten Aktivitäts-Klassen, mit denen man Aktivitäts-Objekte erzeugen kann. Ein Aktivitäts-Objekt enthält eine oder mehrere Aktionen. Eine VPL-*Aktion* ist so etwas wie eine *Methode* in Java oder C#.

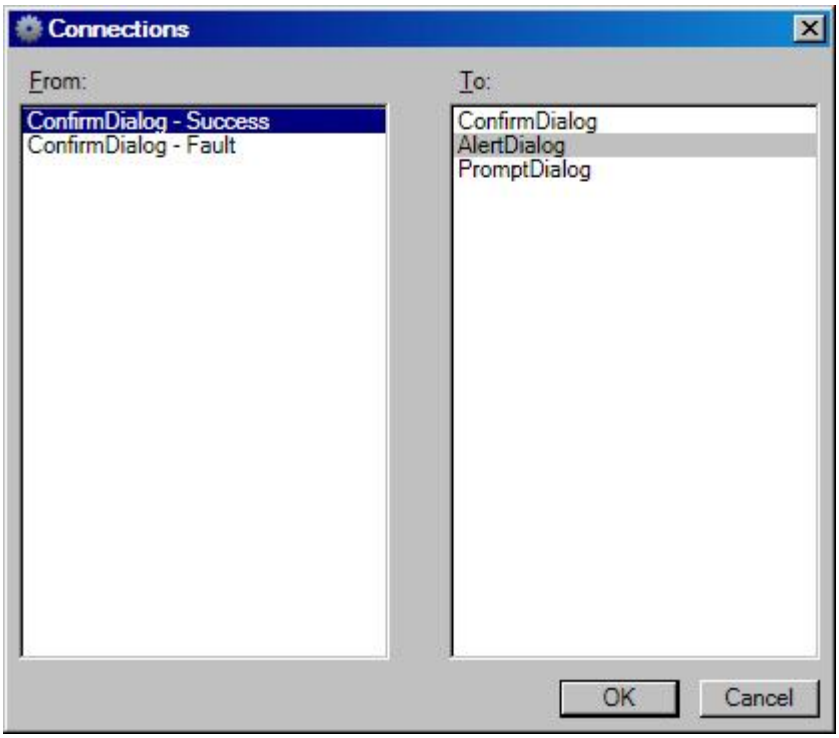
Jeder Block in einem VPL-Diagramm trägt den Namen eines Aktivitäts-*Objekts*, ist aber eigentlich nur ein *Aufruf einer bestimmten Aktion* („einer Methode“) die zu diesem Objekt gehört.

Beispiel-01: Ein `Simple Dialog`-Objekt enthält drei Aktionen namens `AlertDialog`, `ConfirmDialog` und `PromptDialog`. Alle `Simple Dialog`-Blöcke in den bisherigen Beispielen waren Aufrufe der Aktion `AlertDialog`.

Beispiel-02: Das Programm `B02DataSimpleDialogA`

Wenn Sie dieses Programm erstellen, sollten Sie zuerst die drei Blöcke erzeugen. Wenn Sie dann den Pfeil vom `Data`-Block zum ersten `Simple Dialog`-Block ziehen, müssen Sie im `Connections`-Fenster (welches sich automatisch öffnet) unter `To:` die Aktion `ConfirmDialog` wählen (unter `From:` haben Sie keine Wahl, weil dort nur `DataValue` steht). Öffnen Sie dann im Kontextmenü des Pfeiles (Pfeil rechts anklicken) das Fenster `Data Connections` und ersetzen Sie in der Spalte `Value` den Eintrag `null` durch `value`.

Wenn Sie danach den zweiten Pfeil (vom ersten zum zweiten `Simple Dialog`-Block) ziehen, sollten Sie im `Connections`-Fenster (welches sich automatisch öffnet) folgende Wahl treffen:



Damit drücken Sie Folgendes aus: Nur wenn der `ConfirmDialog` *erfolgreich* abgeschlossen wurde, sollen Daten zum nächsten `Simple Dialog`-Block fließen und dieser Block soll die Aktion `AlertDialog` aufrufen (und nicht die Aktion `PromptDialog` oder noch einmal die `ConfirmDialog`-Aktion).

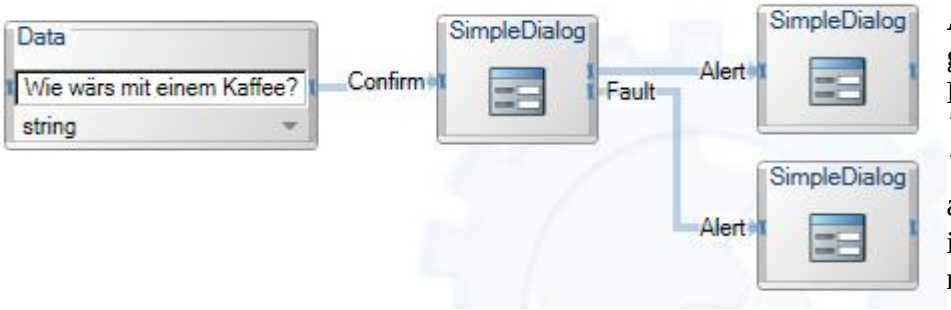
Öffnen Sie dann im Kontextmenü des zweiten Pfeils (Pfeil rechts anklicken) das Fenster `Data Connections` und ersetzen Sie in der Spalte `Value` den Eintrag `null` durch `Confirmed`.

Der Name `Confirmed` bezeichnet hier das boolesche Ergebnis der Aktion `ConfirmDialog` des mittlern Blocks (`true` bzw. `false`, je nachdem ob der Benutzer im `Confirm Dialog`-Fenster auf `OK` oder auf `Cancel` geklickt hat). Dieses Ergebnis wird (oben im Beispiel-02) mit einem Aufruf der `AlertDialog`-Aktion (ganz rechter Block) ausgegeben.

Wenn dieses Programm nicht gleich richtig funktioniert, sollten Sie beide Pfeile entsprechend der Regel-01 überprüfen und evtl. ändern.

Die `ConfirmDialog`-Aktion (Beispiel-02, mittlerer Block) hat (ähnlich wie ein `If`-Block, siehe unten) *zwei* Ausgänge. Im Beispiel-02 haben wir nur vom Ausgang `ConfirmDialog - Success` einen Pfeil ausgehen lassen. Ganz entsprechedn kann man auch vom Ausgang `ConfirmDialog - Fault` einen Pfeil z.B. zu einem weiteren `Simple Dialog`-Block führen. Dazu beginnt man an der selben Stelle des rechten Blockrandes mit der Maus zu „ziehen“ wie beim ersten Pfeil, wählt dann aber im `Connections`-Fenster des neuen Pfeils im Teilfenster `From:` einfach `ConfirmDialog - Fault`. Der neue Pfeil wird als Folge dieser Wahl automatisch wie folgt gezeichnet:

Beispiel-03: Auch der `Fault`-Ausgang der `ConfirmDialog`-Aktion wird belegt



Anmerkung: In VPL-Diagrammen werden Aktions-Namen wie `AlertDialog` und `ConfirmDialog` etc. manchmal voll ausgeschrieben (wie z.B. im Beispiel-02) und manchmal automatisch durch `Alert` und `Confirm` etc. abgekürzt (wie hier im Beispiel-03). Dadurch sollte man sich möglichst wenig verwirren lassen.

Nachdem man auch den `Fault`-Ausgang der Aktion `Confirm(Dialog)` mit einer `Alert(Dialog)`-Aktion verbunden hat, sollte man die neue Verbindung auch testen. Leider habe ich nicht herausgefunden, ob der Benutzer des Programms durch irgendwelche Handlungen bewirken kann, dass die `ConfirmDialog`-Aktion *nicht erfolgreich* (also über den `Fault`-Ausgang) beendet wird. Vielleicht muss man im richtigen Moment den Bildschirm anzünden oder den ganzen PC in Wasser tauchen? :-) Auf jeden Fall scheinen die Aktionen eines `Simple Dialog`-Objekts sehr oft mit Erfolg beendet zu werden.

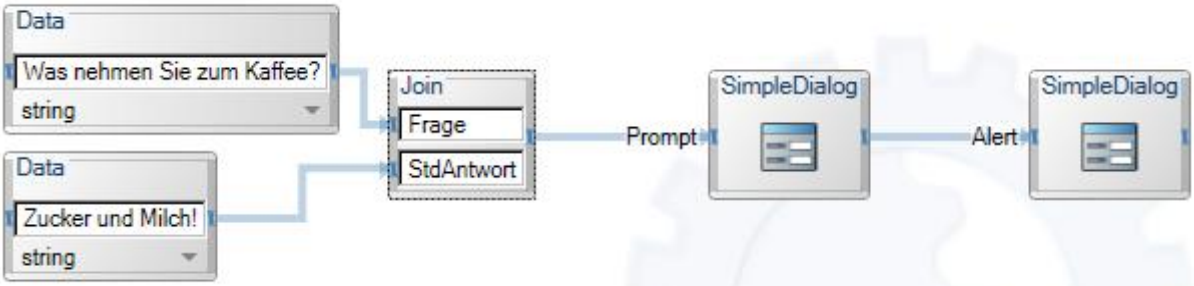
5. Eine Aktion mit mehreren Eingangsparametern

Die Aktionen („Methoden“) `AlertDialog` und `ConfirmDialog` eines `Simple Dialog`-Objekts erwarten jeweils nur *einen* Eingangsparamter vom Typ `string` (den anzuzeigenden Text). Bei einem `AlertDialog` kann der Benutzer dann nur auf `OK` klicken, bei einem `ConfirmDialog` hat er die Wahl zwischen `OK` und `Cancel`.

Die Aktion `PromptDialog` eines `Simple Dialog`-Objekts erwartet dagegen *zwei* Eingangsparameter: Einen *Frage-Text* und eine *Standard-Antwort* (die der Benutzer dann bestätigen oder durch eine andere Antwort ersetzen kann).

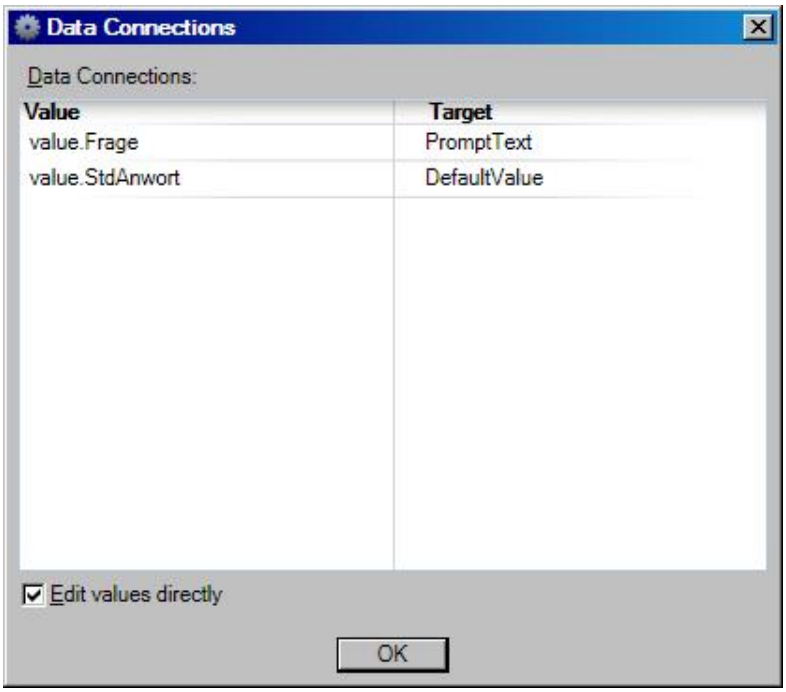
Zu einem `Simple Dialog`-Block darf man nur *einen* Eingabepfeil hinführen. Wenn man *mehrere* Parameterwerte über diesen Pfeil fliessen lassen will, muss man sie vorher mit einem `Join`-Block zu einem Verbund (engl. record oder struct) zusammenfassen. Ein Verbund ist eine Zusammenfassung von Komponenten (engl. fields), die zum selben Typ oder zu verschiedenen Typen gehören können.

Beispiel-01: Das Programm `B02DataSimpleDialogC`



Auch hier sind `Prompt` und `Alert` automatisch eingefügte Abkürzungen für die Aktionsnamen `PromptDialog` und `AlertDialog`.

Der `Join`-Block fasst in diesem Beispiel die Werte, die er von den beiden `Data`-Blöcken bekommt, zu einem Verbund namens `value` zusammen. Dieser Verbund enthält zwei `string`-Komponenten namens `value.Frage` und `value.StdAntwort`.



Besonders interessant ist die Datenverbindung zwischen dem `Join`-Block und der `Prompt(Dialog)`-Aktion. Öffnen Sie im Kontextmenü des entsprechenden Pfeils (Pfeil rechts anklicken) das `Data Connections`-Fenster und wählen Sie darin die rechts abgebildeten Einstellungen:

In diesem Fenster werden die formalen und die aktuellen Parameter der Aktion `PromptDialog` dargestellt. Die formalen Parameter heissen `PromptText` und `DefaultValue` (siehe Spalte `Target`). Als aktuelle Parameter wurden hier die Verbundkomponenten `value.Frage` und `value.StdAntwort` gewählt (siehe Spalte `Value`).

Die Typen der formalen Parameter muss man raten oder durch „Ausprobieren“ herausfinden. Wenn man einen aktuellen Parameter eines falschen Typs angibt, bekommt man manchmal eine gut verständliche Fehlermeldung, in der auch der richtige Typ erwähnt wird. Eine Dokumentation, in der die Aktionen der einzelnen Service-Klassen, die Parameter der Aktionen und ihre Ergebnisse beschrieben sind, konnte ich bisher nicht finden.

Im vorliegenden Beispiel kann man als aktuelle Parameter Werte beliebiger einfacher Typen (`int`, `string`, `double`, ...) angeben, `string`-Werte dürften aber in vielen Fällen am sinnvollsten sein.

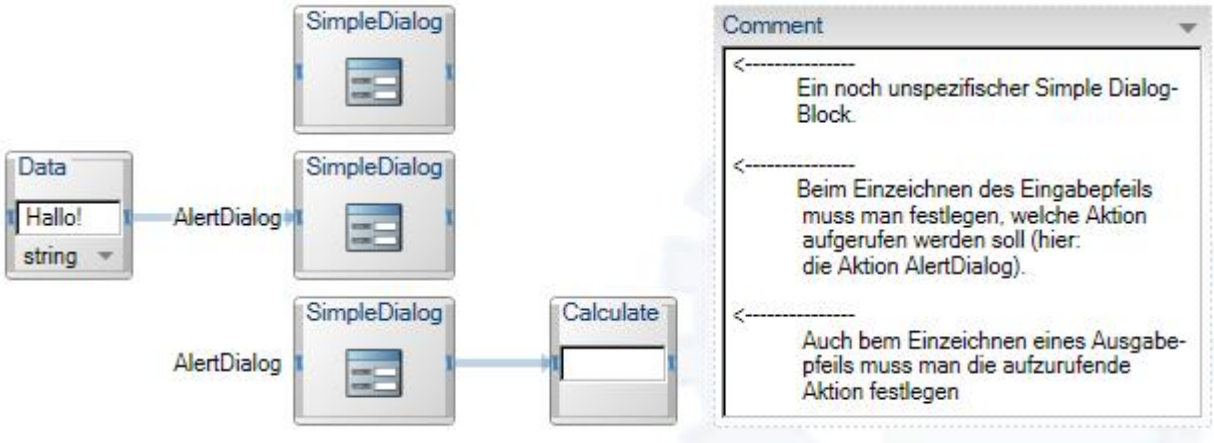
Die CheckBox `Edit values directly` erlaubt es einem, zwischen zwei Notationen zu wählen. Es wird hier empfohlen, die CheckBox *anzuschalten* (wie abgebildet) und dann in der `Value`-Spalte die in vielen Programmiersprachen übliche *Punktnotation* (oder: Verbundnotation) `value.Frage` und `value.StdAntwort` zu verwenden.

6. Von einem Objekt zu einer Aktion

Wenn man eine Aktivitäten-Klasse (aus einem der Fenster Basic Activities oder Services, zum ersten Mal) mit der Maus auf die Arbeitsfläche zieht, wird ein *Objekt* der Klasse erzeugt und als *Block* dargestellt.

Erst und sobald man einen Pfeil einzeichnet, der (von links) zu diesem Block hin oder (nach rechts) von ihm weg führt, legt man fest, welche *Aktion* der Block repräsentieren soll, etwa so:

Beispiel-01: Festlegen, welche Aktion aufgerufen werden soll



Beim obersten `Simple Dialog`-Block ist noch nicht festgelegt, welche Aktion („Methode“) mit ihm aufgerufen werden soll.

Wenn man einen Pfeil einzeichnet, der (z.B. von einem `Data`-Block) zu einem `Simple Dialog`-Block führt, öffnet sich automatisch und unvermeidbar ein `Connections`-Fenster. Darin muss man festlegen, welche Aktion im `Simple Dialog`-Objekt der neue Pfeil mit Daten versorgen soll (`AlertDialog`, `ConfirmDialog` oder `PromptDialog`).

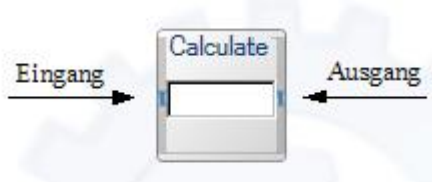
Wenn man einen Pfeil einzeichnet, der von einem `Simple Dialog`-Block wegführt (z.B. zu einem `Calculate`-Block), passiert ganz Entsprechendes.

Aus diesen Beobachtungen folgt: Wenn ein Block z.B. einen Aufruf der Aktion `AlertDialog` darstellt und man ihn so ändern will, dass er z.B. die Aktion `ConfirmDialog` darstellt, muss man zuerst *alle Pfeile* löschen, die mit dem Block verbunden sind.

7. Eingangspfeile und Ausgangspfeile, wieviele sind erlaubt?

Viele Blöcke haben auf der linken Seite einen Eingang und auf der rechten Seite einen Ausgang, jeweils durch eine kleine blaue Markierung dargestellt.

Beispiel-01: Der Eingang und der Ausgang eines Blocks

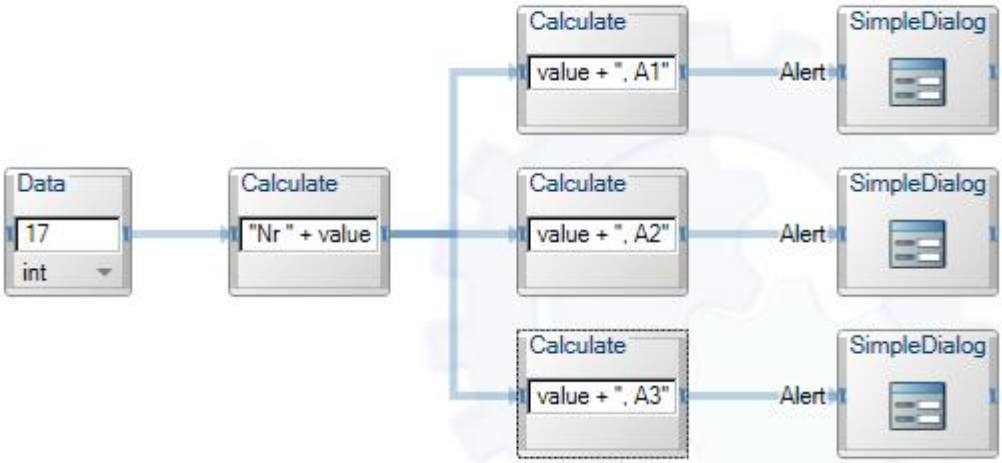


Regel-04: Zum *Eingang* eines Blocks darf höchstens *ein* Pfeil führen (engl. the fan in of an input pin is 1).

Von dieser Regel gibt es 2 Ausnahmen: Ein `Merge`-Block und ein `Join`-Block haben beliebig viele Eingänge (mindestens 2) und zu jedem dieser Eingänge kann *ein* Pfeil führen.

Regel-05: Vom *Ausgang* eines Blocks dürfen *beliebig viele* Pfeile ausgehen (engl. the fan out of an output pin is unbounded).

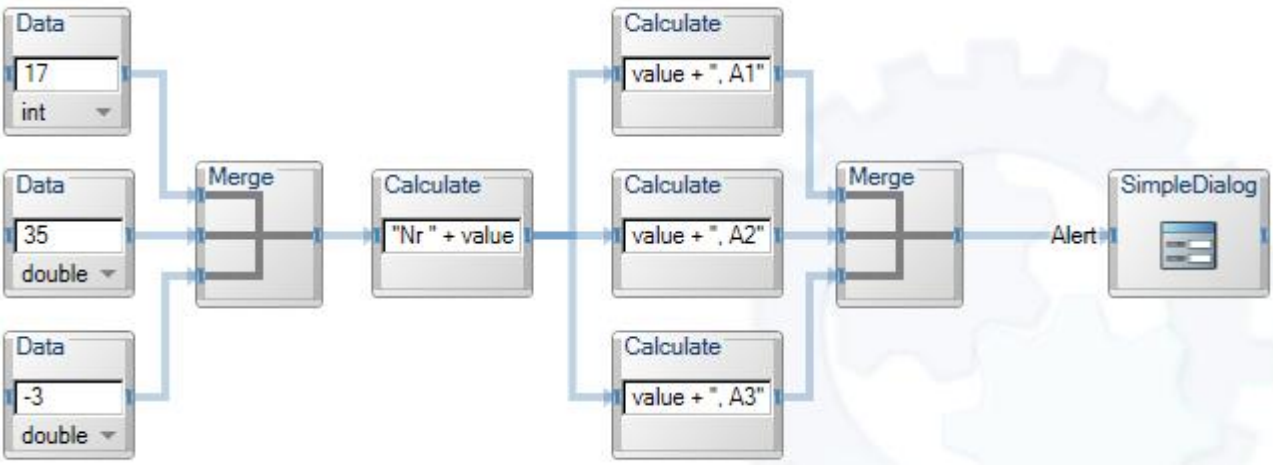
Beispiel-02: Ein `Calculate`-Block mit einem fan out von 3 (Programm `B04FanInOutA`)



Der Name `value` bezeichnet in jedem Block die Ausgabe des davorliegenden Blocks. Im ersten `Calculate`-Block wird der `int`-Wert `17` in einen String umgewandelt und vor das `string`-Literal `"Nr "` konkateniert. Das Ergebnis `"Nr 17"` wird an alle drei weiter rechts stehenden `Calculate`-Blöcke geschickt, wo jeweils ein weiterer kleiner String angehängt wird. Die drei Ergebnisstrings werden dann durch die drei Aufrufe der Aktion `Alert(Dialog)` ganz rechts ausgegeben. Der erste `Calculate`-Block hat einen fan out von 3, d.h. er schickt sein Ergebnis an *drei* weitere Blöcke.

Dass zum Eingang eines (normalen) Blocks nur *ein* Pfeil führen darf, ist weniger einschränkend, als es im ersten Moment scheinen mag. Mit einem (besonderen) `Merge`-Block kann man *beliebig viele* Pfeile zu *einem* Pfeil zusammenführen, etwa so wie im folgenden Beispiel.

Beispiel-03: Das Programm `B04FanInOutB` mit zwei Merge-Blöcken



Ein `Merge`-Block leitet jeden Wert, den er über einen seiner Eingänge bekommt, unverzüglich und unverändert zu seinem Ausgang weiter. In welcher Reihenfolge der erste `Calculate`-Block die drei `int`-Werte `17`, `35` und `-3` bekommt (und weiterleitet), ist durch dieses Programm nicht festgelegt.

Der zweite `Merge`-Block bewirkt, dass alle Strings zu einem einzigen `Simple Dialog`-Block geschickt werden (statt wie im vorigen Beispiel zu drei verschiedenen Blöcken).

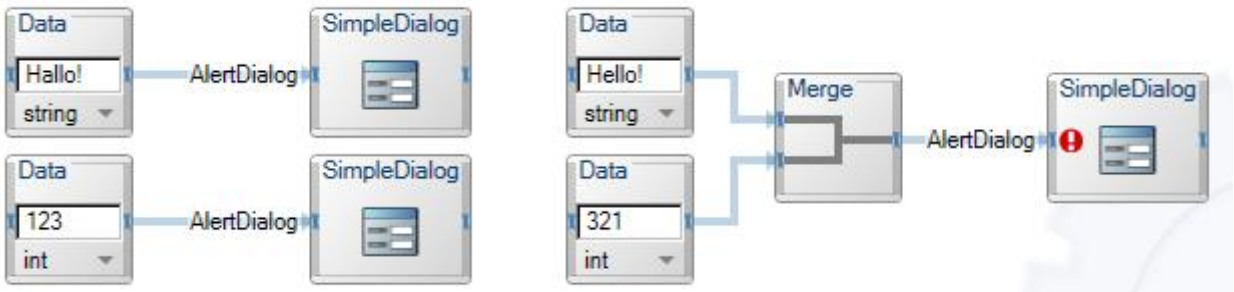
Das Programm im Beispiel-03 gibt insgesamt neun `Alert Dialog`-Fenster aus. Die Reihenfolge der Ausgaben ist durch das Programm nicht festgelegt.

Praktischer Hinweis: Ein neu erzeugter `Merge`-Block hat nur *zwei* Eingänge. Sobald man aber mit der Maus mehrere Pfeile zu einem der Eingänge hinführt, wird die Anzahl der Eingänge automatisch

erhöht. Zu welchem Eingang man einen Pfeil hinführt, ist unwesentlich, nur die zeitliche Reihenfolge zählt: Der zuerst angebrachte Pfeil führt immer zum obersten Eingang, der danach angebrachte Pfeil zum zweitobersten Eingang u.s.w.

Mit einem `Merge`-Block darf man normalerweise nur solche Pfeile zusammenfassen, über die Daten des selben Typs fliessen. Allerdings werden einige Programme, die gegen diese Regel verstoßen, ausgeführt und haben den erwarteten Effekt, z.B. das folgende:

Beispiel-04: Ein falsches aber trotzdem funktionierendes Programm (`B04FanInOutC`)

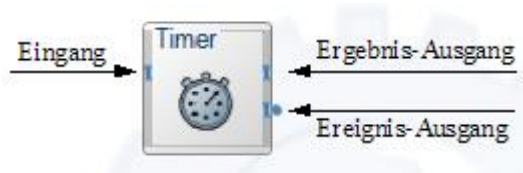


Der rote Kreis mit dem weissen Ausrufezeichen enthält die Fehlermeldung:
`The incoming message type, '(Message -> Unknown)', does not match the input type needed for this activity.`
Trotzdem funktioniert das Programm und gibt die erwarteten vier `Alert Dialog`-Fenster aus.

8. Ergebnis-Pfeile und Ereignis-Pfeile

Viele Service-Blöcke haben einen Ausgang, an dem man nur sogenannte Ergebnis-Pfeile (engl. result output) anbringen darf. Service-Blöcke bestimmter Klassen haben zusätzlich aber noch einen zweiten Ausgang, an dem man sogenannte Ereignis-Pfeile (engl. event output oder notification output) anbringen darf. Dies gilt z.B. für Blöcke der Klassen `Game Controller`, `Direction Dialog` und `Timer`.

Beispiel-01: Ein `Timer`-Block (noch ganz ohne Pfeile)



Die Stelle, wo man *Ereignis*-Pfeile anbringen kann, ist mit einem kleinen blauen Kreis gekennzeichnet.

Einige Aktionen haben Ähnlichkeit mit Funktionen in anderen Programmiersprachen: Über einen Eingabepfeil übergibt man ihnen bestimmte Parameterwerte. Daraus berechnen sie ein Ergebnis und schicken das über einen Ergebnis-Ausgang weiter. Für jede Eingabe liefern sie genau ein Ergebnis.

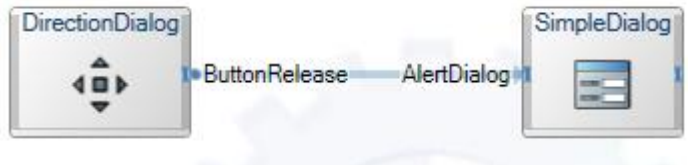
Andere Aktionen sind unabhängiger von irgendwelchen Eingaben und liefern Werte als Reaktionen auf bestimmte Ereignisse, natürlich über einen Ereignis-Pfeil. Eine solche Aktion kann z.B. nach *einer* Eingabe (oder sogar ganz ohne Eingabe) *mehrere* Werte liefern.

Beispiel-02: Das Programm `B05TimerA`

Die Aktion `Tick` im `Timer`-Block liefert pro Sekunde eine Ausgabe (ohne je eine Eingabe zu erwarten). Mit dieser Ausgabe kann man eine Aktion wie `Asterisk-Sound` (klingt etwa wie zwei Tastenanschläge auf einem Klavier) oder andere Programmteile anstoßen, die

regelmäßig ausgeführt werden sollen. Die Aktion `Tick` liefert ihre Ausgabe über einen *Ereignis-Pfeil*, nicht über einen *Ergebnis-Pfeil*.

Beispiel-03: Das Programm `B06DirectionDialogA`



Bei diesem Programm muss man im `Data Connections`-Fenster des Pfeiles (erreichbar über das Kontextmenü des Pfeiles) unter `Value` als Wert `Name` wählen (oder direkt `value.Name` eintragen).

Wenn man das Programm dann ausführen läßt, erscheint auf dem Bildschirm ein `DSS Direction Dialog`-Fenster mit fünf Knöpfen darin, wie rechts abgebildet.

Wenn man über einem der Knöpfe einen Mausknopf wieder loslässt (nachdem man ihn vorher irgendwo und irgendwann runtergedrückt hat), gibt die Aktion `ButtonRelease` im Programm den Namen des Knopfes (als `string`-Wert namens `Name`) über ihren Ereignis-Ausgabepfeil aus. Dieser Name wird dann in einem `Alert Dialog`-Fenster ausgegeben. Die Knöpfe haben die rechts unten dargestellten Namen `Forwards`, `Left`, `Stop` etc.

Wenn man das Programm ausführen läßt, sollte man über einem der Fenster-Knöpfe auf einen Mausknopf drücken, zwei oder drei Sekunden warten und erst dann den Mausknopf wieder loslassen. So kann man sich vergewissern, dass das `Alert Dialog`-Fenster erst beim *Loslassen* des Mausknopfes ausgegeben wird (und nicht schon beim Runterdrücken).

```
          Forwards
Left        Stop      Right
          Backwards
```

Anstelle der Aktion `ButtonRealease` kann man mit dem `Direction Dialog`-Block auch die Aktion `Button Press` ausführen lassen (löschen Sie dazu den Pfeil und zeichnen Sie an seiner Stelle einen neuen Pfeil ein). Nach dieser Änderung des Programms werden die `Alert Dialog`-Fenster schon beim *Runterdrücken* eines Mausknopfes ausgegeben.

9. Mehrere Blöcke die dasselbe Objekt bezeichnen

Wenn man eine Aktivitäts-Klasse aus dem Fenster Services (z.B. die Klasse `Direction Dialog`) *zum ersten Mal* auf die Arbeitsfläche zieht, wird ein Objekt der Klasse erzeugt und als Block dargestellt.

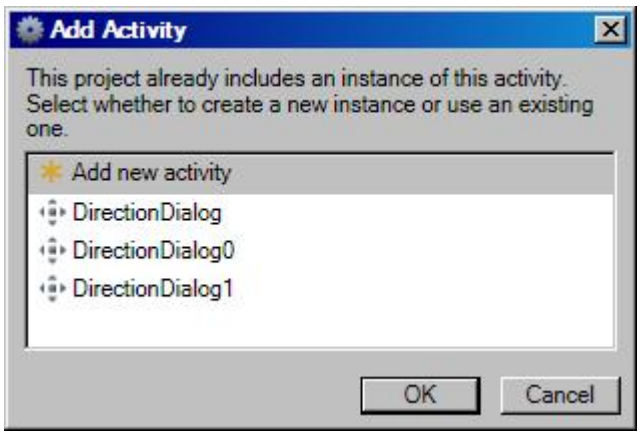
Wenn man dieselbe Klasse *ein weiteres Mal* auf die Arbeitsfläche zieht, wird ein weiterer Block gezeichnet, aber vorher geht automatisch ein `Add Activity`-Fenster auf und man muss entscheiden, ob dieser Block *ein neues Objekt* oder *ein bereits vorhandenes Objekt* bezeichnen soll.

Beispiel-01: Fünf Blöcke, aber nur drei Objekte

Regel-06: Jeder Block ist mit einem Objektnamen beschriftet. Blöcke, die mit *gleichen* Namen beschriftet sind, bezeichnen *dasselbe Objekt.* Blöcke mit unterschiedlichen Namen bezeichnen verschiedene Objekte.

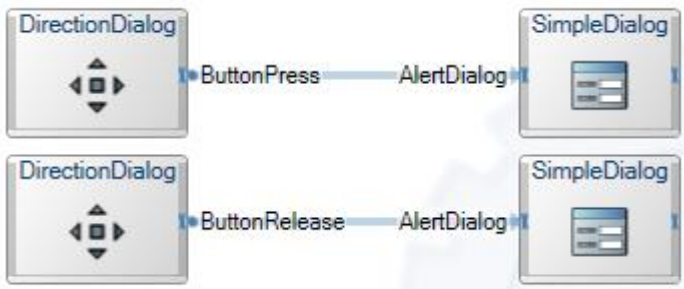
Im Beispiel-01 gibt es drei Objekte namens `DirectionDialog`, `DirectionDialog0` und `DirectionDialog1`, die von zwei bzw. einem bzw zwei Blöcken bezeichnet werden.

Wenn man in dieser Situation die Klasse `Direction Dialog` ein weiteres Mal über die Arbeitsfläche zieht, wird automatisch ein `Add Activity`-Fenster wie das nebenstehend geöffnet. Darin werden einem die Namen aller bereits erzeugten Objekte (der betreffenden Klasse) angezeigt und zusätzlich die Möglichkeit, ein neues Objekt erzeugen zu lassen.



Die folgenden Beispiele sollen den Unterschied zwischen der Verwendung von *einem* Objekt oder von *zwei gleichen* Objekten deutlich machen.

Beispiel-02: Das Programm `B06DirectionDialogB` mit *einem* `Direction Dialog`-Objekt

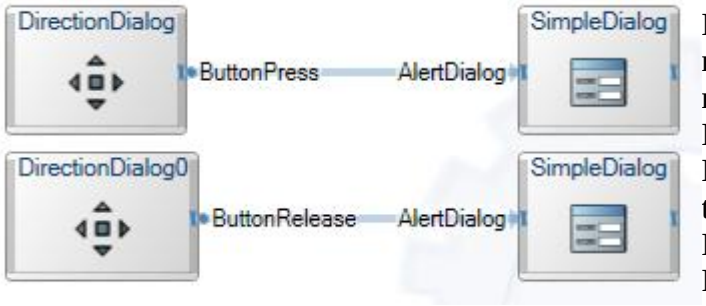


Die beiden `Direction Dialog`-Blöcke in diesem Programm bezeichnen (ein und) dasselbe Objekt. Der obere Block ruft die `ButtonPress`-Aktion in diesem Objekt auf. Der unter Block ruft die `ButtonRelease`-Aktion im selben Objekt auf.

Läßt man das Programm ausführen, so wird das eine Objekt durch *ein* `DSS Direction Dialog`-Fenster mit fünf Knöpfen darin dargestellt (wie im Beispiel-03 des vorigen Abschnitts abgebildet). Klickt man mit der Maus auf einen der fünf Knöpfe, so wird sein Name *zweimal* ausgegeben (einmal beim Runterdrücken und ein zweites Mal beim wieder-Loslassen des Mausknopfes).

Beispiel-03: Das Programm `B06DirectionDialogC` mit *zwei* `Direction Dialog`-Objekten

Dieses Diagramm unterscheidet sich nur durch *ein* Zeichen vom Diagramm im vorigen Beispiel. Hier bezeichnen die beiden `Direction Dialog`-Blöcke zwei Objekte namens `DirectionDialog` und `DirectionDialog0`. Die beiden Objekte sind gleich (etwa wie Zwillinge), aber nicht identisch.



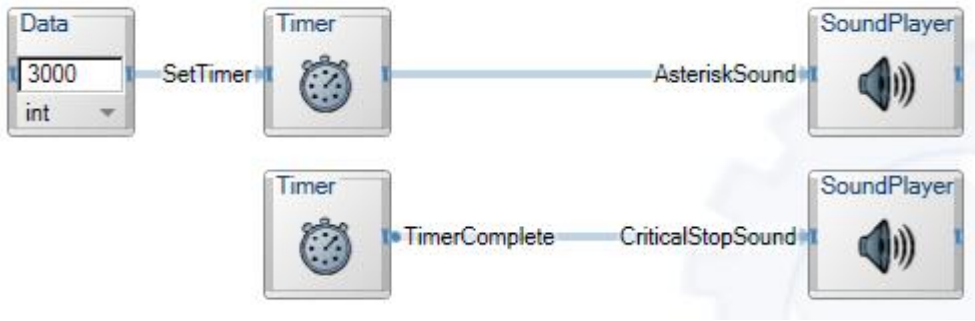
Läßt man das Programm ausführen, so erscheinen *zwei* `DSS Direction Dialog`-Fenster mit je fünf Knöpfen darin. Klickt man mit der Maus auf einen der zehn Knöpfe, so wird sein Name *einmal* ausgegeben (entweder beim Runterdrücken des Mausknopfes oder beim wieder-Loslassen). Welches der beiden 5-Knöpfe-Fenster zu welchem Objekt (`Direction-Dialog` oder `DirectionDialog0`) gehört, kann man nur durch Ausprobieren herausfinden. Es ist zur Zeit nicht möglich, die Beschriftung der Fenster zu modifizieren.

Ob man *ein* Objekt oder *zwei gleiche* Objekte verwendet macht vor allem dann einen Unterschied, wenn die Objekte einen internen, veränderbaren *Zustand* haben. Zum internen Zustand eines `Direction Dialog`-Objekts gehören alle Informationen über seine graphische Darstellung auf dem Bildschirm. Hat man zwei Objekte, so können ihre graphischen Darstellungen (z.B. ihre Positionen auf dem Bildschirm oder die Information, ob ein bestimmter Knopf angeklickt wurde) unabhängig voneinander verändert werden.

Objekte der Basisklassen (im Fenster Basic Activities) haben keine internen Zustände. Deshalb wird man beim Benutzen der Basisklassen nicht gefragt, ob ein neues Objekt erzeugt oder ein schon vorhandenes genommen werden soll (und man kann nicht feststellen, ob der Ausführer jedesmal eine neues Objekt erzeugt oder vorhandene Objekte mehrfach benutzt oder ganz anders vorgeht).

Vermutlich haben auch Objekte der Service-Klasse `Simple Dialog` (und die Objekte einiger weiterer Service-Klassen) keine internen Zustände. Trotzdem wird man beim Erzeugen eines *Service*-Blocks (außer beim ersten Mal) immer gefragt, ob der Block ein neues Objekt oder ein bereits vorhandenes bezeichnen soll.

Beispiel-04: Einen Wecker stellen und klingeln lassen (Beispielprogramm `B05TimerC`)



Mit der Aktion `SetTimer` kann man einem `Timer`-Objekt befehlen, nach einer bestimmten Anzahl von Millisekunden ein Signal auszugeben („zu klingeln“). Der Pfeil vom oberen `Timer`-Block zur Aktion `AsteriskSound` ist mit dem Ergebnis-Ausgang `SetTimer - Success -` des Timer-Blocks verbunden. Der `AsteriskSound` ertönt deshalb, sobald das „Stellen des Weckers“ erfolgreich beendet ist. 3 Sekunden später kommt aus dem Ereignis-Ausgang `TimerComplete` ein Signal und bewirkt, dass der `CriticalStopSound` ertönt. Natürlich funktioniert das nur, wenn die beiden `Timer`-Blöcke *dasselbe* `Timer`-Objekt bezeichnen, was hier der Fall ist.

Um einen „Wecker zu stellen“ wie in diesem Beispiel sind unbedingt *zwei* `Timer`-Blöcke notwendig. Denn sobald man einen Pfeil zum Eingang eines `Timer`-Blocks führt und die Aktion `SetTimer` wählt, *verschwindet* der Ereignis-Ausgang an der rechten Seite des Blocks. Die beiden separaten Timer-Blöcke betonen graphisch, dass das Stellen des Weckers (`SetTimer`) und das Klingeln (`TimerComplete`) nicht eng miteinander gekoppelt sind, sondern nur lose zusammenhängen.

10. Schleifen und Variablen (und Sprachausgabe)

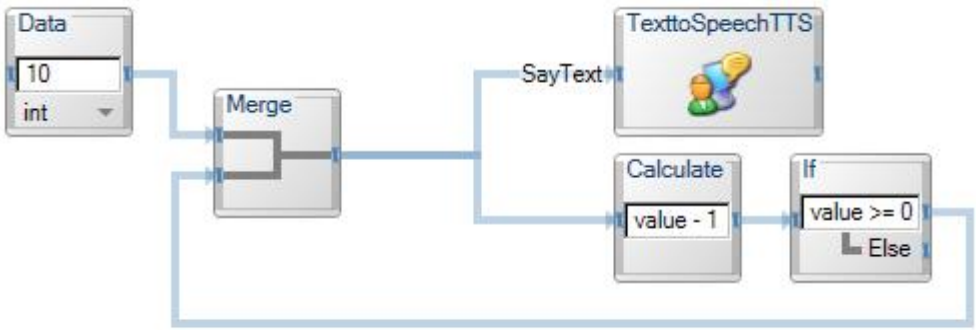
Wenn man zum Testen eines Programms viele Daten mit dem `Simple Dialog`-Service ausgibt, werden evtl. unübersichtlich viele Nachrichtenfenster erzeugt. Ausserdem kann man nur in sehr einfachen Fällen beobachten, in welcher *Reihenfolge* die einzelnen Meldungen ausgegeben wurde. Wir verwenden deshalb ab jetzt neben dem `Simple Dialog`-Service auch den Sprachausgabe-Service `Text to Speech (TTS)`, der zur Grundausstattung des Robotics Studio gehört.

Praktischer Tip: Auf einem PC ohne Lautsprecher kann man die Aufrufe von Sprachausgabeaktionen durch Aufrufe von `Simple Dialog`-Aktionen ersetzen (muss damit aber die oben skizzierten Nachteile in Kauf nehmen).

Praktischer Tip: Falls Sie sich entschließen, Lautsprecher zu kaufen, sollten Sie auch gleich die Anschaffung eines Joysticks erwägen, der in späteren Beispielen benötigt wird. Einen Joystick kann man schon ab etwa 15,- Euro kaufen. Noch günstiger ist es natürlich, sich einen von seinen Kindern oder Nachbarn auszuleihen.

Wenn ein Programmierer möchte, dass bestimmte Aktionen *mehrmals* ausgeführt werden, denkt er an *Rekursion* und an *Schleifen*. Rekursion wird im Abschnitt 16 behandelt.

Beispiel-01: Eine simple Schleife (Programm `B07SchleifeA`)



Wenn man der `Say-Text`-Aktion in einem `Text to Speech (TTS)`-Objekt einen Wert (eines einfachen Typs wie `int`, `double` oder `string` etc.) schickt, wird dieser Wert in englischer Sprache (mehr oder weniger verständlich) ausgesprochen. Das erste `Text to Speech (TTS)`-Objekt in einem Programm heisst immer `TexttoSpeechTTS`.

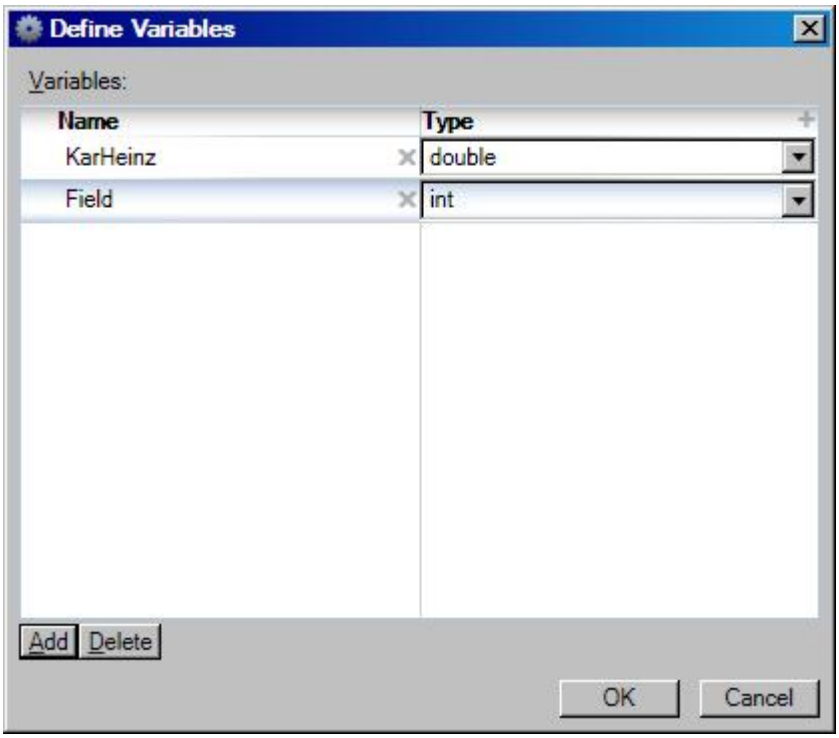
Man beachte, dass der `If`-Block noch einen zweiten (mit `Else` gekennzeichneten) Ausgang hat. Wenn man (wie hier) an diesem Ausgang *keinen* Pfeil anbringt, passiert im `Else`-Fall einfach nichts.

Wenn Sie dieses Programm selbst erstellen und es beim Ausführen stumm bleibt, sollten Sie es entsprechend der Regel-01 (im Abschnitt 2) überprüfen und evtl. korrigieren. Im `Data Connections`-Fenster des Eingabepfeils der `SayText`-Aktion sollte unter `Value` der Standardname `value` stehen. Wenn das Programm nach der Überprüfung immer noch stumm bleibt, sollten Sie die (Strom- und Daten-) Kabel zu Ihren Lautsprechern überprüfen.

Dieses Programm gibt die Worte `ten`, `nine`, `eight`, ..., `one`, `zero` aus. Das Programm garantiert aber nicht für die richtige Reihenfolge der Worte. Die Sprachausgabe eines der Worte dauert ungefähr eine Sekunde. Möglicherweise wird die Aktion `SayText` aber in deutlich kürzeren Abständen aufgerufen (z.B. alle 1/10 oder 1/100 Sekunde). In diesem Fall werden die Aufrufe in einen Puffer geschrieben und von dort abgearbeitet. Von diesem Puffer wird *nicht* garantiert, dass er sich immer an die First-In-First-Out-Regel hält.

Im obigen Programm besteht die Schleife im Wesentlichen aus drei Blöcken: Dem `Merge`-, dem `Calculate`- und dem `If`-Block. Die `SayText`-Aktion ist kein „Bestandteil der Schleife“, sondern wurde nur als eine Art „Wurmfortsatz“ angehängt. Diese Struktur ist ein Grund für das Reihenfolge-Problem.

Wenn man zur Beseitung des Problems die `SayText`-Aktion „in die Schleife“ integriert (indem man den Pfeil vom `Merge`- zum `Calculate`-Block ersetzt durch einen Pfeil vom `TexttoSpeechTTS`- zum `Calculate`-Block) funktioniert die Schleife nicht mehr. Grund: Die `SayText`-Aktion konsumiert die `int`-Werte `10`, `9`, `8`, ... etc. und gibt sie nicht an den `Calculate`-Block weiter, der sie benötigt. Statt dessen liefert die `SayText`-Aktion dem `Calculate`-Block nur eine Art Erfolgsmeldung (technisch: Einen Wert des Typs `DefaultUpdateResponseType`).

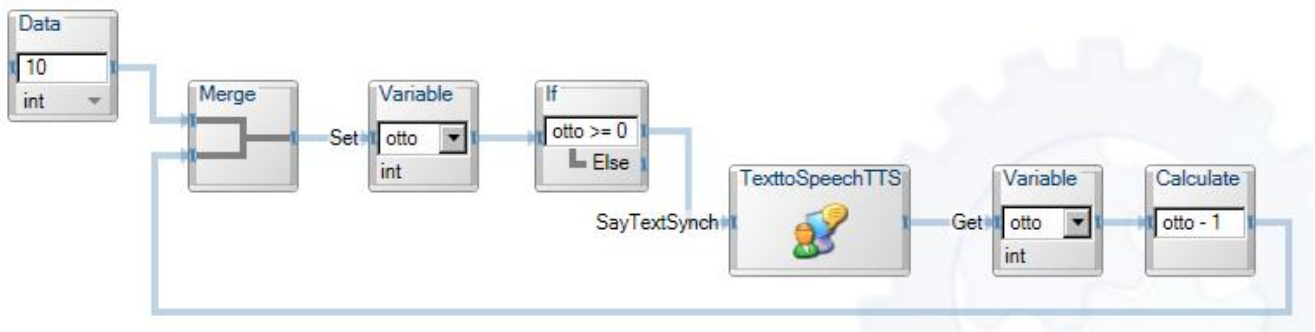


Zur Lösung des Problems setzen wir jetzt ein sehr allgemeines und mächtiges Konstrukt ein: Eine `Variable`. Variablen kann man definieren, indem man im Menü `Edit` den Eintrag `Define Variables` wählt, und im `Define Variables`-Fenster (links abgeildet) unten auf `Add` klickt (für jede Variable einmal). Den Standardnamen `Field` einer neuen Variablen sollte man durch einen besseren Namen ersetzen. Wenn einem der Standardtyp `int` nicht passt, muss man einen anderen Typ wählen.

Wenn man in einer herkömmlichen Sprache wie Fortran, Java oder C# eine Variable definiert hat, kann man (in einem bestimmten Bereich des Programms) auf sie zugreifen, indem man einfach ihren Namen hinschreibt. Das ist in VPL anders: Um einer Variablen einen Wert zuzuweisen oder um ihren Wert zu lesen muss man die Aktion `SetValue` bzw. die Aktion `GetValue` in einem `Variable`-Block aufrufen. Die Klasse `Variable` findet man im Fenster Basic Activities.

Hinweis: In einem VPL-Diagramm werden die Aktionsnamen `SetValue` und `GetValue` häufig durch `Set` und `Get` abgekürzt, und `SaySynch` ist eine Abkürzung für `SayTextSynchronous` (was seinerseits eine Abkürzung für `SayTextSynchronously` ist :-).

Beispiel-02: Eine Schleife mit einer Variablen (Beispielprogramm `B07SchleifeB`)



Im `Calculate`-Block (ganz rechts im Diagramm) darf man den Variablen-Namen `otto` nur deshalb verwenden, weil die `GetValue`-Aktion im vorhergehenden `Variable`-Block den Wert von

`otto` liest und ihn (als Verbund namens `value` mit einer Komponenten namens `value.otto`) an den `Calculate`-Block schickt. Auf die Variable `otto` kann man an beliebigen Stellen des Diagramms zugreifen, aber immer nur über die `GetValue`- oder `SetValue`-Aktion eines entsprechenden `Variable`-Blocks.

Die Aktion `SayTextSynchronous` ist erst dann beendet (und schickt eine Erfolgs-Meldung an den folgenden Block), wenn der betreffende Wert *fertig ausgesprochen* ist. Im Gegensatz dazu ist die Aktion `SayText` schon dann beendet, wenn der auszusprechende Wert *eingetroffen* ist (und vermutlich in einen Puffer geschrieben wurde). Diese Eigenschaft der `SayText`-Aktion („asynchron zu arbeiten“) war ein zweiter Grund für das Reihenfolge-Problem im vorigen Beispiel.

Das folgende Beispiel soll den Unterschied zwischen `SayText` und `SayTextSynchronous` noch einmal deutlich machen.

Beispiel-03: `SayText` (asynchron) und `SayTextSynchronous` (`B07TextToSpeechA`)

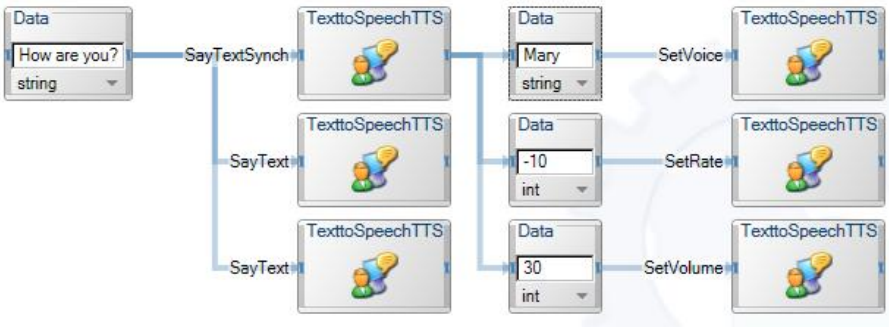
Das Aussprechen der Strings mit den vielen Zahlen darin dauert relativ lange (ein paar Sekunden). Das `Alert Dialog`-Fenster mit dem Text `Fertig 20` erscheint schon, bevor alle 20-er-Zahlen ausgesprochen sind. Damit das Programm dann weiterläuft, muss der Benutzer im `Alert Dialog`-Fenster auf `OK` klicken. Dann werden aller 30-er-Zahlen ausgesprochen und erst danach erscheint die Meldung `Fertig 30`.

Mit den Aktionen `SetVolume`, `SetRate` und `SetVoice` kann man die Lautstärke, die Schnelligkeit und die Stimme (männlich oder weiblich etc.) einer Sprachausgabe festlegen. Die folgenden Werte sind sinnvolle Eingaben für diese Aktionen:

Aktion	*Sinnvolle Eingabewerte*
`SetVolume`	`int`-Werte zwischen `0` und `100`
`SetRate`	`int`-Werte zwischen `-10` und `+10`
`SetVoice`	`string`-Werte aus der Menge {`Sam`}

In der kostenlosen Version des Robotics Studio spricht die Sprachausgabe immer mit der Stimme `Sam`. Es gibt aber noch weitere Frauen- und Männerstimmen (`Mary`, `John`, ... etc.) die man in Zukunft sicherlich aus dem Netz laden oder in einem Laden kaufen kann.

Beispiel-04: Lautstärke und Schnelligkeit der Sprachausgabe ändern (`B07TextToSpeechB`)



Zur Erinnerung: Alle sechs `Text to Speech (TTS)`-Blöcke in diesem Programm repräsentieren dasselbe Objekt (da sie alle mit demselben Namen `TexttoSpeechTTS` beschriftet sind).

Dieses Programm gibt den Text `How are you?` *dreimal* aus, mindestens *einmal* mit den Standardwerten für *Lautstärke* und *Schnelligkeit* (engl. volume and rate), die anderen beiden Male *entweder* mit den Standardwerten *oder* mit neu eingestellten Werten (`-10` für die Schnelligkeit, d.h. sehr langsam, und `30` für die Lautstärke, d.h. ziemlich leise). Der Versuch, die *Stimme* (engl. voice) vom Standardwert `Sam` auf `Mary` umzuschalten hat in meiner Umgebung keine Auswirkungen, weil die Stimme `Mary` dort noch nicht installiert ist.

In einem „ernsthaften Programm“ würde man die Lautstärke und Schnelligkeit der Sprachausgabe in aller Regel nicht nebenläufig zu `SayText`-Aktionen verändern, sondern genau festlegen, welche Ausgabe mit welcher Lautstärke etc. erfolgen soll.

11. Einfache Typen, Werte, Variablen und Verbunde

Was für Daten fließen entlang den Pfeilen in einem VPL-Diagramm? Welche Struktur haben diese Daten? Wie kann man darauf zugreifen? Diese Fragen sollen hier behandelt werden.

Regel-07: In VPL gibt es 14 einfache Typen: `bool`, `byte`, `char`, `decimal`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.

Regel-08: Die Daten, die als Einheit über einen Pfeil zu einer anderen Aktion geschickt werden, heissen immer `value`. Dieser Standardname kann einen *einfachen Wert* (einen Wert eines einfachen Typs) oder einen *Verbund* (-wert, einen Wert eines Verbundtyps) bezeichnen.

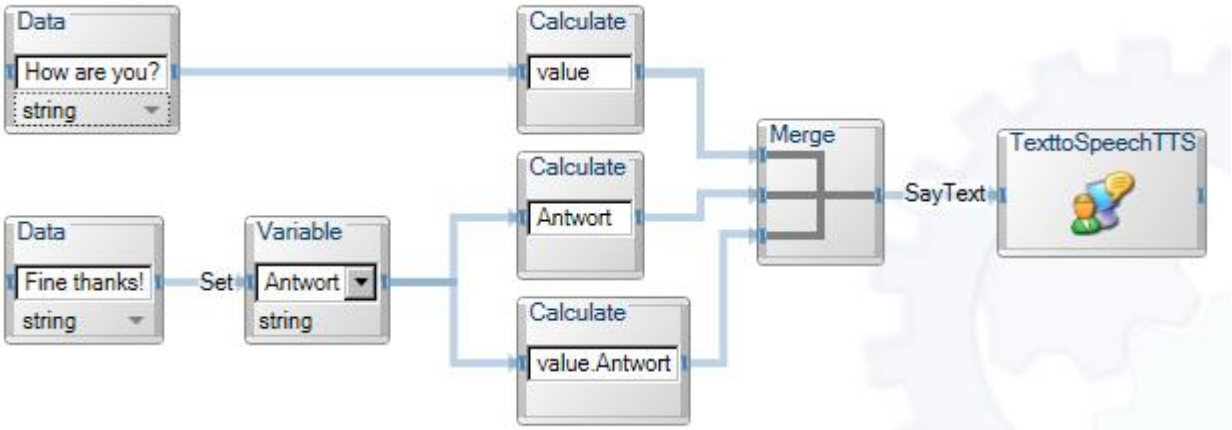
Verbunde werden im Englischen häufig als records oder structs bezeichnet.

Regel-09: Ein Verbund besteht aus einer oder mehreren Komponenten (engl. fields). Die Komponenten eines Verbundes können zu (gleichen oder) verschiedenen einfachen Typen und/oder Verbundtypen gehören.

Regel-10: Auf die Komponenten eines Verbundes kann man mit der üblichen Punktnotation zugreifen.

Z.B. bezeichnet `value.PNr` die Konponente `PNr` des Verbundes `value`, und `value.Person.Gewicht` bezeichnet die Komponente `Gewicht` des Verbundes `value.Person`.

Beispiel-01: Werte und Variablen (das Programm `B08VariablenA`)



Ein `Data`-Block liefert einen *einfachen* Wert namens `value` (siehe Regel-08). Im obersten `Calculate`-Block des Beispiels bezeichnet der Name `value` den `string`-Wert `How are you?`.

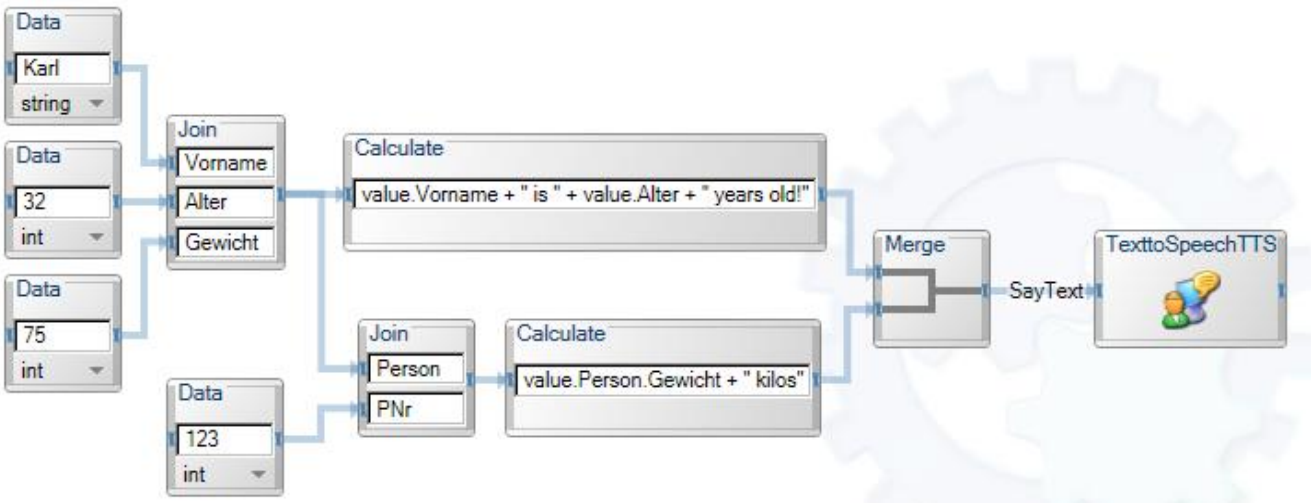
Ein `Variable`-Block liefert einen *Verbund* namens `value`. Dieser Verbund hat nur eine Komponente, die so heisst wie die (im Block angegebene) Variable. Im untersten `Calculate`-Block des Beispiels bezeichnet der (zusammengesetzte) Name `value.Antwort` den `string`-Wert `Fine thanks!`. Statt mit `value.Antwort` kann man diesen Wert auch einfach mit `Antwort` bezeichnen (wie im mittleren `Calculate`-Block des Beispiels).

Das Programm gibt einmal „How are you?“ und zweimal „Fine thanks!“ aus. Die Reihenfolge der Ausgaben wird durch das Programm nicht festgelegt.

Ein `Variable`-Block liefert als Ausgabe immer *einen Verbund mit einer Komponenten*, unabhängig davon, ob man mit dem Block die `SetValue`- (wie hier im obigen Beispiel) oder die `GetValue`-Aktion (wie im Beispiel-02 des vorigen Abschnitts) aufruft. Mir ist nicht klar, warum die Aktion `GetValue` nicht den (einfachen) Wert der Variablen liefert, sondern einen Verbund.

Mit einem `Join`-Block kann man 2 oder mehr Komponenten zu einem Verbund zusammenfassen. Gleichzeitig legt man dabei für jede Komponente einen Namen fest (mit dem man später auf diese Komponente zugreifen kann).

Beispiel-02: Verbunde zusammensetzen und auf die Komponenten zugreifen (`B08VariablenB`)



Der linke `Join`-Block liefert einen Verbund namens `value` mit drei Komponenten namens `value.Vorname` (vom Typ `string`), `value.Alter` (vom Typ `int`) und `value.Gewicht` (ebenfalls vom Typ `int`).

Der zweite `Join`-Block liefert einen Verbund namens `value` mit zwei Komponenten namens `value.Person` (der zu einem namenlosen Verbundtyp gehört) und `value.PNr` (vom Typ `int`).

Das Programm gibt „Karl is 32 years old“ und „75 kilos“ in unbestimmter Reihenfolge aus.

In den `Calculate`-Blöcken darf man anstelle von `value.Vorname`, `value.Alter` und `value.Person.Gewicht` auch einfacher `Vorname`, `Alter` bzw. `Person.Gewicht` schreiben. Die etwas ausführlichere Notation wurde hier verwendet um die Strukturen der Verbunde zu betonen.

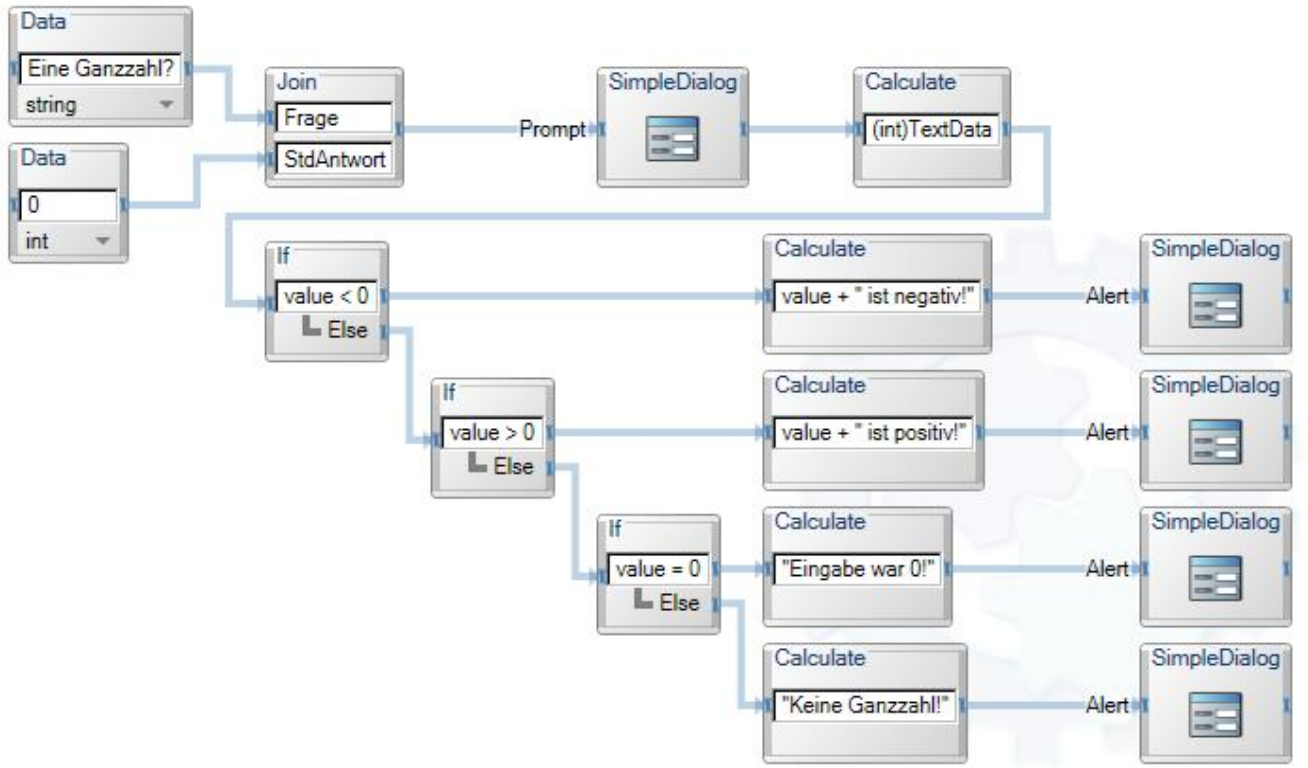
12. If- und Switch-Blöcke

Ein `If`-Block (eigentlich: Die einzige Aktion eines `If`-Blocks) hat einen Eingang, ein kleines Textfensterchen, in dem man einen booleschen Ausdruck eintragen sollte (z.B. `value = 1234` oder `value.gewicht > 17`) und zwei Ausgänge, einen Then-Ausgang rechts neben dem Fensterchen und darunter einen (mit `Else` gekennzeichneten) Else-Ausgang. Am Eingang kann man einen Eingabepfeil und an jedem der beiden Ausgänge beliebig viele Ausgabepfeile (Then-Pfeile bzw. Else-Pfeile) anbringen.

Wenn über den Eingabepfeil ein Wert eintrifft, fließt der entweder über alle Then-Pfeile oder über alle Else-Pfeile weiter, je nachdem ob der Ausdruck im Fensterchen den Wert `true` oder `false` hat.

Anmerkung: Einen Test auf Gleichheit kann man in VPL wahlweise mit *einem* oder mit *zwei* Gleichheitszeichen notieren (z.B. `value = 123` oder `value == 123`), so dass sich sowohl C/Java/C#-geschädigte als auch Algol/Pascal/Ada-verwöhnte ProgrammiererInnen gleich „wie zu Hause bei ihrer Muttersprache“ fühlen :-).

Beispiel-01: Eine Kaskade von drei `If`-Blöcken (Beispielprogramm `B09IfA`)



In einem `Prompt Dialog`-Fenster wird der Benutzer zur Eingabe einer Ganzzahl aufgefordert. Dabei wird ihm als Standard-Antwort die Zahl `0` vorgeschlagen. Im obersten `Calculate`-Block wird die Eingabe des Benutzers zu einem `int`-Wert *gecastet*. Falls der Benutzer keine Ganzzahl (wie z.B. `123` oder `-17` oder `+3`) eingibt, sondern eine andere Zeichenfolge (z.B. `1.5` oder `1,5` oder `1A` oder `ABC`), dann ist dieser `int`-Wert „leer“: Wenn man ihn in einen `string`-Wert umwandelt (um ihn auszugeben) erhält man einen leeren String.

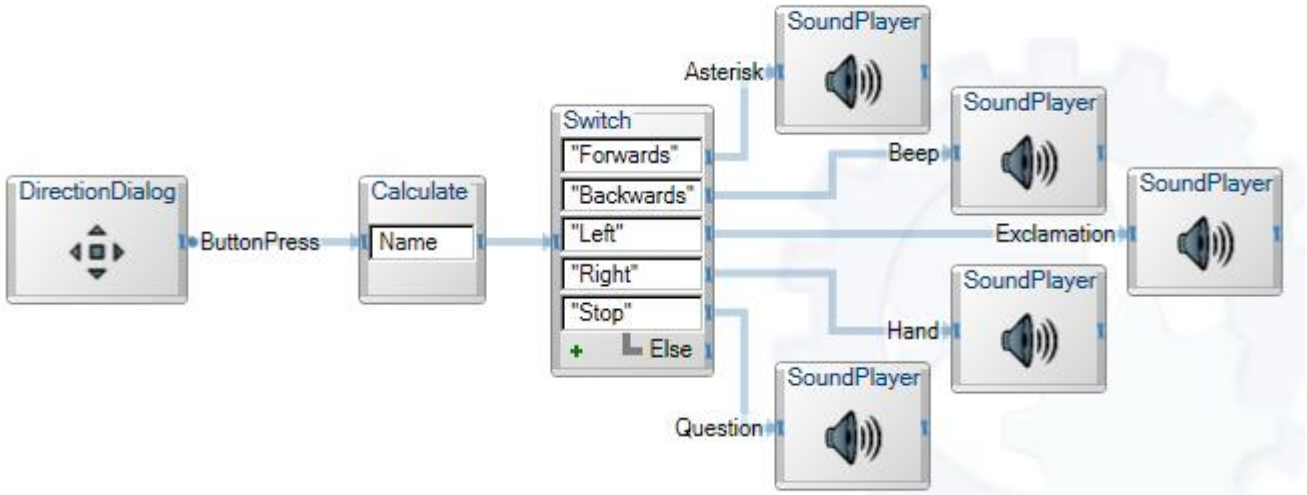
Je nachdem, ob der Benutzer eine negative Zahl, eine positive Zahl, 0 oder eine „unerlaubte Zeichenfolge“ eingegeben hat, wird eine andere kleine Meldung ausgegeben.

Aufgabe: Entwickeln Sie eine Variante dieses Programms, bei der die Meldung im Falle einer unerlaubten Eingabe auch *die falsche Eingabe* anzeigt, etwa so: `"ABC ist keine Ganzzahl"` (statt nur `"Keine Ganzzahl"`). Das Beispielprogramm `B09IfB` ist eine Lösung dieser Aufgabe.

Ein `Switch`-Block (eigentlich: die einzige Aktion eines `Switch`-Blocks) hat einen Eingang und einen (mit `Else` gekennzeichneten) Else-Ausgang und enthält dazwischen anfangs nur einen *Fall*. Ein Fall besteht aus einem Textfensterchen und einem Fall-Ausgang daneben. Jedesmal wenn man auf das kleine Pluszeichen (in der linken unteren Ecke des Blocks) klickt, wird ein weiterer Fall eingefügt. In jedes der Textfensterchen sollte man einen Ausdruck eintragen (z.B. `321` oder `"Hallo"` etc.). Alle Ausdrücke sollten zum selben Typ gehören. Am Eingang kann man einen Eingabepfeil anbringen und an jedem der Ausgänge kann man beliebig viele Gleich- bzw. Else-Pfeile anbringen. Den Wert des Ausdrucks im Fensterchen eines Falls bezeichnen wir kurz als den *Wert des Falls*.

Wenn über den Eingabepfeil ein Wert eintrifft, fließt der entweder über alle Fall-Pfeile eines Falls weiter (wenn er gleich dem Wert des Falls ist) oder über alle Else-Pfeile. Falls mehrere Fälle den gleichen Wert haben, „gewinnt“ nur der oberste Fall (d.h. der Eingabewert fließt nur über seine Pfeile weiter).

Beispiel-02: Ein klangvoller `Switch`-Block mit fünf Fällen (aber ohne einen Pfeil am Else-Ausgang)

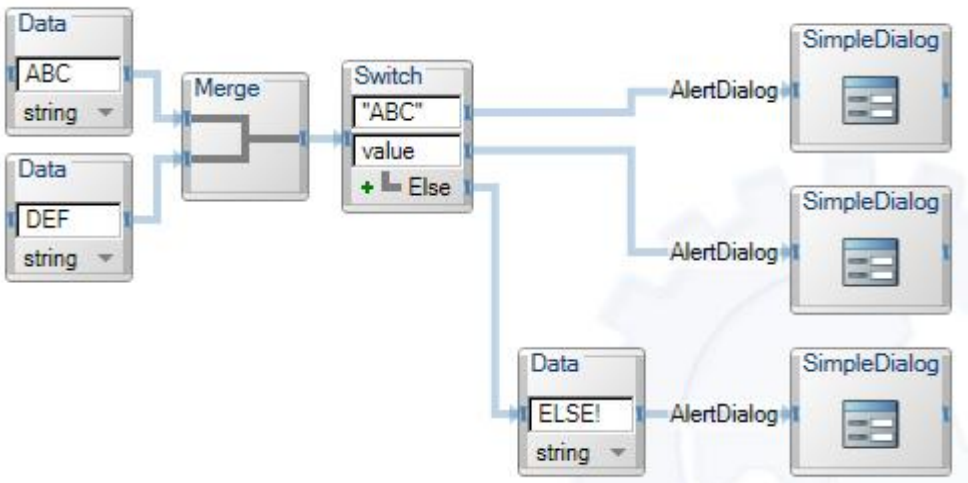


Wenn dieses Programm ausgeführt wird, erscheint auf dem Bildschirm ein `Direction Dialog`-Fenster mit fünf Knöfen darin (wie im Beispiel-03 im Abschnitt 8). Wenn der Benutzer mit der Maus auf einen der Knöpfe klickt, ertönt ein bestimmter Klang.

Anmerkung: Falls einer der Klänge (`AsteriskSound`, `Beep` etc.) „stumm bleibt", kann das an Einstellungen im Betriebssystem liegen (siehe Fenster `Systemsteuerung`, `Sounds und Audiogeräte`, `Sounds`. Nur *die* Sounds, vor denen ein Lautsprecher-Symbol steht, sind „aktiviert". Die anderen kann man im selben Fenster aktivieren).

Anmerkung: `Asterisk` und `Question` etc. als Abkürzung für die Aktionsnamen `AsteriskSound` und `QuestionSound` kann ich leicht nachvollziehen. `Hand` als Abkürzung für `CriticalStopSound` ist etwas schwieriger.

Beispiel-03: Ein Fall mit `value` als Ersatz für Else



Die Else-Alternative in einem `Switch`-Block ist eigentlich überflüssig, da man ihren Effekt leicht auch mit dem Eintrag `value` im letzten Fall-Fensterchen erreichen kann. Im links abgebildeten Beispiel `B09SwitchB` wird die Else-Alternative des `Switch`-Blocks nie ausgeführt.

13. Ein Diagramm in einem Activity-Objekt definieren

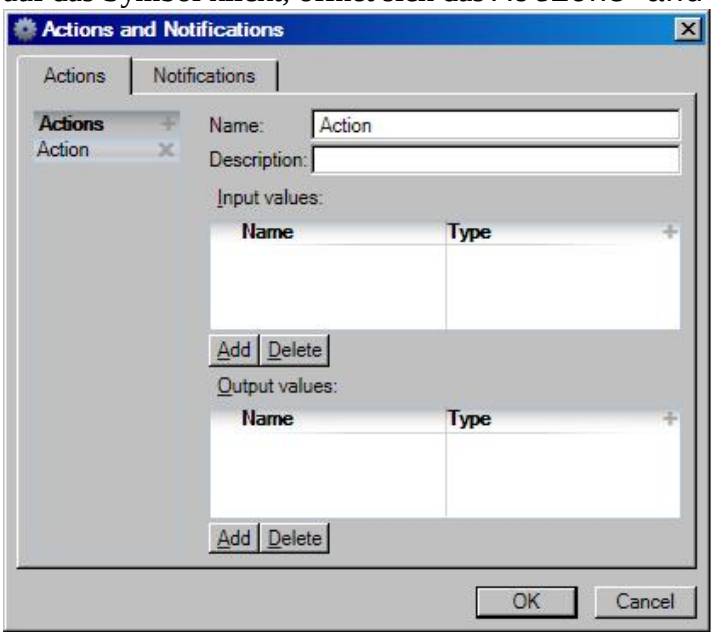
Welche Aktionen ein Objekt enthält, wird normalerweise durch seine Klasse festgelegt. Z.B. enthält jedes `Simple Dialog`-Objekt drei Aktionen namens `AlertDialog`, `ConfirmDialog` und `PromptDialog`. Eine Ausnahme bilden die Objekte der Klasse `Activity`: Sie sind anfangs leer, man kann sie aber öffnen und Aktionen „hineindefinieren". Von jeder der Aktionen kann man festle-

gen, welche Eingangsparameter sie erwartet, welche Ausgangsparameter sie erzeugt und „was sie macht“.

Ziehen Sie die Klasse `Activity` (aus dem Fenster Basic Activities) über die Arbeitsfläche. Dadurch wird ein neues `Activity`-Objekt erzeugt und als Block mit dem Namen `Activity` dargestellt (wie links abgebildet). Mit einem Doppelklick auf diesen Block kann man das Objekt öffnen und die Arbeitsfläche stellt dann das (anfangs leere) Diagramm im Inneren des `Activity`-Objekts dar, etwa so wie unten abgebildet.

Wenn man auf den Reiter `Diagram` klickt, sieht man wieder das *Hauptdiagramm* (welches zur Zeit nur aus dem *einen* `Activity`-Block besteht). Klickt man auf den Reiter `Activity`, sieht man wieder das (zur Zeit noch leere) Innere der `Activity`.

Wenn man das Innere der `Activity` anzeigen läßt, sieht man unter den Reitern `Diagram` und `Activity` ein kleines Fensterchen, in dem der Aktionsname `Action` steht. Links daneben steht als Beschriftung `Action:` und rechts daneben ein kleines Weltkugel-und-Bleistift-Symbol. Wenn man auf das Symbol klickt, öffnet sich das `Actions and Notifications`-Fenster, etwa so:



Wie man sieht ist anfangs der `Actions`-Reiter gewählt (den anderen Reiter, `Notifications`, werden wir erst im Abschnitt 18 benutzen).

Auf der linke Seite sieht man die Liste aller Aktionen. Sie hat die Überschrift Actions (mit s, fett geschrieben und mit einem etwas blaßen Pluszeichen daneben). In der Liste ist anfangs nur eine Aktion namens `Action` (ohne s, mit einem ebenso blaßen `X` daneben) eingetragen.

Jedesmal, wenn Sie auf das Pluszeichen neben Action klicken, wird eine neue Aktion in die Liste eingetragen und diese Aktionen haben erstmal die Standardnamen `Action`, `Action0`, `Action1`... etc.

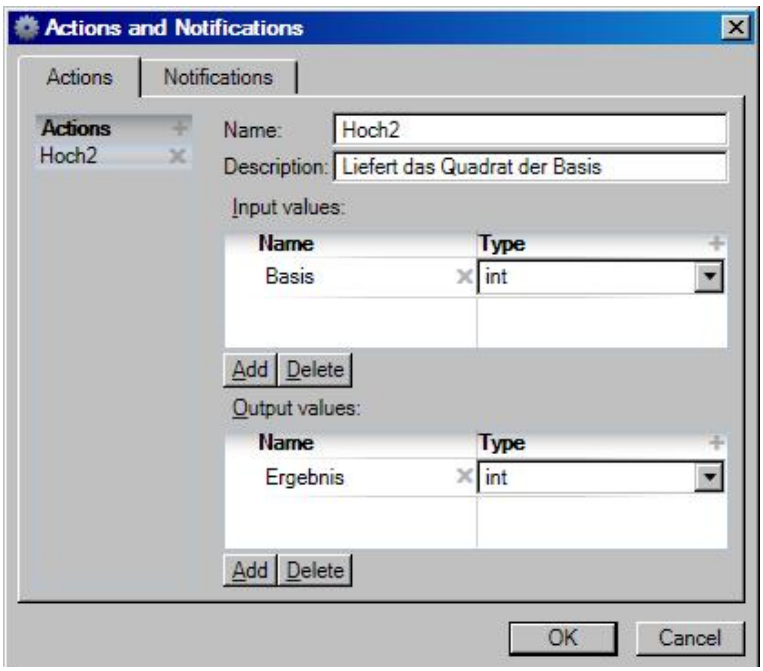
Wenn Sie auf das `X` neben dem Namen einer Aktion klicken, wird die Aktion wieder gelöscht und ihr Name aus der Liste entfernt.

Wenn Sie (in der Liste aller Aktionen) auf den Namen einer Aktion klicken, werden im rechten Teil des Fensters Informationen über *diese* Aktion angezeigt und Sie können gewisse Eigenschaften der Aktion verändern, z.B. ihren Namen, der oben in der mit `Name:` beschrifteten Textzeile steht. Ersetzen Sie dort den Standardnamen `Action` z.B. durch `KarlHeinz` und klicken Sie danach in die `Description:`-Zeile. Der neue Name `KarlHeinz` sollte jetzt in der Liste aller Aktionen angezeigt werden.

Wenn Sie auf Return drücken, wird das `Actions and Notifications`-Fenster geschlossen. Indem Sie erneut auf das Weltkugel-und-Bleistift-Symbol klicken können Sie es wieder öffnen.

In diesem Fenster können Sie nicht nur neue Aktionen erzeugen und ihre Namen verändern, sondern auch die Eingangsparameter und die Ausgangsparameter jeder Aktion festlegen, in der Liste `Input values:` bzw. `Output values:`. Indem Sie auf den `Add`-Knopf unter einer Liste klicken, fügen Sie einen neuen Parameter ein. In der Spalte `Name` können Sie dann den Standardnamen `Field` durch einen selbsgewählten Namen ersetzen und in der Spalte `Type` den Standardtyp `int` stehen lassen oder durch einen anderen Typ ersetzen. Mit einem Klick auf `Delete` (oder auf das X rechts neben dem Namen eines Parameters) können Sie einen Parameter jederzeit wieder löschen.

Ändern Sie jetzt die Eintragungen im `Actions und Notifications`-Fenster so wie in der folgenden Abbildung dargestellt:

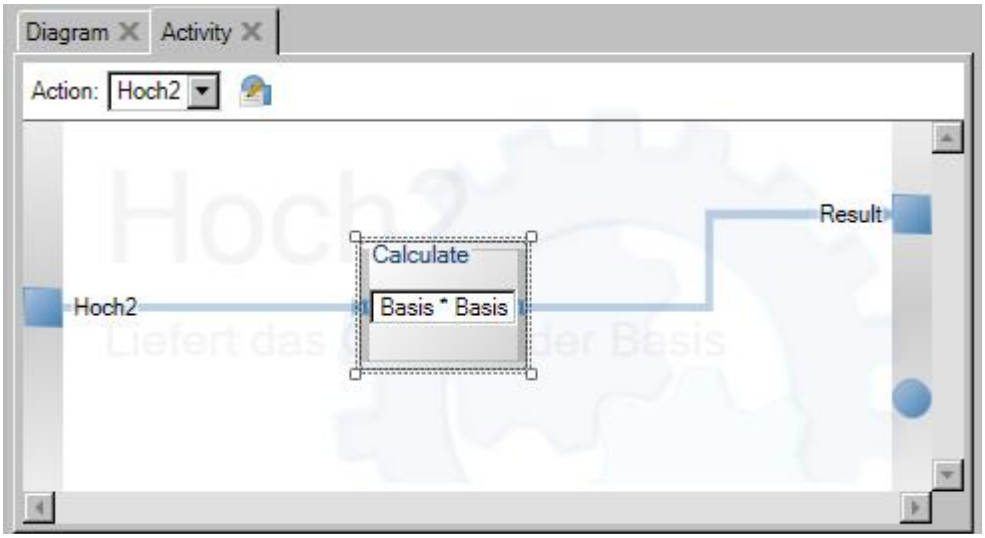


In der Liste aller Aktionen (unter Actions) ist eine Aktion namens `Hoch2` eingetragen. Diese Aktion hat einen Eingangsparameter namens `Basis` vom Typ `int` und einen Ausgangsparameter namens `Ergebnis`, ebenfalls vom Typ `int`. Der Text neben `Description:` ist ein Komentar.

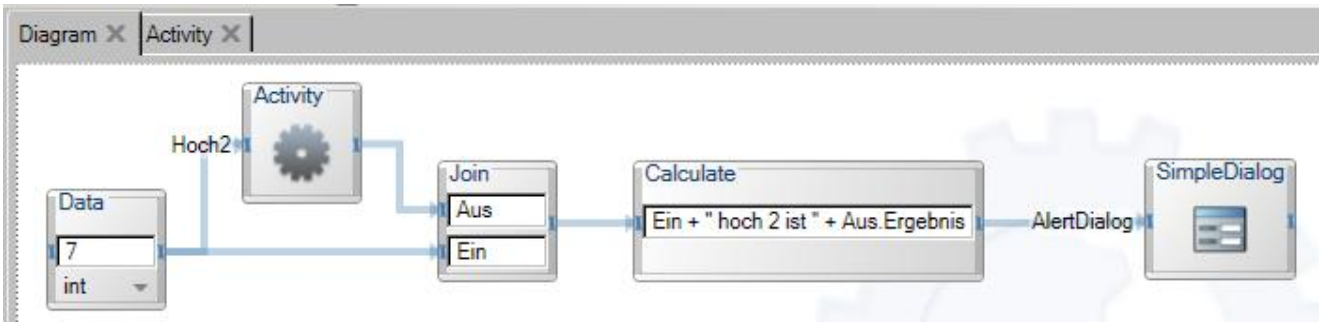
Wenn Sie das `Actions and Notifications`-Fenster schliessen (mit Return oder einem Klick auf OK) und zur Darstellung des Inneren der Activity zurückkehren werden Sie bemerken, dass das blaue Quadrat im linken Rand nicht mehr mit `Action` beschriftet ist, sondern mit dem neuen Aktionsnamen `Hoch2`.

Erzeugen Sie jetzt im Innern der Activity einen `Calculate`-Block und zeichnen Sie zwei Pfeile ein: Einen vom `Hoch2`-Eingang zum `Caculate`-Block und einen vom `Calculate`-Block zum `Return`-Ausgang. Tragen Sie in den `Calculate`-Block den Ausdruck `Basis * Basis` ein, wie unten dargestellt.

Dieses relativ einfache Diagramm (ein Block und zwei Pfeile) beschreibt, wie die `Hoch2`-Aktion aus ihrem Eingangsparameter `Basis` ihren Ausgangsparameter `Ergebnis` berechnet. Kompliziertere Aktionen werden durch kompliziertere Diagramme beschrieben.



Klicken Sie jetzt auf den Reiter `Diagram` und ergänzen Sie das Hauptdiagramm des Programms so, dass es etwa wie in der Abbildung unten aussieht. Im `Calculate`-Block werden die beiden `int`-Werte `Ein` und `Aus.Ergebnis` in `string`-Werte umgewandelt und mit dem Literal `" hoch 2 ist "` konkateniert.

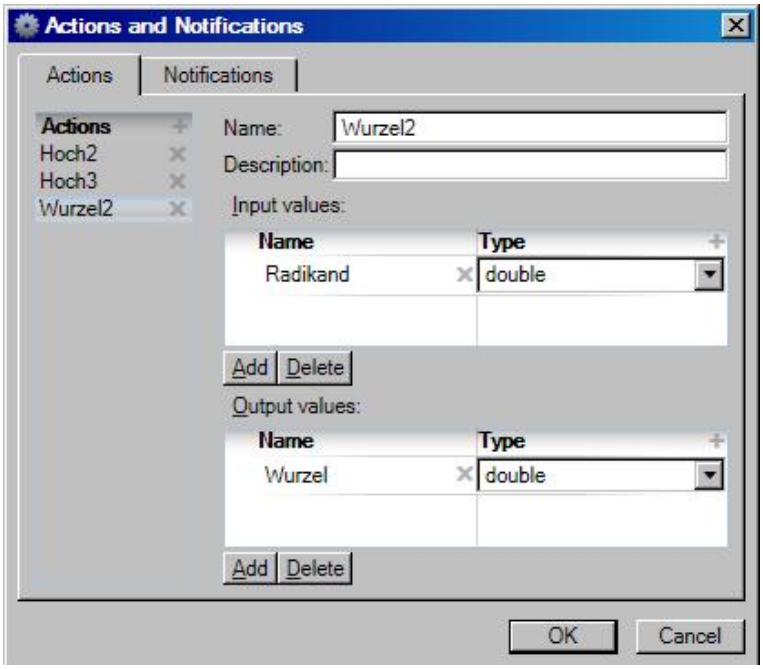


Speichern Sie das Programm z.B. unter dem Namen `B10ActivityA` ab. Wenn Sie es ausführen lassen, sollte ein `Alert Dialog`-Fenster mit der Meldung `7 hoch 2 ist 49` auf dem Bildschirm erscheinen. Falls das Programm nicht gleich funktioniert, sollten Sie alle Pfeile entsprechend der Regel-01 (siehe oben im Abschnitt 2) überprüfen und eventuell korrigieren.

Anmerkung: Der `Join`-Block im Hauptdiagramm liefert einen Verbund mit zwei Komponenten namens `Aus` und `Ein`. Die `Ein`-Komponente (das Ergebnis des `Data`-Blocks) ist vom einfachen Typ `int`. Dagegen ist die `Aus`-Komponente (das Ergebnis unseres `Activity`-Blocks) ein Verbund mit einer Komponente namens `Ergebnis` vom Typ `int`. Deshalb muss im nachfolgenden `Calculate`-Block `Aus.Ergebnis` stehen, und nicht nur `Aus`. Siehe dazu auch den Abschnitt 11.

14. Ein Activity-Objekt mit mehreren Aktionen darin

Im vorigen Abschnitt wurde ein Activity-Objekt mit einer einzigen Aktione (namens `Hoch2`) darin definiert. Ganz ähnlich kann man ein Activity-Objekt mit mehreren Aktionen darin definieren. Beginnen Sie wieder mit einem Activity-Objekt, öffnen Sie es durch Doppelklicken und öffnen Sie das `Actions and Notifications`-Fenster, indem Sie auf das Weltkugel-und-Bleistift-Symbol klicken. Tragen Sie in die *Liste aller Aktionen* (unter Actions) drei Aktionen namens `Hoch2`, `Hoch3` und `Wurzel2` ein. Geben Sie den Aktionen `Hoch2` und `Hoch3` ganz entsprechende Eigenschaften wie der Aktion `Hoch2` im vorigen Beispiel (ein Eingangsparameter namens `Basis` vom Typ `int` und ein Ausgangsparameter namens `Ergebnis` ebenfalls vom Typ `int`). Geben Sie der Aktion `Wurzel2` zur Abwechslung ein bisschen andere Eigenschaften wie oben abgebildet.



In dieser Abbildung erkennt man auch, dass in der Liste aller Aktionen (unter Actions) drei Aktionen eingetragen sind und dass der rechte Teil des Fensters gerade Eigenschaften der Aktion `Wurzel2` anzeigt (der Name der aktuellen Aktion steht oben neben `Name:`).

Die aktuelle Aktion `Wurzel2` hat einen Eingangsparameter namens `Radikand` vom Typ `double` und einen Ausgangsparameter namens `Wurzel` ebenfalls vom Typ `double`.

Die Berechnung der Wurzel selbst in VPL zu programmieren ist theoretisch wohl möglich, praktisch benützt man dazu den vordefinierten Service `Math Functions` (im Fenster Services).

Das hier skizzierte Beispiel findet man auch unter dem Namen `B10ActivityB` bei den Beispielprogrammen. Im Hauptdiagramm dieses Programm werden drei Kopien des „selbst gestalteten `Activity`-Blocks“ dazu benutzt, jede der drei Aktionen einmal aufzurufen und das Ergebnis auszugeben. Die Kopien des `Activity`-Blocks kann man mit den Tastenkombinationen `Strg-C` und `Strg-V` erzeugen.

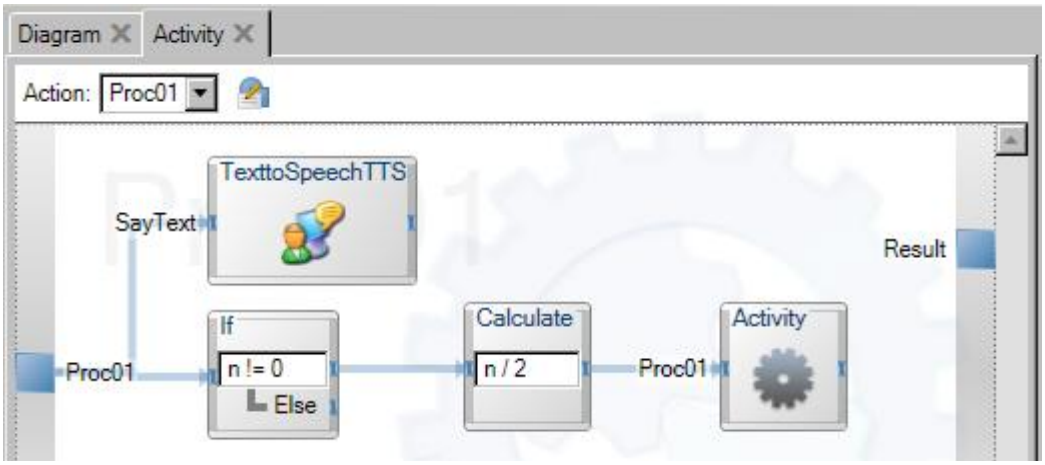
15. Ein beinahe nützliches Activity-Objekt

Das Beispielprogramm `B10ActivityF` (hier nicht abgebildet) enthält ein `Activity`-Objekt mit einer Aktion namens `LiesInt`. Diese Aktion fordert den Benutzer mit einem `Prompt Dialog`-Fenster zur Eingabe einer Ganzzahl auf. Falls der Benutzer statt einer Ganzzahl (z.B. `123` oder `-45` oder `+6789`) eine unerlaubte Zeichenfolge eingibt (z.B. `1.5` oder `1,5` oder `1A` oder `ABC`) wird die Eingabeaufforderung so lange wiederholt, bis der Benutzer wirklich eine Ganzzahl eingibt.

Merkwürdigerweise kritisiert die Entwicklungsumgebung dieses Programm mit einer Fehlermeldung, führt es aber trotzdem aus (mit der intendierten Wirkung). Außerdem scheint die Entwicklungsumgebung Kommentare zu zensieren und hat einige ohne meine Aufforderung oder Zustimmung gelöscht.

16. Ein rekursives Activity-Objekt

Das Beispielprogramm `B10ActivityC` enthält einen `Activity`-Block (mit einer Aktion `Proc01` darin), der in das Diagramm „in seinem eigenen Inneren“ kopiert wurde. Dieser `Activity`-Block wirkt wie ein rekursiver Methodenaufruf. Das Hauptdiagramm des Programms sieht aus wie rechts abgebildet. Das Diagramm im Inneren des `Activity`-Blocks ist unten wiedergegeben:



An diesem Diagramm erkennt man, dass die Aktion `Proc01` kein Ergebnis liefert (sie ist nicht mit dem `Result`-Ausgang verbunden).
Das Kopieren und „rekursive Einfügen“ des `Activity`-Blocks geht völlig problemlos: Im Hauptdiagramm wählt man den Block mit der Maus, kopiert ihn mit `Strg-C`, klickt auf den Reiter `Activity` (um das Diagramm im Innern des `Activity`-Blocks anzeigen zu lassen) und fügt den Block mit `Strg-V` ein.

17. Ein Activity-Objekt mit Variablen

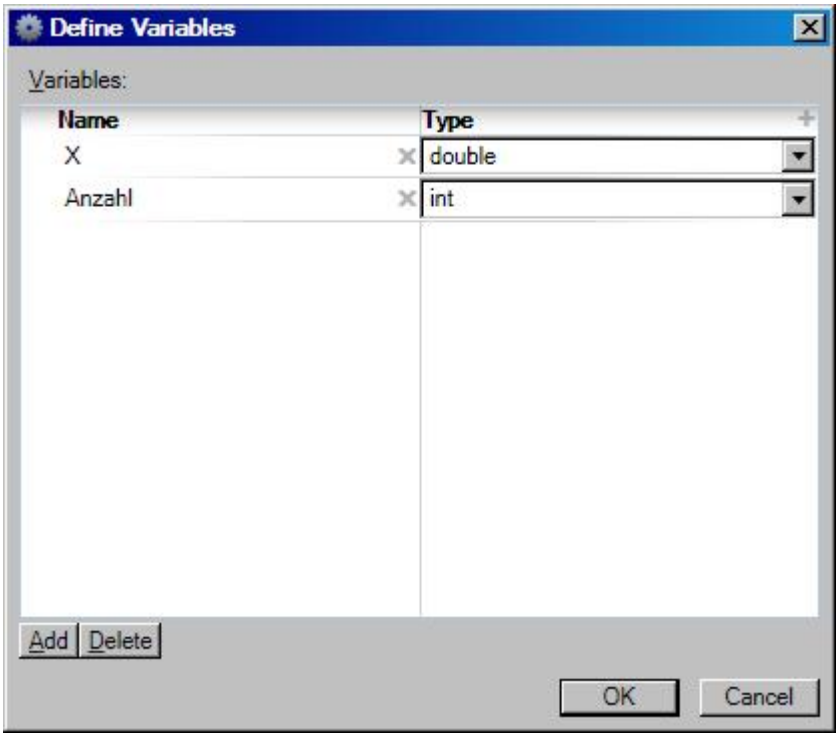
Im Abschnitt 14 wurde gezeigt, wie man mehrere Aktionen in einem Activity-Objekt zusammenfassen kann. Bei dem dort gezeigten Beispiel gab es allerdings keinen *wichtigen* Grund, die beiden Aktionen im selben Objekt unterzubringen, statt zwei Objekte mit je einer Aktion darin zu erzeugen.

Wenn man in einem Activity-Objekt *Variablen* definiert, dann können alle Aktionen des Objekts darauf zugreifen und über diese Variablen miteinander kommunizieren. Das ist ein *wichtiger* Grund dafür, mehrere Aktionen in einem Activity-Objekt zusammenzufassen.

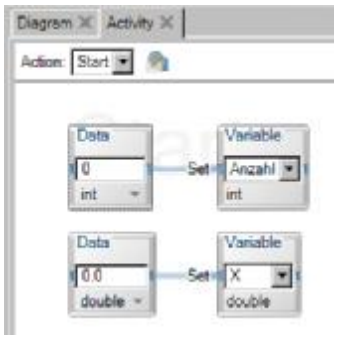
Im folgenden Beispiel werden in einem Activity-Objekt zwei Variablen namens `X` und `Anzahl` und vier Aktionen namens `SetX`, `GetAnzahl`, `AddToX` und `MulByX` zusammengefasst. Die Aktion `AddToX` hat nur einen (`double`-) Parameter, addiert ihn zum Wert der Variablen `X` und liefert die Summe als Ergebnis. Ganz entsprechend funktioniert `MulByX`. Wenn man also z.B. `5` Zahlen mit `33.3` multiplizieren will, braucht man die `33.3` nur einmal der Variablen `X` zuzuweisen (mit der Aktion `SetX`) und muss dann in den 5 Aufrufen der Aktion `MulByX` nur noch den anderen Faktor angeben.

Die `int`-Variable `Anzahl` wird jedesmal um `1` erhöht, wenn eine der Aktionen `AddY` oder `MulByX` aufgerufen wird. Mit `GetAnzahl` kann man den momentanen Wert der Variablen „lesen“.

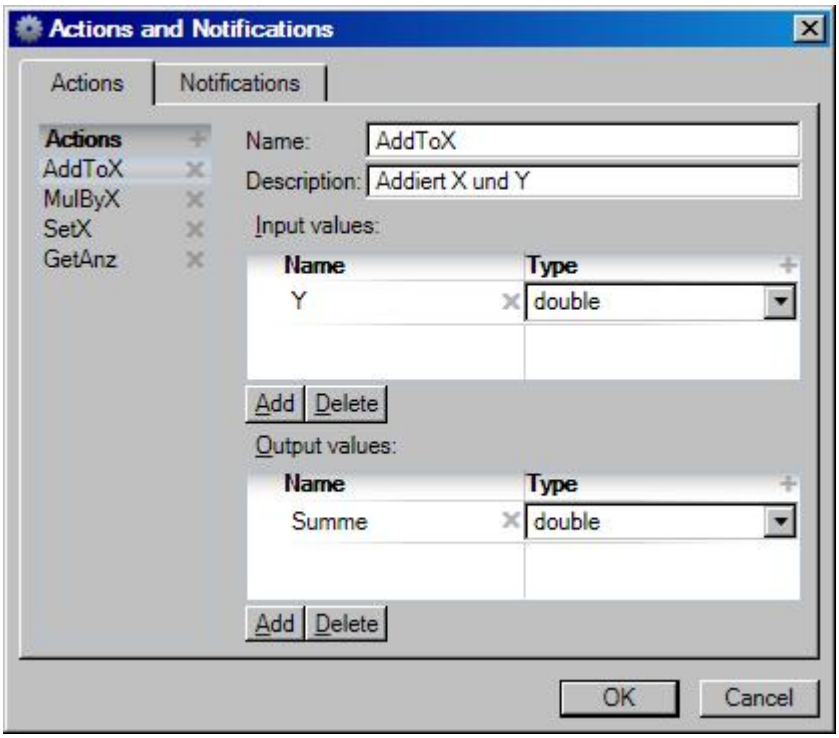
Wir beginnen wieder mit der Erzeugung eines Activity-Objekts, öffnen es durch einen Doppelklick auf seinen Block, öffnen dann im Edit-Menü den Punkt `Define Variables ...` , klicken zweimal auf `Add` und definieren die Variablen `X` und `Anzahl` wie nebenstehend dargestellt.



Wenn das Activity-Objekt zur Laufzeit erzeugt wird, müssen (noch vor dem ersten Aufruf einer Aktion) die beiden Variablen *initialisiert* werden. Dass können wir (nur) im *Start-Teil* des Activity-Objekts erledigen. Der Start-Teil steht (als eine Art „Pseudo-Aktion“) in der Liste aller Aktionen (zwischen dem Wort `Action:` und dem Weltkugel-und-Bleistift-Symbol), wo wir ihn auswählen können.



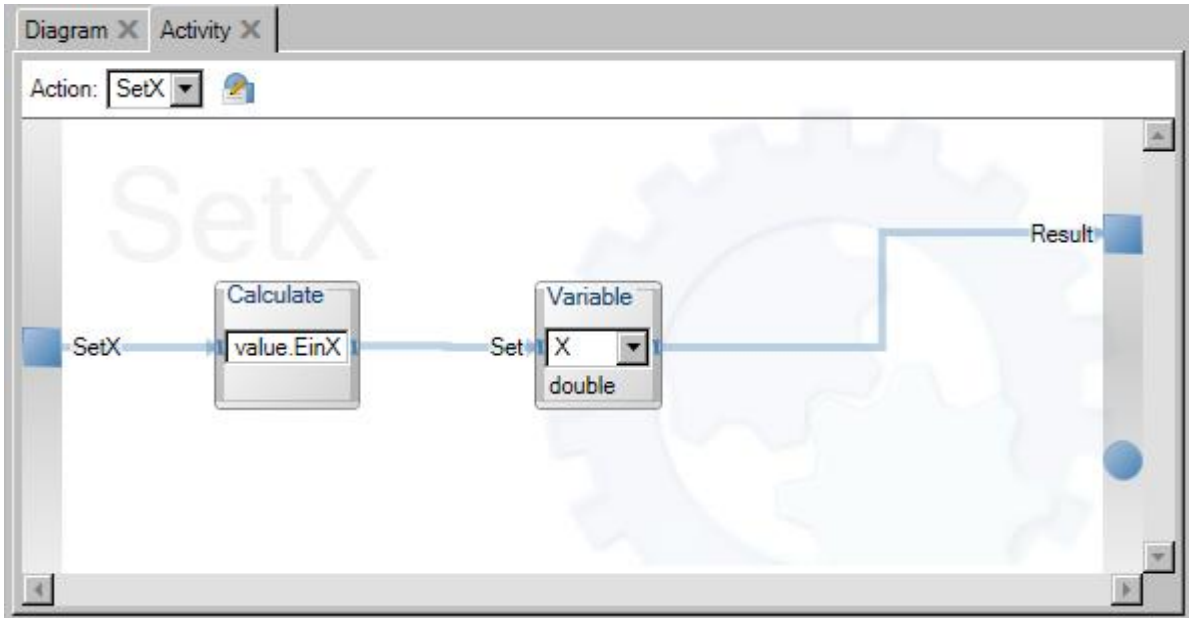
In der Abbildung rechts wird ein Teil des „Inneren“ des Activity-Blocks angezeigt. Im Wahlfeld neben `Action:` wurde `Start` gewählt und in dieser Pseudo-Aktion werden die Variablen `Anzahl` und `X` mit dem `int`-Wert `0` bzw. mit dem `double`-Wert `0.0` initialisiert. Die Pseudo-Aktion `Start` wird (ähnlich wie ein Konstruktor in Java oder C#) nur einmal bei der Erzeugung des Activity-Objekts ausgeführt.



Durch einen Doppelklick auf das Welkugel-mit-Bleistift-Symbol können wir das `Actions and Notifications`-Fenster öffnen und dort wichtige Eigenschaften der vier Aktionen `AddToX`, `MulByX`, `SetX` und `GetAnzahl` festlegen, nämlich ihre Namen und ihre Ein- und Ausgangsparameter.

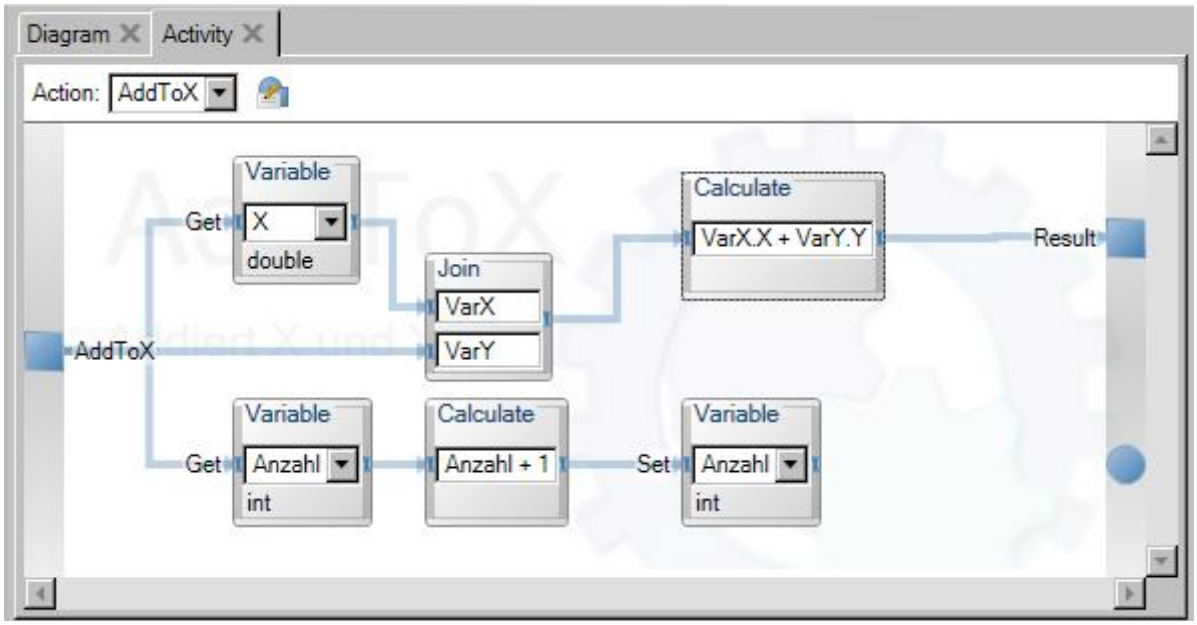
Die nebenstehende Abbildung zeigt, dass die Aktion `AddToX` nur einen Eingangsparameter namens `Y` vom Typ `double` und einen Ausgangsparameter namens `Summe`, ebenfalls vom Typ `double`, hat. Für die anderen Aktionen muss man entsprechende Eigenschaften festlegen (`GetX` braucht keinen Eingangsparameter und `SetX` keinen Ausgangsparameter).

Das Diagramm der Aktion `SetX` sollten etwa wie unten abgebildet definiert werden:



Als Eingabe erwartet diese Aktion einen Verbund namens `value` mit einer `double`-Komponenten namens `EinX`. Der `Calculate`-Block ist notwendig, um aus der Eingabe `value` die Komponente `value.EinX` „herauszupicken“. Anstelle von `value.EinX` darf man auch kürzer `EinX` schreiben, aber die ausführlichere Notation macht den Sinn des `Calculate`-Blocks („einen Verbund auf seine einzige Komponente zu projizieren“) wohl etwas deutlicher.

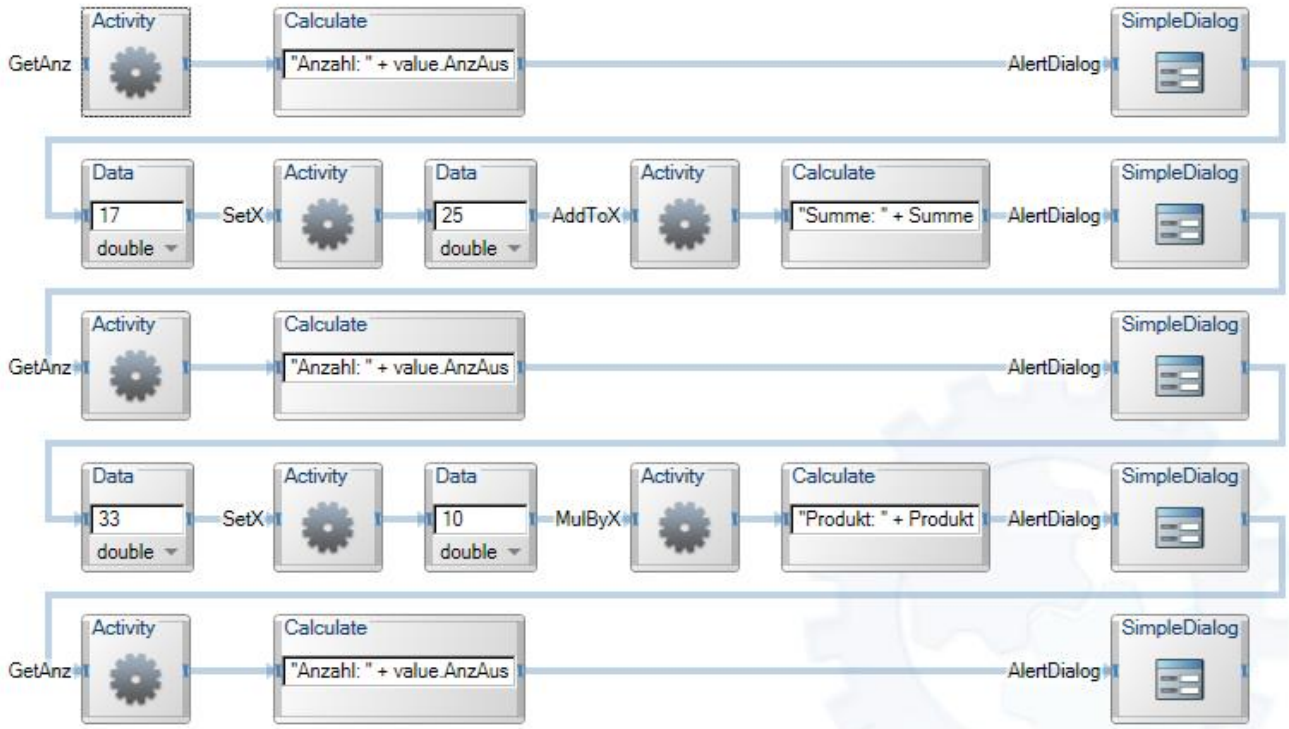
Das Diagramm der Aktion `AddToY` sollte etwa wie folgt definert werden:



Als Eingabe erwartet diese Aktion einen Verbund namens `value` mit einer `double`-Komponente namens `Y`. Der `Variable`-Block liefert einen Verbund namens `value` mit einer `double`-Komponente namens `X`. Der `Join`-Block fasst diese beiden Verbunde zu einem Verbund namens `value` mit zwei Komponenten namens `value.VarX` und `value.VarY` zusammen. Im `Calculate`-Block

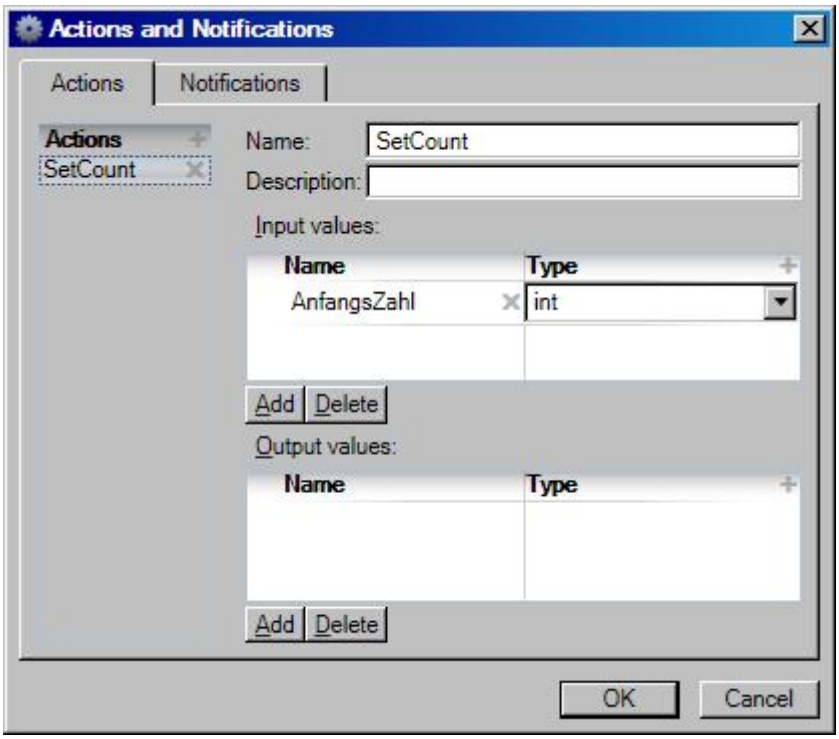
werden die beiden `double`-Werte `value.VarX.X` und `value.VarY.Y` addiert (wobei die zu addierenden Komponenten mit ihren kürzeren Namen `VarX.X` und `VarY.Y` bezeichnet werden).

Das Hauptdiagramm des Programms `B10ActivityD` testet den so definiert Activity-Block, indem es dreimal das Ergebnis der Aktion `GetAnzahl` ausgibt und zwischendurch die Aktionen `SetX` (zweimal), `AddToX` und `MulByX` (je einmal) aufruft und ihre Ergebnisse ausgibt:



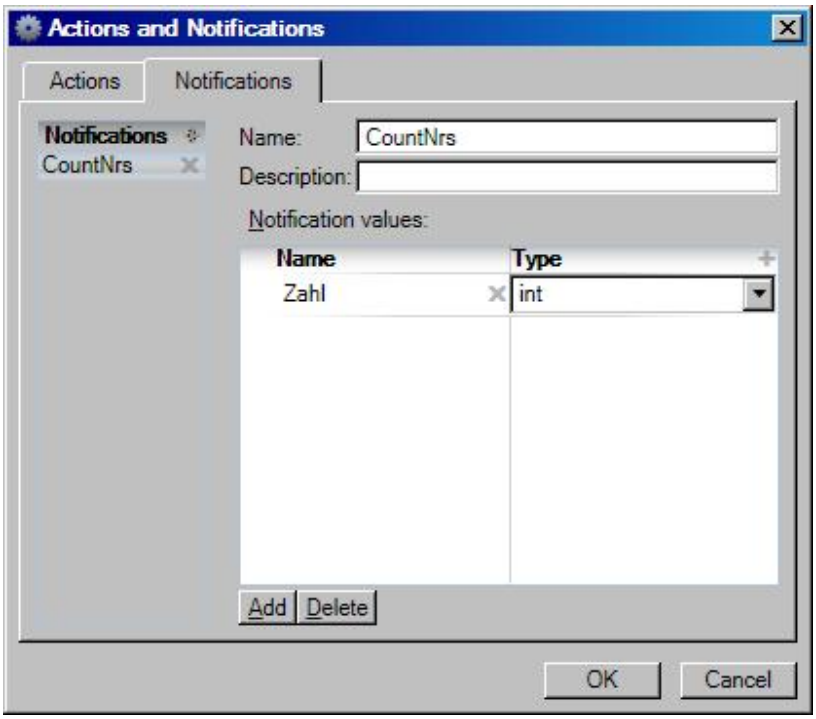
Anmerkung: In der Aktion `AddToX` passieren zwei Dinge *nebenläufig* zueinander: Die Berechnung des Ergebnisses und die Erhöhung der Variablen `Anzahl`. Das ist unsicher (jemand könnte, nachdem er ein `AddToX`-Ergebnis erhalten hat, aber bevor die Variable `Anzahl` erhöht wurde, die Aktion `GetAnzahl` aufrufen und würde ein falsches Ergebnis bekommen). Leider konnte ich für dieses Problem keine sichere und einigermaßen elegante Lösung finden (nur eine, bei der man vor lauter `Join`- und `Calculate`-Blöcken den simplen Algorithmus „erhöhe Anzahl um 1“ kaum noch erkennen kann).

18. Ein Activity-Objekt mit einem Ereignis-Ausgang



Die Activity-Objekte in den vorigen Beispielen hatten jeweils einen *Ergebnis*-Ausgang, über den die Aktionen des Objekts ihre Ergebnisse lieferten. In diesem Abschnitt soll ein Activity-Objekt mit einem *Ereignis*-Ausgang konstruiert werden. Indem man eine Aktion `SetCount` aufruft, soll man dem Objekt eine Ganzzahl `AnfangsZahl` schicken können (z.B. 5). Daraufhin sollen zum Ereignisausgang des Objekts der Reihe nach alle `int`-Werte von der `AnfangsZahl` bis `0` (im Beispiel: 5 4 3 2 1 0) einzeln ausgegeben werden (wie bei einem Countdown).

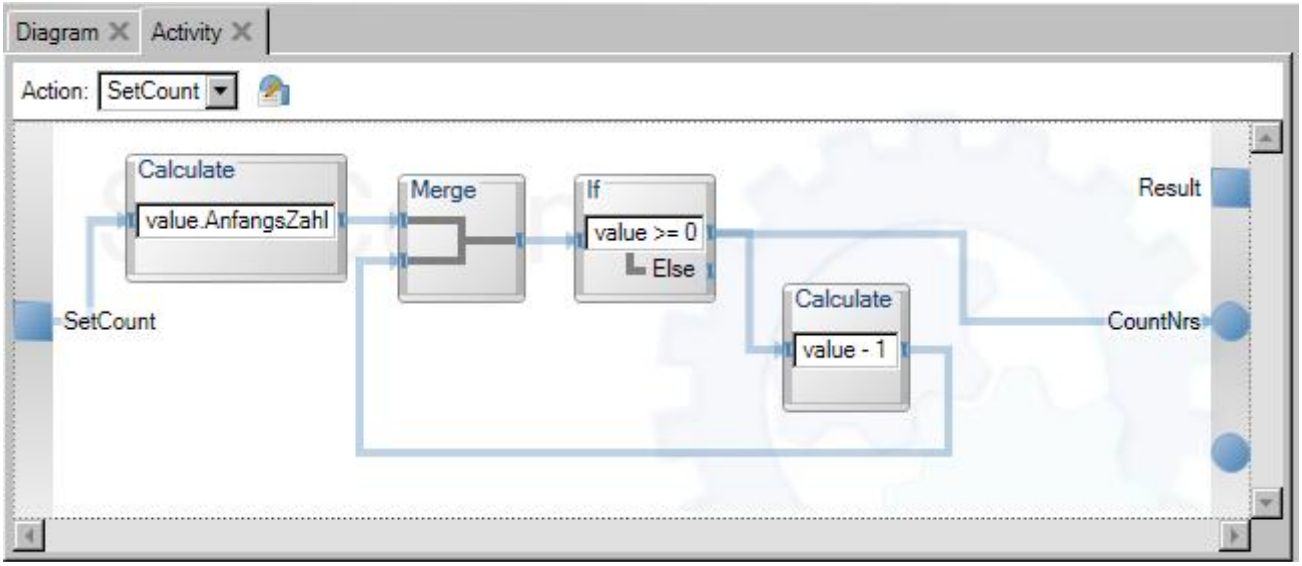
Wir beginnen wieder mit der Erzeugung eines Activity-Objekts und öffnen es durch einen Doppelklick auf seinen Block. Durch einen Klick auf das Welkugel-mit-Bleistift-Symbol öffnen wir das `Actions and Notifications`-Fenster. Darin legen wir die Eigenschaften einer Aktion `SetCount` fest wie in obiger Abbildung.



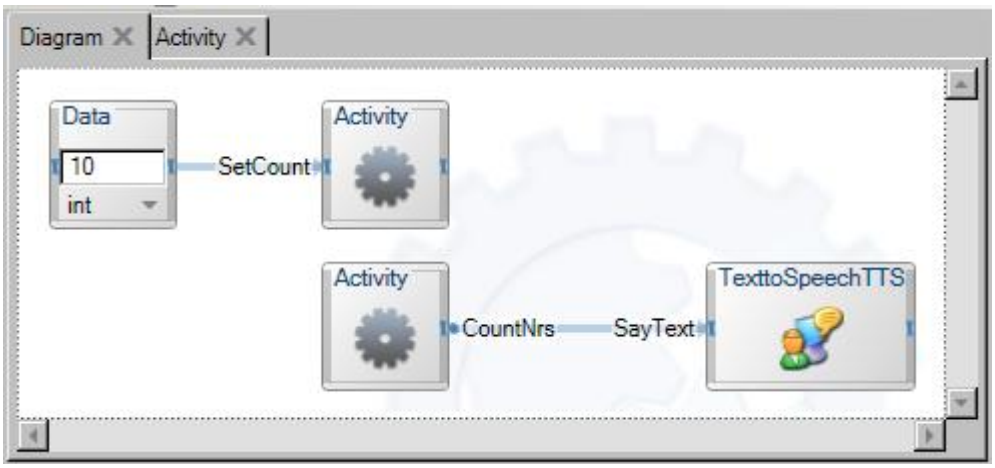
Dann klicken wir (im `Actions and Notifications`-Fenster) auf den Reiter `Notifications` und legen die Eigenschaften eines *Ereignisses* fest: Es soll `CountNrs` heissen und einen Ereignisparameter namens `Zahl` vom Typ `int` haben.

Ein Ereignis (engl. a notification) hat keine Eingangsparameter, nur Ereignisparameter (eng. Notification values). Die *Ereignisparameter* eines Ereignisses entsprechen weitgehend den *Ausgangsparametern* einer Aktion.

Im Innern des Activity-Objekts definieren wir die Aktion `SetCount`. Sie soll eine Schleife enthalten, die von mit ihrem Eingangsparameter `AnfangsZahl` beginnend bis `0` runterzählt und jede Zahl zum Ereignisausgang `CountNrs` ausgibt, etwa so wie in der folgenden Abbildung:



Als Eingabe erwartet die Aktion `SetCount` einen Verbund namens `value` mit einer `int`-Komponenten namens `AnfangsZahl`. Im ersten Calculate-Block kann man anstelle von `value.AnfangsZahl` auch kürzer `AnfangsZahl` schreiben.



Im Hauptdiagramm unseres Programms benötigen wir *zwei* Exemplare des Activity-Blocks, müssen den Block also einmal kopieren (mit `Strg-C` und `Strg-V`). Mit dem einen Block rufen wir die Aktion `SetCount` auf, von dem anderen benutzen wir den Ereignisausgang (das ist der mit dem blauen Punkt), um dort die einzelnen `CountNrs` in Empfang zu nehmen und aussprechen zu lassen.

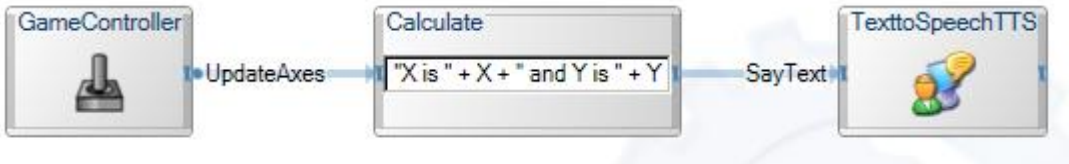
Das Besondere an einem Objekt mit einem Ereignisausgang ist, dass man mit *einer* Eingabe im Prinzip beliebig viele Ausgaben „anstoßen“ kann.

19. Daten von einem Joystick einlesen

Mit einem Objekt der Klasse `Game Controller` (im Fenster Services) kann man Daten von einem *Joystick* einlesen. Wenn Sie keinen Joystick besitzen, dann sollten Sie sich einen von Ihren Kindern oder von den Kindern Ihrer Nachbarn leihen. Und lassen Sie sich gleich zeigen, wie man ihn an einen PC anschließt. Ältere Modelle werden über einen 25-poligen `IEEE 1284`-Stecker oder einen 15-poligen PC `Game Port`-Stecker oder einen `DB 9`-Stecker (mit 9 Polen) angeschlossen, neuere über einen USB-Stecker. Es gibt Adapter-Kabel, die auf der einen Seite einen der älteren Anschlüsse (z.B. einen `Game Port`-Anschluss) und auf der anderen Seite einen USB-Stecker haben. Aber Vorsicht: Die funktionieren offenbar nur für analoge, aber nicht für digitale Joysticks.

Ob ein Joystick an ihrem Windows-PC angschlossen ist, können Sie unter `Systemsteurung`, `Gamecontroller` überprüfen. In diesem Fenster können Sie auch versuchen, bestimmte Anschlussprobleme zu beheben. Wenn ein Joystick korrekt an Ihren PC angschlossen ist, sollte er vom Robotics Studio automatisch erkannt und benutzt werden.

Beispiel-01: Auf Bewegungen des Joysticks in X- und Y-Richtung reagieren (das Beispielprogramm `B11GameControllerA`)



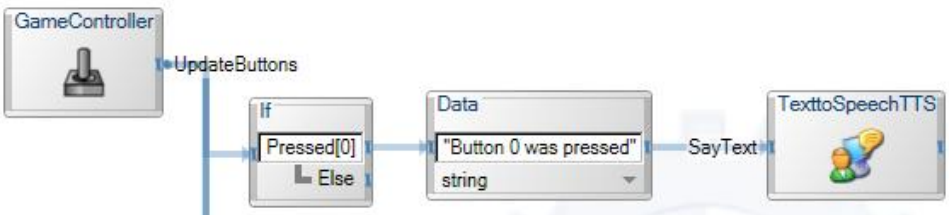
Jedesmal, wenn man den Joystick in Links/Rechts-Richtung (X-Richtung) oder in Vor/Zurück-Richtung (Y-Richtung) bewegt, schickt die Aktion `UpdateAxes` über ihren Ereignisausgang (der mit dem blauen Punkt) einen Verbund namens `value`, der unter anderem zwei `int`-Komponenten namens `X` und `Y` mit Werten zwischen `-1000` und `+1000` enthält.

Billige digitale Joysticks unterscheiden in `X`-Richtung (und ebenso in `Y`-Richtung) nur die drei Werte `-1000`, `0` und `+1000`. Analoge oder teurere digitale Joysticks unterscheiden mehr Werte, einige sogar alle `int`-Werte zwischen `-1000` und `+1000`.

`Warnung`: Wenn man einen guten Joystick z.B. bis zum Anschlag nach Vorn bewegt, schickt er in ziemlich kurzen zeitlichen Abständen Informationen über seine momentane Position. Der `Text to Speech (TTS)`-Block im Beispiel-01 braucht dann evtl. ziemlich lange, bis er all die vielen „Positionsberichte“ ausgesprochen hat. Ein billiger digitaler Joystick meldet dagegen nur die „Extrempositionen“, aber keine Zwischenstände und die Sprachausgabe hat dann weniger zu sagen.

Im folgenden Beispielprogramm passiert etwas, wenn man auf einen der *Knöpfe* an einem Joystick drückt.

Beispiel-02: Wurde der Knopf mit der Nummer 0 gedrückt? (Beispielprogramm `B11GameControllerB`, hier nur teilweise wiedergegeben)



Wenn man auf irgendeinen Knopf des Joysticks drückt, schickt die Aktion `UpdateButtons` über ihren Ereignisausgang (der mit dem blauen Punkt) einen Verbund namens `value`, der unter anderem eine Komponente namens `Pressed` vom Typ `List of bool` enthält. Die Komponenten der Liste sind (wie ind C/C++/Java/C# üblich) mit 0 beginnend durchnummeriert und die einzelnen Komponenten kann man mit Ausdrücken wie `Pressed[0]`, `Pressed[1]`, ... bezeichnen (wie im `If`-Block des Beispiels). Die Komponenten aller *gedrückten* Knöpfe haben den Wert `true`, die anderen Komponenten den Wert `false`.

Die Komponente `value.Pressed.Capacity` vom Typ `int` enthält die Länge der Liste `Pressed` (d.h. die Anzahl ihrer Komponenten). Das ist sehr praktisch, wenn einem die Nachbarskinder nicht verraten haben, wieviele Knöpfe ihr Joystick wirklich hat.

20. Ein virtuelles Robofahrzeug steuern

Das Robotics Studio enthält (im Fenster Services) zahlreiche Klassen, deren Objekte zum Steuern von *realen* Robotern (Lego-, Fischertechnik-, Roomba-Staubsauger- und anderen Robotern) geeignet sind. Für Menschen, die ihren Kindern immer noch keinen Lego- oder Fischertechnik-Baukasten und sich selbst noch keinen Roomba-Staubsauger geschenkt haben, gibt es im Robotics Studio aber auch eine Komponente namens *Visual Simulation Environment* (VSE), die einem auf dem Bildschirm z.B. eine Ebene mit ein paar Hindernissen und einem *virtuellen Robofahrzeug* darauf anzeigen kann. Dieses Robofahrzeug kann man mit einem VPL-Programm dazu veranlassen, sich zu bewegen, auf der Ebene herumzufahren und die Hindernisse zu verschieben oder umzuwerfen.

Warnung: Das Programm *Visual Simulation Environment* stellt ähnlich hohe Ansprüche an die verwendete Hardware wie verbreitete „Ballerspiele“ und funktioniert nur mit einer ziemlich schnellen Graphik-Karte. Ein üblicher „Büro-PC“ mit einem schnellen Pentium-Prozessor reicht *nicht* aus. Leihen Sie sich notfalls den PC Ihrer Kinder oder fragen Sie die Kinder ihrer Nachbarn (deren PC hat mit hoher Wahrscheinlichkeit eine ausreichend schnelle Graphik-Karte).

Allgemein macht die VSE-Komponente es möglich, Steuerungsprogramme erstmal an einem virtuellen Roboter zu testen, ehe man einen realen Roboter damit steuert. Die VSE-Komponente kann also auch für Besitzer von Lego-, Fischertechnik-, Roomba-Staubsauer- und anderen Roboter sehr nützlich sein.

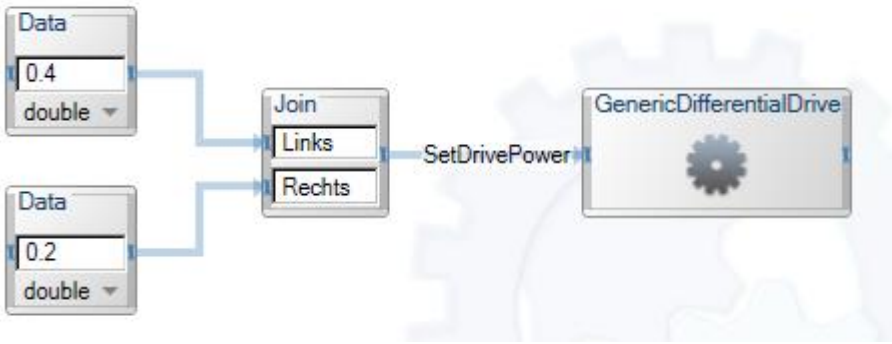
Das fertig mitgelieferte virtuelle Robofahrzeug hat drei Räder, von denen zwei auf einer Achse stecken, aber getrennt voneinander durch je einen (Elektro-) Motor angetrieben werden. Das kleinere dritte Rad rollt nur passiv mit. Wenn die beiden Motoren mit gleicher Stärke arbeiten, fährt das Fahrzeug geradeaus. Um auf der Stelle z.B. links herum zu drehen, kann man den linken Motor rückwärts und den rechten vorwärts drehen lassen (aber nicht zu stark, sonst fällt das Fahrzeug um).

Dieses virtuelle Robofahrzeug kann man mit ein paar Klicks (siehe unten) mit einem `Generic Differential Drive`-Objekt (in einem VPL-Programm) verbinden. Ein solches Objekt enthält unter anderem eine Aktion namens `SetDrivePower`. Die erwartet als Eingabe einen Verbund mit zwei `double`-Komponenten namens `LeftWheelPower` und `RightWheelPower`, von denen die erste zum linken und die zweite zum rechten Motor geschickt wird (als Steuerungssignale). Wenn man geradeaus fahren will, kann man den beiden Motoren ohne weiteres einen Wert aus dem Bereich zwischen `-10.0` und `+10.0` schicken. Beim Drehen auf der Stelle sollte man vorsichtiger sein und dem einen Motor z.B. den Wert `+0.5` und dem anderen `-0.5` schicken.

Achtung: Wenn man den Motoren durch einen Aufruf der Aktion `SetDrivePower` einmal irgendwelche Werte geschickt hat, dann bleiben diese Werte in Kraft, bis man neue Werte schickt (indem man die Aktion `SetDrivePower` erneut aufruft).

Um eine sehr gute Steuerung für das Robofahrzeug zu realisieren, braucht man einen guten Joystick (der viele verschiedene Werte unterscheiden kann). Wenn man keinen Joystick hat, kann man trotzdem eine simple Steuerung programmieren, die ein `Direction Dialog`-Objekt als Eingabegerät verwendet. Eine solche Steuerung wird in den folgenden Beispielen in drei Stufen entwickelt.

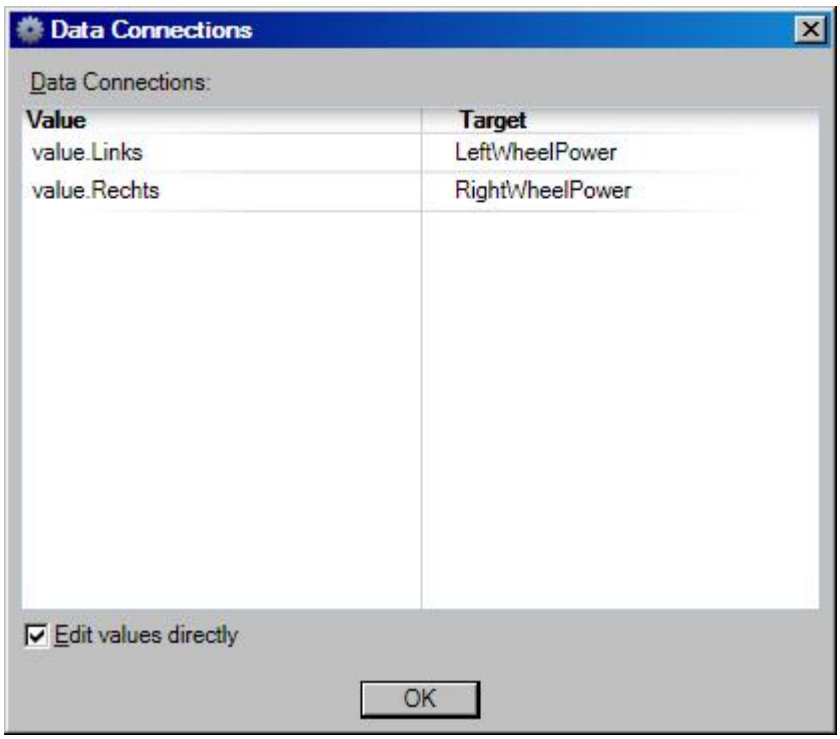
Beispiel-01: Eine „konstante Steuerung“ für das virtuelle Robofahrzeug (`B12VirtualRobotA`)



Dieses Programm schickt ein einziges Mal die `double`-Werte `0.4` und `0.2` zum linken bzw. rechten Motor des Robofahrzeugs. Das Fahrzeug sollte daraufhin „ewig“ (und relativ langsam) rechts herum im Kreis fahren.

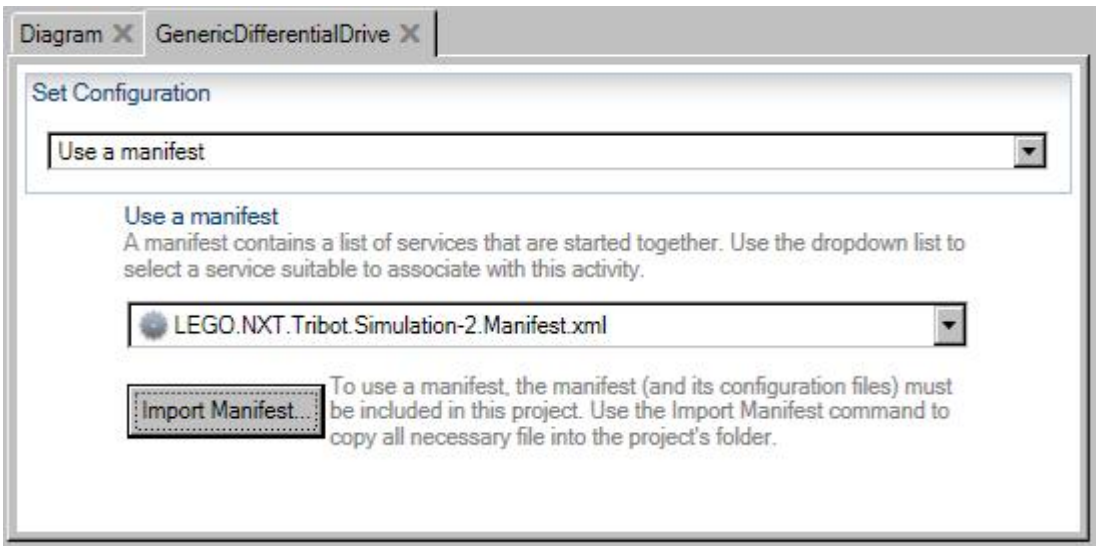
Besonders wichtig sind die Einstellungen im `Data Connections`-Fenster des Pfeils, der zur Aktion `SetDrivePower` führt.

Zur Erinnerung: Das `Data Connections`-Fenster kann man im *Kontextmenü* eines Pfeils öffnen. Das Kontextmenü kann man öffnen, indem man den Pfeil mit der rechten Maustaste anklickt.



Im Beispiel-01 sollte das `Data Connections`-Fenster des rechten Pfeils (vom `Join`- zum `Generic Differential Drive`-Block) aussehen wie nebenstehend abgebildet: Der `Join`-Block liefert einen Verbund namens `value`. Der Aktionsparameter `LeftWheelPower` (bzw. `RightWheelPower`) soll mit der Verbundkomponenten `value.Links` (bzw. `value.Rechts`) initialisiert werden.

Jetzt kommt noch ein entscheidender Schritt der bewirkt, dass bei der Ausführung des VPL-Programms nicht nach einem realen Roboter gesucht, sondern mit dem *Visual Simulation Environment* eine Ebene mit einem virtuellen Robofahrzeug darauf auf dem Bildschirm dargestellt und das `Generic Differential Drive`-Objekt des Programms mit diesem virtuellen Robofahrzeug verbunden wird. Dieser etwas komplizierte Vorgang wird in einem sogenannten *Manifest* beschrieben, welches mit dem Robotics Studio mitgeliefert wird und jetzt nur noch aktiviert werden muss.



Öffnen Sie den `Generic Differential Drive`-Block mit einem Doppelklick. Wählen Sie im sich öffnenden Fenster unter `Set Configuration` die Alternative `Use Manifest`. Klicken Sie dann weiter unten auf den Knopf `Import Manifest` und wählen Sie in dem sich öffnenden `Import Manifest`-Fenster das Manifest **`LEGO.NXT.Tribot.Simulation.manifest.xml`**. Nach dieser Wahl sollte das `Generic Differential Drive`-Fenster wie oben abgebildet aussehen. Lassen Sie sich möglichst wenig davon verwirren, dass in diesem Fenster jetzt ein etwas anderer Manifest-Name steht als *der*, den Sie ausgewählt haben (statt `simulation.manifest` heisst es jetzt `Simulation-2.Manifest`), Hauptsache der Name beginnt mit `LEGO.NXT.Tribot.Simulation` … .

Anmerkung: Bisher wurde nur *ein* Manifest erwähnt. Tatsächlich können Sie eines der folgenden drei Manifeste wählen (nicht nur das erste):

`LEGO.NXT.Tribot.Simulation.manifest.xml`
`MobileRobots.P3DX.Drive.manifest.xml`
`SimulationTutorial3.manifest.xml`

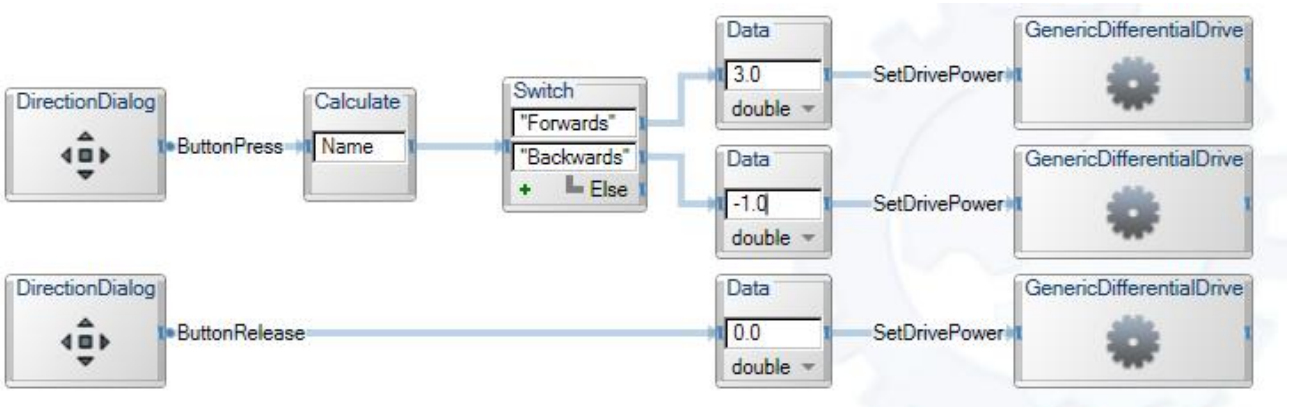
Damit ist unser erstes Steuerungsprogramm für das virtuelle Robofahrzeug fertig. Starten Sie es wie üblich (z.B. mit F5) und seien Sie diesmal besonders geduldig. Nach einiger Zeit sollte sich ein großes buntes Fenster öffnen, auf dem eine sich bis zum Horizont erstreckende Ebene (mit grauem Teppichboden beklebt?) zu sehen ist. Auf der Ebene befinden sich ein paar geometrische Gebilde und ein kleines Robofahrzeug, welches immer rechts herum im Kreis fährt. Das genaue Aussehen des Robofahrzeugs und die Anzahl und Art der geometrischen Gebilde hängt (ein bisschen) vom verwendeten Manifest ab.

Falls das Programm nicht funktioniert, sollten Sie alle Pfeile noch einmal entsprechend der Regel-01 (im Abschnitt 2) überprüfen und eventuell korrigieren. Falls sich zwar ein VSE-Fenster (mit dem Titel Microsoft Visual Simulation Environment) öffnet, aber lange Zeit schwarz bleibt, sollten Sie das Importieren des Manifests noch einmal wiederholen und evtl. ein anderes Manifest verwenden. Wenn sich dann (bei einem erneuten Start des Programms) wieder das VSE-Fenster öffnet aber immer noch schwarz bleibt, ist wahrscheinlich Ihre Graphik-Karte zu langsam.

Das „konstante“ Steuerungsprogramm im Beispiel-01 ist noch sehr primitiv. Mit ihm kann man aber üben, ein Manifest auszuwählen und sicherstellen, dass das VSE-Programm auf der verwendeten Plattform wirklich funktioniert. Wenn das der Fall ist, kann man ein komplizierteres Steuerungsprogramm entwickeln, z.B. das im folgenden Beispiel beschriebene.

Beispiel-02: Eine Steuerung zum Vorwärts- und Rückwärtsfahren (`B12VirtualRobotB`)

Geradeaus zu fahren ist einfacher, als um eine Kurve zu steuern. Mit der folgenden Steuerung kann man das Robofahrzeug in Vorwärts- und Rückwärtsbewegung versetzen und (ganz wichtig) wieder anhalten. Kurven fahren kann mit dieser Steuerung aber noch nicht.



Zur Erinnerung: Die beiden `Direction Dialog`-Blöcke repräsentieren ein und dasselbe Objekt, und ebenso repräsentieren die drei `Generic Differential Drive`-Objekte dasselbe Objekt.

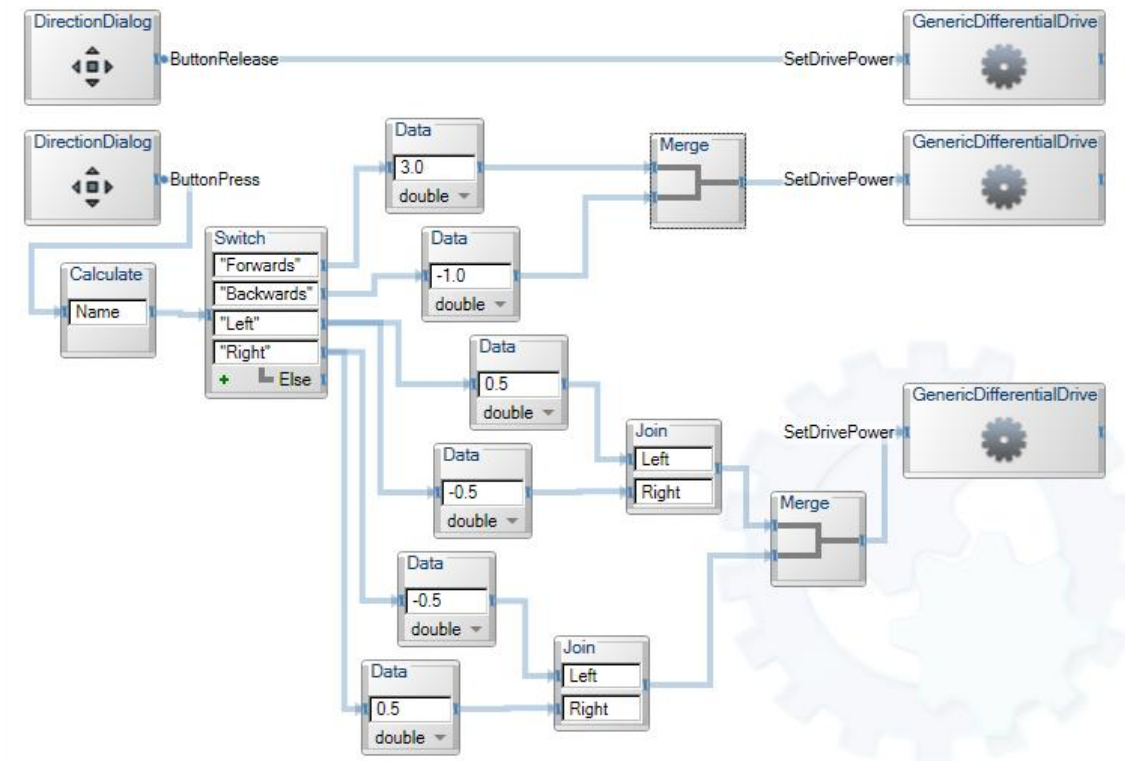
Wenn man dieses Programm startet, sollte nicht nur „das große bunte“ VSE-Fenster mit dem Robofahrzeug darin aufgehen, sondern auch ein kleines `Direction Dialog`-Fenster mit fünf Knöpfen darin (wie im Beispiel-03 im Abschnitt 8 abgebildet). Rechnen Sie damit, dass die beiden Fenster *nicht von allein sichtbar werden*, weil sie sich gegenseitig verdecken oder von anderen Fenstern verdeckt werden. In diesem Falle erkennt man ihre Anwesenheit an ihren Symbolen in der Taskleiste (ganz unten auf dem Bildschirm).

Wenn man im `Direction Dialog`-Fenster den Vorwärts-Knopf *runterdrückt* (`ButtonPress`), soll der `double`-Wert `3.0` zu *beiden* Motoren des Robofahrzeuges geschickt werden (entsprechende Einstellungen muss man im `Data Connections`-Fenster des Ausgabepfeils des obersten `Data`-Block vornehmen). Wenn man den Rückwärts-Knopf runterdrückt, soll das Fahrzeug mit der Stärke `-1` (d.h. rückwärts) fahren. Sobald man den Mausknopf dann wieder *loslässt* (`ButtonRelease`) soll `0.0` an beide Motoren geschickt werden, so dass das Fahrzeug anhält.

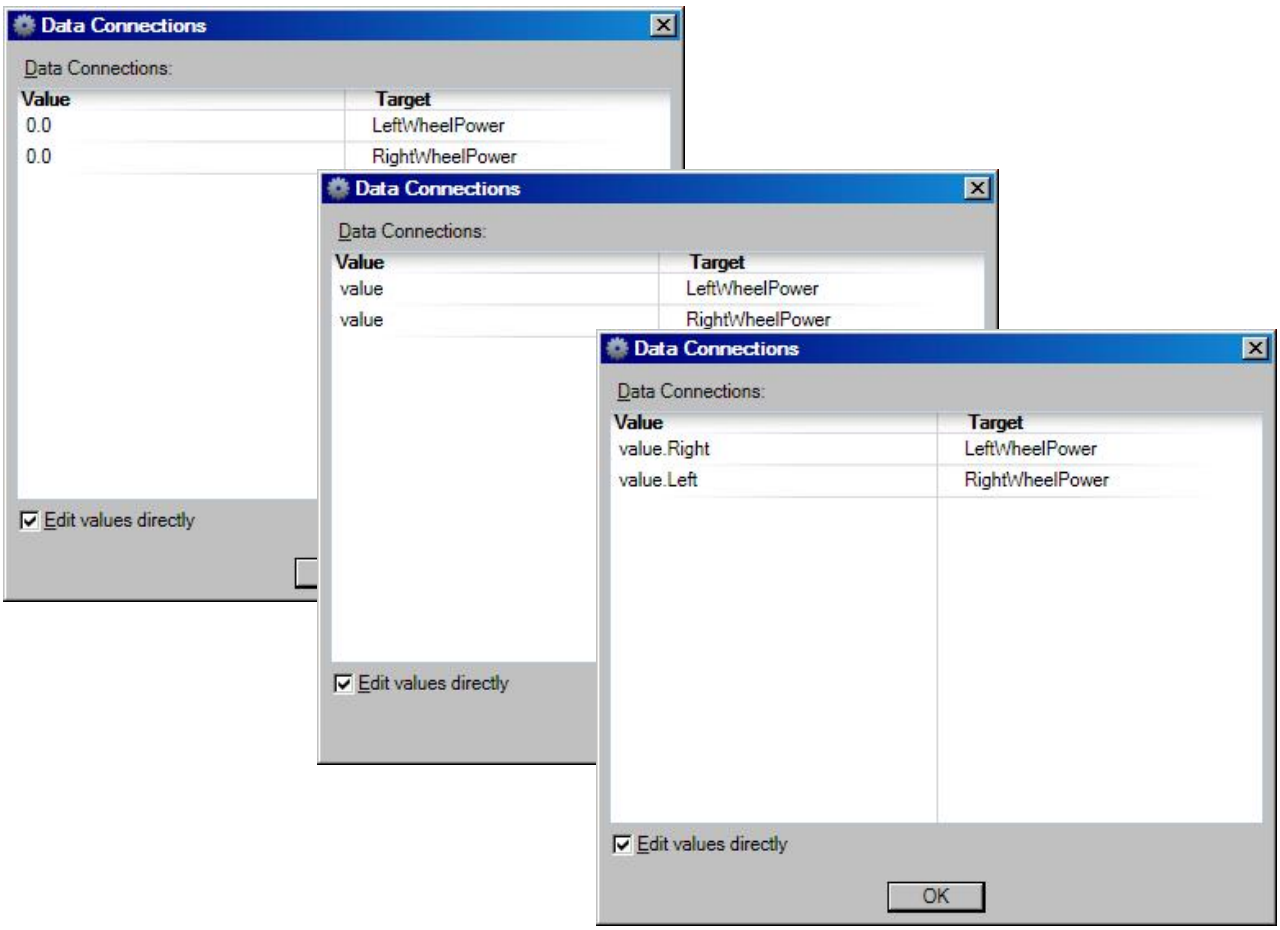
Wenn das Programm funktioniert, kann man die Zahlen `3.0` und `-1.0` in den `Data`-Blöcken variieren und ausprobieren, welchen Effekt größere und kleiner Zahlen haben.

Anmerkung: Den `Data`-Block mit `0.0` darin kann man einsparen und den `Direction Dialog`-Block direkt mit dem `Generic Differential Drive`-Block verbinden. Im `Data Connections`-Fenster des Verbindungspfeils kann man dann den Wert `0.0` (für die beiden Parameter `LeftWheelPower` und `RightWheelPower`) direkt eintragen (nachdem man dort `Edit values directly` gewählt hat). Diese kleine Vereinfachung wird im folgenden Beispiel verwendet.

Beispiel-03: Eine Steuerung mit der man auch Drehen kann



Rechts erkennt man drei `Generic Differential Drive`-Blöcke. Die `Data Connections`-Fenster ihrer Eingabepfeile sollten wie folgt aussehen:



In dieser Steuerung führen insgesamt *fünf Pfeile* zur `SetDrivePower`-Aktion eines `Generic Differential Drive`-Objekts, wobei zweimal je 2 Pfeile mit einem `Merge`-Block zusammengefasst werden.

Anstelle der beiden `Merge`-Blöcke hätte man auch zwei zusätzliche `Generic Differential Drive`-Blöcke verwenden können und jeden der fünf Pfeile zu einem „eigenen“ solchen Block führen können. Es ist wohl Geschmacksache, welche Lösung man bevorzugt.

Andersherum kann man aber nur solche Pfeile mit einem Merge-Block zusammenfassen, die übereinstimmende `Data Connections` haben. Deshalb ist es in diesem Beispiel nicht möglich, mit nur *einem* `Generic Differential Drive`-Block auszukommen.