Snoopy: Ein generisches, adaptives Werkzeug für Graphen

Handbuch für den Administrator

Markus Fieber

Januar 2003

Inhaltsverzeichnis

1 Grundlagen

SNOOPY ist ein Werkzeug zum strukturellen Entwerfen und graphischen Bearbeiten von Graphen. Um diesem Anspruch zu genügen, teilt sich SNOOPY in zwei grundlegende Bereiche. Die **Struktur**-Komponente übernimmt die Organisation der Daten, während eine **Graphik**-Bibliothek die Darstellung ermöglicht.

1.1 Struktur

Um beliebige Graphrepräsentationen definieren zu können, beruht die dem Programm zu Grunde liegende Datenstruktur auf folgenden Festlegungen.

Definition 1 *Graph*
Sei $G = \{\ N, E\ \}$ *ein Graph mit*
$N = \{$ Knoten des Graphen $\}$
$E = \{\ e \in N \times N\ \mid$ Kanten des Graphen $\}$

Die Definition 1 beschreibt ausschließlich die strukturelle Repräsentation eines Graphen. Beide Mengen werden im Programm durch Listen implementiert, die als *Sorten* bezeichnet werden. So existieren eine *Knotenesorte* und eine *Kantensorte*, die diese beiden Mengen repräsentieren.

Diese Definition genügt noch nicht höheren Graphen, die mit verschiedenen Knoten- und Kantenarten die Mengen partitionieren. Um diese Eigenschaften fassen zu können, erweitern wir Definition 1:

Definition 2 *Graph mit partitionierten Mengen*
Sei $G' = \{\ N', E'\ \}$ *ein Graph mit*
$N' = \{\ \bigcup N_i \mid$ mit $\mathrm{N_i}$ Menge von Knoten gleicher Semantik; $i \in \mathbf{N}\ \}$
$E' = \{\ e \in N' \times N'\ \mid$ Kanten zw. Elementen der Mengen aus $N'\ \}$

Die Partitionierung setzt eine Eigenschaftsbeschreibung voraus, anhand derer sich die Elemente der einen oder anderen Partition zurodnen lassen. Das geschieht durch Attribute, die durch ihren Namen und einen Typ definiert sind:
$A = (Name, Typ)$, A ist ein Attribut.

Ein Element einer Partition wird mit seinen Attributen beschrieben. Jedem Bestandteil des Graphen (Knoten und Kanten) ist eine feste Menge von Attributen zugeordnet. Jede Partition ist durch die Gleichheit der Attributmenge ihrer

Elemente beschrieben. Da sich die Elemente einer Partition also nur in der Wertigkeit ihrer Attribute unterscheiden, läßt sich eine solche Partition anhand eines prototypischen Elements beschreiben. Nun kann jedes konkrete Element einer Partition als Ableitung aus dem zugehörigen Prototypen erzeugt werden.

1.2 Graphik

Mit der Strukturdefinition des Graphen haben wir noch keine Möglichkeit, diesen darzustellen. Um dies tun zu können, bedarf es einer Zuordnungsfunktion von graphischen Repräsentationen zu den strukturellen Bestandteilen des Graphen. Diese Zuordnung wird mit *view* gewährleistet:

Definition 3 *Darstellung eines Graphen*
Sei $W = \{ w \mid w$ ist ein Shape$\}$ *eine Menge von graphischen Elementen. Diese graphischen Elemente enthalten die Darstellungen für Knoten, Kanten und Attribute in drei verschiedenen Partitionen, die wir zum Beispiel* W_N *für die Menge der Knotengraphiken nennen.*
Dann existieren folgende Funktionen
$view_N : N \rightarrow W_N$ *die den Elementen aus* N *(Knoten) Elemente aus* W *zuordnet*
$view_E : E \rightarrow W_E$ *die den Elementen aus* E *(Kanten) Elemente aus* W *zuordnet*
$view_A : A \rightarrow W_A$ *die den Elementen aus* A *(Attribute) Elemente aus* W *zuordnet*

Allen graphischen Ausprägungen ist gemein, dass sie in einem Fenstersystem darstellbar sind, ausgedrückt durch $draw(w), w \in W$. Jedem Element des Graphen G wird ein Element aus der Menge der graphischen Ausprägungen W zugewiesen, so dass sich die Darstellung eines Graphen folgendermassen formulieren lässt:
Das Zeichnen eines Graphen bedeutet, seine Knoten und Kanten zu zeichnen.
Das Zeichnen eines Knotens bedeutet, die graphische Ausprägung des Knotens anzuzeigen und alle seine Attribute zu zeichnen.
Das Zeichnen eines Attributs bedeutet, seine graphische Ausprägung anzuzeigen.
Der gleiche Ablauf gilt für die Kanten. Anhand dieses Iterierens über die Datenstruktur mit dem Befehl *draw()* wird die Anzeige aller Bestandteile des Graphen ermöglicht.

1.3 Hierarchisierung

Wenn ein Graph G' von einem Graphen G erbt, dann kann dies nur durch Änderung der Partitionsmengen (N, E) geschehen. Diese ändern sich nur durch das

Hinzufügen oder Entfernen von einzelnen Partitionen oder durch Veränderungen an den bestehenden Partitionen selbst. Änderungen an den Partitionen bedeuten Änderungen an den Attributmengen. Die Definition von *view* zur Zuordnung der Darstellung bleibt davon unberührt und ist weiterhin anwendbar.

Diese Hierarchisierung ist Teil der Philosophie von SNOOPY.

2 Implementierung

Die Umsetzung der Theorie in die Praxis basiert in SNOOPY auf dem Bibliotheksgedanken. Generell ist der Quelltext in die Implementierung von Datenstruktur und Graphik getrennt. Die Datenstruktur enthält alle Bestandteile, die zur Definition der Graphstruktur und den darauf ablaufenden Operationen von Nöten sind. Dazu gehören die Implementierungen der Partitionen (Knoten- und Kantensorte), Bestandteile der Partitionen (Knoten und Kanten), Klassen für verschiedenste Attribute und natürlich der Graph selbst, sowie verschiedene weitere Klassen, die hier erstmal keine Rolle spielen. In der Graphikbibliothek sind die Shapes für die Umsetzung der *view*-Funktionen, sowie alle Klassen, die außerdem noch für die Oberfläche des Werkzeugs selbst benötigt werden, definiert.

Die Verzeichnisse tragen Namen, die sich als Präfix auch an den in ihnen befindlichen Dateien wiederfinden. Dabei stehen die Buchstaben `SP`, für SNOOPY , vor jeder Datei der Applikation. Des Weiteren folgen zwei- bis dreibuchstabige Abkürzungen, die die genaue Aufgabe noch etwas einschränken.

`snoopy`	Wurzelverzeichnis
- `sp-ana`	Analysemethoden und Schnittstellen dafür
- `sp-attr`	Attribute (der Datenstruktur)
- `sp-ds`	Datenstruktur (Klassen, die Knoten, Kanten und deren Sorten beschreiben)
- `sp-gm`	Graphikmanagement, Eventhandler
- `sp-gr`	graphische Knoten, Kanten und Attribute
- `sp-gui`	graphische Oberfläche
- `dialogs`	Dialogklassen
- `windows`	Fensterklassen
- `sp-io`	Ein-/Ausgabeklassen (für Im- und Export)
- `defs`	Schnittstellendefinitionen
- `sp-net`	Netzklassendefinitionen
- `sp-templ`	Templates (Liste), Defines

Die Klasse `SP_DS_Graph`, ist die zentrale Instanz der Datenstruktur. Sie enthält die Knoten- und Kantensorten (die Partitionen). In der `Create`-Methode einer Netzklasse wird die Definition eines `SP_DS_Graph`-Objekts manipuliert. Durch dieses Vorgehen ist die Vererbung realisiert.

```
class SP_DS_Graph: public ...                                          1
{                                                                      2
protected:                                                             3
    mlist <SP_DS_NodeSort*> mNodeSortsList;                            4
    mlist <SP_DS_EdgeSort*>     mEdgeSortsList;                        5
    ...                                                                6
};                                                                     7
```

Die Kernkomponenten eines Graphen sind die Listen der Knoten- und Kantensorten. Die *Sorte* definiert das Aussehen und das Verhalten der Elemente einer Partition anhand des prototypischen Objekts. Für Knoten ist dies ein Objekt der Klasse `SP_DS_Node`, für die Kanten ist dies ein Objekt der Klasse `SP_DS_Edge`.

Man kann einem Graphen Partitionen hinzufügen (`addNodeSort` und `addEdgeSort`) und Partitionen erfragen (`getNodeSort` und `getEdgeSort`). Die Identifikation erfolgt dabei über einen Namen, der der Sorte zur Konstruktionszeit mitgegeben wird, und der im weiteren Verlauf noch manipuliert werden kann (`renameNodeSort` und `renameEdgeSort`). Das prototypische Element einer Partition ist Teil der entsprechenden Sorte. Mit der Methode `GetPrototype()` läßt sich die Vorlage jederzeit anfordern.

Die Attribute des prototypischen Elements werden mittels `addAttribute(SP_DS_Attribute*)` gesetzt. Diese Funktion existiert sowohl für Objekte von `SP_DS_Node` (`SP_DS_Edge`), als auch den zugehörigen Sorten. Dort wird der Aufruf allerdings nur an das prototypische Element weitergereicht. Die Klasse `SP_DS_Attribute` ist Basisklasse für alle existierenden Datenstrukturattribute und implementiert grundlegende Methoden, die von ihren abgeleiteten Klassen jedoch jederzeit überschrieben werden können. Die bisher implementierte Menge der Attribute umfaßt 16 verschiedene Klassen (s. Anhang A.2, S. 23).

Als graphische Ausprägungen der Knoten bietet Snoopy, in der vorliegenden Version, grundlegende geometrische Formen, wie Kreis (Ellipse) und Quadrat (Rechteck) aber auch ein Bitmapshape und Weitere. Als graphische Ausprägung von Kanten existiert momentan nur eine Klasse (s. Anhang A.18, S. 46). Die Assoziation der Bestandteile der Datenstruktur mit graphischen Ausprägungen erfolgt über `SetGraphic(SP_GR_Interface*)`, die von allen Klassen implementiert werden muß. `SP_GR_Interface` ist die Basisklasse der graphischen Bibliothek und definiert die Schnittstellen zu den grundlegenden Funktionen für graphische Elemente. Davon abgeleitet sind weitere Klassen als Basis für Knoten, Kanten und Attribute.

Bei den graphischen Ausprägungen der Attribute ist die Liste noch nicht ganz so umfangreich. Natürlich steht es dem Nutzer frei, eigene graphische Attribute zu definieren, genauso wie er auch noch eigene Datenstrukturattribute hinzufügen

kann. Die bisher implementierte Menge reicht aber schon für viele Anwendungen aus (siehe Seite 46). Während (die graphischen) Knoten und Kanten dem Nutzer die Verbindungen in der Datenstruktur visualisieren, und ihr Verhalten nur in engen Grenzen konfigurierbar ist, bieten sich bei den graphischen Attributen mehrere Möglichkeiten an.

Die Hauptaufgabe der graphischen Attribute ist es, dem Nutzer eine Möglichkeit zu geben, aus der Oberfläche heraus den Wert in der Datenstruktur zu manipulieren. Zu diesem Zweck ist jedes Attribut für sein "Auftreten" im Konfigurationsdialog selbst verantwortlich. Darüberhinaus muß jedes Attribut eine Möglichkeit zur Darstellung in der Oberfläche bieten. Ob diese Darstellung in einer konkreten Netzklasse auch zur Anwendung kommt, ist dem Nutzer freigestellt.

Alle möglichen Erscheinungsformen, oder Anwendungsfälle von graphischen Attributen sind durch die folgenden vier Varianten charakterisiert:

- Darstellung in der Oberfläche, Manipulation im Dialog
- Darstellung in der Oberfläche, keine Manipulation im Dialog
- Keine Darstellung in der Oberfläche, Manipulation im Dialog
- Keine Darstellung in der Oberfläche, keine Manipulation im Dialog

Die ersten drei Forderungen lassen sich über Prädikate jedes graphischen Attributes erfüllen. Für die vierte Forderung existiert ein `SP_GR_NilAttribute`, das weder eine Repräsentation in der Oberfläche hat noch im Eigenschaftendialog des zugehörigen Elements erscheint. Deshalb muß sich der Nutzer beim Entwerfen einer eigenen Netzklasse über die Bedeutung der Attribute im Klaren sein. Zwischen den ersten drei Punkten besteht prinzipiell die Möglichkeit, zu wechseln. Da aber das `SP_GR_NilAttribute` keine Möglichkeit der Einflußnahme aus der Oberfläche heraus bietet, hilft für den Fall, daß man doch einmal eine andere Repräsentation dieses Attributes braucht, nur ein Wechsel der Defintion.

Ausgangspunkt der Graphdefintion ist die Klasse `SP_NT_Net` (das `NT` steht für *Net* oder Netzdefinition). Sie ist die Basis der Vererbungshierarchie (die Graphen betreffend) und definiert die notwendigen Methoden.

```
class SP_NT_Net: public SP_DS_Name                                  1
{                                                                   2
public:                                                             3
    SP_NT_Net(const string&);                                       4
                                                                    5
    virtual bool Create(SP_DS_Graph*) = 0;                          6
    virtual bool Clone() = 0;                                       7
    virtual bool EdgeRequirement( ... ) = 0;                        8
    virtual bool NodeRequirement( ... ) = 0;                        9
```

```
};
```

Die hier definierten Funktionen stellen das wichtigste Werkzeug im Umgang mit Graphen dar. Sämtliche im Programm definierten Netzklassen sind von `SP_NT_Net` abgeleitet. Dem Nutzer bleibt es überlassen, von einer schon vorhandenen Definition zu erben oder für eine neue Netzklasse von dieser Basis abzuleiten. `Create` ist die Konstruktionsmethode, in der die Netzklassendefinition implementiert ist. Durch aufeinanderfolgende Aufrufe dieser Methode von verschiedenen Klassen ist die Eingangs beschriebene Hierarchisierung von Graphen realisiert.

Ein wichtiger Punkt ist die `EdgeRequirement`-Funktion. Mal abgesehen von einfachen Graphen, stellt nämlich die Überprüfung der Zulässigkeit von Kanten ein zentrales Problem der Umsetzung dar. Wie schon besprochen kann ein Graph beliebig viele Knoten- und Kantensorten enthalten. Unter Umständen existiert für jedes Paar aus der Knotenmenge eine eigene Kantendefinition, die nur zwischen diesen beiden Knotentypen erlaubt ist. Um die Überpfüfung solcher Abhängigkeiten gewährleisten zu können, muß die `EdgeRequirement`-Funktion definiert werden:

```
bool
SP_NT_Net::EdgeRequirement(SP_DS_EdgeSort*, SP_DS_Node*, SP_DS_Node*)
```

Diese Methode wird bei jedem Zeichnen einer Kante aufgerufen. Ihr erster Parameter enthält den Typ der zu zeichnenden Kante, die folgenden beiden Parameter enthalten die Objekte der mit dieser Kante zu verbindenden Knoten. Je nach Wert des boolschen Ergebnisparameters wird die Kante zugelassen oder unterbunden. Ähnlich verhält es sich mit der `NodeRequirement`-Funktion, die die Überprüfung von Randbedingungen zur Erstellungszeit eines neuen Knotens erlaubt.

Mit einem ähnlichen Mechanismus lassen sich auch sogenannte *On-the-Fly* Überprüfungen bestimmter Bedingungen und Abhängigkeiten realisieren. Der Aufruf von speziellen Funktionen bei der Änderung von Attributen könnte so zum Beispiel die Lebenszeit eines Objekts beenden oder die Änderung eines anderen Attributes nach sich ziehen.

2.1 Vorgehensweise

Bei der Implementierung eigener Netzklassen sollte immer `SP_NT_Graph` als Basisklasse gewählt werden. Die Klassendefinition sieht dann in aller Regel wie folgt aus:

```
#include "sp_nt_graph.h"

class SP_NT_MyNewGraph: public SP_NT_Graph
{
public:
    SP_NT_MyNewGraph();
    SP_NT_MyNewGraph(const string&);
    bool Create(SP_DS_Graph*);
    bool EdgeRequirement(SP_DS_EdgeSort*, SP_DS_Node*, SP_DS_Node*);
    SP_NT_Net* Clone();
};
```

Alle zur Netzklassendefintion, und zum Arbeiten mit danach erstellten Graphen, notwendigen Klassenattribute und Methoden, stehen durch die Vererbung zur Verfügung. Diese vererbte Definition kann natürlich um weitere, nur in dieser oder in davon abgeleiteten Klassen gültige Attribute und Methoden erweitert werden. Dabei gelten die Regeln der objektorientierten Entwicklung. Um die übernommenen Eigenschaften auch auf die neue Netzklasse anzuwenden, ist es eine unerläßliche Forderung, in der eigenen Create-Methode die entsprechende Funktion der Basisklasse aufzurufen. Dies geschieht nicht automatisch, reicht aber aus, um die Definition der Netzklasse aus der Basisklasse zu übernehmen, wodurch die neue Netzklasse zu einer 1 zu 1 Kopie der Basis wird, woraufhin sie um neue Definitionen erweitert werden kann.

Die Implementierung der Konstruktoren ist simpel. Der Standardkonstruktor dient nur der verkürzten Schreibweise und sollte immer den Konstruktor der Basisklasse mit einer geeigneten Zeichenkette als Namen aufrufen. Der Name wird für die Anzeige in der Oberfläche genutzt und zusätzlich bei der Speicherung in die Datei geschrieben. Im Beispiel sieht das folgendermaßen aus:

```
SP_NT_MyNewGraph::SP_NT_MyNewGraph(): SP_NT_Graph("MyNewGraph")
{ }
```

Die zweite Version des Konstruktors, mit Stringargument, reicht genau dieses Argument an die Basisklasse weiter. Damit können beliebig benannte Instanzen dieser Netzklasse erzeugt werden und in der Vererbung haben alle Netzklassen einer Ableitung den gleichen Namen.

```
SP_NT_MyNewGraph::SP_NT_MyNewGraph(const string& pName)
:SP_NT_Graph(pName)
{ }
```

Darüberhinaus ist es in den Konstruktoren natürlich möglich, bestimmte klassenspezifische Attribute zu initialisieren, die in dieser individuellen Netzklasse benötigt werden. Die Aufrufe der Basisklassenkonstruktoren ist allerdings eine unabdingbare Forderung!

Das Herzstück aller Netzklassen ist die `Create`-Funktion, in der die Partitionen angelegt und deren Prototypen definiert werden. Das als Argument übergebene Objekt von `SP_DS_Graph` wird dabei mit den in dieser Methode vorhandenen Definitionen modifiziert. Dadurch, daß dieses Objekt schon andere Netzklassendefinitionen durchlaufen haben kann, ist die schon beschriebene Vererbung umgesetzt.

Zur Übernahme der Eigenschaften der Basisklasse muß bei einer abgeleiteten Netzklasse die `Create`-Funktion dieser Basis aufgerufen werden. Anschließend kann das Graph-Objekt nach belieben und aufbauend auf die übernommene Definition modifiziert werden.

```
bool
SP_NT_MyNewGraph::Create(SP_DS_Graph* pGraph)
{
    //Create der Basisklasse aufrufen
    SP_NT_Graph::Create(pGraph);
    //Jetzt haben wir die Definition von Graph uebernommen

    ...
}
```

An dieser Stelle enthält die neue Netzklasse alle Definitionen der Basisklasse, in diesem Fall also des einfachen Graphen mit nur einer Knoten- und einer Kantenpartition. In den folgenden Beispielen sind einige Netzklassen mit ihrer vollständigen `Create`-Funktion vorgestellt.

3 Beispiele

3.1 SP_NT_Graph

```
bool                                                                    1
SP_NT_Graph::Create(SP_DS_Graph* pGraph)                                2
{                                                                       3
    //Variablen zur Speicherung                                         4
    SP_DS_NodeSort* lNodeSort;                                          5
    SP_DS_EdgeSort* lEdgeSort;                                          6
    SP_DS_Node* lDsNode;                                                7
    SP_GR_Node* lGrNode;                                                8
                                                                        9
    //Knotensorte und ein Knoten                                        10
    //neue Knoten-Partition, namens Knoten hinzufuegen                  11
    lNodeSort = pGraph->addNodeSort("Knoten");                          12
    //Prototypen sichern                                                13
    lDsNode = lNodeSort->GetPrototype();                                14
    //graphische Auspraegung festlegen (Kreis)                          15
    lGrNode = (SP_GR_Node*)lDsNode->SetGraphic(                         16
                  new SP_GR_Circle(lDsNode)));                          17
    //Initialgroesse festlegen                                          18
    lGrNode->SetSizeVal(50, 30);                                        19
                                                                        20
    //Kantensorte und eine Kante                                        21
    //neue Kanten-Partition, namens Kante hinzufuegen                   22
    lEdgeSort = pGraph->addEdgeSort("Kante");                           23
    //graphische Auspraegung des Prototypen festlegen                   24
    lEdgeSort->GetPrototype()->SetGraphic(                              25
        new SP_GR_Line(lEdgeSort->GetPrototype()));                     26
                                                                        27
    //Definition abgeschlossen                                          28
    return true;                                                        29
}                                                                       30
```

Die Netzklasse *Graph* ist die einfachste Klasse, die überhaupt möglich ist. Die Knoten- und Kantenmenge sind nicht partitioniert, die Sortenlisten sind also einelementig. Damit ist die Definition abgeschlossen. Dieser Graph besitzt noch keine Attribute an seinen Knoten oder Kanten, ist aber schon zur Darstellung in Snoopy geeignet.

Auch an die `EdgeRequirement`-Funktion dieser Netzklasse werden keine hohen Anforderungen gestellt, Kanten sind ohne Einschränkung erlaubt.

```
bool                                                                    1
SP_NT_Graph::EdgeRequirement(SP_DS_EdgeSort* pSort,                     2
                    SP_DS_Node* pSrc, SP_DS_Node* pDest)                3
{                                                                       4
    return true;                                                        5
}                                                                       6
```

3.2 SP_NT_BipartGraph

Zur Veranschaulichung der Graphvererbung soll hier die in SNOOPY implementierte Version eines bipartiten Graphen vorgestellt werden.

```
bool                                                                    1
SP_NT_BipartGraph::Create(SP_DS_Graph* pGraph)                          2
{                                                                       3
    //Variablen zur Speicherung                                         4
    SP_DS_NodeSort* lDsNode;                                            5
    SP_GR_Node* lGrNode;                                                6
                                                                        7
    //Create der Basisklasse aufrufen                                   8
    SP_NT_Graph::Create(pGraph);                                        9
    //Jetzt haben wir die Definition von Graph uebernommen              10
                                                                        11
    //neue Knoten-Partition namens Knoten 2                             12
    lDsNode = pGraph->addNodeSort("Knoten 2");                          13
    //graphische Auspraegung des Prototypen (Rechteck)                  14
    lGrNode = (SP_GR_Node*)lDsNode->GetPrototype()->SetGraphic(         15
        new SP_GR_Rectangle(lDsNode->GetPrototype()));                  16
    //Initialgroesse                                                    17
    lGrNode->SetSizeVal(30, 30);                                        18
    //Fuellfarbe festlegen                                              19
    //false bedeutet, dass dieser Wert nicht geaendert werden darf      20
    lGrNode->SetBrushCol(SP_NAVY, false);                               21
                                                                        22
    return true;                                                        23
}                                                                       24
```

3.3 SP_NT_Samplegraph

Dieser Graph stellt die bisher umfangreichste Netzklasse in SNOOPY dar.

```
class SP_NT_SampleGraph::Create(SP_DS_Graph* pGraph)
{
    //Variablen
    SP_DS_NodeSort* lNodeSort;
    SP_DS_EdgeSort* lEdgeSort;
    SP_DS_Attribute* lDsAttr;
    SP_GR_Attribute* lGrAttr;
    SP_DS_Node* lDsNode;
    SP_GR_Node* lGrNode;
    SP_DS_Edge* lDsEdge;

    //Create der Basisklasse aufrufen
    SP_NT_Graph::Create(pGraph);
    //Jetzt haben wir die Definition von Graph uebernommen

    //*********************************************************
    //Knotenpartition Place aus Knoten gewinnen
    lNodeSort = pGraph->renameNodeSort("Knoten", "Place");
    lDsNode = lNodeSort->GetPrototype();
    //graphische Auspr"agung "uberschreiben
    lGrNode = (SP_GR_Node*)lDsNode->SetGraphic(
                new SP_GR_Circle(lDsNode));
    //Groesse, nicht aenderbar
    lGrNode->SetSizeVal(25, 25, false);
    //Farbe, aenderbar
    lGrNode->SetBrushCol(SP_WHITE, true);

    //neues Zahlenattribut namens ID
    lDsAttr = lNodeSort->AddAttribute(new SP_DS_IntegerAttribute("ID"));
    //graphische Auspraegung des Attributs
    lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
        new SP_GR_NumberAttribute(lDsAttr));
    //bewirkt, dass nur die Sichtbarkeit im Dialog beeinflusst werden kann
    ((SP_GR_NumberAttribute*)lGrAttr)->SetShowOnly(true);
    //Formatvorgabe, an % i wird der Wert eingetragen
    ((SP_GR_NumberAttribute*)lGrAttr)->SetFormatString("_P% i_");

    //Textattribut namens Label
    lDsAttr = lNodeSort->AddAttribute(new SP_DS_StringAttribute("Label"));
    lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
                new SP_GR_StringAttribute(lDsAttr, ""));
    //Startposition relativ zum Mittelpunkt des Knotens
    lGrAttr->SetPositionOffset(20, 20);

    //Zahlenattribut
    lDsAttr = lNodeSort->AddAttribute(new SP_DS_IntegerAttribute("Marking"));
```

```
//besondere graphische Auspraegung in Anlehnung an PED
lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
    new SP_GR_MarkAttribute(lDsAttr, "0"));
lGrAttr->SetPositionOffset(0, 0);

//Wahrheitswertattribut namens Logic, angelehnt an PED
lDsAttr = lNodeSort->AddAttribute(new SP_DS_BoolAttribute("Logical"));
//besondere graphische Auspraegung
lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
            new SP_GR_LogicAttribute(lDsAttr));
//Attribut, nach dessen Wert die Gleichheit entschieden werden soll
((SP_GR_LogicAttribute*)lGrAttr)->SetComparator("Label");

//*********************************************************
//neue Partition namens Transition
lNodeSort = pGraph->addNodeSort("Transition");
lDsNode = lNodeSort->GetPrototype();

lGrNode = (SP_GR_Node*)lDsNode->SetGraphic(
            new SP_GR_Rectangle(lDsNode));
lGrNode->SetSizeVal(25, 25, false);
lGrNode->SetBrushCol(SP_WHITE, false);

lDsAttr = lNodeSort->AddAttribute(new SP_DS_IntegerAttribute("ID"));
lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
            new SP_GR_NumberAttribute(lDsAttr));
((SP_GR_NumberAttribute*)lGrAttr)->SetShowOnly(true);
((SP_GR_NumberAttribute*)lGrAttr)->SetFormatString("_T% i_");

lDsAttr = lNodeSort->AddAttribute(new SP_DS_StringAttribute("Label"));
lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
            new SP_GR_StringAttribute(lDsAttr, ""));
lGrAttr->SetPositionOffset(20, 20);

lDsAttr = lNodeSort->AddAttribute(new SP_DS_BoolAttribute("Logical"));
lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(
    new SP_GR_LogicAttribute(lDsAttr));
((SP_GR_LogicAttribute*)lGrAttr)->SetComparator("Label");

//*********************************************************
//neue Knotenpartition namens Coarse Place
lNodeSort = pGraph->addNodeSort("Coarse Place");
lDsNode = lNodeSort->GetPrototype();

//Elemente dieser Sorte koennen nicht vereinigt werden
lDsNode->SetMergeable(false);
//Elemente dieser Sorte koennen nicht in Unternetze verschoben werden
lDsNode->SetCoarseable(false);

//Classattribute, fuer die Defintion des Unternetzes
lDsAttr = lNodeSort->AddAttribute(
```

```
                    new SP_DS_ClassAttribute(lDsNode, "SubNet",                    98
                        //das Unternetz enthaelt genau den gleichen Graphen        99
                        new SP_IO_SubNet_Def(pGraph), "", false));                 100
                                                                                   101
    //besondere graphische Auspraegung                                             102
    lGrNode = (SP_GR_Node*)lDsNode->SetGraphic(                                    103
                    new SP_GR_CoarsePlace(lDsNode,                                 104
                        pGraph->getNodeSort("Transition"), //Sorte der Randknoten  105
                        pGraph->getNodeSort("Place"))); //Sorte der inneren Knoten 106
    lGrNode->SetSizeVal(25, 25);                                                   107
    //dieses Attribut hat keine graphische Auspraegung                             108
    lDsAttr->SetGraphic(new SP_GR_NILAttribute(lDsAttr));                          109
                                                                                   110
    lDsAttr = lNodeSort->AddAttribute(new SP_DS_StringAttribute("Label"));         111
    lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(                               112
                    new SP_GR_StringAttribute(lDsAttr, ""));                       113
    lGrAttr->SetPositionOffset(20, 20);                                            114
                                                                                   115
    //*********************************************************                    116
    //CoarseTransition                                                             117
    lNodeSort = pGraph->addNodeSort("Coarse Transition");                          118
    lDsNode = lNodeSort->GetPrototype();                                           119
                                                                                   120
    lDsNode->SetMergeable(false);                                                  121
    lDsNode->SetCoarseable(false);                                                 122
                                                                                   123
    lDsAttr = lNodeSort->AddAttribute(                                             124
                    new SP_DS_ClassAttribute(lDsNode, "SubNet",                    125
                        new SP_IO_SubNet_Def(pGraph), "", false));                 126
                                                                                   127
    lGrNode = (SP_GR_Node*)lDsNode->SetGraphic(                                    128
                    new SP_GR_CoarseTransition(lDsNode,                            129
                        pGraph->getNodeSort("Place"),                              130
                        pGraph->getNodeSort("Transition")));                       131
    lGrNode->SetSizeVal(25, 25);                                                   132
    lDsAttr->SetGraphic(new SP_GR_NILAttribute(lDsAttr));                          133
                                                                                   134
    lAttr = lNodeSort->AddAttribute(new SP_DS_StringAttribute("Label"));           135
    lGrAttr = (SP_GR_Attribute*)lDsAttr->SetGraphic(                               136
                    new SP_GR_StringAttribute(lDsAttr, ""));                       137
    lGrAttr->SetPositionOffset(20, 20);                                            138
                                                                                   139
    //*********************************************************                    140
    //Kantenpartition umbenennen                                                   141
    lEdgeSort = pGraph->renameEdgeSort("Kante", "Arc");                            142
    lDsEdge = lEdgeSort->GetPrototype();                                           143
                                                                                   144
    lDsEdge->SetGraphic(new SP_GR_Line(lEdgeSort->GetPrototype()));                145
                                                                                   146
    return true;                                                                   147
};                                                                                 148
```

Dieser Graph entspricht in seinem Aussehen und seinem Verhalten weitestgehend dem in PED fest Implementierten. Wenn die Entwicklung abgeschlossen ist, wird dieser Graph als `SP_NT_PedNet` übernommen werden.

Im Folgenden wird die `EdgeRequirement`-Funktion dieser Netzklasse vorgestellt. Da das PED-Netz nur eine Kante implementiert, ist die Partition für die Bewertung der Zulässigkeit unerheblich. Als erstes erfolgt die Überprüfung, ob schon eine Kante von der Quelle zum Ziel existiert. Ist dies der Fall, wird die Kante verboten. An dieser Stelle sollte die Kante dennoch erlaubt werden, aber der Zähler für die Kantenbewertung dieser Kante um 1 erhöht werden.

Existiert noch keine Kante zwischen den beiden Knoten, wird überprüft, ob das Ziel oder die Quelle generell von einer Partition kommen, zu, oder von der, Kanetn erlaubt sind. Im PED-Netz sind zu den Textfeldern, die für Kommentare gedacht sind, keine Kanten erlaubt. Wenn auch dieser Test positiv ausfällt, bleibt nur noch, zu überprüfen, ob die Kante, wenn sie zu einem Unternetz führt, von einem Knoten kommt, der zur Partition der Randknoten dieses Unternetzes gehört. Für den Fall, daß die Kante aus einem Unternetzknoten kommt, erfolgt die Überprüfung, ob der Zielknoten zu den Randknoten dieses Unternetzes gehört.

```
bool                                                                    1
SP_NT_SampleGraph::EdgeRequirement                                      2
(SP_DS_EdgeSort* pEdge, SP_DS_Element* pSrc, SP_DS_Element* pDst)       3
{                                                                       4
    if (pSrc->HasSuccessor(pDst) || pDst->HasPredecessor(pSrc))         5
    {                                                                   6
        return false;                                                   7
    }                                                                   8
                                                                        9
    bool lSimple =                                                      10
            ((pSrc->GetSort()->GetName() != pDst->GetSort()->GetName())  11
            && (pSrc->GetSort()->GetName() != "Comment")                12
            && (pDst->GetSort()->GetName() != "Comment"));              13
                                                                        14
    if (lSimple)                                                        15
    {                                                                   16
        SP_GR_Element* lGrTest;                                         17
        lGrTest = (SP_GR_Element*)(pDst->GetGraphic());                 18
                                                                        19
        if (lGrTest->GetType() == SP_ELEM_COARSE)                       20
            return (pSrc->GetSort()->GetName() ==                       21
                ((SP_GR_Coarsenode*)lGrTest)->!>GetBorderSort()->GetName()); 22
                                                                        23
        lGrTest = (SP_GR_Element*)(pSrc->GetGraphic());                 24
                                                                        25
        if (lGrTest->GetType() == SP_ELEM_COARSE)                       26
            return (pSrc->GetSort()->GetName() ==                       27
                ((SP_GR_Coarsenode*)lGrTest)->GetBorderSort()->GetName()); 28
```

```
        }                                                    29
                                                             30
        return lSimple;                                      31
    }                                                        32
```

A Anhang

A.1 Präfixe

`sp-ana` `Snoopy-Analyse`:
Schnittstellen und Implementierungen für Analysemethoden

`sp-attr` `Snoopy-Attributes`:
Schnittstelle und Implementierungen aller Attribute der Datenstruktur

`sp-ds` `Snoopy-Datastructure`:
Schnitstellen und Implementierungen der Datenstruktur (Knoten, Kanten und deren Sorten, Graph)

`sp-gm` `Snoopy-Graphical Management`:
Implementierungen der Eventhandler

`sp-gr` `Snoopy-Graphic`:
Schnittstellen und Implementierungen graphischer Elemente, Kanten und Attribute

`sp-gui` `Snoopy-Graphical User Interface`:
graphische Oberfläche, Fenster, Dialoge, Menüs

`sp-io` `Snoopy-Input/Output`:
Schnittstellen und Implementierungen der Ein- Ausgabeformate

`sp-net` `Snoopy-Netclasses`:
Schittstellen und Implementierungen der Netzklassen

`sp-templ` `Snoopy-Templates`:
Implementierungen der SNOOPY-Liste, Defines

A.2 Datenstruktur

A.3 SP_DS_Error

erbt von: -
Als Basisklasse zur Haltung von menschenlesbaren Fehlerinformationen.

A.3.1 Methoden

SP_DS_Error (**void**)
Konstruktor. Initialisiert ausschliesslich die einzige Membervariable (`protected`) zu einer leeren Zeichenkette

---

const string&
GetLastError (**void**)
Liefert den letzten Fehler in der abgeleiteten Klasse.

Das Setzen einer Zeichenkette in die Variable erfolgt zumeist innerhalb der abgeleiteten Klasse. `SP_DS_Error` wurde eingeführt, um bei Funktionen, bei denen es für den Programmfluss unerheblich ist, aus welchem Grund sie fehlschlagen (Rückgabewert **bool**), eine Nachricht ausgeben zu können.

A.4 SP_DS_Name

erbt von: -
Als Basisklasse zur Verwaltung einer Zeichenkette, und deren Zugriffsfunktionen.

Man stelle sich zum Beispiel vor, dass eine Klasse ein Namensattribut haben soll, dessen Inhalt man erfragen (Get) oder setzen kann (Set). Durch das Erben von `SP_DS_Name` erhält jede Klasse automatisch die notwendigen Attribute und Methoden.

Neben der aktuellen Namensvariable ist das eine Liste, welche die sogenannten *Aliase* des Namens enthält. Ein Objekt kann also nicht nur einen Namen haben.

A.4.1 Methoden

SP_DS_Name (**void**)
SP_DS_Name (**const string&** `p_sName`)

p_sName Der Inhalt der Zeichenkette

Konstruktor. Initialisiert die Membervariable (`protected`) zu einer leeren Zeichenkette oder dem übergebenen Wert.

---

string
GetName (**void**)
Liefert den aktuellen Namen zurück.

---

string
SetName (**const string&** `p_sName`)

p_sName Der neue Name

Setzt den aktuellen Namen und fügt ausserdem den Wert in die Liste der *Aliase* ein, sofern er dort noch nicht existiert. Gibt den neuen Namen (also den Wert des Parameters) zurück.

---

bool
isAlias (**const string&** `p_sName`)

p_sName Der zu überprüfende Wert

Liefert **true** für den Fall, dass *p_sName* einem Alias entspricht, **false** sonst.

---

void
addAlias (**const string&** `p_sName`)

p_sName Der einzufügende Name

Fügt den Parameter in die Liste der Aliase ein, ohne den Wert des aktuellen Namens zu verändern.

A.5 SP_DS_Type

erbt von: -
Diese Klasse verwaltet die Kalssifizierung der Elemente in den Netzen nach ihrem Typ. Mittels einer, bei Bedarf erweiterbaren, Enumeration lässt sich so jedes Element nach seinem Typ unterscheiden, wenn nur Objekte dieser Basisklasse ausgetauscht werden.

Ausserdem enthält diese Klasse eine Zählvariable, die jeder abgeleiteten Klasse eine eineindeutige Numemr zuweisst.

A.5.1 Typdefinitionen

enum `SP_SORT_T`
{

```
    SP_NODESORT,
    SP_EDGESORT,
    SP_ATTRIB,
    SP_GRAPH,
    SP_NODE,
    SP_EDGE,
    SP_UNDEF,
};
```

A.5.2 Methoden

SP_DS_Type (SP_SORT_T pType = SP_UNDEF)

pType Typ aus der Enumeration SP_SORT_T

Konstruktor. Definiert eine neue Klasse aus dem Aufzählungstyp, inkrementiert den globalen Zähler der Elemente und weisst dessen Wert der Membervariable zu.

SP_IDCOUNTER_T
SetId (SP_IDCOUNTER_T pId)

pId Id des Objekts

Setzt die Variable Id auf den übergebenen Wert und setzt auch den anwendungsglobalen Zähler, wenn er kleiner als der Parameter ist, auf diesen Wert.

SP_IDCOUNTER_T
GetId (**void**)
Liefert den Wert des Zählers dieses Elements.

SP_SORT_T
GetIdentType (**void**)
Liefert den Eintrag aus dem Aufzählungstypen.

A.6 SP_DS_Parent

erbt von: -
Diese Klasse ist ein Sonderfall, der mit der schon erwähnten, gewählten Datenstruktur zu tun hat.

Die Datenstruktur-Attribute in SNOOPY können sowohl bei Knoten und Kanten, als auch bei Graphen in die Liste der zugehörigen Attribute eingefügt wer-

den. Um nun die Zugehörigkeit im Konstruktor der Attribute nicht beachten zu müssen und trotzdem die Möglichkeit zu haben, jederzeit zu entscheiden, auf welchen Typ Objekte dieses "*Parents*" gecastet werden können, erben Knoten, Kanten und Graphen von dieser Klasse. Der Parameter in Attributen, der das Element beschreibt, zu dem das Attribut gehört (den `Parent`), ist vom Typ `SP_DS_Parent`.

Das bedeutet im Umkehrschluss natürlich, dass Attribute nicht Kinder von anderen Attributen sein können. Es sei denn, eigene Attribute werden auch von dieser Klasse abgeleitet. Bei den im Moment implementierten Attributen ist das aber nicht der Fall.

A.6.1 Typdefinitionen

```
enum SP_DS_PARENT_TYPE
{
    SP_GRAPHPARENT_T,
    SP_EDGEPARENT_T,
    SP_NODEPARENT_T,
};
```

A.6.2 Methoden

SP_DS_Parent (`SP_DS_PARENT_TYPE pType`)

pType Typ aus der Enumeration `SP_DS_PARENT_TYPE`

Konstruktor. Definiert eine neue Klasse aus dem Aufzählungstyp. Der Aufruf des Konstruktors erfolgt explizit mit dem richtigen Typen in den davon abgeleiteten Klassen.

`SP_DS_PARENT_TYPE`
setType (`SP_DS_PARENT_TYPE pType`)

pType Parenttyp, also Knoten, Kante oder Graph

Setzt die Variable Type auf den übergebenen Wert.

`SP_DS_PARENT_TYPE`
getType (**void**)
Liefert den Typ des Objekts zurück.

---

virtual	`SP_DS_Attribute*`
GetAttribute	(**const string&** `pStr`) = 0
pStr	Name des Attributs

Virtuelle Funktion, die von allen abgeleiteten Klassen implementiert wird. Liefert das Attribut des Namens `pStr` oder `NULL`, falls es kein Attribut dieses Namens gibt.

A.7 SP_DS_Graph

erbt von: SP_DS_Error, SP_DS_Type, SP_DS_Name, SP_DS_Parent
Diese Klasse implementiert den Graphen.

A.7.1 Methoden

SP_DS_Graph	(SP_DS_Net* pNetClass, **const string&** pStr)
SP_DS_Graph	(SP_DS_Net* pNetClass)
SP_DS_Graph	(**void**)

pNetClass	Die Netzklasse, auf deren Definition der Graph basiert
pStr	Der Name des Graphen

Konstruktoren. Erstellt einen neuen Graphen der angegebenen Netzklasse und des Namens. Bei Verwendung des leeren Konstruktors muß vor der Arbeit mit dem Graphen wenigstens noch, in weiteren Schritten, die Netzklasse definiert werden.

---

bool	
setNetClass	(SP_DS_Net* pNetClass)

pNetClass	Die Netzklasse, die dieser Graph implementiert.

Setzt die Netzklasse dieses Graphen.

SP_NT_Net*	
getNetClass	(**void**)

Liefert die Netzklasse, zu der dieser Graph gehört.

SP_DS_NodeSort*	
addNodeSort	(**const string&** pStr)

pStr	Der Name der Partition

Erzeugt eine neue Knotenpartition und fügt diese dem Graphen hinzu. Hierzu reicht die Festlegung eines Namens. Der Aufruf des Konstruktors von SP_DS_NodeSort erfolgt automatisch. Die neue Partition wird zurückgegeben.

`SP_DS_EdgeSort*`
addEdgeSort (**const string&** `pStr`)

pStr Der Name der Partition

Erzeugt eine neue Kantenpartition und fügt diese dem Graphen hinzu. Hierzu reicht die Festlegung eines Namens. Der Aufruf des Konstruktors von `SP_DS_EdgeSort` erfolgt automatisch. Die neue Partition wird zurückgegeben.

`SP_DS_NodeSort*`
getNodeSort (**const string&** `pStr`)

pStr Der Name der gewünschten Partition

Liefert die Partition des angegebenen Namens zurück. Wenn keine solche Partition existiert, wird `NULL` zurück gegeben.

`SP_DS_EdgeSort*`
getEdgeSort (**const string&** `pStr`)

pStr Der Name der gewünschten Partition

Liefert die Partition des angegebenen Namens zurück. Wenn keine solche Partition existiert, wird `NULL` zurück gegeben.

`SP_DS_NodeSort*`
renameNodeSort(**const string&** `pScr`, **const string&** `pDst`)

pSrc Alter Name der Partition
pDst Neuer Name der Partition

Benennt die Partition namens `pScr` in `pDst` um. Liefert entweder die manipulierte Partition zurück oder `NULL`, falls es keine solche Partition gibt.

`SP_DS_EdgeSort*`
renameEdgeSort(**const string&** `pScr`, **const string&** `pDst`)

pSrc Alter Name der Partition
pDst Neuer Name der Partition

Benennt die Partition namens `pScr` in `pDst` um. Liefert entweder die manipulierte Partition zurück oder `NULL`, falls es keine solche Partition gibt.

`SP_DS_Attribute*`
AddAttribute (`SP_DS_Attribute* pAttr`)

pAttr Das hinzuzufügende Attribut

Fügt der Attributliste des Graphen das angegebene Attribut hinzu, sofern dieses noch nicht existiert und gibt es wieder zurück.

`SP_DS_Attribute*`
GetAttribute (**const string&** `pStr`)

pStr Name des Attributs

Liefert das Attribut des angegebenen Namens zurück. **NULL**, falls es kein Attribute dieses Namens gibt.

A.8 SP_DS_NodeSort

erbt von: `SP_DS_Error`, `SP_DS_Type`, `SP_DS_Name`
Diese Klasse implementiert eine Knotenpartition des Graphen. Sie enthält einen prototypischen Knoten und eine Liste aller Knoten, welche im Graphen, während der Arbeit, erzeugt werden.

Die konkreten Knoten zur Arbeitszeit werden dabei durch eine identische Kopie aus dem Prototypen erzeugt. Die Netzklassendefinition manipuliert dazu also die Attribute und die graphische Ausprägung dieses Prototypen.

A.8.1 Methoden

SP_DS_NodeSort (`SP_DS_Graph* pParent, const string& pStr`)

pParent	Der Graph, zu dem diese Partition gehört
pStr	Der Name dieser Partition

Konstruktor. Erstellt eine neue Knotenpartition des angegebenen Namens.

Wird in der Regel indirekt über die Verwendung der Methode `SP_DS_Graph::addNodeSort` aufgerufen.

`SP_DS_Node*`
GetPrototype (**void**)

Liefert das prototypische Element zurück.

`SP_DS_Attribute*`
AddAttribute (`SP_DS_Attribute* pAttr`)

pAttr	Das hinzuzufügende Attribut

Fügt der Attributliste des Prototypen das angegebene Attribut hinzu, sofern dieses noch nicht existiert und gibt es wieder zurück.

`SP_DS_Attribute*`
GetAttribute (**const string&** `pStr`)

pStr	Name des zu suchenden Attributs

Gibt das erste Attribute dieses Namens zurück, wenn der Prototyp ein solches besitzt, **NULL** sonst.

---

bool
HasAttribute (**const string&** pStr)

pStr	Name des Attributs

Liefert **true** für den Fall, daß der Prototyp ein Attribute dieses Namens enthält, **false** sonst.

SP_DS_Attribute*
RenameAttribute (**const string&** pSrc, **const string&** pDest)

pSrc	bisheriger Name des Attributs
pDest	neuer Name des Attributs

Benennt das Attribut des Prototypen namens *pSrc* in *pDest* um. Liefert das Attribut zurück, bzw. **NULL**, falls es kein Attribut dieses Namens gibt.

---

bool
RemoveAttribute(**const string&** pStr)

pStr	Name des Attributs

Löscht das erste Attribut des angegebenen Namens aus der Liste der Attribute des Prototyps.

SP_DS_Node*
newElement (**int** pNetNumber = 1, **bool** pCloneAll = **true**)

pNetNumber	Nummer des Unternetzes in dem dieser Knoten erzeugt werden soll
pCloneAll	Steuert, ob die graphische Ausprägung des Prototypen auch kopiert werden soll

Erzeugt einen neuen Knoten nach dem Vorbild des Prototypen, trägt diesen in die Liste der Knoten ein und liefert ihn zurück.

SP_DS_Node*
getElement (**SP_IDCOUNTER_T** id)

id	ID des zu suchenden Elements dieser Sorte.

Liefert das Element dieser Sort, welches die angegebene ID hat, oder **NULL**.

`SP_GR_Interface*`
SetGraphic (**SP_GR_Interface*** pGr, **int** pNr = 1)

pGr	Graphische Ausprägung, die dem Prototypen zugewiesen wird.
pCloneAll	Netznummer, die dieser Graphik zugeordnet wird.

Setzt den Parameter als Vorlage der graphischen Ausprägung des Prototypen dieser Sorte.

A.9 SP_DS_EdgeSort

erbt von: `SP_DS_Error`, `SP_DS_Type`, `SP_DS_Name`
Diese Klasse implementiert eine Kantenpartition des Graphen. Sie enthält eine prototypische Kante und eine Liste aller Kanten, welche im Graphen, während der Arbeit, erzeugt werden.

Jede Kante ist dabei eine identische Kopie des Prototypen, der in der `Create`-Methode der Netzklasse manipuliert wurde.

A.9.1 Methoden

SP_DS_EdgeSort (`SP_DS_Graph* pParent, const string& pStr`)

pParent	Der Graph, zu dem diese Partition gehört
pStr	Der Name dieser Partition

Konstruktor. Erstellt eine neue Kantenpartition des angegebenen Namens.

Wird in der Regel indirekt über die Verwendung der Methode `SP_DS_Graph::addEdgeSort` aufgerufen.

`SP_DS_Edge*`
GetPrototype (**void**)

Liefert das prototypische Element zurück.

`SP_DS_Attribute*`
AddAttribute (`SP_DS_Attribute* pAttr`)

pAttr	Das hinzuzufügende Attribut

Fügt der Attributliste des Prototypen das angegebene Attribut hinzu, sofern dieses noch nicht existiert und gibt es wieder zurück.

`SP_DS_Attribute*`
GetAttribute (**const string&** `pStr`)

pStr Name des zu suchenden Attributs

Gibt das erste Attribute dieses Namens zurück, wenn der Prototyp ein solches besitzt, **NULL** sonst.

---

bool
HasAttribute (**const string&** `pStr`)

pStr Name des Attributs

Liefert **true** für den Fall, daß der Prototyp ein Attribute dieses Namens enthält, **false** sonst.

`SP_DS_Attribute*`
RenameAttribute (**const string&** `pSrc`, **const string&** `pDest`)

pSrc bisheriger Name des Attributs
pDest neuer Name des Attributs

Benennt das Attribut des Prototypen namens *pSrc* in *pDest* um. Liefert das Attribut zurück, bzw. **NULL**, falls es kein Attribut dieses Namens gibt.

---

bool
RemoveAttribute(**const string&** `pStr`)

pStr Name des Attributs

Löscht das erste Attribut des angegebenen Namens aus der Liste der Attribute des Prototyps.

`SP_DS_Edge*`	
newEdge	(**int** pNetNumber = 1, **bool** pCloneAll = **true**)
	(`SP_DS_Node* pSrc, SP_DS_Node* pDest`)
pNetNumber	Nummer des Unternetzes in dem diese Kante erzeugt werden soll
pCloneAll	Steuert, ob die graphische Ausprägung des Prototypen auch kopiert werden soll
pSrc	Knoten, von dem diese Kante ausgeht.
pDest	Knoten, an dem diese Kante endet.

Erzeugt eine neue Kante nach dem Vorbild des Prototypen, trägt diese in die Liste der Kanten ein und liefert sie zurück. Die zweite Version ruft auch die `EdgeRequirement`-Funktion auf.

`SP_DS_Edge*`	
getEdge	(**SP_IDCOUNTER_T** id)
id	ID der zu suchenden Kante dieser Sorte.

Liefert die Kante in dieser Sort, welche die angegebene ID hat, oder **NULL**, wenn es keine in dieser Partition gibt.

`SP_GR_Interface*`	
SetGraphic	(**SP_GR_Interface*** pGr, **int** pNr = 1)
pGr	Graphische Ausprägung, die dem Prototypen zugewiesen wird.
pNr	Netznummer, die dieser Graphik zugeordnet wird.

Setzt den Parameter als Vorlage der graphischen Ausprägung des Prototypen dieser Sorte.

A.10 SP_DS_Attribute

erbt von: `SP_DS_Error`, `SP_DS_Type`, `SP_DS_Name`
Diese Klasse implementiert die grundlegenden Methoden aller Attribute und ist außerdem Basis für alle konkreten Implementierungen. Momentan implementierte Attribute sind von einem der folgenden Typen:

A.10.1 Typdefinitionen

```
enum SP_DS_AttributeTypes
{
      SP_DS_UNDEF_ATTR,
      SP_DS_STRING_ATTR,
      SP_DS_INT_ATTR,
      SP_DS_BOOL_ATTR,
      SP_DS_INTINTERVAL_ATTR,
      SP_DS_PASSATTR,
      SP_DS_CLASS_ATTR,
      SP_DS_INTLIST_ATTR,
      SP_DS_ELEMENT_ATTR,
      SP_DS_INTVECTOR_ATTR,
      SP_DS_STRINGVECTOR_ATTR,
      SP_DS_LONGVECTOR_ATTR,
      SP_DS_ATTRIB_ATTR,
      SP_DS_ELEMENTLIST_ATTR,
      SP_DS_EDGELIST_ATTR,
};
```

Im Abschnitt auf Seite 44 werden einige dieser Attribute etwas näher erklärt. Die folgenden Methoden sind alle in dieser Basisklasse implementiert und bedürfen eigentlich, für den normalen Einsatz, keiner Reimplementierung, obwohl sie alle als `virtual` deklariert sind, was diese Möglichkeit also offen läßt.

A.10.2 Methoden

SP_DS_Attribute (`SP_DS_Parent* pParent, const string& pStr, const string& pRem, SP_DS_AttributeTypes pType`)
(`SP_DS_Parent* pParent, const string& pStr, SP_DS_AttributeTypes pType`)

(`SP_DS_Parent* pParent, SP_DS_AttributeTypes pType`)
(`SP_DS_AttributeTypes pType`)

pParent	Das Element, die Kante, oder der Graph zu dem dieses Attribut gehört
pStr	Der Name des Attributs
pRem	Remark, Bemerkung
pType	Typ aus der Enumeration `SP_DS_AttributeTypes`

Konstruktoren. Erstellen ein neues Attribut.

`SP_GR_Interface*`
SetGraphic (**SP_GR_Interface*** `pGraphic`, **int** *pNetNumber* = 0)

pGraphic	Graphische Ausprägung
pNetNumber	Unternetznummer

Assoziiert die graphische Ausprägung mit dem Attribut. Da von `SP_GR_Interface` keine Objekte erzeugt werden können, muß dieser Parameter ein Objekt sein, welches von `SP_GR_Attribute` abgeleitet ist. Die Netznummer ist zur Defintion der Netzklasse nicht notwendig, sie wird erst während der Arbeit mit Graphen dieser Netzklasse benötigt. Diese Methode liefert das Objekt der graphischen Ausprägung zurück.

---

bool
TestValidGraphic (`SP_TYPE_ATTRIBUTE` *pType*)

pType	typ der graphischen Ausprägung

Die Aufgabe dieser Methode ist es, die Gültigkeit von zugewiesenen graphischen Ausprägungen zu validieren. In der Basisklasse, die hier gerade erläutert wird, liefert diese Methode immer `true`. Einige der abgeleiteten, konkreten Attribute allerdings validieren den übergebenen Typen und entscheiden, ob die dazugehörige graphische Ausprägung in die Liste aufgenommen wird. Somit soll vermieden werden, dass zum Beispiel ein Datenstrukturattribut, welches Zeichenketten verwalten soll, als graphische Ausprägung die Darstellung von Integerwerten erhält.

Unabhängig davon, ob diese Methode in abgeleiteten Klassen neu implementiert ist, wird sie in der Methode `SP_DS_Attribute::SetGraphic` verifiziert. Allerdings ist in der vorliegenden Version die Rückmeldung eines Fehlerzustands noch nicht sehr weit ausgearbeitet. Im *worst case* endet ein Datenstrukturattribut, welchem der Nutzer eine nicht erlaubte graphische Ausprägung zuweisen will, mit einem `NULL` Pointer an der entsprechenden Stelle. (Stand: Dez. 2002)

---

virtual string
SetValueString (**const string&** `pVal`) = 0

pVal Der zu setzende Wert des Attributs in einem String

Setzt den Wert des Attributs. Da die Klasse selbst keine Werte speichert, dient diese Methode nur der Erleichterung der Schreibweise und muß von allen abgeleiteten Klassen implementiert werden!

---

virtual string
GetValueString (**void**) = 0

Liefert den Wert dieses Attributs als Stringrepräsentation zurück. Diese Methode muß von allen abgeleiteten Klassen implementiert werden und interagiert mit `SetValueString`, um Werte mehrerer Attribute zu setzen, ohne deren Typ zu kennen.

`SP_DS_Parent*`
SetParent (`SP_DS_Parent* pPar`)

pPar Das Elternobjekt

Setzt das Elternobjekt (Knoten, Kante oder Graph) des Attributs.

`SP_DS_Parent*`
GetParent (**void**)

Liefert den Knoten, die Kante oder den Graphen, zu dem dieses Attribut gehört.

---

virtual bool
Equal (`SP_DS_Attribute`* pAttr)

pAttr Vergleichsattribut

Liefert **true** für den Fall, daß das übergebene Attribut den gleichen Wert enthält und vom selben Typ ist, **false** sonst. In dieser Klasse liefert die Methode schon bei Typgleichheit **true**. In abgeleiteten Klassen wird diese Methode überschrieben.

A.11 SP_DS_Node

erbt von: `SP_DS_Error`, `SP_DS_Type`, `SP_DS_Parent`
Diese Klasse implementiert die Knoten des Graphen.

A.11.1 Methoden

SP_DS_Node	(`SP_DS_Graph* pParent, SP_DS_NodeSort* pGrp, bool pMerge = true, bool pCoarse = true`)
pParent	Der Graph, zu dem dieser Knoten gehört
pGrp	Die Partition zu der dieser Knoten gehört
pMerge	Knoten kann mit Elementen der selben Partition vereinigt werden
pCoarse	Knoten kann in Unternetze verschoben werden

Konstruktor. Erstellt einen neuen Knoten der angegebenen Partition als Element des angegebenen Graphen.

`SP_DS_Attribute*`	
AddAttribute	(`SP_DS_Attribute* pAttr`)
pAttr	Das hinzuzufügende Attribut

Fügt der Attributliste des Knoten das angegebene Attribut hinzu, sofern dieses noch nicht existiert und gibt es wieder zurück.

---

bool	
HasAttribute	(**const string&** `pStr`) (**SP_DS_AttributeTypes** `pType`, **const string&** `pStr`)
pStr	Name des Attributs
pType	Attributtyp

Liefert **true** für den Fall, daß der Knoten ein Attribute dieses Namens enthält, **false** sonst. Die zweite Version liefert **true** für den Fall, daß der Knoten ein Attribute dieses Typs und dieses Namens enthält, **false** sonst.

`SP_DS_Attribute*`	
GetAttribute	(**const string&** `pStr`)
pStr	Name des Attributs

Liefert das Attribut des angegebenen Namens zurück. **NULL**, falls es kein At-

tribut dieses Namens gibt.

`SP_DS_Attribute*`
RenameAttribute (**const string&** `pSrc`, **const string&** `pDest`)

pSrc	bisheriger Name des Attributs
pDest	neuer Name des Attributs

Benennt das Attribut namens *pSrc* in *pDest* um. Liefert das Attribut zurück, bzw. **NULL**, falls es kein Attribut dieses Namens gibt.

`void`
RemoveAttribute (**const string&** `pStr`)

pStr	Name des Attributs

Entfernt das Attribut des Namens *pStr* aus der Liste der Attribute dieses Knotens.

`SP_GR_Interface*`
SetGraphic (**SP_GR_Interface*** `pGraphic`, **int** *pNetNumber* = 0)

pGraphic	Graphische Ausprägung
pNetNumber	Unternetznummer

Assoziiert die graphische Ausprägung mit dem Knoten. Da von `SP_GR_Interface` keine Objekte erzeugt werden können, muß dieser Parameter ein Objekt sein, welches von `SP_GR_Node` abgeleitet ist. Die Netznummer ist zur Defintion der Netzklasse nicht notwendig, sie wird erst während der Arbeit mit Graphen dieser Netzklasse interessant. Die Methode liefert das Objekt der graphischen Ausprägung zurück.

A.12 SP_DS_Edge

erbt von: `SP_DS_Error`, `SP_DS_Type`, `SP_DS_Parent`
Diese Klasse implementiert die Kanten des Graphen.

A.12.1 Methoden

SP_DS_Edge	(`SP_DS_Graph* pParent, SP_DS_EdgeSort* pGrp`)
SP_DS_Edge	(`SP_DS_Graph* pParent, SP_DS_EdgeSort* pGrp, SP_DS_Node* pSrc, SP_DS_Node* pDest`)

pParent	Der Graph, zu dem diese Kante gehört
pGrp	Die Partition zu der diese Kante gehört
pSrc	Knoten, von dem diese Kante ausgeht.
pDest	Knoten, an dem diese Kante endet.

Konstruktoren. Erstellt eine neue Kante der angegebenen Partition als Element des angegebenen Graphen. Die Angabe der Quell- und Zielknoten kann im Konstruktor erfolgen, oder in zwei Schritten, mit `SetNodes` oder `SetSource` und `SetDestination`.

`SP_DS_Attribute*`	
AddAttribute	(`SP_DS_Attribute* pAttr`)

pAttr	Das hinzuzufügende Attribut

Fügt der Attributliste des Knoten das angegebene Attribut hinzu, sofern dieses noch nicht existiert und gibt es wieder zurück.

---

bool	
HasAttribute	(**const string&** `pStr`)
	(**SP_DS_AttributeTypes** `pType`, **const string&** `pStr`)

pStr	Name des Attributs

Liefert **true** für den Fall, daß der Knoten ein Attribute dieses Namens enthält, **false** sonst. Die zweite Version liefert **true** für den Fall, daß der Knoten ein Attribute dieses Typs und dieses Namens enthält, **false** sonst.

`SP_DS_Attribute*`	
GetAttribute	(**const string&** `pStr`)

pStr	Name des Attributs

Liefert das Attribut des angegebenen Namens zurück. **NULL**, falls es kein At-

tribut dieses Namens gibt.

`SP_DS_Attribute*`
RenameAttribute (**const string&** `pSrc`, **const string&** `pDest`)

pSrc	bisheriger Name des Attributs
pDest	neuer Name des Attributs

Benennt das Attribut namens *pSrc* in *pDest* um. Liefert das Attribut zurück, bzw. **NULL**, falls es kein Attribut dieses Namens gibt.

---

void
RemoveAttribute (**const string&** `pStr`)

pStr	Name des Attributs

Entfernt das Attribut des Namens *pStr* aus der Liste der Attribute dieses Knotens.

`SP_GR_Interface*`
SetGraphic (**SP_GR_Interface*** `pGraphic`, **int** `pNetNumber` = 0)

pGraphic	Graphische Ausprägung
pNetNumber	Unternetznummer

Assoziiert die graphische Ausprägung mit der Kante. Da von `SP_GR_Interface` keine Objekte erzeugt werden können, muß dieser Parameter ein Objekt sein, welches von `SP_GR_Edge` abgeleitet ist. Die Netznummer ist zur Defintion der Netzklasse nicht notwendig, sie wird erst während der Arbeit mit Graphen dieser Netzklasse interessant. Die Methode liefert das Objekt der graphischen Ausprägung zurück.

---

bool
SetNodes (`SP_DS_Node* pSrc, SP_DS_Node* pDest`)

pSrc	Quelle
pDest	Ziel

Setzt den Quell- und Zielknoten für die Kante. Dabei wird die `EdgeRequirement`-Funktion der Netzklasse zur Überprüfung, ob diese Kante erlaubt ist, herangezogen. Liefert **true** für den Erfolgsfall, **false** sonst.

---

void
SetSource (`SP_DS_Node* pSrc`)

pSrc Quellknoten

Setzt den Quellknoten der Kante. Achtung: dabei wird nicht überprüft, ob dieser Knoten als Ausgangspunkt der Kante erlaubt ist!

---

void
SetDestination (`SP_DS_Node* pDest`)

pDest Zielknoten

Setzt den Zielknoten der Kante. Achtung: dabei wird nicht überprüft, ob dieser Knoten als Endpunkt der Kante erlaubt ist!

`SP_DS_Node*`
GetSource (**void**)

Liefert den Startknoten der Kante zurück.

`SP_DS_Node*`
GetDestination (**void**)

Liefert den Zielknoten der Kante zurück.

A.13 Attribute

Die Attribute sind ein elementarer Bestandteil von SNOOPY und werden in diesem Abschnitt gesondert vorgestellt. Grundphilosophie ist es, für alle denkbaren Datentypen oder zu beschreibenden Verhaltensweisen eine eigene Klasse zu implementieren, die von `SP_DS_Attribute` (siehe Seite 36) abgeleitet wird und sowohl Funktionalitäten übernimmt, als auch einige wichtige Methoden selbst implementieren *muss*.

Prinzipielle Eigenschaft eines jeden Attributs ist es, die ihm zugewiesene Datenstruktur verwalten zu können, sprich die Accessormethoden `Set` und `Get` zu implementieren. Die Kopplung und die Interaktion mit den graphischen Ausprägungen erfolgt, ebenso wie das Speichern und Lesen in eine Datei über Methoden der Basisklasse `SP_DS_Attribute.`

Ich werde in diesem Abschnitt deshalb nicht auf die Methoden und ihre Parameter eingehen, sondern nur die wichtigsten strukturellen Attribute vorstellen, deren Datentypen nennen und welche graphischen Ausprägungen sie akzeptieren.

A.14 SP_DS_StringAttribute

Datentyp:	**string**
SP_DS_Type	`SP_DS_STRING_ATTR`
Graphik:	`SP_GR_StringAttribute`, `SP_GR_NILAttribute`
Besonderheiten:	unterstützt das `ShowOnly`-Flag

Das wohl am häufigsten benutzte Attribut. Dient zur Speicherung von Zeichenketten im `C++` Typen **string**. Die erste graphische Ausprägung unterstützt dabei sowohl die Darstellung und Bearbeitung mehrzeiliger, wie auch einzeiliger Zeichenketten.

A.15 SP_DS_IntegerAttribute

Datentyp:	**int**
SP_DS_Type	`SP_DS_INT_ATTR`
Graphik:	`SP_GR_NumberAttribute`, `SP_GR_MarkAttribute`, `SP_GR_TokenAttribute`, `SP_GR_NILAttribute`
Besonderheiten:	unterstützt das `ShowOnly`-Flag

A.16 SP_DS_BoolAttribute

Datentyp:	**bool**
SP_DS_Type	SP_DS_BOOL_ATTR
Graphik:	SP_GR_BoolAttribute, SP_GR_LogicAttribute, SP_GR_NILAttribute
Besonderheiten:	unterstützt das ShowOnly-Flag

A.17 SP_DS_PassAttribute

Datentyp:	**string**
SP_DS_Type	SP_DS_PASSATTR
Graphik:	SP_GR_PassAttribute, SP_GR_NILAttribute
Besonderheiten:	-

Dieses Attribute ist im Prinzip ein einzeiliges StringAttribute, mit dem Unterschied, dass es im Dialog zwei Eingabefelder darstellt, die beide den bekannten Passworteingabefeldern nachempfunden sind, die Eingabe also mit * maskiert wird. Dieses Attribut aktualisiert seinen Wert nur, wenn beide Eingabefelder im Dialog den gleichen Inhalt haben.

A.18 Graphik

Die graphischen Ausprägungen für Knoten, Kanten und Attribute (also die Elemente eines Netzes) folgen im Prinzip dem Aufbau der strukturellen Pendants, mit dem Unterschied, dass sie dazu dienen, die Werte der Datenstruktur in der Fensterumgebung darzustellen und editierbar zu machen.

Graphische Ausprägungen sind immer an ein Element des Graphen gebunden. Die Graphik eines Knotens wird zum Beispiel in der Liste der graphischen Ausprägungen des Objekts vom Typ `SP_DS_Node` gehalten, analog mit den Kanten und Attributen. Jede der Klassen implementiert eine `SetGraphic`-Methode, welche die Verbindung initialisiert und in der Netzklassendefinition verwendet wird.

Alle graphischen Ausprägungen erben von der Klasse `SP_GR_Interface`, die auch als Typ an den Accessormethoden in der Datenstruktur verwendet wird. Sie enthält eine Membervariable, die den Typ des Objekt nach Knoten, Kante oder Attribut klassifiziert. Ebenfalls analog zur Datenstruktur erfolgt die weitere Verfeinerung in die Klassen `SP_GR_Node`, `SP_GR_Edge` und `SP_GR_Attribute`, wobei sich die Attribute wiederum in verschiedene konkrete Attribute verfeinern.

Im Unterschied zur Datenstruktur, wo es für die Abbildung eines Knotens oder einer Kante nur je einer Klasse bedarf, ist es in der Graphik erforderlich, auch für die strukturell gleichen Objekte verschiedene Klassen zu implementieren. SNOOPYist in der vorliegenden Version schon mit einer Vielzahl von graphischen Ausprägungen ausgestattet, welche allerdings ebenso leicht, wie neue Attribute geschrieben werden können, erweitert werden können.

Da es vom entwicklungstechnischen Standpunkt keinen Sinn machte, die Interaktion und die Darstellung der Graphik mit dem Betriebssystem neu zu implementieren, wurde die Wahl getroffen, eine freie Bibliothek, ein sogenanntes GUI-Toolkit zu verwenden. Mit WXWINDOWS fiel die Wahl dabei auf ein freies Werkzeug, welches in grossem Maße plattformunabhängiges Arbeiten erlaubt und ausserdem von einer engagierten *Community* am Leben erhalten und aktiv weiter entwickelt wird. Damit basieren also alle graphischen Ausprägungen im Kern auch auf den Funktionen von WXWINDOWS.

Eine Besonderheit der in WXWINDOWS verwendeten Graphikbibliothek ist die Kopplung von graphischen Objekten mit sogenannten Eventhandlern, welche Methoden als `virtual` defniieren und diese aus der Bibliothek heraus, zum Beispiel beim Auftreten eines Klicks mit der Maus auf ein Objekt, aufrufen. Dem Programmierer steht es frei, diese Methoden in abgeleiteten Klassen zu überschreiben und so auf Ereignisse in der Interaktion mit dem Nutzer zu reagieren.

Aus diesem Grund existieren in SNOOPYsozusagen Äquivalente zu den Eventhandlern in WXWINDOWS. Im Verzeichnis `sp-gm` finden sich die Klassen für Knoten, Kanten und Attribute, die alle im Prinzip die gleichen Methoden implementieren und von `wxShapeEvtHandler` erben. Die Bindung der Eventhandler an die Objekte erfolgt in der vorliegenden Version im Konstruktor des graphischen

Objekts und es ist (noch) nicht vorgesehen, diese Bindung im Nachhinein zu ändern. Im Augenblick ist die vorhandene Funktionalität aber ausreichend und bedarf keiner solcher Anforderungen.

Die folgenden Abschnitte geben einen Überblick über die zur Verfügung stehenden Klassen, ohne im Detail auf jede Funktion einzugehen.

A.19 SP_GR_Interface

Die Basisklasse jeder graphischen Ausprägung. Sie definiert im Grossen und Ganzen einzig die Methoden, die den kleinsten gemeinsamen Nenner der abgeleiteten Klassen bilden und unterscheidet ausserdem ihre Ableitungen nach ihrem Typ:

A.19.1 Typdefinitionen

```
enum SP_TYPES
{
      SP_T_NODE,
      SP_T_EDGE,
      SP_T_ATTRIB,
      SP_T_UNDEFINED,
};
```

Hinzu kommt die Haltung von globalen Werten, die jeder Graphik eigen sind, so zum Beispiel die Nummer des Netzes, in der das konkrete graphische Element dargestellt werden soll. Das gewinnt allerdings erst an Bedeutung, wenn man sich Netze vorstellt, die mehrere "Ebenen" haben können.

Die Intention, diese Klasse einzuführen resultierte aus der Überlegung, die Kommunikation aus der Datenstruktur einzig darüber abzuwickeln. Zum Einen gab es Anfangs Probleme, die auch von wxWindows abgeleiteten Klassen in der Datenstruktur zu verwenden, zum Anderen sollte die Kopplung der beiden Hauptbestandteile von Snoopy so locker wie möglich sein, um bei Bedarf die Graphikbibliothek leicht austauschen zu können.

A.20 SP_GR_Node

Die Basisklasse aller Knoten eines Graphen ist einzig von `SP_GR_Interface` abgeleitet. Da diese Klasse keinerlei Funktionalität von wxWindows übernimmt, können keine Objekte von ihr dargestellt werden. Sie enthält wiederum eine Typisierung der zur Verfügung stehenden Knoten und dient als Container für die Verweise auf die Elemente in der Datenstruktur.

A.20.1 Typdefinitionen

```
enum SP_TYPE_NODE
{
    SP_NODE_UNDEFINED,
    SP_NODE_RECTANGLE,
    SP_NODE_CIRCLE,
    SP_NODE_TEXT,
    SP_NODE_COARSE,
    SP_NODE_DRAWNCOARSE,
    SP_NODE_BMPCOARSE,
    SP_NODE_BITMAP,
};
```

A.20.2 wichtige Methoden

Die folgenden Methoden sind in allen abgeleiteten Klassne verfügbar und beeinflussen das Aussehen und das Verhalten der Objekte.

bool	
SetWidth	(**double** `pWidth`, **bool** pAlways)
SetHeight	(**double** `pHeight`, **bool** pAlways)
SetSizeVal	(**double** `pWidth`, **double** `pHeight`, **bool** pAlways)
pWidth	Breite
pWidth	Höhe
pAlways	Indikator, ob dieser Wert änderbar sein soll

Diese drei Methoden setzen die Werte für die Breite und Höhe des Knotens. Der boolsche Parameter setzt die Möglichkeit, diese Werte zur Laufzeit zu manipulieren. Ein Wert von **true** zeigt sich zum Beispiel in der Anwendung durch die Anzeige von 8 Kontrollpunkten an den Ecken und den Längsseiten der Knoten, an denen das Objekt mit der Maus skaliert werden kann. Ist diese Möglichkeit ausgeschalten, werden die Objekte nur mit einem dickeren Rand selektiert und der Nutzer kann sie nicht in der Grösse ändern.

bool	
SetPenWidth	(**int** `pWidth`, **bool** pAlways)
pWidth	Breite
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Dicke des zur Darstellung verwendeten "Stifts". Also die Breite mit der die Umrandungslinien gezogen werden.

bool	
SetPenCol	(**int** `pPen`, **bool** pAlways)
SetPenCol	(`wxColour` `pPen`, **bool** pAlways)
pPen	Die Farbe, entweder ein Palettenindex, oder ein `wxColour`-Objekt
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Farbe des zur Darstellung verwendeten "Stifts". Das `wxColour`-Objekt kann entweder selbst erzeugt werden, oder aus der Liste der vordefinierten Farben in der Datei `sp_defines.h` entnommen werden.

bool	
SetBrushCol	(**int** `pPen`, **bool** pAlways)
SetBrushCol	(`wxColour` `pPen`, **bool** pAlways)
pPen	Die Farbe, entweder ein Palettenindex, oder ein `wxColour`-Objekt
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Farbe des zur Darstellung verwendeten Pinsels, also die Füllfarbe des Objekts. Das `wxColour`-Objekt kann entweder selbst erzeugt werden, oder aus der Liste der vordefinierten Farben in der Datei `sp_defines.h` entnommen werden.

A.20.3 abgeleitete Klassen

SP_GR_Rectangle

Diese Klasse implementiert ein einfaches Rechteck. Sie erbt neben `SP_GR_Node` auch von `wxRectangleShape` und ist damit direkt zur Darstellung in Netzen verwendbar.
verwendeter Eventhandler: `SP_GR_ShapeHandler`

SP_GR_Circle

Diese Klasse implementiert einen einfachen Kreis. Sie erbt neben `SP_GR_Node` auch von `wxEllipseShape` und ist damit direkt zur Darstellung in Netzen verwendbar. Genau genommen ist diese Klasse sogar eine Ellipse, wie an der Vererbung zu sehen ist.
verwendeter Eventhandler: `SP_GR_ShapeHandler`

SP_GR_Bitmap

Diese Klasse implementiert ein Icon als Repräsentation eines Knotens. Sie erbt neben `SP_GR_Node` auch von `wxBitmapShape` und ist damit direkt zur Darstellung in Netzen verwendbar.

Diese Klasse erhält im Konstruktor einen Pointer auf ein Array of **char**, in welchem im .xpm-Format die graphischen Informationen abgelegt sind. In SNOOPY ist im Augenblick genau eine Grafik eincompiliert, welche verwendet wird, falls der entsprechende Parameter ungültig sein sollte. Eine Alternative liegt in der Verwendung der Methode
`SP_DS_Data::GetImage(string pPath, string pFile, string pDrive)`
über das Applikationsglobale Objekt `g_SP_DS_AppData`. Sie erhält als Parameter den Pfad, den Dateinamen und optional den Laufwerksbuchstaben zu der Datei, die geladen werden soll. Dabei wird in dem Pfad '/' als Pfadseparator verwendet und die Datei muss eine .xpm-Datei sein.
verwendeter Eventhandler: `SP_GR_ShapeHandler`

SP_GR_Coarsenode

Diese Klasse ist die Basisklasse für alle Coarsenodes.
verwendeter Eventhandler: -

Die von dieser Klasse abgeleiteten Klassen sind `SP_GR_BitmapCoarsenode` und `SP_GR_DrawnCoarsenode`. Diese weitere Schicht hat sich als sinnvoll herausgestellt, da hier die wichtigen Methoden für bestimmte Ereignisse implementiert werden konnten, während sich in den abgeleiteten Klassen nur noch um die Darstellung gekümmert wird.

Die erwähnten Ereignisse sind das Anfügen von eingehenden oder ausgehenden Kanten an diese Knoten, was eine Reihe von Abläufen nach sich zieht, die in keiner anderen Knotenklasse sonst gebraucht wird, und auch für den Netzklassenadministrator nicht weiter von Bedeutung sind.

Als wichtigste Methode beim Erzeugen von Coarsenodes möchte ich nur eine einzige nennen, die sich auf die Darstellung bezieht. Die im Folgenden erwähnten Klassen überschreiben diese im `protected`-Teil der Klasse definierte Funktion. In ihr wird sozusagen der Knoten gemalt.

---

void
MyDraw (**bool** `pSelect` = `FALSE`) = 0

pSelect Status "Selektiert"?

Diese Methode wird in abgeleiteten Klassen implementiert, um den Knoten zu zeichnen.

Von dieser Klasse können keine Objekte dargestelt werden, dazu ist immer ein

Objekt einer der folgenden Klassen notwendig.

SP_GR_DrawnCoarsenode

Diese Klasse implementiert einen Coarsenode, der zur Darstellung die Möglichkeit beliebiger Figuren verwendet. Es ist in dieser Klasse möglich, mit einfachen Anweisungen *in* das Objekt zu malen, so als wäre es ein Devicecontext. Sie erbt neben `SP_GR_Coarsenode` auch von `wxDrawnShape` und ist damit direkt zur Darstellung in Netzen verwendbar. Aus der Klasse `wxDrawnShape` kommen auch die Methoden, die in `MyDraw` verwendet werden und es ermöglichen beliebige Figuren aus Kreisen, Kreisbögen, Strichen oder Rechtecken zu erzeugen.

Als Beispielimplementierung möchte ich auf `SP_GR_CoarsePlace` und `SP_GR_CoarseTransition` verweisen, die mit den Mitteln von `wxDrawnShape` die entsprechenden Elemente aus PED nachempfinden. Die Implementierung von `MyDraw` in dieser Klasse selbst reduziert sich auf ein Quadrat, welches in den Diagonalen Linien enthält.
verwendeter Eventhandler: `SP_GR_CoarseHandler`

SP_GR_BitmapCoarsenode

Diese Klasse implementiert einen Coarsenode, der zur Darstellung die Möglichkeit von Bildern verwendet. Sie erbt neben `SP_GR_Coarsenode` auch von `wxBitmapShape` und ist damit direkt zur Darstellung in Netzen verwendbar. Mit dieser Klasse ist es möglich, Icons zur Visualisierung der Knoten zu erwenden. Die einzige Anfroderung ist die Verwendung von Beschreibungen im .xpm-Format.

Näheres zu den in SNOOPY verwendeten Bitmaps findet sich bei der Beschreibung von `SP_GR_Bitmap` weiter oben.
verwendeter Eventhandler: `SP_GR_CoarseHandler`

A.21 SP_GR_Edge

Die Basisklasse aller Kanten eines Graphen ist einzig von `SP_GR_Interface` abgeleitet. Da diese Klasse keinerlei Funktionalität von wxWindows übernimmt, können keine Objekte von ihr dargestellt werden. Sie enthält wiederum eine Typisierung der zur Verfügung stehenden Kanten und dient als Container für die Verweise auf die Elemente in der Datenstruktur.

A.21.1 Typdefinitionen

```
enum SP_TYPE_NODE
{
    SP_EDGE_UNDEFINED,
    SP_EDGE_UNIDIRECTIONAL,
    SP_EDGE_BIDIRECTIONAL,
};
```

A.21.2 wichtige Methoden

Die folgenden Methoden sind in allen abgeleiteten Klassne verfügbar und beeinflussen das Aussehen und das Verhalten der Objekte.

---

bool
SetIsSpline (**bool** pVal)

pVal Spline Ja/Nein

Diese Methode stellt die Kante nun an als Spline dar. Dabei wird die Funktionalität von wxWindows genutzt, die leider in der aktuellen Version noch einige Schwächen, bezüglich der Hittests (Entscheidung, ob ein Objekt mit der Maus getroffen wurde), hat.

---

bool
SetWidth (**int** `pWidth`, **bool** pAlways)

pWidth Breite
pAlways Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Dicke, mit der die Linie gezeichnet wird in Pixeln.

---

bool
SetPenCol (**int** `pPen`, **bool** pAlways)
SetPenCol (`wxColour` `pPen`, **bool** pAlways)

pPen	Die Farbe, entweder ein Palettenindex, oder ein `wxColour`-Objekt
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Farbe des zur Darstellung verwendeten "Stifts". Das `wxColour`-Objekt kann entweder selbst erzeugt werden, oder aus der Liste der vordefinierten Farben in der Datei `sp_defines.h` entnommen werden.

---

bool
SetBrushCol (**int** `pPen`, **bool** pAlways)
SetBrushCol (`wxColour` `pPen`, **bool** pAlways)

pPen	Die Farbe, entweder ein Palettenindex, oder ein `wxColour`-Objekt
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Die Farbe des zur Darstellung verwendeten Pinsels, also die Füllfarbe des Objekts. Das `wxColour`-Objekt kann entweder selbst erzeugt werden, oder aus der Liste der vordefinierten Farben in der Datei `sp_defines.h` entnommen werden.

---

bool
SetStyle (**int** `pStyle`, **bool** pAlways)

pWidth	Breite
pAlways	Indikator, ob dieser Wert änderbar sein soll (*unbenutzt*)

Der Style, mit dem die Linie gezeichnet wird. Zu finden sind alle vorbereiteten Styles (gestrichelt, gepunktet, etc.) in der Datei `sp_defines.h`.

A.21.3 abgeleitete Klassen

SP_GR_Line

Diese Klasse implementiert eine einfache unidirektionale Kante mit einer Pfeilspitze an ihrem Zielknoten. Sie erbt neben `SP_GR_Edge` auch von `wxLineShape` und ist damit direkt zur Darstellung in Netzen verwendbar.
verwendeter Eventhandler: `SP_GR_LineHandler`

SP_GR_Arc

Diese Klasse implementiert eine unidirektionale Linie ohne Pfeile. Sie erbt neben `SP_GR_Edge` auch von `wxLineShape` und ist damit direkt zur Darstellung in Netzen verwendbar.
verwendeter Eventhandler: `SP_GR_LineHandler`

A.22 SP_GR_Attribute

Die Basisklasse aller Attribute eines Graphen ist einzig von `SP_GR_Interface` abgeleitet. Da diese Klasse keinerlei Funktionalität von wxWindows übernimmt, können keine Objekte von ihr dargestellt werden. Sie enthält wiederum eine Typisierung der zur Verfügung stehenden Kanten und dient als Container für die Verweise auf die Elemente in der Datenstruktur.

A.22.1 Typdefinitionen

```
enum SP_TYPE_ATTRIBUTE
{

    SP_ATTR_UNDEFINED,
    SP_ATTR_STRING,
    SP_ATTR_NUMBER,
    SP_ATTR_BOOL,
    SP_ATTR_PASS,
    SP_ATTR_MARK,
    SP_ATTR_TOKEN,
    SP_ATTR_LOGIC,
    SP_ATTR_COLLECTVIEW,
    SP_ATTR_LINK,
    SP_ATTR_NIL,
};
```

A.22.2 wichtige Methoden

Die folgenden Methoden sind in allen abgeleiteten Klassen verfügbar und beeinflussen das Aussehen und das Verhalten der Objekte.

---

void
SetPositionOffset(**double** x, **double** y)

x	Koordinate in x-Richtung
y	Koordinate in y-Richtung

Der Versatz zum Bezugspunkt des Elternelements (Knoten oder Kante). Gerade bei Elementen mit mehreren Attributen kann man so bereits bei der Netzklassendefinition das Attribut an eine Startposition verschieben. Bei Knoten ist der Bezugspunkt die Mitte, bei Kanten entweder der mittelste Knickpunkt, oder die Mitte der Linie, die von den beiden mittleren Knickpunkten begrenzt wird.

---

bool
SetStatic (**bool** pStatic)

pStatic Statisch Ja/Nein

Statisch bedeutet, dass sich das Attribut nicht verschieben lässt. Ein *Drag and Drop* eines statischen Attributes wird an das Elternelement propagiert und bewegt so das ganze Element (nur wirklich sinnvoll bei Knoten).

---

bool
SetHidden (**bool** pHidden)

pStatic Versteckt Ja/Nein

Versteckt bedeutet, dass das Attribut nicht im Fenster angezeigt werden soll, auch wenn es prinzipiell dazu in der Lage wäre. So kann man verhindern, dass die Ansicht unübersichtlich wird, sobald bestimmte Knoten 5 oder mehr Attribute besitzen. Dieses Attribut lässt sich zur Laufzeit nicht beeinflussen.

---

bool
SetShowInDialog (**bool** pShow)

pShow Anzeigen Ja/Nein

Ein boolscher Wert von `false` setzt die Anzeige des Attributs im Dialog aus. Mithin kann der Nutzer den Wert nicht manipulieren, während die Anzeige bleibt.

A.22.3 abgeleitete Klassen

SP_GR_StringAttribute

Diese Klasse implementiert die graphische Ausprägung von Zeichenketten. Sie erbt neben `SP_GR_Attribute` auch von `wxTextShape` und ist damit direkt zur Darstellung in Netzen verwendbar.

Eine besondere Fähigkeit dieses Attributs ist in der folgenden Methode beschrieben:

---

bool
SetCenterOnParent (**bool** pCenter)

pCenter Zentriert Ja/Nein

Zentriert bedeutet, dass das Attribut sein Elternelement in der Grösse verändert,

so dass es immer in der Mitte von diesem zu Steehn kommt und das Element fast ganz ausfüllt (nur wirklich sinnvoll bei Knoten). Diese Eigenschaft ignoriert die gesetzten Attribute bezüglich der Grösse!

Es ist nicht sinnvoll, mehreren Attributen dieses Flag zu setzen, da immer nur das letzte über die endliche Grösse entscheidet.

---

verwendeter Eventhandler: `SP_GR_AttributeHandler`

SP_GR_NumberAttribute

Diese Klasse implementiert die graphische Ausprägung von Integerwerten. Sie erbt neben `SP_GR_Attribute` auch von `wxRectangleShape` und ist damit direkt zur Darstellung in Netzen verwendbar.
verwendeter Eventhandler: `SP_GR_AttributeHandler`

SP_GR_BoolAttribute

Diese Klasse implementiert die graphische Ausprägung von boolschen Werten. Sie erbt neben `SP_GR_Attribute` auch von `wxRectangleShape`, ist allerdings nur indirekt zur Darstellung in Netzen verwendbar. Ihre Anzeige im Fenster ist in der vorliegenden Version ohne Funktion. Im Dialog, in dem sich die Attribute editieren lassen, ist für das `SP_GR_BoolAttribute` eine Checkbox zu sehen.
verwendeter Eventhandler: `SP_GR_AttributeHandler`

SP_GR_LogicAttribute

Diese Klasse implementiert die indirekte graphische Ausprägung von boolschen Werten. Sie erbt ausschliesslich von `SP_GR_Attribute` und ist deshalb nicht selbst im Fenster zu sehen. Im Dialog, in dem sich die Attribute editieren lassen, ist für dieses Attribut eine Checkbox zu sehen.

Die indirekte Auswirkung und Darstellung bezieht sich auf die Tatsache, dass das Setzen dieses Attributes (also das Setzen des Häkchens in der Checkbox) verschiedene Funktionen abruft, die das zu diesem Attribut gehörende Element in den Zustand "logisch" versetzt, so wie er von PED bekannt ist. In SNOOPY heisst das, dass von nun an alle Attribute und auch die graphische Ausprägung auf je nur noch ein Datentrukturelement verweisen.
verwendeter Eventhandler: -

SP_GR_MarkAttribute

Diese Klasse implementiert eine graphische Ausprägung von Integerwerten, ähnlich der in PED. Sie erbt neben `SP_GR_Attribute` auch von `wxDrawnShape` und ist deshalb direkt für die Darstellung in Graphen verwendbar.

Dabei verhält es sich mit der Darstellung so, dass die Ziffern 1 bis 3 als Anordnung von schwarzen, gefüllten Kreisen dargestellt werden, während ab 4 das normale Zahlzeichen verwendet wird.
verwendeter Eventhandler: `SP_GR_AttributeHandler`

SP_GR_TokenAttribute

Diese Klasse implementiert eine graphische Ausprägung von boolschen Werten. Sie erbt ausschliesslich von `SP_GR_MarkAttribute` und limitiert den möglichen Wertebereich auf $[0, 1]$. Im Dialog, in dem sich die Attribute editieren lassen, ist für dieses Attribut eine Checkbox zu sehen.
verwendeter Eventhandler: indirekt `SP_GR_AttributeHandler`

SP_GR_NilAttribute

Diese Klasse implementiert fast gar nichts. Sie erbt ausschliesslich von `SP_GR_Attribute` und erfüllt die Anforderungen an das Funktionsinterface. Dies Klasse hat keine Darstellung im Fenster und auch keine im Dialog. Mit ihr werden Attribute assoziiert, die immer denselben Wert haben sollen, oder andere, höhere Aufnahmen übernehmen.
verwendeter Eventhandler: –