ISA Dialog Manager

Benutzerdefinierte Attribute und Methoden

A.06.03.b

Dieses Handbuch erläutert die Verwendung von benutzerdefinierten Attributen und Methoden.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

<Kleinbuchstabe> ::= a | b | c | … x | y | z

<Großbuchstabe> ::= A | B | C | … X | Y | Z

Inhalt

1 Einführung

In diesem Handbuch finden Sie eine Übersicht über die Definition der

- benutzerdefinierten Attribute
- benutzerdefinierten Methoden

und deren Anwendung im Dialog Manager.

Benutzerdefinierte Attribute

Benutzerdefinierte Attribute dienen der Definition applikationspezifischer Informationen. Damit können Dialog und Anwendung vollständig voneinander getrennt werden. D.h., dass die Anwendung keine Information über den Dialog mehr benötigt.

Benutzerdefinierte Methoden

Benutzerdefinierte Methoden dienen der Definition applikationspezifischer Informationen. Mit Hilfe dieser Methoden kann ein Dialog "objektorientiert" programmiert werden.

2 Benutzerdefinierte Attribute

Mit Hilfe der im folgenden beschriebenen Möglichkeiten, können Dialog und Anwendung vollständig voneinander getrennt werden. D.h., dass die Anwendung keine Information über den Dialog mehr benötigt (siehe auch Handbuch "Programmiertechniken" -> "Verwendung von benutzerdefinierten Attributen und objekt-orientierte Programmierung").

Die existierende Grenze von 16 Parametern pro Anwendung besteht weiterhin. Jedoch kann nun jeder Parameter für sich eine Struktur sein.

Zum einen kann für jedes Objekt das vom Dialog Manager bereitgestellte Attribut *.userdata* verwendet werden, um applikationspezifische Informationen zu definieren. Dieses Attribut ist für "normale" Objekte (z.B. statischer Text) je einmal und für Objekte mit Listencharakter (Listbox, Poptext, Tablefield) für jeden Eintrag vorhanden.

Zum anderen werden **benutzerdefinierte Attribute** zur Verfügung gestellt. Diese werden identisch behandelt wie interne DM-Attribute, d.h. sie können ebenfalls von einer Vorlage oder einem Default geerbt werden.

Für die Definition dieser Attribute gibt es folgende prinzipiellen Möglichkeiten:

- skalare Attribute
- indizierte Attribute
- Attribute als Verweis auf andere Attribute

2.1 Definition benutzerdefinierter Attribute

Syntax

Skalare Attribute

```
<Datentyp> <Attributname> { := <Wert> } ;
```

Indizierte Attribute

```
<Datentyp> <Attributname> [ <integer-Wert> ] ;
```

shadow-Attribute

```
<Datentyp> <Attributname> shadows { instance }
    <Objekt> <Attribut> { [ <Index> ] } ;
```

<Datentyp>
Hier kann einer der im IDM verfügbaren Datentypen eingesetzt werden, z.B. *integer*, *object*, *string*, *boolean*…

<Attributname>
Hier kann ein vom Benutzer frei gewählter Name für das Attribut eingesetzt werden. Dieser Name muss der DM-Namenskonvention für Identifikatoren entsprechen (Großbuchstaben am Anfang usw.).

Beispiel

» Deklaration ohne Punkt:

```
pushbutton MyButton
{
  integer Position;
}
```

» Referenzierung mit Punkt:

```
MyButton.Position := 1;
```

<Wert>
Wert, der dem vorne beschriebenen Datentyp entspricht.

<integer-Wert>
Zahl, die die Größe des Feldes definiert.

Mit dem Keyword **shadows** kann ein uneigentliches Attribut definiert werden. Dieses merkt sich seinen Wert nicht selbst, sondern greift (lesend und schreibend) auf das nachfolgend angegebene Objekt-Attribut zu. Wenn zusätzlich das Schlüsselwort **instance** verwendet wird, bedeutet das, dass bei der Instanziierung eines Modells der Verweis auf die neu generierte Instanz verändert werden soll und nicht beim Modell bleiben soll.

<Objekt>
Objekt, auf das das Attribut zugreift.

<Attribut>
Attribut des Objektes, auf das das Attribut zugreift.

<Index>
Angabe von Zeilen bzw. Spalten für Listenobjekte bzw. deren Attribute.

Beispiele

» Definition eines ***pushbutton***-Modells mit einem benutzerdefinierten Attribut „Position“, das an alle Instanzen mit dem Initialwert *17* vererbt wird.

```
model pushbutton MyModel
{
  integer Position := 17;
}
```

» Definition eines ***pushbutton***-Modells mit einem indizierten, benutzerdefinierten Attribut (Feld) „Active“, das *5* Elemente vom Datentyp *boolean* aufnehmen kann.

```
model pushbutton MyModel
{
  boolean Active[5];
}
```

» Definition eines ***pushbutton***-Modells, dessen benutzerdefiniertes *string*-Attribut „CurrentValue“ seinen Wert aus dem Inhalt des ***edittextes*** „Et“ bezieht.

```
model pushbutton MyModel
{
  string CurrentValue shadows Et.content;
}
```

» Definition eines ***window***-Modells mit einem benutzerdefinierten Attribut „CurrentValue“, welches sich bei der Instanziierung des Modells auf die Instanz des ***edittextes*** „Et“ bezieht.

```
model window MyModel
{
  string CurrentValue shadows instance Et.content;
  child edittext Et
  {
  }
}
```

2.2 Zugriff auf benutzerdefinierte Attribute

Nach der Definition eines benutzerdefinierten Attributs kann dieses wie jedes andere Attribut in den Regeln oder bei der Objektdefinition angesprochen werden.

Beispiel

```
MyModel.Position := 28;
```

oder

```
MyModel.Active[0] := true;
```

oder

```
MyModel.Currentvalue := "NewContent"
```

Diese Zuweisung ändert in Wirklichkeit den Inhalt des shadow-Objektes (vgl. o. editierbarer Text).

2.3 Erweiterte Pfadreferenzierung bei Attributen

Pfade im statischen Teil von IDM-Dateien erlaubten bisher nur den Zugriff auf ein Objekt bzw. exportiertes Objekt. Nun wird zusätzlich die Werteauflösung beim Zugriff auf Attribute sowie Variablen unterstützt, sodass statische Pfade sich ähnlich zu Pfaden in Regeln verhalten, d.h. dass Pfadteile, die kein Kindobjekt, sondern eine Variable oder Attribut bezeichnen, deren Wert für die weitere

Pfadauflösung verwenden. Natürlich gilt bei weiteren Pfadauflösungen die Zugriffsbeschränkung auf exportierte Objekte.

Grundsätzlich unterstützen nur statische Setzungen an benutzerdefinierte Attribute und Variablen die vollständige Werteauflösung. Vordefinierte Attribute benötigen weiterhin zwingend im letzten Pfadteil die Angabe eines **Kind**-Objektes, bei den vorangegangenen Pfadteilen wird eine Werteauflösung durchgeführt.

Um die Kompatibilität mit bisherigen IDM-Anwendungen zu gewährleisten, wird bei einer Variablenreferenzierungen als letztem Pfadteil nur der Wert zurückgeliefert wenn die Ziel-Variable bzw –Attribut nicht den Typ anyvalue oder object besitzt.

Es ist weiterhin zu Beachten das die Pfadauflösung vor der Initialisierung der Objekte (Aufruf der :init-Regeln) erfolgt, die Attribut- bzw. Variablenwerte also vorher gesetzt sein müssen. Entsprechend wird während bei Attribut-Zugriffen auch keine vordefinierten :get-Methoden für die Werteauflösung aufgerufen.

Wichtig

Trotz der damit erreichten Angleichung zwischen „statischen“ Pfaden und Pfaden im dynamischen Regelteil verhalten diese sich gerade im Umgang mit globalen Variablen anders. Bei Pfaden im Regelteil hat die Werteauflösung Vorrang, die Variablenreferenzierung ist durch Benutzung von *.self* möglich. Dies ist bei statischen Initialisierungen anders, hier hat quasi die Referenz Vorrang vor dem Wert entsprechend dem Ziel-Datentyp.

Von der Verwendung von statische Initialisierungen die mittels Werteauflösung funktionieren wird bei Benutzung des IDM Editors abgeraten, da beim Speichern diese Pfade nicht wieder herausgeschrieben/restauriert werden können (analog zu string-Attributen die mit Text-Resourcen initialisiert werden).

Beispiel

Dieses Beispiel veranschaulicht die möglichen Verwendungsarten und beinhaltet auch Fälle die einen Pfad-Auflösungsfehler aufzeigen.

Datei: bsp.mod

```
module M
export record ERec {
  object Ref := ERec.Sub2;
  export child record Sub1 {
    integer X := 2001;
  }
  child record Sub2 {
    integer Y := 1998;
    child record Sub3 {
      integer Z := 2010;
    }
  }
}
```

Datei: bsp.dlg

```
dialog D

import I1 "~:bsp.if";

variable integer V1 := 1918;
variable object  V2 := Defs;

text Tx "Hallo Welt";

// Definition von Attribute für eine statisch Initialisierung
record Defs {
  integer D1 := 640;
  integer D2;
  string D3 := "Guten Tag!";
  integer D4[4];
  .D4[2] := 666;
  :init() {
    this:super();
    this.D2 := 2;
  }
  record SubDefs {
  }
}
record Rec {
  integer A := V1;       // Pfad-Wertauflösung bei einer Variablen
  integer B := Defs.D1;  // Pfad-Wertauflösung bei einem Attribut
  object  C := D.V1;     // Pfad-Auflösung zu einer ID/object-Ref.
//  integer D := Defs.D2;  // FEHLER: Wert zum Auflösezeitpunkt Ungesetzt!
  string  E := D.V2.D3;  // Mehrfache Werteauflösung über object-Ref.
  integer F := ERec.Sub1.X; // Auflösung über exportiertes Kindobjekt
//  integer G := ERec.Sub2.Y; // FEHLER: Sub2 nicht exportiert!
  integer H := ERec.Ref.Y;  // Umweg über object-Attribut geht
//  integer I := ERec.Ref.Sub3.Z;  // FEHLER: Sub3 nicht exportiert!
//  integer J := Rec.K; // FEHLER: Gegenseitige Pfade nicht auflösbar!
//  ingeger K := Rec.J; // FEHLER: Gegenseitige Pfade nicht auflösbar!
  integer L := Defs.D4:[2]; // Pfad-Auflösung auf einzelnes Feldelement
}
window Wi {
//  .title Defs.D3; // FEHLER: Pfad-Endstueck ist kein Objekt
//  .width Defs.D1; // FEHLER: Pfadauflösung nur vordef. object-Attributen
//  .userdata Defs.D1; // FEHLER: Pfad-Endstueck ist kein Objekt
  .userdata V2.SubDefs; // Pfad-Werteaufloesung am Anfang OK
  edittext Et {
    .content Tx;  // Wertauflösung für Textresourcen ging schon immer
  }
```

```
  on close {
    exit();
  }
}
```

2.4 Attribute der benutzerdefinierten Attribute

Die benutzerdefinierten Attribute haben ihrerseits Attribute, die man in den Regeln abfragen und ändern kann.

Die Abfrage des Attributes bzw. das Verändern des Attributes erfolgt nach folgendem Schemata:

```
<Objekt><.Zu änderndes Attribut>[<Benutzerdefiniertes Attribut>]
```

Beispiel

```
print Window.count[.FensterNummer];
```

2.4.1 .convformat

Identifikator:

.convformat

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	String
C-Definition:	AT_convformat,
C-Datentyp:	DT_string
COBOL-Definition:	AT-convformat
COBOL-Datentyp:	DT-string
Zugriff:	set, get

Mit Hilfe dieses Attributes kann man einem benutzerdefinierten Attribut einen beliebigen String zuweisen. Normalerweise wird dieses Attribut von der CICS-Schnittstelle benutzt, in dem Elementen in Records der Datentyp zugewiesen wird, der im CICS für das Record-Element gelten soll. Natürlich kann man sich hier auch andere Informationen hinterlegen, die man beim Zugriff auf das Record-Element beachten möchte.

Beispiel

```
record Rec1
{
  string Str1 convformat("PIC X(20)");
```

```
  integer Int1 convformat ("PIC 9(4) comp-5");
}
```

2.4.2 .count

Identifikator:

.count

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Integer
C-Definition:	AT_count,
C-Datentyp:	DT_integer
COBOL-Definition:	AT-count
COBOL-Datentyp:	DT-integer
Zugriff:	set, get

Mit Hilfe dieses Attributes kann die Größe eines Feldes eines benutzerdefinierten Attributes erfragt bzw. gesetzt werden.

Beispiel

```
record Rec1
{
  string Str1[10];
  integer Int1[10];
}
```

Dieses definiert einen Record mit zwei Elementen, die jeweils als Feld der Größe 10 definiert sind.

```
print Rec1.count[.Str1] ;    => 10

Rec1.count[.Str1] := 20;
print Rec1.count[.Str1] ;    => 20
```

2.4.3 .shadowattr

Identifikator:

.shadowattr

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Attribut
C-Definition:	AT_shadowattr,
C-Datentyp:	DT_attribute
COBOL-Definition:	AT-shadowattr
COBOL-Datentyp:	DT-attribute
Zugriff:	set, get

Mit Hilfe dieses Attributes wird definiert, auf welches Attribut dieses Attribut verweisen soll. Dieser Verweis kann dynamisch verändert werden, in dem man diesem Attribut ein neues Attribut zuweist.

Beispiel

```
record Rec1
{
  string Str1 shadows ET.content;
}
```

Dieses definiert einen Record mit einem Element, das als Verweis auf das Eingabefeld ET und auf das Attribut .content definiert ist.

```
print Rec1.shadowattr[.Str1] ;    => .content
```

2.4.4 .shadowindex

Identifikator:

.shadowindex

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Integer, Index
C-Definition:	AT_shadowindex,
C-Datentyp:	DT_integer, DT_index
COBOL-Definition:	AT-shadowindex
COBOL-Datentyp:	DT-integer, DT-index
Zugriff:	set, get

Mit Hilfe dieses Attributes wird definiert, auf das wievielte Element eines vektoriellen Attributes dieses Attribut verweisen soll. Dieser Verweis kann dynamisch verändert werden, in dem man diesem Attribut einen neuen Index (ein- oder zweidimensional, je nach Attribut) zuweist.

Beispiel

```
record Rec1
{
  string Str1 shadows Listbox.content[1];
  string Str2 shadows Table.content[1,2];
}
```

In diesem Fall verweist das Attribut Str1 auf den Inhalt der ersten Zeile in der Listbox, das Attribut Str2 verweist auf den Inhalt der Zelle in der Zeile 1, Spalte 2 in der Tabelle

2.4.5 .shadowinstance

Identifikator:

.shadowinstance

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Boolean
C-Definition:	AT_shadowinstance
C-Datentyp:	DT_boolean
COBOL-Definition:	AT-shadowinstance
COBOL-Datentyp:	DT-boolean
Zugriff:	set, get

Mit Hilfe dieses Attributes wird definiert, dass Verweise auf ein Attribut innerhalb desselben Modells bei dessen Instanziierung auf die Instanz umgehängt werden sollen.

Beispiel

Bei der Instanz eines hierarchischen Modells sollen die Attribute eines Records auf die Instanzobjekte gesetzt werden.

```
model window WModel
{
    child edittext ET1
    {    .ytop 1;
        .xleft 1;
        .format "NNNN";
```

```
    }
    child edittext ET2
    {
        .ytop 3;
        .xleft 1;
    }
    record WData
    {
        string [20]    Value1;
        string [20]    Value2;
    }
}
```

Das Generieren der Instanz und das Umsetzen der Record-Attribute erfolgt nun in den Regeln wie folgt:

```
rule void CreateInstance ( )
{
    variable object Obj;

    Obj:= create (WModel, WModel.dialog);
    if    (Obj <> null) then
        Obj.WData.shadowobject [.Value1]
            := Obj.ET1;
        Obj.WData.shadowattr [.Value1]
            := .content;
        OBj.WData.shadowobject [.Value2]
            ::= Obj.ET2;
        Obj.WData.shadowattr [.Value2]
            := .content;
    endif
}
```

Einfacher wäre das Ganze gewesen, wenn die Attribute des Records als Verweis auf die Instanz der Objekte definiert gewesen wäre. Dieses funktioniert aber nur, wenn die Objekte innerhalb desselben Modells definiert sind.

```
model window WModel
{
    child edittext ET1
    {    .ytop 1;
        .xleft 1;
        .format "NNNN";
    }
    child edittext ET2
    {
        .ytop 3;
        .xleft 1;
```

```
    }
    record WData
    {
        string [20]    Value1 shadows instance ET1.content;
        string [20]    Value2 shadows instance ET2.content;
    }
}
```

2.4.6 .shadowobject

Identifikator:

.shadowobject

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Objekt
C-Definition:	AT_shadowobject,
C-Datentyp:	DT_object
COBOL-Definition:	AT-shadowobject
COBOL-Datentyp:	DT-object
Zugriff:	set, get

Mit Hilfe dieses Attributes wird definiert, auf welches Objekt dieses Attribut verweisen soll. Dieser Verweis kann dynamisch verändert werden, in dem man diesem Attribut ein neues Objekt zuweist.

Beispiel

```
record Rec1
{
  string Str1 shadows ET.content;
}
```

Dieses definiert einen Record mit einem Element, das als Verweis auf das Eingabefeld ET und auf das Attribut .content definiert ist.

```
print Rec1.shadowobject[.Str1] ;    => ET

Rec1.shadowobject[.Str1] := Window1;
print Rec1.shadowobject[.Str1] ;    => Window1
```

2.4.7 .type

Identifikator:

.type

Klassifizierung:	Attribut von benutzerdefinierten Attributen

Definition

Argumenttyp:	Datentyp
C-Definition:	AT_type,
C-Datentyp:	DT_datatype
COBOL-Definition:	AT-type
COBOL-Datentyp:	DT-type
Zugriff:	set, get

Mit Hilfe dieses Attributes kann man einem benutzerdefinierten Attribut den Datentyp setzen bzw. abfragen. Durch dieses lassen sich auch zur Laufzeit dynamisch Attribute generieren bzw. zerstören. Generiert werden Attribute, indem man einem Attribut, das in Form eines Strings angegeben wird, einen Datentyp zuweist. Zerstört wird das Attribut, indem man ihm den Datentyp *void* zuweist.

Beispiel

Deklaration:

```
listbox LBFiles
{
  .ytop 1;
  .yleft 1;
}
```

Nachträgliche Generierung in einer Regel:

```
LBFiles.type[".CurrentFile"] := string;
```

Dies entspricht folgender Deklaration:

```
listbox LBFiles
{
  .ytop 1;
  .yleft 1;
  string CurrentFile;
}
```

Dieses Attribut kann man durch die nachfolgende Zuweisung wieder löschen:

```
LBFiles.type[.CurrentFile] := void;
```

2.5 Assoziative Arrays

An jedem Objekt kann der Dialogdesigner eigene Attribute, sogenannte benutzerdefinierte Attribute, angeben. Diese können dazu benutzt werden, um Statusinformation oder sonstige Daten direkt am Objekt abzulegen. Der Dialog Manager bietet hier die Möglichkeit der einfachen Attribute, der Feldattribute und assoziative Arrays.

Assoziative Felder werden hier detailliert beschrieben und anhand von Beispielen Anwendungsmöglichkeiten dargeboten.

2.5.1 Assoziationen und assoziative Arrays

Eine Assoziation ist eine gedankliche Verbindung zwischen Dingen oder Tätigkeiten, die per se nichts miteinander zu tun haben. Als Beispiel für Assoziationen kann ein Fernsehsender dienen, mit "RTL" verbindet oder assoziiert ein Zuschauer "Spielfilme", ein Jurist "Privatfernsehen", ein Werbefachmann "Quoten" und ein Fernsehtechniker den Kanal "65". In einem Programm können diese Assoziationen entsprechend programmiert werden, sodass ein Fernsehtechniker sich selbst nicht mehr den Kanal merken muss, sondern nur noch "RTL", das Programm unterstützt und entlastet ihn von unnötiger Arbeit. Die Umrechnung muss das Programm übernehmen. Hierbei ist Voraussetzung, dass sich die Assoziationen programmieren lassen. Der Dialog Manager unterstützt den Dialogprogrammierer, indem er ihm ein mächtiges Strukturierungselement an die Hand gibt; die assoziativen Arrays. Im allgemeinen ist eine Assoziation gerichtet, d.h. sie ist nicht unbedingt umkehrbar, so denkt der Fernsehtechniker bei "65" eher an Rente als einen Fernsehsender oder der Jurist bei "Privatfernsehen" eher an viel Arbeit und Verträge als an "RTL". Eine einzelne Assoziation lässt sich im Dialog Manager bereits mit normalen Attributen leicht definieren beziehungsweise ist unnötig zu definieren, da diese Assoziation im Normalfall direkt in den Code hineinprogrammiert wird. Zum Beispiel wird der Fernsehtechniker auch bei anderen Sendern gewisse Kanäle assoziieren. Der Dialog Manager bietet für solche Gruppen oder Reihen von gleichartigen Assoziationen das Mittel der assoziativen Arrays. Diese Beziehung könnte auch als Funktion aufgefasst werden, als Parameter wird die Quelle der Assoziation wie zum Beispiel "RTL" übergeben und als Ergebnis erhält man den Kanal 65. Diese Funktion zu programmieren ist sehr aufwendig, da in einer Regel ein "case" über alle Werte programmiert werden und die Funktion den möglichen Wertebereich abdecken muss oder nur statische Assoziationen besitzt. Der Wartungsaufwand ist unverhältnismäßig hoch im Gegensatz zu den assoziativen Arrays. Bei diesen Arrays oder Feldern kann mit der Quelle der Assoziation indiziert werden; der Inhalt dieses Feldes ist der entsprechende Endpunkt der Assoziation. In dem Senderbeispiel wurde mit dem String "RTL" indiziert und als Ergebnis wurde 65, also der Inhalt des Arrays, zurückgeliefert.

Der Dialog Manager kann an jedem beliebigen Objekt (Fenster, Poptext, Pushbutton, aber auch Dialog, Module oder Applikation) solche assoziativen Arrays definieren und benutzen. Diese assoziativen Arrays können fast wie normale Arrays von benutzerdefinierten Attributen definiert werden. Der Index wird jedoch als Datentyp angegeben und nicht als Bereich eingetragen.

Assoziative Arrays werden wie folgt definiert:

```
<DM Datentyp 1> <Attribute Name> [ <DM Datentyp 2> ];
.<Attribute Name>[ <Wert 2> ] := <Wert 1>;
```

Assoziative Arrays können maximal 65.528 Einträge aufnehmen. Wenn versucht wird mehr Einträge anzulegen, wird dies als "[SV] FAILED (IDM-E-IndexRange)" protokolliert.

Beispiel

```
window WMain
{
integer Kanal [ string ];
  .Kanal [ "RTL" ] := 65;
  .Kanal [ "ARD" ] := 14;
  .Kanal [ "ZDF" ] := 11;
}
```

Im Beispiel besitzt das Fenster ein assoziatives Array, in dem zu verschiedenen Strings, den Sendern, die entsprechenden Kanäle gespeichert werden.

Das obige Beispiel kann zu einem funktionsfähigen Dialog ausgebaut werden.

```
dialog SenderWahl

window WMain
{
integer Kanal [ string ];
  .Kanal [ "RTL" ] := 65;
  .Kanal [ "ARD" ] := 14;
  .Kanal [ "ZDF" ] := 11;
  child poptext Sender
  {
    .text [ 1 ]  "ARD";
    .text [ 2 ]  "ZDF";
    .text [ 3 ] "RTL";
    on select
    {
      print WMain.Kanal [ this.content ];
    }
  }
  on close { exit(); }
}
```

Wählt der Benutzer aus dem Poptext einen Sender aus, dann wird der aktuell gewählte String aus dem Poptext (this.content) als Index für den assoziativen Array Kanal benutzt. So erhält man auf elegante Art und Weise den zugehörigen oder assoziierten Kanal.

Man beachte, dass auf den assoziativen Arrays **keine** Ordnung besteht. Eine Ordnung ist oftmals nicht gegeben bzw. schwer berechenbar. Auch hier im Beispiel besteht bei den Sendern keine Ordnung, abgesehen von historischen Gründen bei ARD, ZDF und den "dritten" Regionalprogrammen.

2.5.2 Dynamische Verwaltung

Der Dialog Manager übernimmt die vollständige Verwaltung der assoziativen Felder. Für das Anlegen eines neuen Eintrages muss keine Vergrößerung des Arrays oder sonstige Verwaltungsarbeit vorgenommen werden. Es genügt die einfache Zuweisung eines Inhaltes an ein Feld; ist der Index noch nicht vorhanden, dann wird das Feld automatisch angelegt. Es ist also kein Aufruf einer :insert() Methode nötig.

Hierzu ein Beispiel eines Stacks oder Kellerspeichers, der beliebige Dialog Manager Daten (anyvalue) speichern kann:

```
dialog StackVerwaltung
record Stack
{
      integer Top := 0;
      anyvalue  Entry[ integer ];
      object Push := Rule_Push;
      object Pop := Rule_Pop;
}
rule void Rule_Push ( anyvalue Value)
{
      Stack.Entry[Stack.Top] := Value;
      Stack.Top := Stack.Top + 1;
}
rule anyvalue Rule_Pop ()
{
       if Stack.Top = 0
       then
              !! Error, stack is empty
        else
              Stack.Top := Stack.Top - 1;
        endif
        return (Stack.Entry[ Stack.Top ]);
}
on dialog Start
{

     Stack.Push( "Eintrag" );    !! "Eintrag"
     Stack.Push( "String" );     !! "Eintrag" - "String"
     Stack.Push( 42 );           !! "Eintrag" - "String" - 42
     Stack.Push( .visible );     !! "Eintrag" - "String" - 42
                                - .visible
     print Stack.Pop();          !! "Eintrag" - "String" - 42
     Stack.Push( .width );       !! "Eintrag" - "String" - 42
                                - .width
     print Stack.Pop();          !! "Eintrag" - "String" - 42
     print Stack.Pop();          !! "Eintrag" - "String"
     print Stack.Pop();          !! "Eintrag"
```

```
}
```

Mit Hilfe des obigen, funktionsfähigen Beispiels kann sehr leicht eine eigene Stackverwaltung aufgebaut werden. In Zeile 11 wird ein neuer Eintrag angelegt, ohne dass hierzu Verwaltungsaufwand getrieben werden muss. Im Beispiel wird nicht mit Strings indiziert oder Assoziationen aufgebaut, sondern es werden hier die integer Zahlen benutzt. Hier wird aber nicht die natürliche Ordnung der Zahlen im Array angewendet, es wird nur zu bestimmten Zahlen eine Beziehung oder Assoziation aufgebaut. Das Beispiel kann so abgewandelt werden, dass statt in Einer-Schritten in verschiedenen Schrittweiten hoch- bzw. runtergezählt wird. Das Verhalten bleibt identisch.

2.6 Arbeiten mit assoziativen Arrays

Manchmal ist es notwendig, die assoziativen Arrays in ihrer Gesamtheit zu bearbeiten oder einzelne Assoziationen, die überflüssig geworden sind, zu löschen. Mit Hilfe des Attributes *.itemcount* ist es möglich, die Anzahl der vorhandenen Einträge abzufragen, hierzu muss der Name des Arrays als Index dienen.

```
window WMain
{
  integer Kanal [ string ];
    .Kanal [ "RTL" ] := 65;
    .Kanal [ "ARD" ] := 14;
    .Kanal [ "ZDF" ] := 11;
}
rule integer Senderzahl()
{
  return ( WMain.itemcount[ .Kanal ]);  !! im Beispiel, 3
}
```

2.6.1 Die Methode index

Es ist auch möglich, die innere Ordnung der assoziativen Arrays abzufragen. Die innere Ordnung dient dem Dialog Manager nur zur Verwaltung. Der Anwender kann die innere Ordnung benutzen, um damit sein assoziatives Array zu durchlaufen.

Achtung

Die innere Ordnung der assoziativen Arrays dient nur zur internen Verwaltung. Diese innere Ordnung kann sich ändern. Sollten Sie eine Ordnung auf den Daten benötigen, was dann im allgemeinen auf einen Missbrauch der assoziativen Arrays schließen lässt, so müssen sie diese selbst berechnen. In diesem Fall empfehlen wir jedoch normale Arrays.

Mit der Methode :index() kann der Index aus der inneren Ordnung berechnet werden.

```
<Objekt>:index ( attribute <Name>, integer <InnereIndex> );
```

Wobei <Objekt> das Objekt ist, an dem der Array <Name> definiert ist. Der <InnereIndex> ist ein Index nach innerer Ordnung, diese Ordnung wird über integer Zahlen von 1 bis zur Anzahl der Einträge definiert.

Beispiel

```
window WMain
{
  integer Kanal [ string ];
    .Kanal [ "RTL" ] := 65;
    .Kanal [ "ARD" ] := 14;
    .Kanal [ "ZDF" ] := 11;
}
rule void DruckeSender()
{
  variable integer I;
  for I : = 1 to WMain.itemcount[ .Kanal ]
  do
    print WMain.Kanal[ WMain:index(.Kanal, I ) ];
  endfor
}
```

In Zeile 11 wird die Schleife von 1 bis zur Anzahl der Einträge durchlaufen, gemäß der inneren Ordnung. In Zeile 13 wird der Index berechnet (WMain:index(.Kanal, I)) und mit diesem dann das assoziative Array .Kanal indiziert.

2.6.2 Die Methode delete

Um beliebige Einträge in dem assoziativen Feld zu löschen, steht dem Anwender die Methode :delete () zur Verfügung.

```
<Objekt>:delete ( attribute <Name>, anyvalue <Index> );
```

Wobei <Objekt> das Objekt ist, an dem der Array <Name> definiert ist. Der <Index> gibt den Eintrag an, der gelöscht werden soll.

Beispiel

```
window WMain
{
  integer Kanal [ string ];
    .Kanal [ "RTL" ] := 65;
    .Kanal [ "ARD" ] := 14;
    .Kanal [ "ZDF" ] := 11;
}
rule void LoescheSender()
{
  variable integer I;
  for I : =  WMain.itemcount[ .Kanal ] to 1 step -1
```

```
  do
    WMain:delete( .Kanal, WMain:index(.Kanal, I ) );
  endfor
}

!! Zweite, eleganter programmierte Loeschroutine
rule void LoescheSender_2()
{
  while (WMain.itemcount[ .Kanal ] > 0)
  do
    WMain:delete( .Kanal, WMain:index(.Kanal, 1 ) );
  end
}

!! Dritte Variante der Loeschfunktion
!! Hier wird der Name direkt benutzt
rule void LoescheSenderPerNamen()
{
  WMain:delete( .Kanal, "RTL" );
  WMain:delete( .Kanal, "ZDF" );
  WMain:delete( .Kanal, "ARD" );
}
```

Die ersten zwei Regeln haben die identische Funktion, bei der ersten wird die innere Ordnung ausgenutzt und per Schleife von Ende bis zum Anfang durchlaufen. Man beachte hier, dass sich die innere Ordnung durch den Löschvorgang natürlich ändert, deshalb wird hier von hinten gelöscht. Solange Einträge vorhanden sind, gibt es einen ersten Eintrag gemäß der inneren Ordnung. Dies nützt die zweite Regel welche immer den ersten Eintrag löscht.

Die dritte Regel funktioniert nur, wenn man die Indizes der Einträge kennt.

2.7 Methoden für Arrays benutzerdefinierter Attribute

Die Methoden **:clear()**, **:delete()**, **:exchange()**, **:insert()** und **:move()** werden für normale, also nicht assoziativer Felder benutzerdefinierter Attribute unterstützt. Die dabei zulässigen Methoden werden im folgenden aufgeführt.

2.7.1 :clear()

Mit Hilfe dieser Methode können die Inhalte von Elementen in Feldern (indizierte, nicht assoziative, benutzerdefinierte Attribute) gelöscht werden.

Im Unterschied zur **:delete()**-Methode werden mit **:clear()** nur die Inhalte der Elemente gelöscht. Die Elemente selbst bleiben als „leere“ Elemente erhalten, die – sofern vorhanden – den Standardwert aufweisen. **:clear()** verringert also die Anzahl der Elemente nicht.

Besonderheit

Die **:clear()**-Methode kann auch dafür benutzt werden, **alle** Elemente eines **assoziativen** Felds zu löschen. In diesem Fall dürfen weder der *Start*- noch der *Count*-Parameter angegeben werden. Es ist also nicht möglich, mit **:clear()** einzelne Elemente aus einem assoziativen Feld zu löschen. Außerdem werden in diesem Fall nicht nur die Inhalte sondern die Elemente selbst gelöscht. Die Anzahl der Elemente ist danach also *0*.

Definition

```
boolean :clear
(
      attribute Attr input
  { , integer Start input }
  { , integer Count input }
)
```

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

*integer **Start** input*

Dieser Parameter definiert die Position, ab der die Inhalte der Elemente gelöscht werden sollen. Der Wertebereich für *Start* ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert gelöscht werden.
Wenn weder *Start* noch *Count* angegeben sind, werden die Inhalte aller Elemente gelöscht.

*integer **Count** input*

Dieser optionale Parameter definiert die Anzahl der Elemente, deren Inhalte gelöscht werden sollen.
Wenn weder *Start* noch *Count* angegeben sind, werden die Inhalte aller Elemente gelöscht.
Wenn *Start* angegeben wird, ist der Standardwert von *Count 1*.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Inhalte der Elemente gelöscht werden konnten.

Wenn beim Löschen der Inhalte ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

2.7.2 :delete()

Mit Hilfe dieser Methode können Elemente aus Feldern (indizierten benutzerdefinierten Attributen) gelöscht werden.

Im Unterschied zur **:clear()**-Methode werden die Elemente mit **:delete()** vollständig gelöscht, d.h. die Anzahl der Elemente verringert sich.

Definition

```
boolean :delete
(
      attribute Attr input,
      anyvalue Position input
  { , integer Count := 1 input }
)
```

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

anyvalue *Position* input

In diesem Parameter wird der Index übergeben, ab dem die Elemente gelöscht werden sollen. Bei nicht-assoziativen Feldern hat *Position* den Datentyp *integer*, bei assoziativen Feldern den Datentyp, mit dem das assoziative Feld indiziert ist.
Der Wertebereich für *Position* bei nicht-assoziatven Feldern ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert gelöscht werden. Wird der Standardwert gelöscht, dann wird der Wert des ersten Elements nach den gelöschten Elementen zum Standardwert.

*integer **Count** := 1 input*

Dieser optionale Parameter definiert die Anzahl der zu löschenden Elemente. Wird der Parameter nicht angegeben, so wird *1* angenommen.
Count kann bei assoziativen Feldern nicht verwendet werden, es kann immer nur *1* Element gelöscht werden.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente gelöscht werden konnten.

Wenn beim Löschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

2.7.3 :exchange()

Mit Hilfe dieser Methode können zwei Elemente eines Feldes (indizierten, nicht-assoziativen, benutzerdefinierten Attributs) miteinander vertauscht werden.

Definition

```
boolean :exchange
(
  attribute Attr input,
  integer Position1 input,
  integer Position2 input
)
```

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position1* input
integer *Position2* input

In diesen Parametern werden die Indizes der beiden Elemente übergeben, die miteinander vertauscht werden sollen.
Der Wertebereich für beide Parameter ist *0 … .count[Attr]*, d.h. es kann auch der Standardwert mit dem Wert eines anderen Elements vertauscht werden.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente vertauscht werden konnten.

Wenn beim Vertauschen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

2.7.4 :insert()

Mit Hilfe dieser Methode können neue Elemente in Felder (indizierte, nicht-assoziative, benutzerdefinierte Attribute) eingefügt werden. Die neu eingefügten Elemente sind „leer“ bzw. weisen – sofern vorhanden – den Standardwert auf.

Definition

```
boolean :insert
(
      attribute Attr input,
      integer Position input
  { , integer Count := 1 input }
)
```

Parameter

attribute *Attr* input

Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position* input

In diesem Parameter wird der Index übergeben, an dem die neuen Elemente eingefügt werden sollen. Wenn sein Wert *0* ist, werden die Elemente am Ende angehängt.

*integer **Count** := 1 input*

Dieser optionale Parameter definiert die Anzahl der einzufügenden Elemente. Wird der Parameter nicht angegeben, so wird *1* angenommen.

Rückgabewert

Die Methode gibt *true* zurück, wenn die Elemente eingefügt werden konnten.

Wenn beim Einfügen ein Fehler aufgetreten ist, gibt die Methode ein `fail` zurück.

2.7.5 :move()

Mit Hilfe dieser Methode können Elemente eines Feldes (indizierten, nicht-assoziativen, benutzerdefinierten Attributs) an eine andere Stelle im Feld verschoben werden.

Definition

```
void :move
(
  attribute Attr input,
  integer Position input,
  integer Target input,
  integer Count input
)
```

Parameter

attribute *Attr* input
: Dieser Parameter definiert das benutzerdefinierte Attribut, auf das die Methode angewendet werden soll.

integer *Position* input
: In diesem Parameter wird der Index des ersten Elements angegeben, das verschoben werden soll.

integer *Target* input
: Dieser Parameter legt fest, wohin die Elemente verschoben werden sollen.

integer *Count* input
: Dieser Parameter definiert, wie viele Elemente verschoben werden.

Rückgabewert

Keiner.

3 Benutzerdefinierte Methoden

Die Methoden sind Regeln am Objekt. Sie können dort wie benannte Regeln im Dialog in der gewohnten Weise definiert werden. Der Aufruf dieser Methoden lehnt sich an die vordefinierten Methoden an.

Beispiel

```
dialog Beispiel
window Win
{
  child statictext Stext
  {
    !! Definition und Deklaration der Methode
    rule void PrintEvent(string S)
    {
      this.text := "Win was " + S;
    }
  }
  on move
  {
    !! Aufruf der Methode
    Stext:PrintEvent("moved");
  }
  on resize
  {
    Stext:PrintEvent("resized");
  }
}
```

3.1 Das this-Objekt

Bisher war das Schlüsselwort this in der Regelsprache mit der Bedeutung belegt, dass es das Objekt repräsentiert, das die Regelverarbeitung ausgelöst hat. Diese Bedeutung ist nach wie vor unverändert. Lediglich innerhalb von Methoden wurde die Bedeutung so verändert, dass in dem this das Objekt gespeichert ist, mit dem die Methode aufgerufen worden ist. Muss in Methoden zusätzlich auf das Objekt zugegriffen werden auf dem das auslösende Ereignis (also nicht die Methode) auftrat, so kann der Entwickler dies über das Objekt thisevent.object erfragen.

Im obigen Beispieldialog wurde in der Methode bereits implizit davon ausgegangen, dass das this der "statictext Stext" und nicht das "window Win" ist, auf dem das Ereignis stattfand. Wird in einer Methode eine weitere Methode aufgerufen, wird die Belegung des this entsprechend verändert.

Bemerkung

Regeln, die am Dialog definiert sind, sind keine Methoden des Dialogs, es ändert sich somit nicht die Bedeutung von bestehenden Dialogen.

3.2 Modelle und Methoden

Der Dialog Manager unterstützt die Methoden auch in den Modellen und in der Vererbung. Es können also zentral am Modell oder am Default Methoden definiert werden. Damit steht dem Entwickler ein weiteres mächtiges Instrument zur Verfügung. Instanzen der Modelle können dynamisch auf diese Modellmethoden zugreifen und sie verwenden.

```
dialog Beispiel
model statictext Mstatic
{
  rule void PrintEvent (string S)
  {
    this.text := "Win was " + S;
  }
}
window Win
{
  child Mstatic S1 {}
  child Mstatic S2 {}
  on move
  {
    !! Aufruf der Methode
    S1:PrintEvent("moved");
  }
  on resize
  {
    S2:PrintEvent("resized");
  }
}
```

Das Beispiel oben zeigt, dass die Methode zentral am Modell Mstatic definiert wurde. Hier kann wie in den bekannten Ereignisregeln das this Objekt benutzt werden. Das this ist dann immer mit dem entsprechenden Objekt belegt, mit dem die Methode in Wirklichkeit aufgerufen wurde. Im Beispiel bei dem Ereignis move des Fensters Win hat this den Wert S1, beim resize von Win den Werte S2. Mit Hilfe dieses Mechanismus lassen sich leicht allgemeine Regeln oder besser Methoden formulieren, die am Modell verankert werden können.

3.3 Der Aufruf von Modell-Methoden

Bei Methoden entsteht das Problem, das am Default oder an relativ weit oben stehenden Modellen, allgemeine Methoden formuliert sind. An weiter unten stehenden Objekten geschieht dann die

Spezialisierung dieser Methoden. Um eine Codeverdoppelung zu vermeiden, können die Methoden an oben stehenden Modellen auch mitbenutzt d.h. von hierarchisch tieferliegenden Methoden aufgerufen werden. Es besteht aber kein Automatismus, wie etwa bei den Ereignisregeln (mit before und after), mit der dies kontrolliert werden kann. Es wird dem Entwickler die Methode :super zur Verfügung gestellt, diese ruft die entsprechende Methode des Modells auf.

```
Objekt:super( [Parameter...] );
```

Beispiel

```
dialog Beispiel

model window MWin
{
    integer Status := 0;
    rule void CleanAfterClose()
    {
        !! Das Attribut zurücksetzen
        this.Status := 0;
    }

    !! Beide Regeln reagieren darauf, wenn das Fenster
    !! unsichtbar wird und stoßen dann zentral die
    !! "Aufräum-Regel" an.
    on close before
    {
        this:CleanAfterClose();
    }
    on .visible changed before
    {
        if this.visible = false then
            this:CleanAfterClose();
        endif
    }
}
model MWin MMainWin
{
    integer MainStatus := 0;
    rule void CleanAfterClose()
    {
        !! Nur die Attribute, die hier definiert wurden
        !! zurücksetzen
        this.MainStatus := 0;
        !! Den Rest aufräumen
        this:super();
    }
}
```

```
MMainWin MainWin
{
}
```

Dieses Beispiel zeigt, wie nach dem Schließen eines Fensters, bestimmte Attribute wieder zurückgesetzt werden. Die Methoden stehen direkt an dem Objekt, wo die Attribute definiert wurden. Damit wird die Information zentral an dem Objekt gespeichert. Mit der Methode :super ist die Reihenfolge des Ablaufs der Methoden in der Modellhierarchie selbst festzulegen. Der Dialog Manager kann hier nicht vorhersehen, in welcher Reihenfolge bzw. welche Bedingungen erfüllt sein müssen, damit die jeweilige Methode korrekt funktioniert und das gewünschte Ergebnis erzeugt. this:super() sucht immer die nächste Methode in der Modellhierarchie des aktuellen Objektes.

3.4 Der indirekte Aufruf von Methoden

Der Dialog Manager besitzt den Datentyp method. In Variablen oder Attributen lassen sich damit Methoden oder besser Methodennamen speichern und aufrufen. Das gleiche Problem besteht bei Attributen, wenn in einer Variable ein Attribut gespeichert wird, dann kann nicht einfach Objekt.Variable geschrieben werden. Der Dialog Manager kann an dieser Stelle nicht unterscheiden, ob der Variablenname nicht auch ein Attribut dieses Objektes sein könnte. Bei Attributen kann über getvalue (Objekt,Variable) dieses Problem gelöst werden. Bei Methoden ist, wie gesagt, das gleiche Problem vorhanden. Objekt:Variable kann der Dialog Manager wegen der Möglichkeit von Mehrdeutigkeiten nicht auflösen. Deshalb gibt es hierzu eine ":call"-builtin-Methode.

Der Aufruf ist

```
Objekt:call( <Methode>, [Parameter...] );
```

Also kann, wenn in einer Variable eine Methode gespeichert wurde, diese Methode dann aufgerufen werden mit Objekt:call(Variable). Der Aufruf von :call ruft indirekt die angegebene Methode auf. Der Rückgabewert entspricht dem der indirekt aufgerufenen Methode. Der :call Parameter kann jedoch nur maximal 15 weitere Parameter besitzen. Hierauf ist beim Design von Methoden zu achten. Die Methode :call kann auch eingebaute Methoden wie etwa :insert oder :delete aufrufen.

Beispiel

```
dialog Beispiel
record Rec
{
  string String;
  rule void Method {}
  rule boolean ZeigeAn (object Obj input) {}
}

on dialog start
{
  Rec:call(:Method);         // ruft die Methode Method auf
  Rec:call(:ZeigeAn, this);  // ruft die Methode ZeigeAn mit
                             // dem Dialog als Objekt auf.
```

```
}
```

3.5 Existenz einer Methode

Der Dialog Manager kann zur Laufzeit, also dynamisch, abfragen, ob ein bestimmtes Objekt eine bestimmte Methode hat oder ein bestimmtes Attribut besitzt. Dieses Abfragen wird über die Methode :has realisiert. Der Aufruf erfolgt über Objekt:has(Attribut oder Methode).

```
Objekt:has( <Methode> | <Attribut> );
```

Beispiel

```
dialog Beispiel
record Rec
{
  string String;
  rule void Method {}
}
on dialog start
{
  print Rec:has(:Method);  // ergibt true
  print Rec:has(.String);  // ergibt true
  print Rec:has(:Print);   // ergibt false
  print Rec:has(.Data);    // ergibt false
}
```

Index