Markus Fieber

Entwurf und Implementierung eines generischen, adaptiven Werkzeugs zur Arbeit mit Graphen

Diplomarbeit

Institut für Informatik
Brandenburgische Technische Universität Cottbus

Markus Fieber, Matr.-Nr. 9501454

Diplomarbeit an der

Brandenburgischen Technischen
Universität Cottbus
Institut für Informatik
Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit
betreut von Prof. Dr.-Ing. Monika Heiner

und bei der

AntzSystem GmbH
Am Nordrand 40
03044 Cottbus
betreut von Dipl.-Inform. Thomas Menzel

Mai 2004

Selbstständigkeitserklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Markus Fieber
Cottbus, 26. Mai 2004

Inhaltsverzeichnis

Vorwort

Das Thema und die Bedeutung dieser Arbeit, für mich persönlich, sind mit „Entwurf und Implementierung eines generischen, adaptiven Werkzeugs zur Arbeit mit Graphen“ nur unzureichend beschrieben.

Genau genommen handelt es sich um eine endlose Geschichte, die mich fast mein ganzes Studentenleben durch begleitet und immer zum Ziel hatte, einige lang gediente Werkzeuge abzulösen.

Durch meine Arbeit am Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit, sowie den Besuch der Lehrveranstaltungen “Einführung in die Nebenläufigkeit” und “Entwurf und Analyse von Petrinetzen” kam ich zeitig in Kontakt mit *dem* zentralen Werkzeug zur Erzeugung und Bearbeitung der in Forschung und Lehre verwendeten Modelle.

Seit Jahren wird für diesen Zweck PED eingesetzt, eine Software, welche die Konstruktion von hierarchischen Platz/Transitions Netzen und den Export der Daten zur Analyse, in verschiedenen Formaten, unterstützt. Das Programm hat eine lange Entwicklung hinter sich und unterstützt mittlerweile eine Vielzahl von Bearbeitungsfunktionalitäten. Mit der Beschränkung auf SUN/Solaris bzw. Linux als Plattform und der häufig nur durch weitreichende Eingriffe möglichen Modifikation und Erweiterung der Funktionalitäten, stand lange der Wunsch nach einer Reimplementierung im Raum.

Ich arbeitete lange Zeit am Lehrstuhl mit Herrn Dipl.-Inform. Thomas Menzel zusammen, der im Rahmen seines Studiums selbst ein Werkzeug entworfen und prototypisch implementiert hat, welches die Möglichkeit bot, die mit PED entworfenen Graphen zu animieren, das heisst, das “Markenspiel” zu spielen. In seiner Diplomarbeit [6] erarbeitete er die Grundlagen einer Software, die durch ihr Design als Werkzeug für die Erstellung und Bearbeitung der verschiedensten Graphen geeignet ist.

Was fehlte, war die Umsetzung dieses Designs in eine letztendlich wirklich benutzbare Applikation. Ich kam, als Mitarbeiter am Lehrstuhl, schon bald in den Genuss, Teile der Arbeiten an diesem Projekt zu übernehmen. Aus den, anfänglich nur die Oberfläche betreffenden, Teilen wurde schon bald die Delegation von zahlreicheren Aufgaben und während ich mir, etwas übertrieben formuliert, in den Vorlesungen gerade die Grundlagen der objektorientierten Softwareentwicklung aneignete, wurde dieses, mittlerweile SNOOPY getaufte, Projekt mehr und mehr Teil meiner Arbeit.

Die vorliegende Arbeit, zusammen mit der einsatzbereiten Implementierung, stellt das vorläufige Ende eines langen Weges dar, den zu beschreiten mir eine Vielzahl von Erfahrungen, auch abseits des informatorischen Handwerks gebracht hat.

Das Erstellen der der Software hat den Hauptteil der Zeit in Anspruch genommen und das Ergebnis ist eine Applikation, die den definierten Anforderungen genügt und darüber hinaus das bislang grösste Projekt darstellt, das ich persönlich und in Eigenregie zu einem Abschluss gebracht habe.

SNOOPY II bietet alle Möglichkeiten der Definition von Graphen und ausserdem eine homogene Umgebung, die Entwürfe auch wirklich anzuwenden und zu benutzen. Das schliesst eine vollständige Anwendungs-Oberfläche ein, sowie sämtliche Elemente, die sich aus den Eigenheiten der definierten Graphen heraus ergeben und weitgehend selbstständig die Vorgänge hinter den Kulissen steuern. Darüber hinaus wurden eine Vielzahl von Arbeitserleichterungen entworfen und implementiert, die einzig der Anwendung dienen und nicht Gegenstand des theoretischen Entwurfs der Anfangsphase waren. Dazu gehören Mechanismen zur Modifikation der sichtbaren Objekte (Transform, Rotate, Flip und Mirror) sowie eine funktionierende Implementierung für das oft gewünschte Copy'n'Paste, das graphische Editoren erst wirklich benutzbar macht. Darüber bietet sich jedem die Möglichkeit, eine selbst definierbare Art der Animation in jeden Graphen einzubauen. Es handelt sich bei dieser Arbeit hauptsächlich um die Anwendung der „handwerklichen“ Aspekte der Informatik – weniger um das Betreten akademischen, theoretischen Neulands. Dafür ist das Ergebnis in Form der vorliegenden Software explizit für den Einsatz entwickelt.

Die zentralen Ideen und deren Umsetzung, wie sie Einfluss in die Software genommen haben, sind Gegenstand dieser schriftlichen Arbeit und sollen an manchen Stellen eine Erklärung für getroffene Entscheidungen darstellen und immer das Verständnis für die Funktionsweise, mit Blick auf die Möglichkeiten eigener Erweiterungen fördern.

Markus Fieber
Cottbus, Mai 2004

1 Motivation

Die im Rahmen dieser Diplomarbeit zu erstellende Software soll die Möglichkeit bieten, Graphen innerhalb Oberfläche zu definieren und mit ihnen zu arbeiten.

Zu diesem Zweck müssen die Eigenschaften jedes Graphen generisch beschrieben und die Möglichkeit gegeben werden, die identifizierten Strukturen zu manipulieren. Darüber hinaus muss eine Oberfläche implementiert werden, die sich an den definierten Eigenschaften der zu bearbeitenden Graphen orientiert und deren Erstellen und Manipulieren erlaubt.

Die identifizierten Strukturen sollen die Möglichkeit bieten, die Graphen einfach nach eigenen Vorgaben und Anforderungen um Funktionalitäten erweitern zu können. Dabei soll die Notwendigkeit des Eingriffs auf wenige Stellen beschränkt sein und trotzdem ein Höchstmass an Flexibilität bieten.

Die Eigenschaften sollen sich sowohl auf die strukturelle Definition, als auch auf die Präsentation innerhalb der Oberfläche beziehen, um auch das Aussehen individuell festlegen zu können, die programmtechnische Verbindung zwischen den, die Graphstruktur beschreibenden Teilen und allem, was der Darstellung dient, soll so schwach wie möglich sein, um einen Austausch der Elemente zur Visualisierung zu ermöglichen.

Die Hauptrichtung der Entwicklung war die Implementierung eines Beispielgraphen, der die Funktionalitäten des Platz-Transitions-Netz-Editors PED, wie er am Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit an der BTU Cottbus eingesetzt wird, nachempfindet. Die meisten der implementierten Bestandteile der Graph-Struktur und des Aussehens, wurden mit Blick auf diesen Schwerpunkt entwickelt, soweit es sich um individualisierbare Eigenschaften handelt.

Als Ergebnis der durchgeführten Entwicklungsarbeit steht ein Graph-Werkzeug zur Verfügung, das sich insbesondere durch die folgenden zwei Eigenschaften auszeichnet:

Generierbarkeit

Leichte Erweiterbarkeit um neue Graphklassen, etwa Datenflussgraphen, Fehlerbäume, Ursache-Wirkungs-Graphen sowie Varianten zeitbewerteter Petrinetze bzw. speziell interpretierte Petrinetze, wie sie für den Hardware-Entwurf oder in der Systembiologie eingesetzt werden.

Adaptionsfähigkeit

Alle definierten Graphklassen können nebeneinander unter einer homogenen Oberfläche benutzt werden, wobei sich die diese dynamisch den Eigenheiten der Graphklasse in dem jeweilig aktiven Fenster anpasst.

Die automatisch für jeden definierten Graphen zur Verfügung stehenden Grundfunktionalitäten umfassen sowohl typische Funktionen zum Editieren (Cut, Copy, Paste) und zur Layoutanpassung (Mirror, Flip, Rotate) als auch die als Teil der Graphdefinition aktivierbaren, gerade für umfangreiche Netze hilfreichen, Konzepte der logischen Knoten und der Hierarchisierung, die ein Hierarchiebrowser automatisch unterstützt.

Darüber hinaus wurde ein Konzept entwickelt und umgesetzt, das für eine grosse Anzahl von Graph-Animationsarten als Basis dienen und ebenfalls mit Blick auf Erweiterbarkeit einfach an individuelle Anforderungen angepasst werden kann.

2 Grundlagen

Da das Thema dieser Arbeit der Entwurf und die Entwicklung eines Werkzeugs zur Erstellung und Bearbeitung von Graphen ist, gebe ich eine Einführung in die Grundlagen der Graphentheorie um die verwendeten Begriffe bekannt zu machen, und so die getroffenen Entscheidungen im Rahmen bekannter Definitionen erläutern zu können.

2.1 Graphentheorie

Anschaulich gesprochen handelt es sich bei einem Graphen um eine Menge von Punkten zwischen deren Elementen Linien verlaufen können. Die Elemente der Punktmenge bezeichne ich im Folgenden als Knoten, die Linien zwischen diesen Elementen als Kanten.

Anhand der Eigenschaften der Knoten und Kanten und den Möglichkeiten der Verbindungen lassen sich Graphen in verschiedene Klassen unterteilen.

So genannte einfache Graphen sind dadurch charakterisiert, dass zwischen zwei Knoten nur maximal eine Kante existiert. Graphen, bei denen diese Regel nicht gilt, es also mehrere Kanten zwischen gleichen Knoten geben kann, bezeichnet man als Multigraphen.

Kanten können in allen Graphen ungerichtet oder gerichtet sein, wonach man den Graphen als gerichteten, respektive ungerichteten Graphen bezeichnet. Gerichtete Kanten zeichnen sich dadurch aus, dass sie die Knoten, die sie verbinden, explizit in eine Quelle und ein Ziel unterscheiden. Graphisch wird diese Eigenschaft häufig durch einen Pfeil am Ende der Kante visualisiert. Bei ungerichteten Kanten entfällt diese Klassifizierung.

Def. 1 Graph

Ein Graph *G* ist ein Tupel *(N, E)*, wobei *N* die Menge der Knoten und *E* eine höhere Menge von Kanten bezeichnet. Dabei ist *E* in

- ***ungerichteten Graphen ohne Mehrfachkanten*** *eine Teilmenge aller 2-elementigen Teilmengen von N,*
- ***gerichteten Graphen ohne Mehrfachkanten*** *eine Teilmenge des kartesischen Produkts N x N,*

- ***ungerichteten Graphen mit Mehrfachkanten*** *eine Multimenge über der Menge aller 2-elementigen Teilmengen von N,*
- ***gerichteten Graphen mit Mehrfachkanten*** *eine Multimenge über dem kartesischen Produkt N x N.*

Umgangssprachlich verzichtet man auf den Zusatz „ohne Mehrfachkanten" und bezeichnet nur Graphen mit Mehrfachkanten gesondert als „Multigraphen". Ebenso verzichtet man häufig auf das Attribut „ungerichtet" und bezeichnet nur gerichtete Graphen speziell. Ungerichtete Graphen ohne Mehrfachkanten bezeichnet man häufig als „einfach" oder „schlicht" und in Anlehnung an die englische Übersetzung nennt man gerichtete Graphen auch DiGraph (für engl. *directed graph*).

Um kürzer schreiben zu können, definiert man gewöhnlich zwei Abbildungen N und E, die einem Graphen $G = (N,E)$ seine Knoten- bzw. Kantenmenge zuordnen, d.h. $N(G) := N$ und $E(G) := E$. Man beachte, dass trotz der selben Symbole (N bzw. E) die Knoten- bzw. Kantenmenge etwas anderes als die gerade definierten Abbildungen darstellen. Wenn nicht anders gesagt, meinen N bzw. E ohne Argument gewöhnlich die Knoten- bzw. Kantenmenge des gerade betrachteten Graphen. N bzw. E mit Argument meinen hingegen immer die Abbildungen, deren Ergebnis wieder eine Knoten- bzw. Kantenmenge ist, und zwar die des als Argument angegebenen Graphen.

Ist G ein Graph, so sagt man allgemein n ist **Knoten** von G, wenn n zu $N(G)$ gehört. Ferner sagt man, falls G

- ungerichteter Graph ohne Mehrfachkanten ist und e zu $E(G)$ gehört, e ist eine **ungerichtete Kante** von G,
- gerichteter Graph ohne Mehrfachkanten ist und e zu $E(G)$ gehört, e ist eine **gerichtete Kante** von G,
- ungerichteter Graph mit Mehrfachkanten ist und $E(G)(e) > 0$, e ist eine **ungerichtete Kante** von G,
- gerichteter Graph mit Mehrfachkanten ist und $E(G)(e) > 0$, e ist eine **gerichtete Kante** von G.

In einer ungerichteten Kante $e = \{s,t\}$ bezeichnet man s und t als **Endknoten** von e. In einer gerichteten Kante $e = (s,t)$ bezeichnet man s als **Startknoten** und t als **Endknoten** von e.

Ist in Multigraphen sogar $E(G)(e) > 1$, so spricht man auch von einer **Multi-** oder **Mehrfachkante**. $E(G)(e)$ bezeichnet man auch als die **Vielfachheit** von e. Hat eine Kante e in gerichteten Graphen die Form (n, n), so spricht man von einer **Schleife**. Ist e in einem Multigraphen G zusätzlich eine Mehrfachkante,

so spricht man von einer Mehrfachschleife. Eine 1- oder 2-elementige Menge von geordneten Paaren mit der Eigenschaft, dass sie neben (v,w) auch (w,v) enthält (im 1-elementigen Fall ist dies natürlich nur bei $v = w$ möglich) nennt man, falls G

- gerichteter Graph ohne Mehrfachkanten ist und sowohl (v,w) als auch (w,v) Kante von G sind, **ungerichtete Kante** von G,
- gerichteter Graph mit Mehrfachkanten ist und $E(G)((v,w)) = E(G)((w,v))$, **ungerichtete Kante** von G.

1-elementige ungerichtete Kanten in gerichteten Graphen sind offensichtlich also immer Schleifen. Umgekehrt lässt sich zu jeder Schleife e eine ungerichtete Kante konstruieren (nämlich $\{e\}$). Schleifen sind in diesem Sinne also immer ungerichtet, weshalb man gewöhnlich das Attribut „gerichtet" weg lässt. Ist jede Kante eines gerichteten Graphen G Element einer ungerichteten Kante von G, so nennt man G auch **symmetrisch**.

Man lässt gewöhnlich in ungerichteten Graphen das Attribut „ungerichtet" für Kanten weg. In gerichteten Graphen lässt man das Attribut „gerichtet" für Kanten gewöhnlich nur dann weg, falls man keine *ungerichteten* Kanten betrachtet, der Graph also ausschliesslich gerichtete Kanten enthält. Gerichtete Graphen ohne Schleifen nennt man **schleifen los** oder **schleifen frei**.

Als **Knotenzahl** $n(G) = |N(G)|$ eines Graphen G bezeichnet man die Anzahl seiner Knoten, als **Kantenzahl** $m(G)=|E(G)|$ bezeichnet man die Anzahl seiner Kanten (in Multigraphen summiert man über die Vielfachheit der Kanten).

Einen Graphen, dessen Knotenmenge endlich ist, nennt man **endlicher Graph**. Üblicherweise ist in diesen auch die Kantenmenge endlich. Im Gegensatz dazu nennt man einen Graph, dessen Knotenmenge unendlich ist, **unendlicher Graph**. Meist betrachtet man nur endliche Graphen und lässt daher das Attribut „endlich" weg, während man „unendliche Graphen" explizit kennzeichnet.

2.2 Partitionierung

Sowohl die Knoten- als auch die Kantenmenge eines Graphen kann (und muss) häufig unterteilt werden. Einige Modelle können nur angewandt werden, wenn es der Graph erlaubt, seine, ihn ausmachenden, Mengen zu diversifizieren. Für solche Fälle spricht man von einer Partitionierung der Mengen der Knoten und Kanten, so dass sie in Teilmengen zerfallen, deren Elemente sich durch bestimmte Eigenschaften auszeichnen und von Elementen der anderen Partitionen abgrenzen.

Diese Eigenschaft der Partitionierung kann ermittelt werden, indem ein bestehender Graph auf Beziehungen zwischen seinen Elementen hin untersucht wird. Als Beispiel sei der Test der Zweifärbbarkeit einer Knotenmenge mittels Tiefensuche erwähnt. Für unsere Anwendung spielt diese nachträgliche Ermittlung keine Rolle. Wir machen die Partitionierung zu einer Eigenschaft des Graphen zur Definitionszeit und liefern die Möglichkeiten, diese Partitionierung und die damit vom Designer implizierten Eigenschaften, sicherzustellen.

Die Partitionierung an sich bezeichnet folgende Eigenschaft: Sei $G := (N, V)$ ein Graph mit N der Menge der Knoten und V der Menge der Kanten. Dann bezeichnen wir $N' := \{ n \mid n \in N' \wedge N' \subseteq N(G) \}$ als Partition der Knotenmenge $N(G)$. Für alle Elemente von $N(G)$ gilt, dass sie jeweils genau einer der definierten Partitionen zugeordnet sind. Darüber hinaus sind alle Partitionen paarweise disjunkt und ihre Vereinigung ergibt wieder die Menge der ursprünglichen $N(G)$. Wir definieren eine Abbildung $NodePart(n) := N'$ die zu jedem Knoten $n \in N(G)$ die zugehörige Partitionsmenge angibt.

Für die Menge der Kanten $E(G)$ ergibt sich dadurch eine Vielzahl neuer Möglichkeiten der Definition und Beschränkung der Beziehungen ihrer Elemente zu denen aus $N(G)$. Analog zu N' definieren wir E' als Teilmenge von E(G) und bezeichnen E' als eine Partition der Kantenmenge. $EdgePart(e) := E'$ ist die Partition von $E(G)$, von der e Element ist. Auch die Mengen der Kanten-Partitionen sind paarweise disjunkt und ergeben durch Vereinigung wieder E(G).

Eine bekannte Partitionierung beschreibt zum Beispiel ein bipartiter Graph, dessen Knoten sich in zwei Teilmengen aufteilen lassen, so dass es zwischen den Elementen innerhalb einer Teilmenge keine Kanten gibt. Man kann zum Einen beliebige Graphen auf diese Eigenschaft hin untersuchen (durch Tiefensuche) oder andererseits die Partitionierung schon zur Definitionszeit festlegen. In der hier vorgestellten Arbeit ist die Partitionierung Teil der Definition der Graphen.

3 Entwurf

Alles, was im Folgenden als *Datenstruktur* bezeichnet wird, meint, im Gegensatz zur *graphischen Ausprägung*, im weitesten Sinne das, was im Abschnitt „Graphentheorie“ auf Seite 7 eingeführt wurde. Also die mengenmässige Beschreibung eines Graphen, seiner Elemente und deren Beziehungen.

Dem gegenüber stehen die graphischen Ausprägungen, also alles, was die Verknüpfung der Elemente der Datenstruktur mit individuellen Möglichkeiten der Darstellung in einem Fenstersystem zum Gegenstand hat, sowie die Erfassung aller Aspekte, die sich aus der Arbeit mit einem Fenstersystem ergeben, also das GUI-Management und die Reaktion auf, durch den Nutzer ausgelöste, Ereignisse.

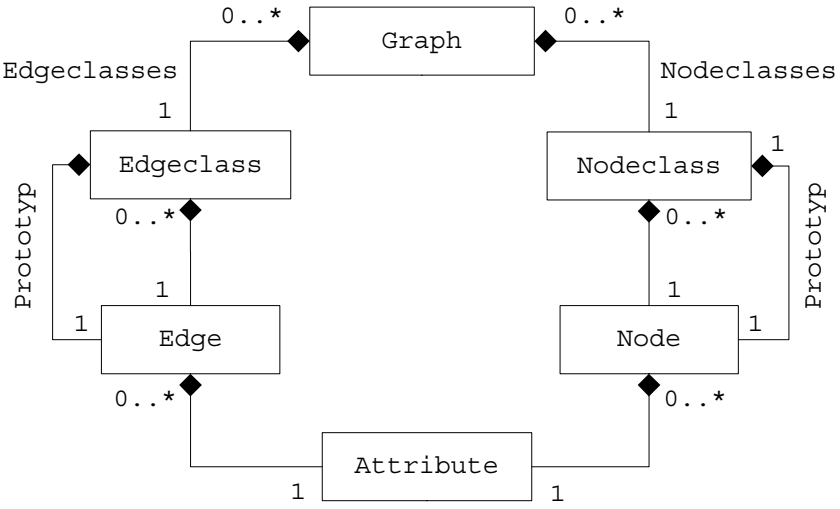
Der erste Entwurf entsprang zu Beginn den Überlegungen im Zusammenhang mit der Datenstruktur und wurde im Laufe der Entwicklung den Bedürfnissen der Zusammenarbeit mit dem graphischen Frontend an manchen Stellen angepasst. Der im Folgenden vorgestellte Entwurf ist das Ergebnis der Erkenntnisse aus diesem ersten Versuch und stellt damit eine Weiterentwicklung dar.

3.1 Datenstruktur

Zentrales Element der Datenstruktur ist der Graph, der die (beiden) Mengen der Knoten und Kanten verwaltet und Zugriffsmethoden zur Modifikation seiner Struktur zur Verfügung stellt. Jeder Graph wird durch eine Instanz der *Graph*-Klasse abgebildet, die einmalig ist, das heisst es existieren keine Ableitungen im objektorientierten Sinne. Jede Instanz von *Graph* soll anhand eines Namens identifiziert werden.

Zur Umsetzung der Partitionierung sind die Mengen der Knoten und Kanten durch so genannte Knoten- und Kantenklassen (*Nodeclass* und *Edgeclass*) separierbar. Einfache Graphen sind zum Beispiel durch genau eine Instanz der *Nodeclass*- und genau eine Instanz der *Edgeclass*-Klasse definiert. Sowohl die *Nodeclass*- als auch die *Edgeclass*-Klasse arbeiten nach demselben Prinzip: Zur Definitionszeit erlauben sie das Manipulieren der Eigenschaften eines prototypischen Elements, während sie zur Arbeitszeit die Menge ihrer konkreten Elemente verwalten, die alle als Kopien aus dem Prototypen erzeugt wurden. Ausserdem soll auch jede Instanz von *Nodeclass* und *Edgeclass* anhand eines Namens identifiziert werden.

Die von *Nodeclass* und *Edgeclass* als Prototypen und Elemente ihrer Mengen, sind Instanzen von *Node* bzw. *Edge*. Diese beiden Klassen bilden sozusagen die kleinste Einheit der im Abschnitt Graphentheorie behandelten Elemente eines Graphen und auch von diesen sollen keine Ableitungen existieren. Eine Instanz von *Node* verwaltet hauptsächlich die Mengen der an ihr ein- und ausgehenden Kanten und *Edge* wiederum enthält Verweise auf Objekte von *Node*, die sie verbindet.



Zeichnung 1 - Entwurf zur Graphhierarchie

3.1.1 Attribute zur Unterscheidung

Da es nur eine *Node*- beziehungsweise nur eine *Edge*-Klasse geben soll, und sich so die Elemente von *Nodeclass* oder *Edgeclass* nicht unterscheiden (können) wurde die Definition von Knoten und Kanten durch so genannte Attribute erweitert. Ein *Node* ist nun also nicht nur durch seine Zugehörig-

keit zu einer *Nodeclass*-Instanz oder die Menge seiner ein- und ausgehenden Kanten definiert, sondern er verwaltet ausserdem eine Menge von Instanzen der Klasse *Attribute*.

Um die Nutzung so flexibel wie möglich zu gestalten, wird *Attribute* als Klasse definiert, von der Ableitungen im objektorientierten Sinne existieren müssen und ausserdem selbst als eine Klasse, von der selbst keine Objekte instanziiert werden können, da sie auch ausschliesslich abstrakte Methoden definiert. Die Attribute sind die eigentlichen „Informationsträger" der Elemente des Graphen. Für diese Informationen enthalten alle Ableitungen von *Attribute* als Kern einen Wert (*value*). Das bedeutet, es existiert zum Beispiel eine, von *Attribute* abgeleitete, Klasse, die zur Speicherung von textuellen Informationen dient. Jedes konkrete Attribut stellt dabei Methoden zum Setzen und Erfragen des aktuellen Wertes zur Verfügung, die jeweils Rückgabewerte oder Parameter vom konkreten Typ des *value* haben. Um dennoch auch über die gemeinsame Schnittstelle *Attribute* auf den Instanzen Arbeiten zu können, muss jede abgeleitete Klasse eine Möglichkeit der Serialisierung seiner Daten implementieren. *Node* und *Edge* werden um Methoden erweitert, die es erlauben, die Menge der ihnen zugeordneten Attribute zu manipulieren.

3.1.2 Die Notwendigkeit des Klonens

Die Definition der Partitionen als Menge von Kopien nach dem Vorbild eines Prototypen impliziert die Möglichkeit, diese Vorlagen kopieren zu können. Zu diesem Zweck wird der Klasse der Knoten und Kanten, ebenso wie der der Attribute diese Fähigkeit zwingend vorgeschrieben.

Es ist also integraler Bestandteil der Klassen *Node*, *Edge* und *Attribute* dass sie es ermöglichen, exakte Kopien ihrer selbst zu erzeugen. Für *Node* und *Edge* ist der Mechanismus immer derselbe, da es, wie gefordert, keine Ableitungen gibt. Sie erzeugen also beim Kopieren eine neue Instanz ihrer selbst, der sie die identische Anzahl von Kopien nach der Vorgabe ihrer Attributmenge mitgeben. Das Kopieren der Klasse *Attribute* muss allerdings in jeder Ableitung separat implementiert werden, da die Kopien sich in dem, durch sie verwalteten, Typ unterscheiden.

Die *Graph*-Klasse verwaltet also die beiden Mengen von Instanzen der *Nodeclass*- und *Edgeclass*-Klasse und bietet Möglichkeiten zur Manipulation dieser Mengen, also das Erzeugen neuer und das Entfernen oder Ändern bestehender Elemente. *Nodeclass* und *Edgeclass* wiederum bieten, wie schon

erwähnt, Zugriff auf ihre Prototypen oder die Menge der konkreten Elemente und *Node* und *Edge* am Ende erlauben das Hinzufügen oder Entfernen von Attributen.

3.1.3 Die Netzklassen als Vorlage

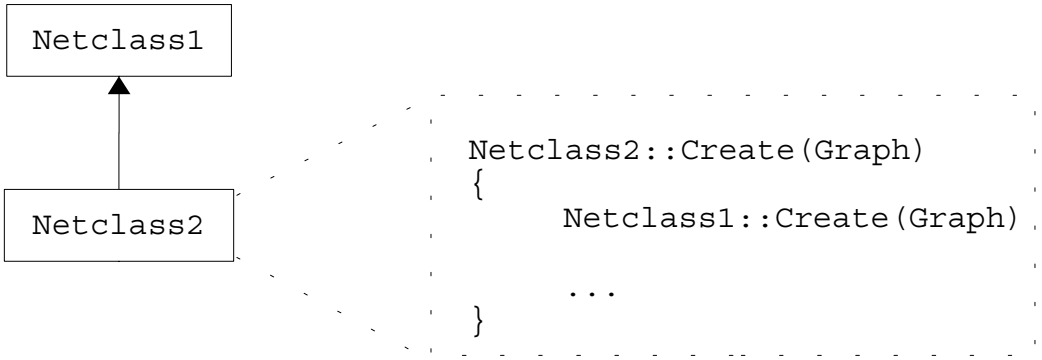
Damit sind die Möglichkeiten der Definition eines Graphen beschrieben. Um diese Arbeiten zentral erledigen zu können, wurde in den Entwurf die Definition von so genannten Netzklassen (*Netclass*) aufgenommen. Diese sind der eigentliche Kern des Entwurfs von Graphen zur Verwendung innerhalb der Software.

Netclass selbst ist dabei eine Klasse, von der Ableitungen existieren können und die durch einen Namen identifizierbar sein soll. Ihre Daseinsberechtigung liegt, unter anderem, in der Kapselung der Definitionsschritte, die einen konkreten Graphen ausmachen. Die wichtigste Methode ist dafür das Manipulieren (*Create*) eines *Graph*-Objekts nach den Möglichkeiten der vorab erwähnten Struktur eines Graphen. Denn nicht die *Netclass* selbst instantiiert ein Objekt von *Graph*, sondern sie manipuliert immer eine bestehende Instanz. Das schliesst die Erweiterung der Knoten- und Kantenpartitionen um neue Elemente sowie das Entfernen von Elementen aus diesen Mengen genauso ein, wie das Manipulieren der in *Nodeclass* oder *Edgeclass* gehaltenen Elemente – bis hin zu den Attributen an *Node* und *Edge*. Nicht ausschliesslich, aber hauptsächlich, sollte sich die Einflussnahme an dieser Stelle, der Manipulation der Prototypen widmen.

3.1.4 Abgeleitete Netzklassen

Durch die Möglichkeit der Vererbung im objektorientierten Sinne innerhalb der von *Netclass* vererbten Klassen, bietet sich auch die Möglichkeit, Graphen aufeinander aufzubauen – wenngleich die Bezeichnung „vererben“ im objektorientierten Sinne den Mechanismus nicht ganz beschreibt.

Zeichnung 2 zeigt symbolisch, was notwendig ist, um Graphen aufeinander aufbauen zu lassen. Erste Voraussetzung ist, dass die Netzklassen im Sinne der objektorientierten Entwicklung voneinander vererbt wurden. Dann kann vor der Implementierung der *Create*-Funktion der erbenden Klasse (*Netclass2*) die *Create*-Funktion der vererbenden Klasse (*Netclass1*) mit demselben *Graph*-Argument aufgerufen werden, das auch der erbenden Klasse zur



Zeichnung 2 - Vererbung von Netzklassen

Manipulation übergeben wurde. Das Ergebnis ist ein Graph, der den Definitionen der vererbenden Klasse entspricht. Da auch diese Klasse ihrerseits auf anderen Definitionen aufbauen kann, ist diese „Vererbung“ von Graphdefinitionen beliebig staffelbar. Es ist aber immer Aufgabe des Designers der entsprechenden Netzklasse, dass er diese Übernahme der Definition veranlasst, dies kann nicht durch einen objektorientierten Mechanismus erzwungen werden.

3.1.5 Erweiterte Funktionen der Netzklasse

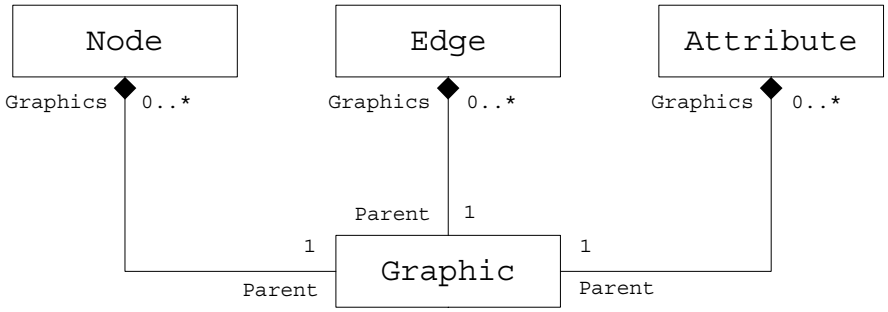
Neben *Create* bietet eine *Netclass*-Klasse noch zwei weitere, wichtige, Möglichkeiten der Einflussnahme. Wie schon im Abschnitt „Partitionierung“ auf Seite 10 ausgeführt, ist es möglich, die erlaubten Verbindungen von Kanten zwischen den Elementen der Knoten-Partitionen zu beschränken. Diese Forderung ermöglicht erst das gezielte Definieren von, zum Beispiel, Graphen ohne Schleifen oder auch Graphen, in denen es keine Mehrfachkanten geben soll. Um diese Möglichkeit nicht an die konkreten Instanzen von *Node* oder *Edge* zu binden, bietet jede *Netclass* zwei Methoden, die zum Überprüfen der Zulässigkeit von Kantenverbindung oder auch des Hinzufügens von Knoten dienen und automatisch bei der Anfrage des Erzeugens neuer Elemente aufgerufen werden sollen. Im *NodeRequirement* wird die konkrete Partition des zu erzeugenden Knotens als Information übergeben. Bei *EdgeRequirement* sind es die Partition der zu erzeugenden Kante sowie die gewählten Instanzen von *Node*, die verbunden werden sollen.

Durch den automatischen Aufruf dieser Methoden zur Überprüfung der Zulässigkeit kann das Verhalten vollständig in der Instanz von *Netclass* kontrolliert werden.

3.2 Graphische Ausprägungen

Neben der Notwendigkeit der generischen Definition von Graphen wurde im Entwurf auf alle Belange, die graphische Darstellung betreffend, Wert gelegt. Dazu gehört die Festlegung der zur Darstellung zu verwendenden graphischen Elemente, sowohl für Knoten, Kanten als auch Attribute. Für Attribute musste darüber hinaus ein Weg gefunden werden, die Manipulation ihrer Werte nach individuellen Regeln zu ermöglichen.

Alle graphischen Elemente sind Derivate der Klasse *Graphic*, die das grundlegende Verhalten graphischer Elemente beschreibt und die Beziehungen zwischen den Elementen der Datenstruktur und verschiedenen Elementen der Graphic verwaltet. Daraus folgt, dass die Elemente, die einen Graphen in der Datenstruktur beschreiben um Methoden erweitert werden müssen, um sie um graphische Ausprägungen zu erweitern. Diese Anforderung gilt erst unterhalb der Ebene der Knoten- und Kantenklassen, also für *Node*, *Edge* und *Attribute* und dessen Ableitungen. Von *Graphic* selbst soll kein Objekt instanziierbar sein, da es eine reine Sammlung grundlegender Aufgaben ist und der Definition der Schnittstellen dient. Das impliziert, dass es Ableitungen geben muss, die, wie bei den Attributen, jeweils für spezielle Eigenschaften oder hier eben spezielles Aussehen, stehen. So kann zum Beispiel einem Knoten eine kreisförmige, rechteckige oder beliebige andere graphische Ausprägung zugeordnet werden und den Attributen, je nach Intention des Netz-Administrators, eine Visualisierung der Werte. Dabei ist es frei gestellt, welche Darstellung für einen konkreten Typen gewählt wird. So kann zum Beispiel ein numerisches Attribut der Datenstruktur einmal eine textuelle oder ebenso eine symbolische graphische Ausprägung haben.



Zeichnung 3 - Assoziation von Daten zur Graphik

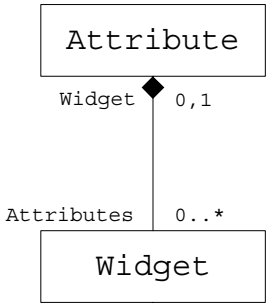
Die Festlegung der zu verwendenden graphischen Ausprägungen erfolgt ebenfalls in der *Create*-Methode der Netzklasse. Also muss, neben der Liste der Attribute, beim *Clone* von *Node* und *Edge* auch das Vervielfältigen der graphischen Ausprägungen veranlasst werden, ebenso wie es bei den Attributen, neben der reinen Duplizierung der Werte, um das Kopieren der assoziierten Graphiken geht. Um diese Folge von Kopiervorgängen für beliebig strukturierte Graphen generisch erledigen zu können, muss jede Ableitung von *Graphic* ebenfalls ein *Clone* implementieren.

3.3 Widgets, die Fensterelemente

Neben den Elementen der graphischen Darstellung musste noch eine weitere Anforderung im Entwurf berücksichtigt werden. Die Attribute, die einen Knoten oder eine Kante ausmachen, müssen vom Nutzer der Anwendung (also dem Benutzer des definierten Graphen) für Änderungen der Werte zugänglich sein. Anders als in der ersten Implementierung wurde dieses Verhalten, also die Darstellung in Dialogen der Oberfläche, von den konkreten graphischen Ausprägungen von *Attribute* getrennt. Stattdessen ist für jedes Attribut, neben der Festlegung seines Aussehens, noch eine Festlegung über die Art der möglichen Manipulation notwendig.

Zu diesem Zweck wurde die Klasse *Widget* eingeführt, die als Basis für so genannte Window-Gadgets (Bedienelemente in der Oberfläche) fungiert und für die Darstellung in einem Dialog vorgesehen ist. Wie auch bei *Attribute* können von *Widget* beliebige Ableitungen unter Einhaltung einer minimalen Schnittstelle definiert werden. Denkbar ist zum Beispiel eine Möglichkeit, einen Text zu editieren oder einen Zahlenwert zu ändern. Zur Definitionszeit der Netzklasse werden die Attribute, die der Nutzer bearbeiten können soll, mit einem solchen, von *Widget* abgeleiteten Darstellungselement verknüpft. Dabei erfolgt diese Verknüpfung nicht statisch in der Klasse *Attribute*, sondern anhand einer eindeutigen Identifizierung in einer Registrierung. Um anschliessend, zur Laufzeit auch die entsprechenden Widgets anzeigen zu können, müssen auch die *Widget*-Derivate ein *Clone* implementieren, über das sie, aus der Registrierung heraus, für jedes Attribut erzeugt und anschliessend wieder zerstört werden können.

Es ist übrigens nicht zwingend erforderlich, dass Datenstrukturelemente eine graphische Ausprägung zugeordnet haben oder dass Attribut für ein *Widget* registriert sein müssen. Es soll also möglich sein, zum Beispiel ein Attribut zu



Zeichnung 4 - Verbindungen zwischen Attributen und Widgets im Entwurf

definieren, dass zwar keine graphische Ausprägung hat, sich aber trotzdem im Dialog zur Bearbeitung der Eigenschaften manipulieren lässt. Das gilt auch für jede andere Kombination dieser beiden Möglichkeiten.

Das Festlegen der Schnittstellen und die Reduzierung der Komplexität auf einfache Strukturen, ermöglicht die Erweiterung des Systems. Neue Netzklassen sind einfach definier- und hinzufügbar. Dabei bleiben die zugrunde liegenden Operationen über den verschiedenen Mengen der Klassen, der Knoten, Kanten, Attribute und ihrer Graphiken in aller Regel für den Nutzer transparent, da er für eine Erweiterung der Netzklassen nur auf das Implementieren von Attributen, graphischen Ausprägungen und Elementen zur Manipulation der Werte angewiesen ist.

3.4 Hierarchien und Mehrdeutigkeiten

Die Arbeit mit existierenden Graph-Editoren zeigte häufig die Möglichkeit der hierarchischen Organisation der verschiedenen Elemente von Knoten und Kanten. Das heisst, es wurde die Möglichkeit geboten, einen Graphen in Teilgraphen (ich bezeichne diese als Netze) zu separieren, wobei diese Aufteilung meistens optischer Natur ist. Jedem Element von N und E wurde zu diesem Zweck eine Nummer zugeordnet, die es als zu einem bestimmten Netz zugehörig kennzeichnet. Ausgezeichnete Elemente der Knotenmenge standen dann, als Teil ihrer Partition, für die Existenz eines neuen Teilgraphen oder auch Unternetzes.

Das a priori existierende Ur-Netz, trägt immer die Netz-Nummer 1 und damit bekommen alle, in dieser Ansicht erzeugten, Elemente diese Nummer zugeordnet. Definiert ein Graph eine besondere Partition, deren Elemente für jeweils ein neues Unternetz stehen, so kann das erste Element dieser Partiti-

on auch nur im Ur-Netz angelegt werden. Allerdings bietet ein solches Element die Möglichkeit, eine neue Sicht zu definieren und in der Arbeit mit den Elementen der Partitionen in dieser Ansicht fortzufahren. Um bei dem Beispiel zu bleiben, würde ein solches Element zwar dem Netz mit der Nummer 1 zugeordnet sein, aber seinerseits eine weitere Ebene in der Hierarchie mit der Nummer 2 einfügen. Alle in dieser neuen Ansicht angelegten Elemente erhalten nun die Netz-Nummer 2.

Diese Hierarchisierung ist rein optischer Natur. Jedes Element bleibt weiterhin genau einer der Partitionen des Graphen zugeordnet. So kann es Knoten in ein und derselben Partition geben, die sich nur anand ihrer Netz-Nummer unterscheiden. Man kann weiterhin alle Knoten und Kanten ohne Ansicht der Netzzugehörigkeit betrachten, die Möglichkeit der Hierarchisierung ist in diesem Fall ausschliesslich eine Zugabe, die der Übersicht bei der Arbeit mit umfangreichen Graphen dienen soll. Anwendung findet diese Art der Hierarchisierung zum Beispiel in dem Platz-Transitions-Graphen-Editor PED mit seinem Konzept der *Coarse*-Nodes (für engl. c*oarse* = grob).

Trotz der Tatsache, dass eine solche Hierarchisierung keine Erweiterung der „Fähigkeiten“ des Graphen mit sich bringt, erfordert sie einiges an Definitionsaufwand um sie generisch beschreiben und nutzen zu können. Darüber hinaus lässt so eine Möglichkeit auch den Wunsch nach dem nächsten logischen Schritt aufkommen, nämlich dass es in einem solchen Element einer besonderen Partition einen völlig anderen Graphen zur Bearbeitung geben soll, dessen Elemente nichts mit den Elementen des Graphen in dem die Knoten zur Hierarchisierung definiert sind, zu tun haben.

Für beide Anforderungen wurde eine Erweiterung namens *Coarse* entworfen, die den Elementen zugeordnet werden kann, selbst die Anforderung einer *Clone*-Methode erfüllt, und die Elemente um die Fähigkeit erweitert, eine Netznummer zu generieren, diese zu speichern und ausserdem Informationen über den darzustellenden Graphen zu verwalten. Dadurch, dass die Klasse *Coarse* selbst die Informationen zu dem von ihr dargestellten Graphen hält, sind beide Anforderungen realisierbar. Einmal kann also die gespeicherte Referenz auf dieselbe *Graph*-Instanz zeigen, in der auch der Knoten, dem die *Coarse*-Eigenschaft zugeordnet wurde, definiert ist. Oder die Instanzen sind verschieden, was es ermöglicht, Graphen aus unterschiedlichen Netzklassen ineinander darzustellen.

Die zweite Arbeitserleichterung, die in einigen bestehenden Werkzeugen Verwendung findet, ist die Möglichkeit, mehrere graphische Ausprägungen zu ein und demselben Datenstrukturelement zu definieren. Zu diesem Zweck müssen alle Operationen, die der Zuordnung von Datenstruktur zu

Graphik dienen, auf Mengen operieren. Ein Knoten wird also nicht mehr nur durch genau eine graphische Repräsentation zur Ansicht gebracht, sondern er kann jederzeit über multiple Instanzen von *Graphic* an unterschiedlichen Positionen und in unterschiedlichen Netzen verfügen. Analog gilt das für Kanten und daraus folgend auch für alle nachgeordneten Attribute.

Die Existenz mehrerer graphischer Ausprägungen für eines der Elemente bedeutet nicht gleichzeitig die Existenz einer Netzhierarchie innerhalb des Graphen. Es ist durchaus denkbar, dass ein Knoten in ein und demselben Netz (also derselben Ansicht) mehrere graphische Ausprägungen definiert. Da jede der entsprechenden Instanzen von *Graphic* zu ein und derselben Instanz von *Node* gehört, verweisen sie auch alle auf dieselben Informationen. Unabhängig davon, zu welcher graphischen Ausprägung des Knotens zum Beispiel eine Kante existiert, sie wird in der Datenstruktur immer der entsprechenden Instanz von *Node* eingeschrieben. Mehrere graphische Ausprägungen werden oft zum Zwecke der Reduktion von Kantenzügen eingesetzt, falls grosse Distanzen überbrückt werden müssten um eine Verbindung zwischen zwei Knoten herzustellen. Wir bezeichnen diese Möglichkeit der Mehrdeutigkeit von graphischen Ausprägungen als *logic* oder sagen, die graphischen Repräsentationen dieses Knotens oder jener Kante sind logisch zueinander und verweisen auf dasselbe Element in der Datenstruktur.

4 Implementierung

Das folgende Kapitel stellt die vorliegende Implementierung in Auszügen vor. Diese beziehen sich hauptsächlich auf die im Kapitel Entwurf vorgestellten Strukturen. Die meisten der, die Oberfläche betreffenden Klassen sind davon ausgenommen, da sie in der Regel von Änderungen für eigene Netzklassen nicht betroffen sind und sich automatisch Anpassen.

4.1 Verwendete Software

Wie schon die erste Version setzt diese Software auf die Verwendung von *wxWidgets* [10] (vormals *wxWindows*) als Bibliothek für die Oberflächenelemente, welche erfahrungsgemäss eine Arbeitserleichterung im Bezug auf die Portierbarkeit darstellt. Zum Zeitpunkt des Abschlusses war die verwendete Version 2.4.2. Die in wxWidgets enthaltene Erweiterung *OGL 3* (*Object Graphics Library*) dient, mit einer Modifikation, zur Darstellung der Graphelemente. Alle Hauptroutinen zur Ein- und Ausgabe des XML-Datenformats basieren auf *XercesC++* 2.5.0, dem SAX- und DOM-Parser der *Apache Software Foundation* [14].

Die Beschreibung aller grundlegenden Funktionalitäten und des vollständigen Repertoires zur Graphdefinition sind Gegenstand dieser Arbeit. Zur tiefer gehenden Einflussnahme auf die Vorgänge in der Software ist die Einarbeitung in die Mechaniken von *wxWidgets* allerdings unerlässlich. Die Software macht in ihrem Kern von den Prinzipien *MDI* und *Document/View*, wie sie in *wxWidgets* Verwendung finden, gebrauch. *MDI* steht für *multiple document interface* und beschreibt die Fähigkeit einer GUI-Anwendung, mehrere Dokumente gleichzeitig unter einer einheitlichen Oberfläche zur Bearbeitung anzubieten. *Document/View* steht für das Prinzip der Trennung von Datenhaltung und Datenansicht. Integraler Bestandteil dieses Konzepts ist die Fähigkeiten, verschiedene Sichten (*Views*) auf die gleichen Daten (*Documents*) zu ermöglichen. Die Vermittlung des Verständnisses der Arbeitsweise dieser beiden Muster in *wxWidgets* ist nicht Gegenstand dieser Arbeit und wird für die Definition eigener Netzklassen auch nicht benötigt.

Die zur Darstellung aller graphischen Elemente verwendete Bibliothek *OGL* ist Teil der *wxWidgets* Installation und setzt auf einen ähnlichen Entwurf zur Verwaltung ihrer Objekte, wie diese Arbeit. Einige Entscheidungen bezüglich der Implementierung sind bei mir von der Funktionalität und Mechanik

innerhalb der *OGL* abhängig gemacht worden. Dennoch denke ich, dass das Ziel der Trennung von Struktur und Graphik in der vorliegenden Version ungleich besser gelungen ist, als im Vorgänger. An den entsprechenden Stellen werde ich näher auf diese Entscheidungen eingehen, sehe aber auch das Verständnis der *OGL* als Voraussetzung zur eigenen Forschung, „unter der Oberfläche“, an.

4.2 Namenskonventionen

In Anlehnung an die *Ungarische Notation* wurde innerhalb der Software eine Konvention für alle verwendeten Bezeichner eingesetzt, die aus einem Präfix und einem aussagekräftigen Namen besteht. Für Klassennamen besteht das Präfix immer aus der Abkürzung `SP` (für SNOOPY) sowie maximal einem oder keinem, durch Unterstrich '_' abgetrennten und gross geschriebenen Kürzel für die Zugehörigkeit innerhalb der Quellcodehierarchie. Immer folgt anschliessend, durch '_' getrennt der Name der Klasse.

Die folgende Tabelle listet alle verwendeten Abkürzungen auf, die zwischen `SP` und dem Namen der Klasse verwendet wurden.

Kürzel	***Bedeutung***	***Erklärung***
`DS`	*datastructure*	Elemente der Datenstruktur
`GR`	*graphic*	Graphische Elemente
`GRM`	*graphical management*	Eventhandler für graphische Elemente
`GUI`	*graphical user interface*	Fenster und Fensterelemente
`GM`	*GUI management*	Elemente des Document/View Frameworks
`MDI`	*multiple document interface*	Elemente des MDI Frameworks
`WDG`	*window gadget*	Bedienelemente, Fensterinhalte
`DLG`	*dialog*	Dialoge

Tabelle 1 - Verwendete Kürzel in Klassennamen

Für Variablen gilt ebenfalls die Verwendung von definierten Präfixen, die in diesem Fall allerdings als erstes den Gültigkeitsbereich und anschliessend den verwendeten Typen definieren. Die Kürzel für Gültigkeitsbereich und Typ sind wieder durch einen Unterstrich '_' getrennt und werden immer Klein geschrieben. Anschliessend folgt, beginnend mit einem Grossbuchstaben, der Bezeichner der Variable.

Kürzel	***Bedeutung***	***Erklärung***
`g`	*global*	Globale Variablen
`m`	*member*	Membervariablen von Klassen
`p`	*parameter*	Funktionsparameter
`l`	*local*	lokale Variablen

Tabelle 2 - Abkürzung für den Gültigkeitsbereich von Variablen

Blocklokale Variablen im innersten Block, zum Beispiel in Schleifenkörpern, müssen nicht das `'l_'` Präfix führen.

Die Abkürzungen für die verwendeten Typen orientieren sich an der *Ungarischen Notation*, verwenden also eine Abkürzung für das Verdeutlichen von Zeigervariablen. Wann immer ein Typenkürzel mit `'p'` beginnt, ist die Variable ein Pointer auf den Typen, der im nachfolgenden Kürzel kodiert ist, `'a'` seinerseits zeigt an, dass es sich um einen Array-Typen handelt, der also mit `[]` alloziiert wurde.

Kürzel	***Bedeutung***	***Erklärung***
`c`	*class object*	Klassenobjekt, häufig als Pointer also `'pc'`
`s`	*string*	Zeichenkette, Objekt von `wxString`
`ch`	*char*	Standard C character
`l`	*list*	Liste, fast immer nach `std::list`
`n`	*number*	Fest- oder Fliesskommazahl
`ts`	*set*	Menge, immer `std::set`
`m`	*map*	Map, immer `std::map`
`it`	*iterator*	Iterator
`b`	*bool*	Boolsche Variable

Tabelle 3 - Abkürzungen der Typen in Variablennamen

Vereinzelt, vor allem im Zusammenhang mit dem Map-Typen, wurde auch der Typ des Value-Eintrags im Namen kodiert (`'m_mlAnimatorRegistry'` steht zum Beispiel für Map von Listen).

4.3 Erweiterung des Entwurfs

Die Erfahrungen der ersten Implementierung hatten gezeigt, dass der zugrunde liegende Entwurf nur mit einigen Erweiterungen als tragende Grundlage für eine einsatzfähige Software dienen konnte. Diese Änderungen sollten nichts am grundlegenden Design ändern, da dessen Kern, also die Identifikation der elementaren Bestandteile eines Graphen und dadurch die Möglichkeit der generischen Definition, Grundvoraussetzung für die Anwendung ist.

Ich stelle in diesem Abschnitt alle Klassen vor, die genau diesen Kern ausmachen und gehe im Detail auf einige der Änderungen im Vergleich zum ursprünglichen Entwurf ein und begründe diese Änderungen.

4.3.1 Basisklassen, Templates und Makros

Die Separierung von Funktionalitäten und ihre Kapselung in eigene Klassen unter der Massgabe der Wiederverwendung ist einer der Eckpfeiler des objektorientierten Entwurfs und ebenso der vorliegenden Implementierung.

Die grundlegendsten Klassen, die in der Software Verwendung finden, sind `SP_Name`, `SP_Type`, `SP_NetElement`, `SP_Error` und `SP_IdCounter`. Sie sind Basis für zum Teil verschiedenste Klassen in der Vererbungshierarchie und kapseln einen minimalen Satz an Funktionalität:

`SP_Error`

Objekte von abgeleiteten Klassen besitzen die Fähigkeit, einen aufgetretenen Fehler zu Speichern und an Andere Objekte auf Nachfrage Auskunft zu erteilen. In der vorliegenden Implementierung ist es eine textuelle Repräsentation, die auch zur Darstellung in Dialogen genutzt wird.

`SP_IdCounter`

Jedes Objekt einer Ableitung erhält zur Konstruktionszeit eine eineindeutige Identifikationsnummer und erlaubt den Zugriff auf diese ID.

`SP_Name`

Zugriff auf ein und Speicherung eines String-Attributs, das den Objekten von abgeleiteten Klassen einen „Namen“ gibt.

`SP_NetElement`

Steht für alle Funktionalitäten, die ein Objekt als Teil eines Graphen ausmachen. Dazu gehört, vor allem, die Verwaltung der Zugehörigkeit zu einem bestimmten Teilnetz anhand einer Nummerierung.

`SP_Type`

Erlaubt anhand seines Enumerations-Attributs die Typisierung von abgeleiteten Objekten um Unterscheidungen treffen zu können (z.B. zwischen Knoten, Kante und Attribut).

Einige der von diesem Klassen definierten Methoden sind als virtuell deklariert und können und werden in einigen abgeleiteten Klassen überschrieben.

Die meisten der, im Entwurf definierten Mengen sind durch eine Liste implementiert. Dabei wurde fast überall der Einsatz von `std::list`, also der Implementierung der Liste aus der STL (*standard template library*), forciert. Gleiches gilt für die ebenfalls verwendeten Datenstrukturen für Maps (`std::map`), Sets (`std::set`) oder Vektoren (`std::vector`). Die Templatedefinitionen wurden allerdings allesamt in eigenen Templates gekapselt.

`SP_List`

Templatetyp nach der Vorlage von `std::list`, erweitert um eine lineare Suche `SP_List::find`.

`SP_Map`

Templatetyp nach der Vorlage von `std::map`.

`SP_Set`

Templatetyp nach der Vorlage von `std::set`.

`SP_Vector`

Templatetyp nach der Vorlage von `std::vector`.

Darüber hinaus wurden mehrere Makros definiert, die beim Verständnis der abgedruckten Quelltexte eine Rolle spielen. Sie dienen der vereinfachten Schreibung und damit der besseren Lesbarkeit des Quelltextes und verhalten sich, je nach gewähltem Ziel (Debug- oder Release) verschieden.

`SP_LOGMESSAGE(s)`

Wird vom Präprozessor im Debug-Modus zu einer Anweisungsfolge expandiert, welche die übergebene Zeichenkette `'s'` durch eine Methode von *wxWidgets* auf das gerade aktive Logtarget umleitet. Dieses Target ist ein Textfenster, das im Debug-Modus aktiv ist und so über die Vorgänge informieren soll, ohne den Debugger starten zu müssen.

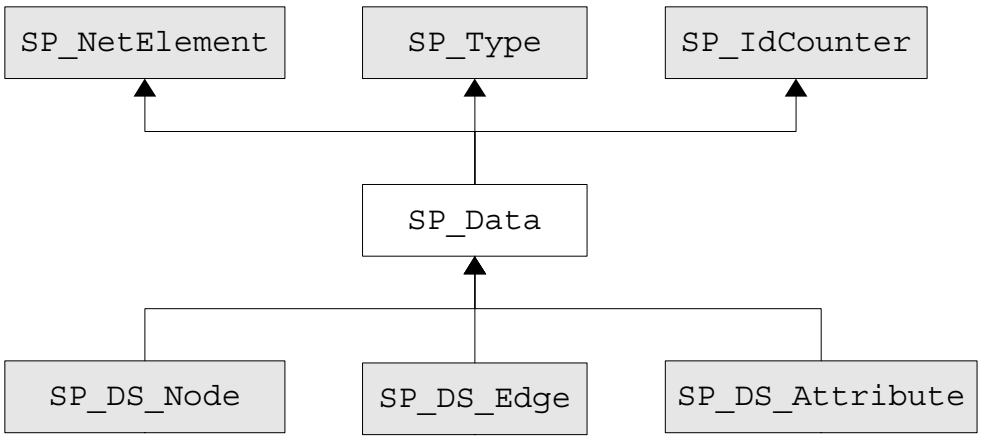
`CHECK_POINTER(P, RET)`

Kann von allen Klassen verwendet werden, die von `SP_Error` abgeleitet werden und testet den Parameter `'P'` auf die Identität mit `NULL`. Im Falle, dass diese Identität zutrifft, wird eine Zeile mit einem generischen Fehlertext sowie dem Dateinamen und der entsprechenden Zeile der Quelltextdatei generiert und über die entsprechende Methode von `SP_Error` gespeichert, sowie diese Fehlerzeile mittels `SP_LOGMESSAGE` ausgegeben.

`CHECK_BOOL(B, RET)`

Arbeitet wie `CHECK_POINTER`, abgesehen davon, dass der Test nicht auf `NULL` erfolgt und darüber hinaus der Text der generierten Fehlermeldung einen Hinweis auf die Tatsache, dass es sich um einen booleschen Ausdruck handelte, enthält.

4.3.2 SP_Data



Zeichnung 5 - SP_Data als Erweiterung des Entwurfs

Eine Vielzahl von Operationen, der im Entwurf identifizierten Klassen, sind identisch. Das bezieht sich vor allem auf die Verbindung der Datenstruktur-Elemente zu den assoziierten graphischen Ausprägungen. Für diese Beziehung ist es zur Laufzeit unerheblich, von welchem konkreten Typen zum Beispiel eine Graphik ist. Aus diesem Grund wurde der Entwurf durch die Klasse `SP_Data` ergänzt. Als Komplement wurde `SP_Graphic` genau als die Klasse *Graphic* aus dem Entwurf umgesetzt.

Durch den einzigen öffentlich zugänglichen Konstruktor von `SP_Data`, der mit dem Eintrag aus einem Aufzählungstypen aufgerufen werden muss, wird gefordert, dass alle drei Ableitungen sich identifizierbar machen. Das bedeutet, dass eine Instanz von `SP_DS_Node` ihre Basis `SP_Data` immer mit `SP_ELEMENT_NODE` initialisiert, was wiederum durch `SP_Type` gespeichert und verwaltet wird. Gleiches gilt für die Kanten und Attribute und ermöglicht es, jedes `SP_Data`-Objekt in der ersten Ebene einer der drei Kategorien, die die einzigen Unterscheidungen in der Datenstruktur darstellen, zuzuordnen.

Durch `SP_IdCounter` erhält jede Instanz einen individuellen numerischen Identifikator zugeordnet, der im Zusammenhang mit den Vorgängen beim Laden und Speichern der Strukturen eine wichtige Rolle spielt. `SP_NetElement` kapselt alle Funktionalitäten, die eine Instanz als zu einem bestimmten Netz zugehörig definieren, also einfach gesprochen eine Netz-Nummer und die Zugriffsmethoden.

SP_Data selbst fügt zu diesen Möglichkeiten noch alle wichtigen Methoden zur Verwaltung der graphischen Ausprägungen hinzu. Die wichtigsten Funktionalitäten sind in dem folgenden Ausschnitt zusammengefasst.

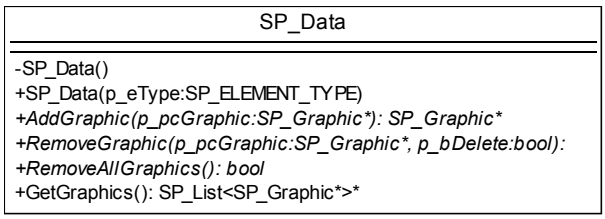


Abbildung 1 - SP_Data, ausgewählte Methoden

Datenhaltung

Eine Besonderheit ist die Datenhaltung der Liste graphischer Ausprägungen für die, von SP_Data abgeleiteten, Objekte. In der ersten Implementierung aggregierte jedes Datenstrukturobjekt eine Liste, die mit dem Template aus std:list verknüpft war und nicht dynamisch instanziiert wurde. In Vorbereitung der neuen Version haben Tests auf einem PC unter Verwendung der, mit dem Microsoft-C++-Compiler, ausgelieferten STL, gezeigt, dass die Aggregation einer solchen Liste 12 Byte beansprucht. Das bezieht sich auf die Grösse des Typs, also nicht die Grösse eines instanziierten Objekts mit wirklichen Inhalten in der Liste.

Da es eine nicht unerhebliche Menge solcher Instanzen geben kann, da jeder Knoten, jede Kante und jedes Attribut der Datenstruktur von SP_Data erben, erschien es mir wichtig, auf die Belange des Speicherverbrauchs zu achten. Aus diesem Grund wurde die Verknüpfung der SP_Data-Objekte mit der Liste ihrer graphischen Ausprägungen in eine globale Datenstruktur ausgelagert, deren Typ insgesamt 16 Byte ausmacht. Sie bildet die SP_Data-Objekte als Key in einer Map auf Listen von SP_Graphic-Objekten ab.

Um diese Implementierung transparent zu machen, blieben die notwendigen Methoden (graphische Ausprägung hinzufügen/löschen/erfragen) Element der Klasse SP_Data, arbeiten aber alle auf der globalen Map-von-Listen. Damit ist der Typ SP_Data im Moment genau 28 Byte gross, im Gegensatz zu 40 Byte, bei Verwendung einer aggregierten Liste direkt in der Klasse, die unter Umständen keine Einträge enthält, da nicht jedes Element der Datenstruktur eine graphische Ausprägung haben *muss*.

Die drei Ableitungen SP_DS_Node, SP_DS_Edge und SP_DS_Attribute entsprechen den jeweiligen Klassen im Entwurf und SP_Data definiert nur einen minimalen Satz an Funktionalität, der allen drei Ableitungen gemein ist und legt darüber hinaus die wichtigsten Schnittstellen fest. So könnte theoretisch jede Ableitung von SP_Data auf die Möglichkeiten der Verbindung mit anderen Objekten durch Kanten reagieren – die entsprechenden Methoden sind allerdings bislang nur in SP_DS_Node überschrieben und mit Sinn gefüllt worden.

4.3.3 SP_DS_Coarse

SP_DS_Coarse steht für eine der Forderungen des Abschnitts „Hierarchien und Mehrdeutigkeiten“ auf Seite 18, in dem es um die Möglichkeit der Strukturierung von Graphen in Unternetze ging.

Diese Klasse ist nur von SP_Error abgeleitet um über aufgetretene Fehler informieren zu können und wird von Objekten von SP_Data aggregiert, wodurch sie, wenn es einen gültigen Wert in SP_Data::m_pcCoarse geben sollte, an allen Vorgängen zur Laufzeit automatisch beteiligt wird.

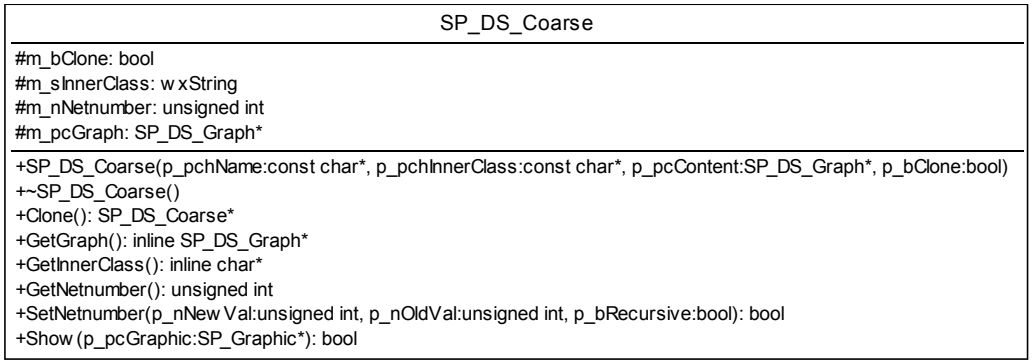


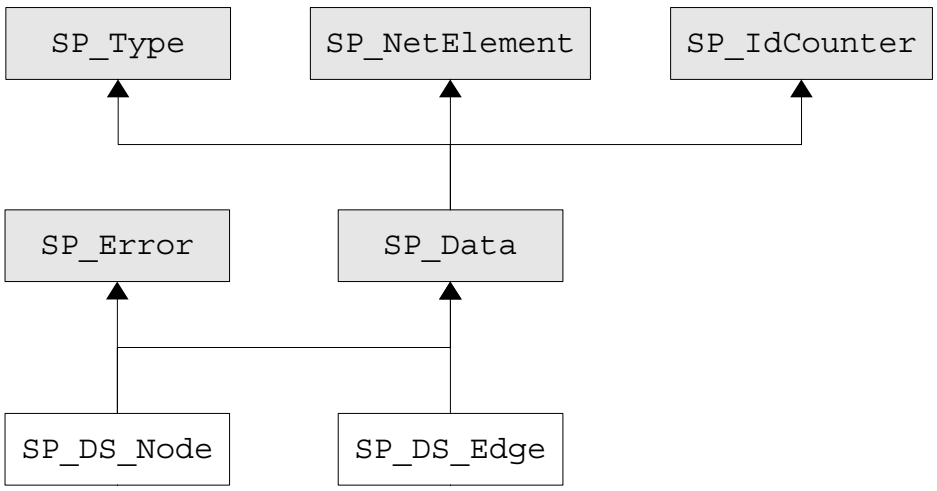
Abbildung 2 - SP_DS_Coarse, ausgewählte Methoden und Attribute

Die zentralen Informationen, die SP_DS_Coarse verwalten muss sind zum Einen ein numerischer Identifikator, der für die Nummer des Unternetzes steht, das ein Objekt dieser Klasse repräsentiert (m_nNetnumber). Ausserdem den Bezeichner einer Partitionsmenge, von deren Typ die Elemente am „inneren Rand“ sein sollen (m_sInnerClass). Das sind genau die Elemente, die eine Verbindung zu Elementen aus dem Obernetz haben dürfen. Anhand dieses Bezeichners kann bei Bedarf ein Element in der

Verantwortlichkeit dieser Klasse erzeugt werden, ohne dass der Nutzer direkt darauf Einfluss nimmt. Und ausserdem hält SP_DS_Coarse noch eine Referenz auf ein Objekt der Klasse SP_DS_Graph, in deren Partitionen alle Elemente innerhalb des Subnetzes definiert sein müssen (m_pcGraph). Das boolesche Flag m_bClone kann über den Konstruktor initialisiert werden und dient als Entscheidungshilfe, wenn es darum geht, eine Kopie anzulegen, denn auch die Klasse SP_DS_Coarse muss, durch seine Aggregation in SP_Data, die Fähigkeit zur Reproduktion identischer Kopien haben.

4.3.4 SP_DS_Node und SP_DS_Edge

SP_DS_Node und SP_DS_Edge sind die wichtigsten Derivate von SP_Data und repräsentieren Elemente der Knoten- bzw. Kantenmenge eines Graphen. Sie sind Blätter in ihrem jeweiligen Zweig der Vererbungshierarchie, mithin ist jeder Knoten in der Datenstruktur Instanz der Klasse SP_DS_Node und jede Kante Instanz der Klasse SP_DS_Edge.



Zeichnung 6 - SP_DS_Node und SP_DS_Edge

Von SP_Data erben beide Klassen alle Attribute und Methoden, die ihnen das Verwalten ihrer graphischen Ausprägungen erlauben, sowie die Identifikation anhand eines Typs (SP_ELEMENT_NODE bzw. SP_ELEMENT_EDGE), die eineindeutige Identifikation anhand eines numerischen Wertes und die

Zuordnung zu einer Netznummer. Durch die Ableitung von `SP_Error` sind beide Klassen in der Lage, auftretende Fehler bei der Arbeit innerhalb ihrer Methoden anzuzeigen.

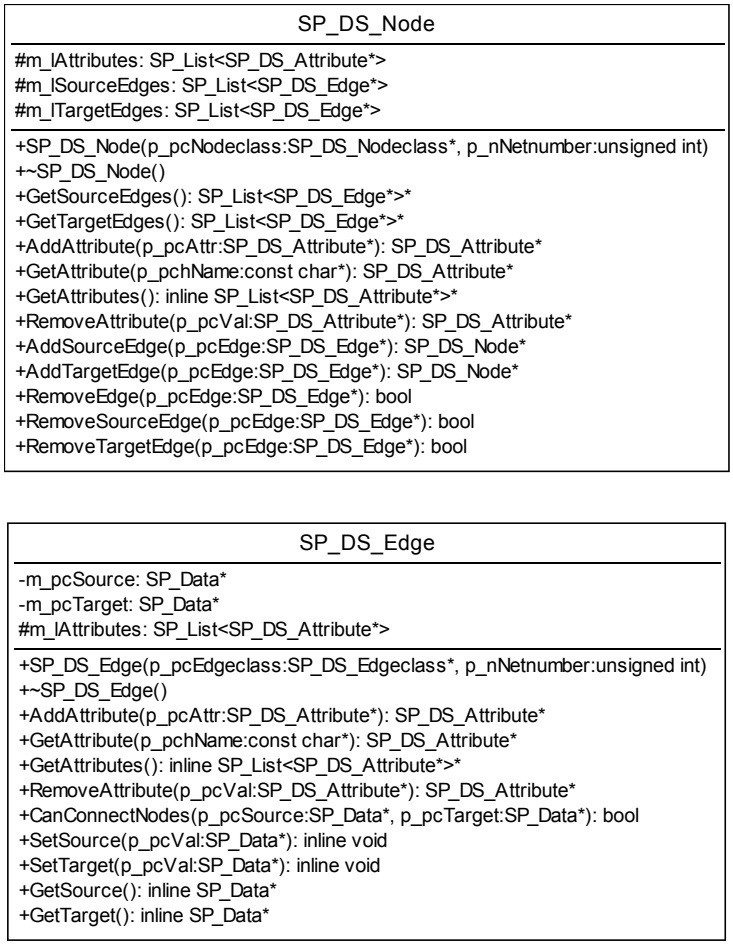


Abbildung 3 - SP_DS_Node, SP_DS_Edge, ausgewählte Methoden

Einzigartig für beide Klassen ist die Aggregation einer Datenstruktur für die, ihnen zugeordneten, Attribute. Hierbei wurde bewusst auf das Auslagern der verwendeten Liste in eine separate Struktur verzichtet, da es eine essentielle Eigenschaft für Knoten und Kanten ist, Attribute zu besitzen. Der Fall, dass ein Objekt einer der Klassen keine Attribute in der Datenstruktur besitzt, sollte ein seltener Fall sein.

Als Möglichkeiten der Manipulation dieser Liste von Attributen bieten beide Klassen ähnliche Methoden. Dazu gehört das Hinzufügen eines neuen Attributs, das Entfernen eines solchen und der Zugriff auf die Elemente, entweder durch Iteration oder gezielten Zugriff anhand des Namens oder Typens (siehe auch Abschnitt „SP_DS_Attribute“ auf Seite 34).

Die Klasse `SP_DS_Node` enthält darüber hinaus noch Methoden und Attribute zur Verwaltung von ausgehenden oder eingehenden Kanten. Das bedeutet, es können Einträge in die Listen `m_lSourceEdges` und `m_lTargetEdges` hinzugefügt und aus ihnen entfernt werden. Analog dazu bietet die Klasse `SP_DS_Edge` Methoden zum Setzen der Attribute `m_pcSource` und `m_pcTarget`, die entsprechenden Instanzen von `SP_DS_Node` an den jeweiligen Enden. Diese Namenswahl impliziert nicht zwangsläufig eine Richtung im Sinne der Graphentheorie, sondern verdeutlicht nur die „Richtung“, die der Benutzer der Software beim Ziehen der Kante in der Oberfläche vorgegeben hat. Die Semantik dieser Attribute, wie auch die der Aufteilung in eingehende und ausgehende Kanten in der Klasse `SP_DS_Node`, liegt im Ermessen des Designers der Netzklasse.

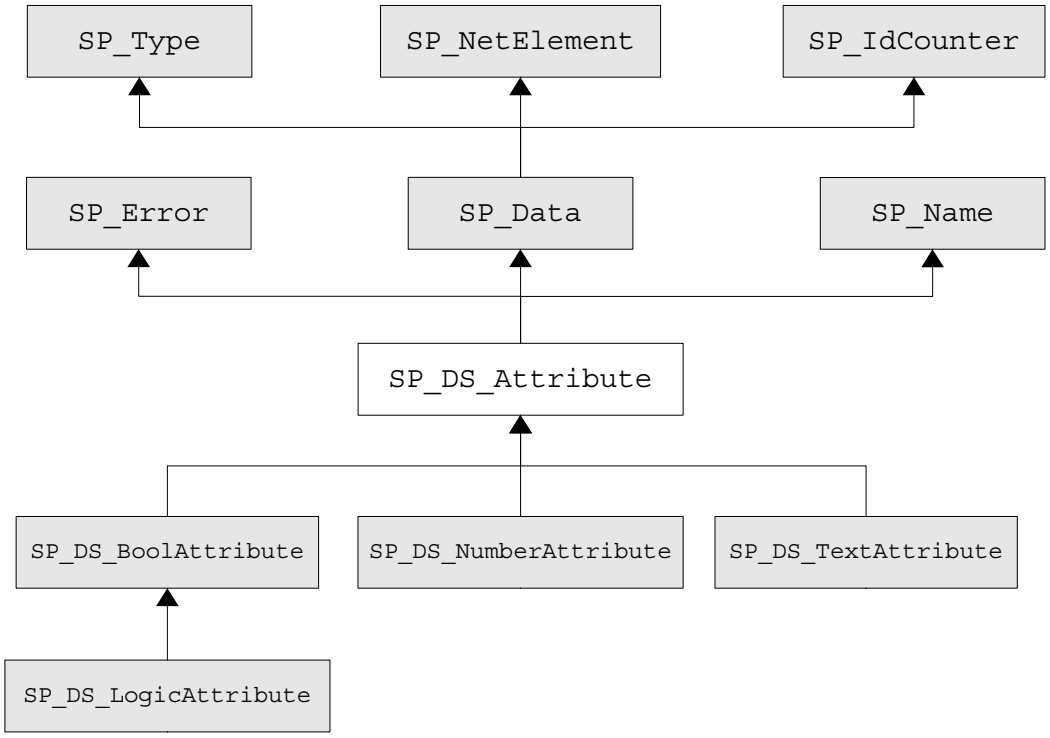
Alle Methoden der Klasse `SP_DS_Node` und `SP_DS_Edge`, die sich mit dem Entfernen von Attributen beschäftigen, sind nicht destruktiv, das heisst, es wird kein Speicher durch den Aufruf von `delete` wieder verfügbar gemacht. Das ermöglicht es zum Beispiel, ein bestehendes Attribut aus einem Objekt zu entfernen und in ein anderes einzufügen. Zu diesem Zweck wird bei den entsprechenden Methoden überprüft, ob das Attribut schon zu einem anderen Element gehört, beziehungsweise beim Entfernen diese Beziehung wieder zurückgesetzt.

Eine Ausnahme bezüglich des Speichermanagements bildet der Aufruf des Destruktors. Sowohl `~SP_DS_Edge` als auch `~SP_DS_Node` iterieren über die Liste ihrer Attribute und geben den Speicher der dort vermerkten Einträge frei. Das hat seine Ursache darin, dass jede der beiden Klassen über ihre `Clone`-Methode identische Abbilder von Objekten ihres Typs erzeugen kann und die Liste der Attribute dabei ebenfalls für jeden Eintrag kopiert und vervielfacht wird. Da es keine Attribute ohne zugehöriges „Elternelement“ geben soll, endet die Lebenszeit der entsprechenden Objekte, per Definition, mit dem Ende der Lebenszeit des Knotens oder der Kante.

Ausserdem fordert keine Instanz von `SP_DS_Node` beim Umgang mit seinen ein- und ausgehenden Kanten Speicher für diese an, noch wird der Speicher wieder frei gegeben, wenn einer der Einträge entfernt wird. Es werden lediglich Einträge in die entsprechenden Listen hinzugefügt und wieder entfernt. Gleiches gilt für die Referenzen in der Klasse `SP_DS_Edge`, wo das „Entfernen“ eines Knotens nur das Setzen des dafür vorgesehenen Zeigers auf `NULL` bedeutet.

Die Begründung dafür liegt in der Tatsache, dass das Erzeugen der konkreten Instanzen Aufgabe der übergeordneten *Klasse* (`SP_DS_Nodeclass` bzw. `SP_DS_Edgeclass`) ist, was diese ebenso auch für das Zerstören und Freigeben des Speichers zuständig macht.

4.3.5 SP_DS_Attribute



Zeichnung 7 - Vererbungshierarchie für SP_DS_Attribute

Die Klasse `SP_DS_Attribute` definiert die Schnittstelle für alle konkreten Attribute, die Teil der Datenstruktur sind. Sie selbst, oder eine ihrer Ableitungen muss Basis für jedes weitere Attribut sein, das zur Erweiterung der Funktionalitäten eingeführt wird.

Wie SP_DS_Node und SP_DS_Edge erhält SP_DS_Attribute durch die Vererbung alle Fähigkeiten zur Verwaltung von graphischen Ausprägungen, die Identifikation anhand eines Typs (SP_ELEMENT_ATTRIBUTE) und einer eineindeutigen Nummerierung, die Zugehörigkeit zu einem Teilnetz, sowie durch SP_Error die Möglichkeit, auftretende Fehler zu speichern und weiterzumelden. Neu ist die Vererbung von SP_Name, wodurch jede Ableitung durch einen Namen, also eine textuelle Repräsentation identifizierbar wird. Diese Festlegung hat damit zu tun, dass ein Attribut an einem Knoten oder einer Kante zur Definitionszeit, keine Möglichkeit der Identifikation bietet. Da an Attribute in der Regel eine Bedeutung geknüpft ist, sollte sich diese in dem assoziierten Namen wiederfinden, was es dem „Benutzer“ des fertigen Graphen erleichtert, die vorhandenen Attribute zu füllen. In einem Graphen, der zum Beispiel eine Landkarte modellieren soll, macht es Sinn, einem Knoten ein Textattribut einzuschreiben, das den Namen der Stadt, für den dieser Knoten des Graphen steht, tragen soll. Sinnvollerweise würde dieses Attribut dann „Name“ heissen und sich dadurch zum Beispiel von einem weiteren Textattribut namens „Bundesland“ abgrenzen.

SP_DS_Attribute fügt eine weitere Typisierung ein, die all ihre Ableitungen nicht nur als Attribut im Sinne der Datenstruktur identifizierbar macht (über die Vererbung von SP_Type), sondern darüber hinaus auch noch als Objekt eines bestimmten Attribut-Typs. So wurden für die vorhandenen Ableitungen schon Typen, wie SP_ATTRIBUTE_TEXT oder SP_ATTRIBUTE_NUMBER eingeführt und jede neue Ableitung muss sich durch einen eigenen Eintrag in dem entsprechenden Enumerationstypen verewigen, da diese Möglichkeit der Typisierung zum Beispiel im Zusammenhang mit der Darstellung in Dialogen eine wichtige Rolle spielt.

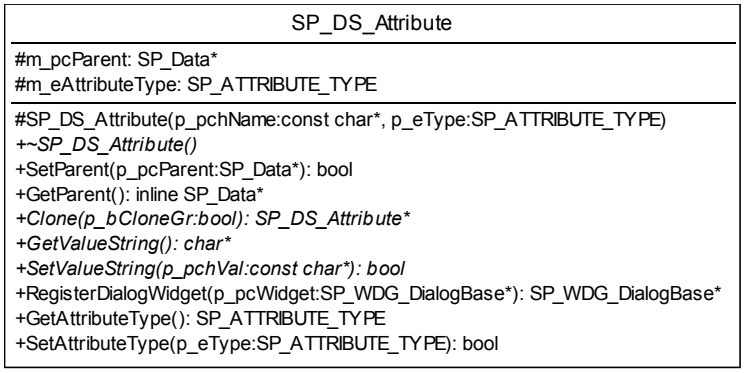


Abbildung 4 - ausgewählte Methoden

Die Klasse `SP_DS_Attribute` definiert genau drei wichtige abstrakte Methoden. Das ist zum Einen die Methoden zum Erzeugen exakter Kopien (`SP_DS_Attribute::Clone`) sowie zwei Methoden, die eine einheitliche Möglichkeit des Zugriffs auf den Wert eines Attributs ohne Kenntnis seines Typs erlauben. Aus den Erfahrungen der ersten Implementierung habe ich mich auch hier dazu entschlossen, eine Serialisierung in ein textuelles Format über die Methoden `::GetValueString` und `::SetValueString` zu fordern. Für die bislang implementierten Datenstruktur-Attribute ist diese Möglichkeit der Serialisierung einsehbar praktikabel. Das `SP_DS_TextAttribute` benötigt keine weitere Umwandlung und liefert den Wert seines entsprechenden Attributs, beziehungsweise übernimmt einfach den Wert des Parameters. Das `SP_DS_NumberAttribute` „druckt“ seinen Wert in eine Zeichenkette, oder versucht die Konvertierung des Parameters bei `SetValueString` in eine numerische Entsprechung. Das `SP_DS_BoolAttribute` arbeitet ebenfalls auf Ziffern, nur dass es als Kodierung seines Zustandes „0“ (für `FALSE`) und „1“ (für `TRUE`) verwendet.

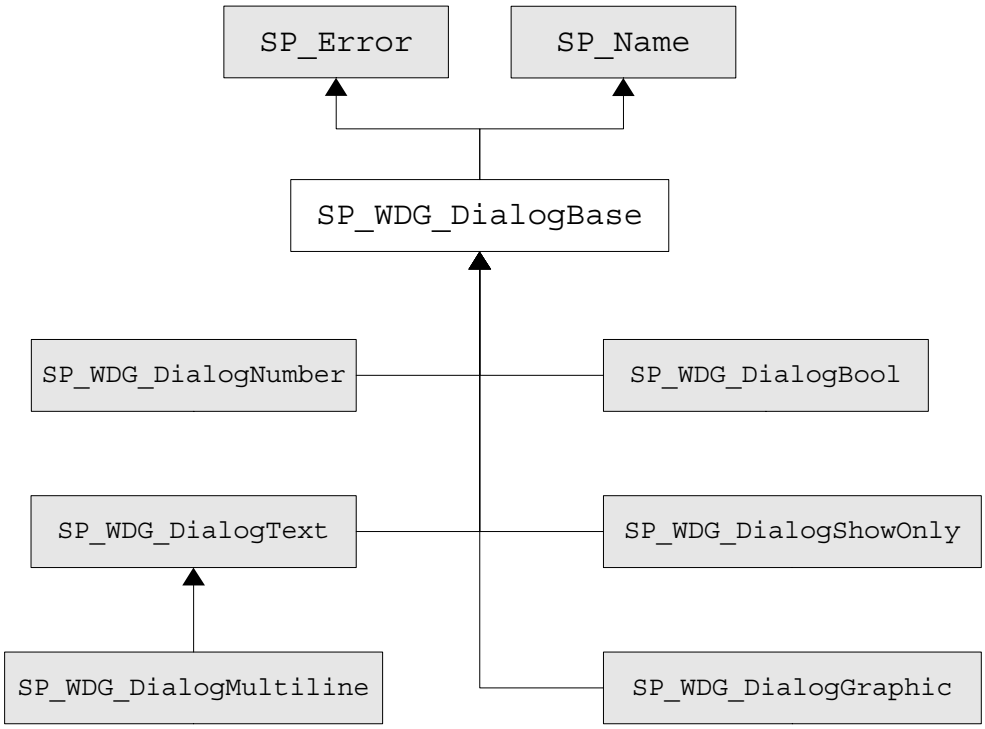
Durch die Anforderung der abstrakten Definition, so dass jedes neue Attribut diese beiden Methoden selbst implementieren muss, liegt es in der Verantwortlichkeit des Entwicklers, für eine sichere Konvertierung seines individuellen Attributs in und aus einer Textvariable zu sorgen. Selbst komplexe Strukturen können durch die Verwendung eineindeutiger Separatoren in Kombination mit dem Kodieren und Dekodieren der dazwischenliegenden Teile (zur Verhinderung von Fehlerkennungen, falls der Separator Teil des Wertes sein sollte) in diesem einfachen Format transportiert werden.

Die Fähigkeit zur Serialisierung und Deserialisierung ermöglicht es, diesen kleinsten gemeinsamen Nenner sowohl zur Anzeige oder Bearbeitung, als auch zum Laden und Speichern einzusetzen. Die vorliegende Implementierung setzt dabei auf ein `XML`-Format, wodurch die Attribut-Werte in so genannten `CDATA`-Knoten (für *character data*) gespeichert und aus diesen wieder geladen werden können – wohingegen es die erste Implementierung erforderte, dass jedes Attribut selbst die Methoden zum Laden und Speichern in einem `XML`-Format implementieren muss.

4.3.6 SP_WDG_DialogBase

Wie schon im Entwurf erwähnt, soll es den Nutzern der erstellten Netze möglich sein, Werte in den verwendeten Attributen zu modifizieren. Der in der ersten Implementierung verfolgte Ansatz, jedem Attribut die Implementierung von Methoden zur Darstellung in einem Dialog zwingend vorzuschreiben, wurde verworfen. Stattdessen kann nun jedem Attribut zur Definitionszeit ein Widget vorgeschrieben werden, das zur Darstellung im Dialog genutzt werden soll und das für das Zurückschreiben der Werte in die Datenstruktur verantwortlich ist.

Diese Verknüpfung erfolgt nicht innerhalb der Klasse SP_DS_Attribut. Stattdessen wird die Registrierung eines Attributs für ein Dialog-Widget durch `SP_DS_Attribute::RegisterDialogWidget` in einer zentralen Registry vermerkt und bei Bedarf mit Kopien dieses registrierten Widgets gearbeitet (siehe auch Abschnitt „SP_Core“ auf Seite 49). Zu diesem Zweck wurde eine Hierarchisierung eingeführt, die sich auf verschiedenste Widgets konzentriert und durch Ableitung von einer Basisklasse die Schnittstellen festlegt.



Zeichnung 8 - Widgets in Dialogen und ihre Vererbung

Die Definition von `SP_WDG_DialogBase` gibt drei Arten von virtuellen Methoden vor, die in ihren Ableitungen überschrieben werden können und Instanzen von ihrem Typen werden im dazugehörigen Dialog gesammelt und zur Anzeige gebracht.

SP_WDG_DialogBase
#m_tlAttributes: SP_List<SP_DS_Attribute*>
+SP_WDG_DialogBase(p_sPage:const char*) *+~SP_WDG_DialogBase()* *+Clone(): SP_WDG_DialogBase** *+AddToDialog(p_ptlAttributes:SP_List<SP_DS_Attribute*>*, p_pcDlg:SP_DLG_ShapeProperties*):* *+AddToDialog(p_plGraphics:SP_List<SP_Graphic*>*, p_pcDlg:SP_DLG_ShapeProperties*): bool* *+OnDlgOk(): bool*

Abbildung 5 - SP_WDG_DialogBase, ausgewählte Methoden

Dazu gehört zum Einen die Forderung, wie sie auch schon aus der Datenstruktur bekannt ist, dass jedes Widget exakte Kopien seiner selbst über seine `Clone`-Methode erzeugen kann. Die beiden `::AddToDialog`-Methoden, die

automatisch immer dann aufgerufen werden, wenn es darum geht, ein Attribut zur Bearbeitung anzuzeigen, sind der Platz für die Festlegung der Dialogelemente zur Modifikation der Werte. Also zum Beispiel eine Eingabezeile für Texte, oder Buttons für Eventbearbeitung, und so weiter. Die zweite Definition, die auf einer Liste graphischer Ausprägungen arbeitet, hat bislang nur im Zusammenhang mit `SP_WDG_DialogGraphic`, dem Widget zur Bearbeitung graphischer Attribute wie Vorder- und Hintergrundfarbe, eine Bedeutung. Da es in einigen Situationen mehrere Attribute des gleichen Typs und des gleichen Namens geben kann (wenn zum Beispiel mehrere identische Elemente zur Bearbeitung selektiert wurden), hält jedes Widget eine Liste der konkreten Attribute, für die es zur Darstellung genutzt wird. Dadurch kann jedes Widget in Eigenverantwortung beim Aufruf von `::OnDlgOk` die Änderungen in seine Attribute schreiben. Dieser Aufruf erfolgt, wie schon die Aufforderung zum Einfügen der Widgets in den Dialog, automatisch.

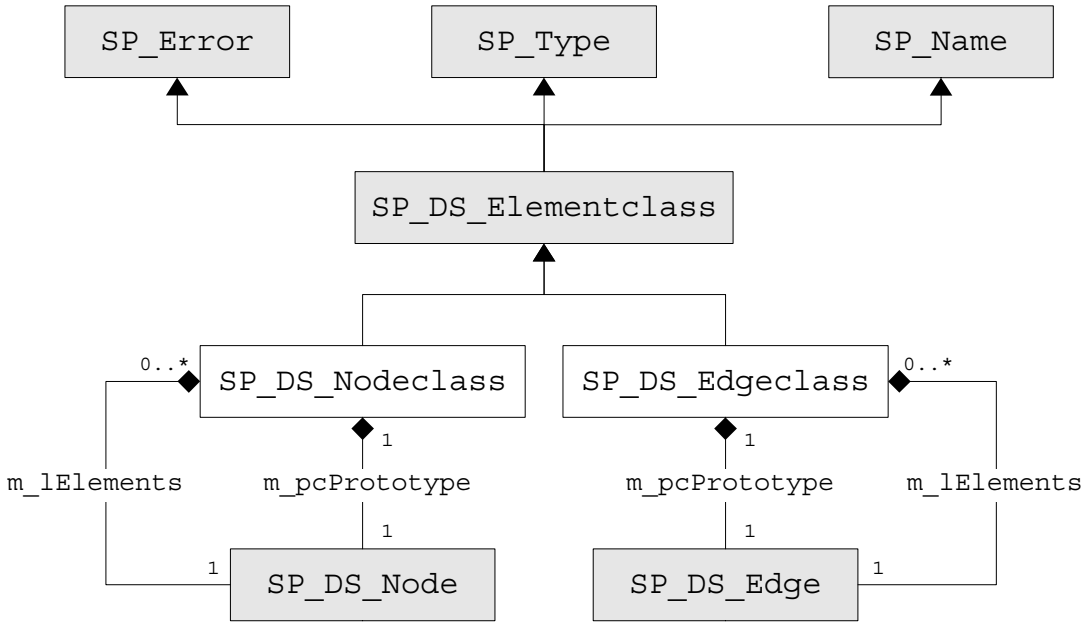
Die bestehenden Ableitungen stehen für verschiedene Eingabemöglichkeiten für textuelle (auch mehrzeilige), numerische und boolesche Attribute. `SP_WDG_DialogShowOnly` kann genutzt werden, um nur die Anzeige des Attributs ein- oder auszuschalten ohne die Werte zu manipulieren und `SP_WDG_DialogGraphic` stellt, wie schon erwähnt, Elemente zur Wahl von Vorder und Hintergrundfarbe der gewählten graphischen Ausprägungen zur Verfügung.

Für alle Attribute, die sich nicht in einem Dialog bearbeiten lassen sollen, entfällt einfach der Aufruf von `::RegisterDialogWidget` zur Definitionszeit, woraufhin es bei der Suche nach registrierten Widgets für dieses Attribut keine Treffer gibt.

Wie am einzigen Konstruktor von `SP_WDG_DialogBase` ersichtlich ist, werden alle Widgets mit einem Namen initialisiert (zur Speicherung dieses Wertes auch die Ableitung von `SP_Name`), der in diesem Fall aber eine etwas andere Bedeutung als zum Beispiel bei Attributen hat. Der Dialog, der zur Anzeige der verschiedenen Widgets benutzt wird, bietet die Möglichkeit, seine Fenster-Elemente auf so genannten Notebook-Pages zu organisieren. Das sind in der Anwendung einzelne Seiten, die über eine Leiste angesprochen werden können, die wie eine Karteiablage organisiert ist. Jede dieser Seiten kann einen eigenen Namen haben und mit diesem Mechanismus ist es möglich, bestimmte Widgets (und damit bestimmte Attribute) in verschiedenen Unterseiten zu organisieren. Der Dialog stellt eine Methode zur Verfügung, die entweder seinem Notebook eine entsprechende Seite eines neuen Namens hinzufügt, oder eine schon existierende Seite zurück liefert, wodurch jedes Widget darauf vertrauen kann, dass es genau der Seite zugeordnet wird, die der Designer der Netzklasse im Sinn hatte.

4.3.7 SP_DS_Nodeclass und SP_DS_Edgeclass

Diese beiden Klassen entsprechen den Klassen *Edgeclass* beziehungsweise *Nodeclass* aus dem Entwurf und repräsentieren eine Partition aus der Gesamtmenge der Knoten oder Kanten eines Graphen.



Zeichnung 9 - Klassendiagramm für SP_DS_Nodeclass und SP_DS_Edgeclass

`SP_DS_Elementclass` kapselt durch Ableitung von `SP_Error`, `SP_Type` und `SP_Name` die Eigenschaften beider Arten von Partitionen, dass sie über einen Typen beziehungsweise einen Namen identifizierbar sind, und dass sie über, in ihrem Zuständigkeitsbereich aufgetretene, Fehler berichten können. Darüber hinaus hält `SP_DS_Elementclass` noch die Information über den Graphen, als dessen Teil diese Partition definiert wurde.

Abgesehen von den konkreten Typen ihrer Elemente verfügen beide Partitionen über den gleichen Satz an Methoden zum Hinzufügen, zum Erfragen und zum Entfernen von Elementen. Die Aggregation einer Instanz von `SP_DS_Node` beziehungsweise `SP_DS_Edge` als prototypisches Element, also die Vorlage, die zur Definitionszeit modifiziert wird, erfolgt automatisch zur Konstruktionszeit einer Instanz dieser Klassen. Das Erzeugen eines neu-

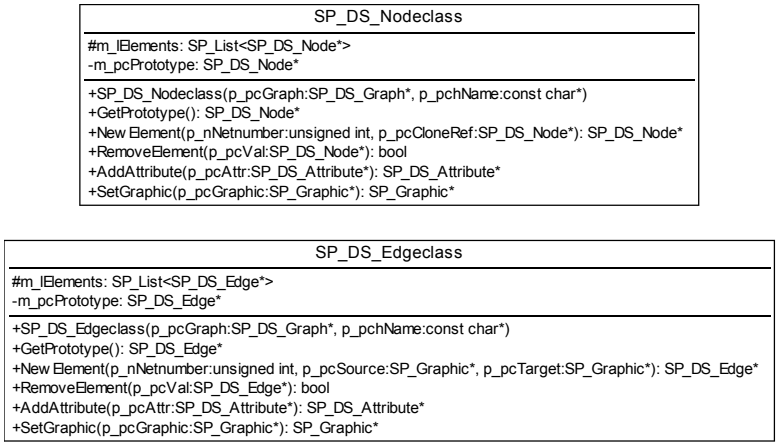


Abbildung 6 - ausgewählte Methoden

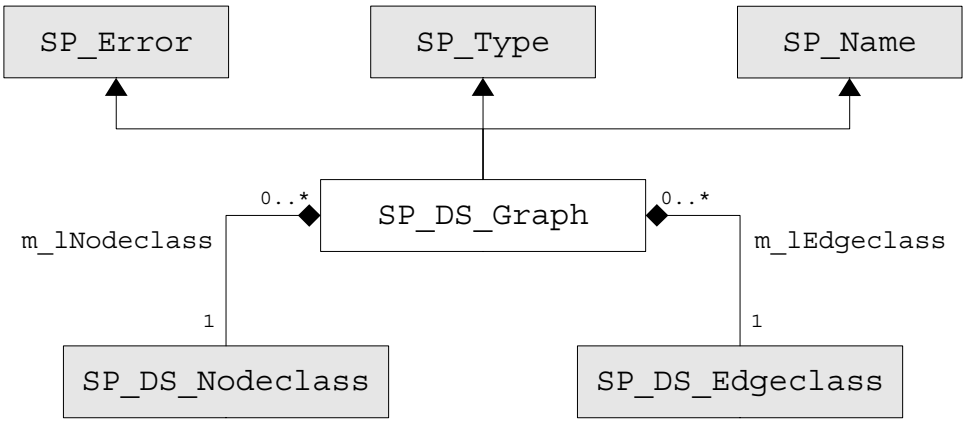
en Elements und das Einfügen in die Liste erfolgt in der Regel implizit durch `::NewElement`, in der eine Kopie des Prototypen durch den Aufruf von dessen `::Clone`-Methode erzeugt und diese im Erfolgsfall der Liste der Elemente hinzugefügt wird. Das stellt sicher, dass sich in der Liste der Elemente immer nur exakte Kopien des Prototypen befinden, diese sich also in der Anzahl und den Typen ihrer Attribute gleichen.

Zur einfacheren Arbeit mit den Einstellungen am prototypischen Element, bieten beide Klassen Methoden, die denen der Klasse des prototypischen Elements entsprechen und direkt an die Instanz des Prototypen weitergereicht werden. Das schliesst vor allem das Hinzufügen und Entfernen eines Attributs und das Assoziieren des Prototypen mit einer graphischen Ausprägung ein.

Von keiner dieser beiden Klassen existieren Ableitungen im Sinne des objektorientierten Designs, das bedeutet, alle Knoten-Partitionen sind Instanzen von `SP_DS_Nodeclass` und jede Kantenpartitionen ist Instanz von `SP_DS_Edgeclass`. Wie man den Konstruktoren ansehen kann, erfolgt die Instanziierung anhand eines Graphen, zu dem die Partition als Teil definiert wird und eines Namens, der diese Partition identifiziert.

4.3.8 SP_DS_Graph

`SP_DS_Graph` ist die Entsprechung der Klasse *Graph* aus dem Entwurf. Jede Instanz dieser Klasse bietet Zugriff auf die verschiedenen Partitionen ihrer Knoten- und Kantenmenge, was das Erweitern und Entfernen von Einträgen einschliesst.



Zeichnung 10 - SP_DS_Graph

Durch die Vererbung von `SP_Error`, `SP_Type` und `SP_Name` sind die Identifikation anhand eines Typen und eines Namens, sowie die Möglichkeit der Speicherung von und Auskunft zu aufgetretenen Fehlern gegeben.

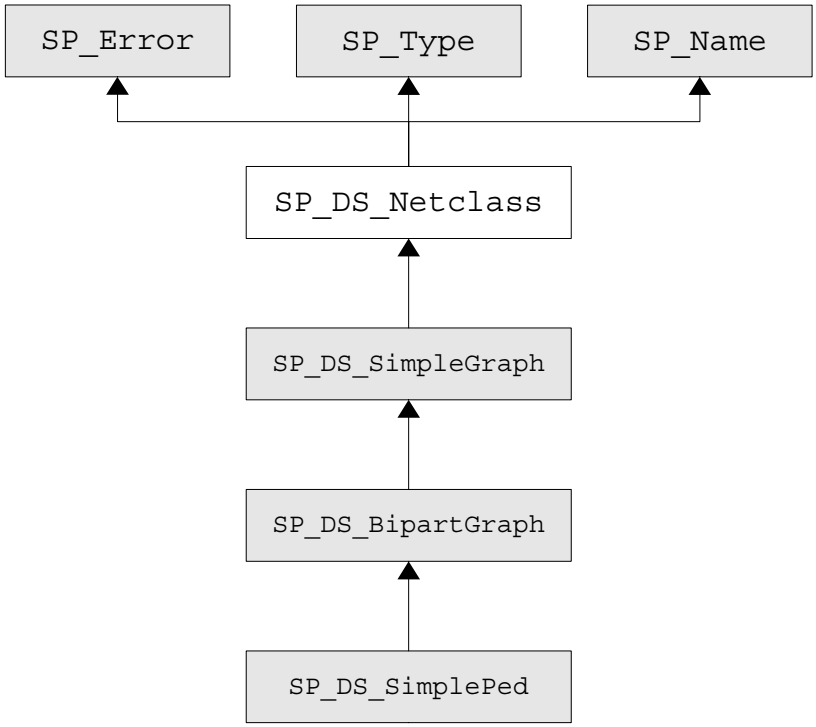
SP_DS_Graph
#m_pcNetclass: SP_DS_Netclass* #m_lNodeclass: SP_List<SP_DS_Nodeclass*> #m_lEdgeclass: SP_List<SP_DS_Edgeclass*>
+SP_DS_Graph(p_pcNetclass:SP_DS_Netclass*) +AddNodeclass(p_pcClass:SP_DS_Nodeclass*): SP_DS_Nodeclass* +GetNodeclass(p_pchName:const char*): SP_DS_Nodeclass* +AddEdgeclass(p_pcClass:SP_DS_Edgeclass*): SP_DS_Edgeclass* +GetEdgeclass(p_pchName:const char*): SP_DS_Edgeclass* +GetNetclass(): SP_DS_Netclass*

Abbildung 7 - ausgewählte Methoden

Das Hinzufügen von neuen Einträgen in die Menge der Knoten- oder Kantenpartitionen (`AddNodeclass` bzw. `AddEdgeclass`) erfolgt direkt über die Übergabe einer neuen Instanz der jeweiligen Partition. Da jede der Partitionsklassen über einen Namen identifizierbar ist, existieren auch Zugriffsmethoden, die die entsprechende Instanz zu einem Namen liefern.

4.3.9 SP_DS_Netclass

Diese Klasse entspricht der Klasse *Netclass* aus dem Entwurf und ist der Dreh- und Angelpunkt der Definition eines eigenen Graphen.



Zeichnung 11 - Vererbungshierarchie von SP_DS_Netclass

Durch die Ableitung von `SP_Type` und `SP_Name` erhalten die Instanzen die Möglichkeit der Identifizierung anhand eines Typen und eines Namens, wobei letzterer auch zur Anzeige in der Oberfläche und zur Speicherung der Daten verwendet wird. Durch `SP_Error` erhält jede Instanz die Fähigkeit aufgetretene Fehler zu speichern und darüber Auskunft zu erteilen.

SP_DS_Netclass
#m_sExtension: wxString
+SP_DS_Netclass(p_pchName:const char*) +~SP_DS_Netclass() *+CreateGraph(p_pcGraph:SP_DS_Graph*): SP_DS_Graph** *+NodeRequirement(p_pcNode:SP_DS_Node*): bool* *+EdgeRequirement(p_pcEdgeclass:SP_DS_Edgeclass*, p_pcNode1:SP_Data*, p_pcNode2:SP_Data*):* +GetExtension(): inline wxString

Abbildung 8 - SP_DS_Netclass, ausgewählte Methoden

Wie auch schon im Entwurf ausgeführt, muss jede Ableitung von `SP_DS_Netclass` drei wichtige Methoden implementieren. Dazu gehört einmal die *Create*-Methode, die hier `CreateGraph` heisst und die zentrale Stelle zur Definition darstellt. Diese Methode wird automatisch aus der Oberfläche heraus aufgerufen, wenn zum Beispiel ein neuer Graph von dieser Netzklasse erzeugt oder ein bestehender geladen werden soll, und hier ist der Platz, an dem mit Hilfe der vorgestellten Methoden das übergebene `SP_DS_Graph`-Objekt modifiziert werden kann.

Die im Klassendiagramm in Zeichnung 11 auf Seite 43 abgeleiteten Klassen stellen konkrete Implementierungen dar. Dabei ist darauf zu achten, dass der Aufruf der `CreateGraph`-Methode der Basisklasse explizit durch den Designer erfolgen muss. Das bedeutet also, dass zum Beispiel `SP_DS_Bipart-Graph::CreateGraph` mit dem Aufruf von `SP_DS_SimpleGraph::CreateGraph` mit dem übergebenen Parameter beginnen sollte. Dieses Vorgehen ist nicht zwingend erforderlich, stellt aber sonst die Ableitung im objektorientierten Sinn in Frage.

Die beiden anderen wichtigen Funktionen sind `EdgeRequirement` und `NodeRequirement` und entsprechen den im Entwurf beschriebenen Funktionalitäten. Über diese kann der Netz-Designer Einfluss auf das Erstellen von Elementen aus der Knoten und Kantenmenge des Graphen nehmen.

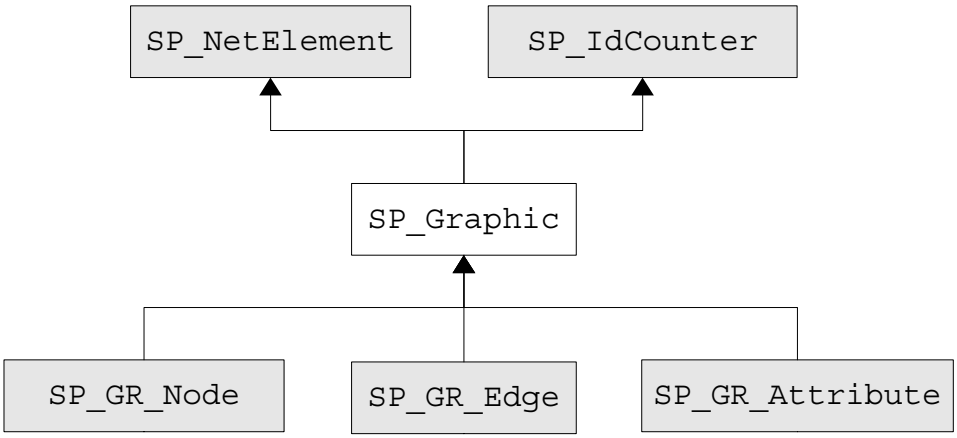
In `NodeRequirement` bekommt man den neu erstellten Knoten übergeben, bevor er in die Liste der Elemente der `SP_DS_Nodeclass` eingefügt wird. Denkbare Notwendigkeiten, diesem Einfügen zu widersprechen wäre zum Beispiel die Limitierung der Anzahl von Knoten einer bestimmten Partition.

In `EdgeRequirement` wird über die anzulegende `SP_DS_Edgeclass` und die beiden zu verbindenden Elemente aus der Knotenmenge informiert. Hier kann man zum Beispiel verhindern, dass Knoten aus der gleichen `SP_DS_Nodeclass` miteinander verbunden werden können, beziehungsweise dass das nur durch Kanten eines bestimmten Typs geschehen darf.

In beiden Fällen entscheidet der boolesche Rückgabewert der Funktionen über die Existenz des Elements und der Aufruf der entsprechenden Methode erfolgt automatisch aus den ereignisgesteuerten Routinen des Hauptprogramms.

4.3.10 SP_Graphic

`SP_Graphic` ist die Umsetzung der Klasse *Graphic* aus dem Entwurf und dient als Basis für die drei, analog zur Datenstruktur ausgelegten, Klassen `SP_GR_Node`, `SP_GR_Edge` und `SP_GR_Attribute`.



Zeichnung 12 - Vererbungshierarchie für SP_Graphic

Durch `SP_Graphic` wird ein Interface definiert, dessen Mindestanforderungen alle Ableitungen genügen müssen. Dazu zählt, als wichtigste der rein abstrakten Methoden, `Clone`, die, wie es der Entwurf fordert, die Möglichkeit zur Erzeugung exakter Kopien bereit stellt.

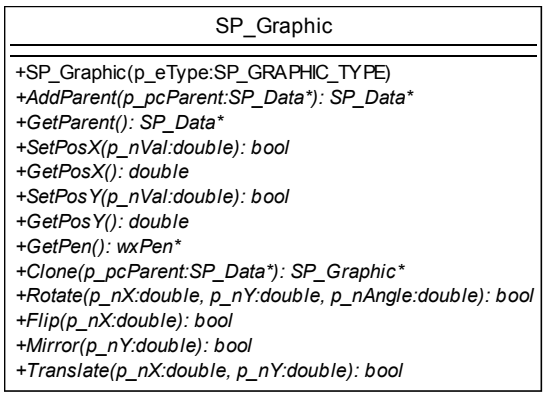
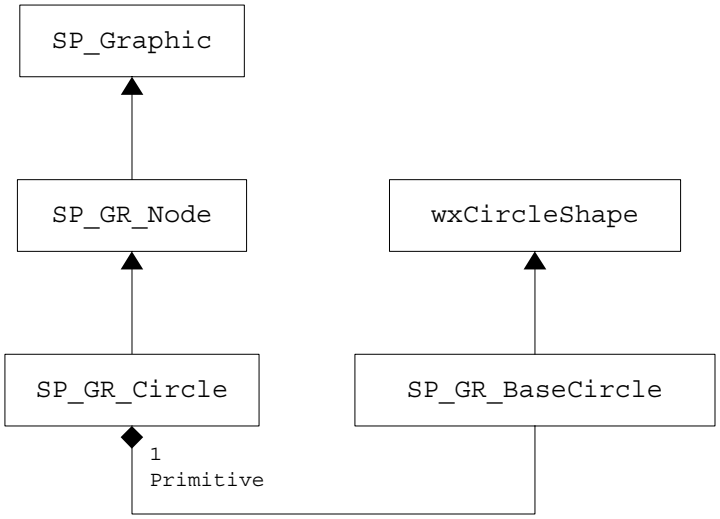


Abbildung 9 - SP_Graphic, ausgewählte Methoden

Der Konstruktor, der mit dem Element eines Aufzählungstypen aufgerufen werden kann, entspricht der Methodik aus der Datenstruktur, anhand derer man ein konkretes graphisches Objekt jederzeit durch seinen Typen identifizieren kann. Wie auch in der Datenstruktur gibt es unter anderem je einen Wert für konkrete Instanzen von Knoten, Kanten oder Attributen.

Die anderen, in der Abbildung aufgeführten Methoden sollen den Charakter dieser Klasse verdeutlichen, dass sie ein Interface zur Manipulation von graphischen Objekten darstellt.

Zu beachten ist, dass weder von `SP_Graphic`, noch von einer der drei Ableitungen konkrete Objekte erzeugt werden können, da auch `SP_GR_Node`, `SP_GR_Edge` und `SP_GR_Attribute` noch einmal abstrakte Methoden enthalten und als eine weitere Ebene der Spezialisierung auftauchen.



Zeichnung 13 - SP_Graphic und die OGL

Die Bindung an die Objekte aus der *OGL*, die letztendlich auf der Oberfläche dargestellt werden, ist nicht mehr direkt durch Ableitung von den entsprechenden Klassen aus der *OGL* und von `SP_Graphic`, wie es in der ersten Implementierung der Fall war, realisiert. Stattdessen besitzen die „konkreten" graphischen Objekte aus Sicht der Software alle ein so genanntes Primitiv, das sie zur Darstellung nutzen. Dieses ist von ihnen aggregiert und müsste, vom Designstandpunkt her, nicht noch einmal durch eine Vererbunsgebene an eine SNOOPY-Klasse gebunden werden. Dass das doch der Fall ist, hat nur den Grund, dass *OGL* bei der Implementierung bestimmter Verhaltensweisen auf die Vererbung besteht, man also eigene Objekte, prinzipbedingt, durch Vererbung von bestehenden *OGL*-Objekten ableitet.

Da `SP_GR_Circle` einen Kreis repräsentieren soll, liegt die Verwendung von `wxCircleShape` als Element aus der *OGL* nahe. Die Klasse SP_GR_Circle neben SP_GR_Node, auch von `wxCircleShape` erben zu lassen erscheint auf den ersten Blick sinnvoll – ich habe mich trotzdem dagegen entschieden. Die einzige Notwendigkeit, eine eigene Klasse von `wxCircleShape` abzuleiten bestand in der schon erwähnten Anforderung der *OGL*, über bestimmte Ereignisse aus der Oberfläche nur durch Ableitung und Überschreiben von abstrakten Methoden informiert zu werden. Das bezieht sich hauptsächlich auf die Benachrichtigung über das Anklicken oder auch die Änderung des Verhaltens, wenn es darum geht, den Zustand der Selektion anzuzeigen.

Eine weitere Entscheidung bezüglich des Designs von `SP_Graphic` geht auf eine Anforderung eines graphischen Objekts zurück, die ich an einem Beispiel zeige:

Die Objekte der Datenstruktur, die für Knoten und Kanten vorgesehen sind, besitzen eine Liste von ihnen zugeordneten Attributen. Jedem dieser Attribute ist es, durch die Vererbung von `SP_Data` möglich, graphische Ausprägungen assoziiert zu haben, wie es auch dem zugehörigen Knoten oder der Kante möglich ist. Wenn jetzt ein Knoten in der Oberfläche angezeigt werden soll, propagiert er diese Aufforderung an alle seine Attribute, die sich daraufhin ebenfalls darstellen. Das führt dazu, dass es neben dem Bild des Knotens unter Umständen noch Anzeigen für die Attribute geben kann.

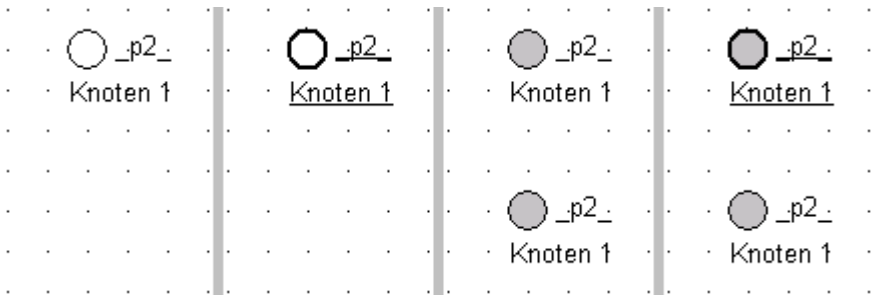


Abbildung 10 - 4 Beispiele für Knotendarstellungen und ihre Attribute

In Abbildung 10 sind 4 Ansichten nebeneinander gestellt, die der Motivation dienen sollen. In der Ersten, der linken, Darstellung sieht man das Symbol eines Knotens (der Kreis) und zwei seiner Attribute, die ebenfalls eine graphische Darstellung besitzen. Die zweite Darstellung zeigt den Knoten, nachdem er ausgewählt wurde, dabei fällt auf, dass auch beide Attribute ihr

Aussehen geändert haben, was sie erstens als „Kinder“ dieses Knotens ausweist und sie zweitens ebenfalls in eventuelle Manipulationen einbezieht, also zum Beispiel Verschiebeoperationen oder das Löschen.

Zur Haltung der Beziehung dieser verschiedenen Darstellungen (der Kreis und 2 Texte als graphische Kinder) ist jedem `SP_Graphic`-Objekt eine Liste von `SP_Graphic`-Objekten assoziiert, die, für den Fall des Knotens, genau die graphischen Ausprägungen der beiden Attribute enthält. So kann einfach auch die Aufforderung, seine Anzeige so zu ändern, dass man als „ausgewählt“ erscheint, einfach an alle Abhängigkeiten delegiert werden. Das Iterieren über die Menge der, dem `SP_Data`-Knoten zugeordneten, Attribute und das Auswählen der graphischen Ausprägungen dieser Attribute ist keine Option, wie die verbleibenden beiden Darstellungen zeigen sollen.

Dabei stehen beide Knoten für ein und das selbe Element in der Datenstruktur (`SP_DS_Node`). Also enthält die Liste der graphischen Ausprägungen des Knotens und beider Attribute jeweils 2 Einträge. Damit ist nicht mehr einwandfrei entscheidbar, welche der Attributgraphiken bei Auswahl welcher Knotengraphik als selektiert markiert werden soll – wenn nicht wenigstens eine von zwei Beziehungen gegeben ist: die vom graphischen Attribut zu seinem graphischen Elternelement oder eben die Liste der graphischen Kinder am Knoten. Tatsächlich gibt es beide Beziehungen in der vorliegenden Implementierung, wobei die Liste der graphischen Kinder eines `SP_Graphic`-Objekts den elegantesten Entwurf und eben darum auch den erfolgversprechendsten Kandidaten für die Einsparung von Speicherplatz darstellte.

Dabei wurde derselbe Ansatz verfolgt, wie bei der Liste der graphischen Ausprägungen für `SP_Data`-Objekte. Es existiert also eine globale Map von SP_Graphic-Objekten als Key zu einer Liste von SP_Graphic-Objekten als Value (`g_mlGraphicChildren`). Wieder reduziert sich die Grösse des Typen und in diesem Fall ist das eine nicht zu unterschätzende Menge, da `SP_GR_Attribute`-Ableitungen keine Kinder assoziiert haben und in der Regel die Masse der `SP_Graphic`-Derivate ausmachen.

SP_Graphic definiert auch die Methoden zur Verknüpfung und Arbeit mit Kanten aus der Oberfläche heraus. Diese sind in der Definition von SP_Graphic mit einer leeren Implementierung hinterlegt und werden im vorliegenden Stand der Arbeiten nur von SP_GR_Node überschrieben und mit Leben gefüllt, prinzipiell lässt diese Implementierung allerdings auch Raum für die Möglichkeit, dass Kanten durch Kanten verbunden werden.

4.3.11 SP_Core

In dieser Klasse manifestiert sich eine Designentscheidung, die so im ursprünglichen Entwurf nicht vorkommt und hauptsächlich den Erfahrungen aus der ersten Implementierung geschuldet ist, nämlich dem Fehlen einer zentralen Instanz zur Sammlung und vor allem der Abfrage von Informationen.

`SP_Core` ist nach dem *Singleton*-Designprinzip [9] als Klasse ohne öffentlichen Konstruktor angelegt. Der einzig mögliche Zugriff erfolgt über eine statische Methode (`SP_Core::GetInterface`) die im Bedarfsfall ein neues Objekt durch ihren geschützten Konstruktor erzeugt oder das schon bestehende zurück liefert. Dadurch ist sicher gestellt, dass zur Laufzeit der Software nur immer genau ein Objekt dieser Klasse existiert.

Der wichtigste Punkt im Hinblick auf die Verfügbarkeit von Informationen war der Wunsch, jederzeit zu einem gegebenen Objekt der *OGL* einfach die Information zu dem dazugehörigen Objekt aus der Menge der von `SP_Graphic` abgeleiteten Objekte zu bekommen. Wie in dem Abschnitt über `SP_Graphic` schon dargelegt, visualisieren sich die graphischen Objekte aus Sicht der Software „nur“ über aggregierte Instanzen von Objekten der *OGL*. Durch diese Trennung der Adressen des SNOOPY-Kreises und des Kreises der *OGL* (um beim Beispiel von `SP_GR_Circle` zu bleiben) liess sich ein einfaches, aber effizientes Mapping einführen. Zu diesem Zweck wird bei jeder Erzeugung eines `SP_GR_Node`- (respektive `SP_GR_Edge`- und `SP_GR_Attribute`-) abgeleiteten Objekts ein Eintrag in eine Map vorgenommen, die die Adresse des Objekts aus SNOOPY mit der Adresse des Objekts aus der *OGL* in Verbindung bringt. Mit diesem Schritt und der Möglichkeit des Zugriffs über eine zentrale Klasse (nämlich `SP_Core`) besteht jederzeit die Möglichkeit, von einem Objekt einfach und schnell in seine jeweilige Entsprechung zu übersetzen.

Ein Beispiel soll helfen, die Intention zu verdeutlichen.

Die *OGL* bietet die Möglichkeit, eine Vielzahl von unterschiedlichen graphischen Objekten in einem speziellen Fenster darzustellen und zu manipulieren. Eine wichtige Voraussetzung dafür ist die Möglichkeit, aus einer Anzahl von so genannten „*shapes*“ in dem Fenster mit der Maus zu wählen, und so die nachfolgenden Aktionen auf die Auswahl einzuschränken (zum Beispiel: Links-Klick/löschen). Das Bestimmen der Position des Mauszeigers zum Zeitpunkt der Aktion und damit das Bestimmen betroffener Shapes erledigt die *OGL*. In aller Regel erhält der Softwareentwickler durch den Aufruf einer definierten Funktion in seiner, vom *OGL*-Shape (Basis: `wxShape`) abgeleiteten Klasse, Kenntnis von einem Ereignis, wie dem Links-Klick aus dem obigen Beispiel.

Um nicht für jede Ableitung die Methoden zur Ereignisreaktion implementieren zu müssen, bietet die OGL eine Möglichkeit der Auslagerung der Eventbehandlung in so genannte Eventhandler. Tatsächlich ist jedes Shape der *OGL* selbst eine Ableitung der Klasse `wxShapeEvtHandler` und bietet mit `SetEventhandler` die Möglichkeit, sich selbst in eine zentralisierte Behandlung einzugliedern.

In der Klasse `SP_GRM_EventHandler` findet sich die Basisimplementierung der Ereignisbehandlung für graphische Objekte in SNOOPY. Der Links-Klick mit der Maus, um beim Beispiel zu bleiben, wird durch Aufruf der Funktion `OnLeftClick` signalisiert. Der Aufruf von `OnLeftClick` garantiert, dass das Ereignis einem graphischen Objekt widerfahren ist und nicht etwa auf eine leere Fläche des Fensters erfolgte. Welches Objekt das ist, kann man jederzeit durch den Aufruf von `GetShape` erfragen. Unglücklicher- aber naheliegenderweise ist der Rückgabewert dieser Methode vom Typ *pointer-to-*`wxShape`, also der Basis aller *OGL*-Objekt.

Durch die Tatsache, dass wir zum Einen unsere Registrierung jederzeit erreichen können (`SP_Core::GetInstance`) und zum Anderen bei der Erzeugung unserer, von `SP_Graphic` abgeleiteten, Objekte die Beziehungen zwischen deren Adresse und den Adressen der zugehörigen, von `wxShape` abgeleiteten, Objekte „registriert“ haben, ist es uns möglich, in den Überladung von *OGL*-Methoden in unserer eigenen Sprache zu sprechen. Durch Aufruf von `SP_Core::ResolveExtern` mit dem Parameter des `wxShape`-Objekts erhält man die Adresse eines `SP_Graphic`-Objekts, oder `NULL`, wenn es keinen Eintrag in der Map gibt, was im Allgemeinen als untrügliches Indiz für einen Fehler in der Datenkonsistenz gilt, da es keine, von `wxShape` abgeleiteten, Objekte ohne Entsprechung in der Hierarchie unterhalb von `SP_Graphic` geben soll.

Die Widget Registry

Wie schon im Kapitel „SP_WDG_DialogBase“ auf Seite 37 erwähnt, existiert eine Registry für die Zuordnung der Attribute eines Graphen zu den zu verwendenden Widgets zur Darstellung im Dialog. Sinnigerweise wird dafür natürlich auch SP_Core eingesetzt und zwar nach folgendem Prinzip.

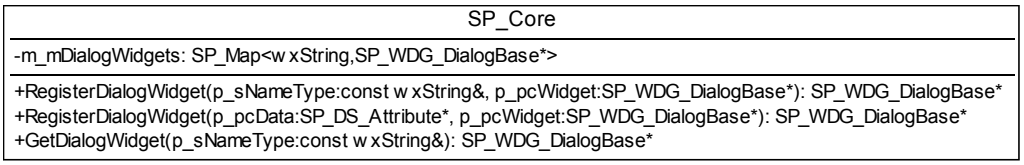


Abbildung 11 - SP_Core und die Widget-Registrierung

Das `m_mDialogWidgets`-Attribut von `SP_Core` ist eine Map von Strings auf Pointer von `SP_WDG_DialogBase`-Derivaten. Die Zugehörigkeit ist in den Strings kodiert, die sich aus folgenden Informationen zusammensetzen:

```
<ClassName> '|' <AttributeName> '|' <AttributeType>
```

mit

`<ClassName>` := Name der Klasse, zu deren Elementen dieses Attribut definiert wurde, also der Name der `SP_DS_Elementclass`-Klasse

`<AttributeName>` := Name des Attributs, wie vom Designer der Netzklasse vergeben

`<AttributeType>` := numerische Entsprechung des Eintrags aus dem Aufzählungstypen, der den Typen des Attributs angibt (also zum Beispiel `SP_ATTRIBUTE_TEXT`)

Der Separator `'|'` dient nur der optischen Abgrenzung, falls man sich einmal die gespeicherten Werte anzeigen lassen sollte, von daher ist es kein schweres Problem, wenn `<ClassName>` oder `<AttributeName>` ebenfalls dieses Zeichen enthielten, da es dennoch in den meisten Fällen zu einem eineindeutigen und reproduzierbaren String führen sollte. Es ist mir bewusst, dass es natürlich eine beliebige Anzahl von Konstruktionen beider Namen gibt, die für verschiedene Attribute in verschiedenen Elementen denselben Key in der Map erzeugen, habe diese Behandlung allerdings in der vor-

liegenden Version vertagt. Da es sich um eine lokale Generierung der entsprechenden Keys handelt, ist für diesen Fall eine Anpassung der Generierung leicht vorzunehmen.

Initialise Netclasses

Eine, auch für den Designer neuer Netzklassen, wichtige Methode von `SP_Core` ist `InitialiseNetclasses`. An dieser Stelle werden beim Programmstart die implementierten Netzklassen in die interne „Datenbank" übernommen, woraufhin sie beim Erstellen neuer, sowie beim Laden bestehender, Graphen als Vorlage und bei der Wahl der Dateierweiterung zur Verfügung stehen.

Daher ist es wichtig, neben dem Ableiten von einer bestehenden Netzklasse oder auch direkt von SP_DS_Netclass, die so geschaffene neue Vorlage für Graphen in InitialiseNetclasses bekannt zu machen. Die dafür notwendigen Schritte sind immer dieselben:

```
1    bool
2    SP_Core::InitialiseNetclasses(SP_GM_Docmanager* p_pcDocmanager)
3    {
4        SP_DS_Netclass l_pcNet;
5        ////////////////////////////////////////////////////////////////
6        l_pcNet = new SP_DS_SimpleGraph("Simple Graph");
7        if (l_pcNet->AddToGui(p_pcDocmanager))
8            AddNetclass(l_pcNet);
9        else
10           delete l_pcNet;
11
12       ////////////////////////////////////////////////////////////////
13       l_pcNet = new SP_DS_BipartGraph("Bipart Graph");
14       if (l_pcNet->AddToGui(p_pcDocmanager))
15           AddNetclass(l_pcNet);
16       else
17           delete l_pcNet;
18
19       ////////////////////////////////////////////////////////////////
20       l_pcNet = new SP_DS_SimplePed("Simple Ped");
21       if (l_pcNet->AddToGui(p_pcDocmanager))
22           AddNetclass(l_pcNet);
23       else
24           delete l_pcNet;
25
26       return TRUE;
27   }
```

Die Datenhaltung erfolgt in `SP_GM_Docmanager` einer Klasse, die von einer bestehenden *wxWidgets*-Klasse abgeleitet wurde und die damit automatisch Teil des Document/View- und MDI-Prinzips ist. Über AddToGui werden Instanzen der, nach den Vorgaben des Document/View-Modells vorhandenen Klassen innerhalb von SNOOPY erzeugt und dem Dokumenten-Manager be-

kannt gemacht. Es besteht keine Notwendigkeit, diese Methodik in einer Ableitung neu zu definieren. Alle zum jetzigen Zeitpunkt identifizierten Informationen, die zum generischen Hinzufügen zu dieser Verwaltungsstruktur erforderlich sind, sind schon als Anforderungen an die Definition einer Netzklasse gestellt.

4.3.12 SP_DS_Animation

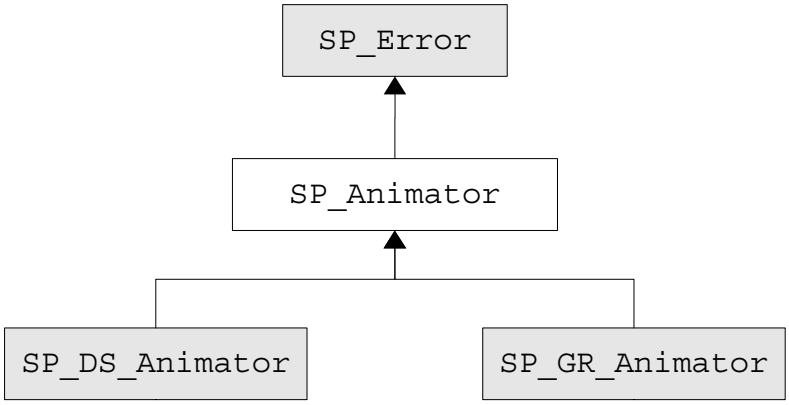
Da für einen konkreten Anwendungsfall gefordert wurde, dass der Graph animierbar sein soll galt es, einen Weg zu finden, eine solche Möglichkeit für alle denkbaren Graphen einzubauen. Diese Möglichkeit war im ursprünglichen Entwurf nicht vorgesehen.

Mein Ansatz zielte darauf ab, alle Aufgaben so gut wie möglich zu verteilen, so dass kein beteiligtes Objekt mehr als die maximal notwendige Arbeit verrichten oder Verantwortung übernehmen muss, um es umgangssprachlich zu formulieren. Wie auch schon an allen anderen Stellen sollte es eine Vorlage für die Vererbungshierarchie geben, von der je nach Bedarf und unter Beachtung der Anforderungen an die Schnittstellen eigene Ableitungen zur Erfüllung spezieller Aufgaben angelegt werden konnten, da nicht jeder Wunsch jedes Netzdesigners vorauszuahnen ist. Dieser Ansatz sollte derselbe wie bei der Erweiterung der Struktur oder der Graphik um neue Attribute und graphische Ausprägungen sein.

Darüber hinaus sollte die Erweiterung der bestehenden Klassen auf ein Minimum reduziert werden, gerade weil „Animation“ nicht unbedingt zu den essentiellen Bestandteilen jedes Graphen gehört und somit jedes neue Attribut an einer zentralen Klasse den Speicherbedarf erhöhen würde, ohne unbedingt von Nutzen zu sein. Also entschied ich mich für ein Verfahren, dass ähnlich der Registrierung der Dialogelemente funktionieren sollte.

Kern dieses Ansatzes war die Überlegung, dass nicht unbedingt ein Graph eine Animation steuern *muss*, sondern im Gegenteil, eine Animation einen Graphen steuern *kann*. Und diese Beeinflussung sollte sich auch fast ausschliesslich auf die Datenstrukturelemente eines Graphen beschränken lassen. Diese Feststellung hat mit der Tatsache zu tun, dass in der Datenstruktur alle relevanten Informationen zu allen Elementen der Knoten- und Kantenpartitionen, deren Verknüpfungen und vor allem den dazugehörigen Attributen vorhanden sind, während die konkreten graphischen Ausprägungen dieser Strukturinformationen erst einmal zweitrangig sind und sich selbst in der Regel nur an der Datenstruktur orientieren.

Das Ergebnis dieser Überlegung war die Klasse SP_Animator, deren wichtigste Ableitung in SP_DS_Animator bestand.



Zeichnung 14 - SP_Animator und Ableitungen

SP_Animator selbst erfüllt dabei, ähnlich wie SP_Type in der Datenstruktur, nur die Aufgabe, die abgeleiteten Klassen noch, wenn sie als Objekt vom Typ ihrer Basis vorliegen, unterscheiden zu können. Zu diesem Zweck verwaltet SP_Animator den Wert eines Aufzählungstypen, der die beiden Ableitungen als der Datenstruktur oder der Graphik zugehörig ausweist. Nach der Vorrede jetzt auch einen SP_GR_Animator zu sehen, mag verwirren, ich verschiebe die Vorstellung dieser Klasse trotzdem noch ein wenig nach hinten und möchte vorab etwas genauer auf SP_DS_Animator eingehen.

SP_DS_Animator stellt den Initiator aller Veränderungen in der Datenstruktur dar, die durch Animation ausgelöst werden können. Zu diesem Zweck muss er Kenntnis von einem Element der Datenstruktur erhalten, für dass er verantwortlich sein soll. Das bedeutet, ein Objekt vom Typ SP_Data ist sein Bezugselement und alle Elemente eines Graphen können aus Sicht der Animation mit einem Animator ausgestattet werden. Ausserdem muss es eine Möglichkeit geben zu entscheiden, ob ein Animator auch wirklich etwas tun soll, und diese Entscheidung muss regelmässig überprüft und unter Umständen revidiert werden. Aus diesem Grund muss jeder Animator 3 Funktionen implementieren, die eine Animation ausmachen:

`PreStep`

Zeitpunkt der Entscheidung, welcher Animator in der nächsten Zeit aktiv werden kann. In diesem Schritt kann jeder Animator sowohl lokal als auch durch Informationen anderer Animatoren entscheiden, ob er im nächsten Schritt „aktiv“ sein wird.

`Step`

Der eigentliche Schritt in einer Animationsphase. Hier soll jeder im vorherigen Schritt „aktivierte“ Animator seine „Bewegung“ ausführen.

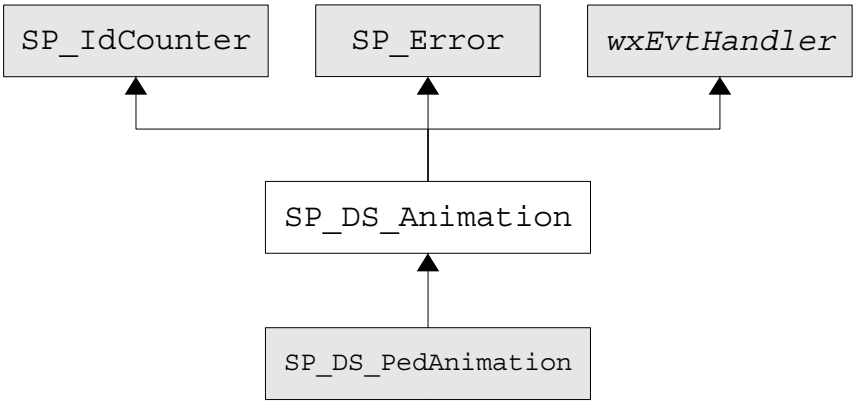
`PostStep`

Platz zum Setzen von Werten in den assoziierten Strukturen des Graphen und zum Rücksetzen von lokalen Zuständen und Variablen jedes Animators.

Die Assoziation eines Animators zu den Elementen eines Graphen muss zur Definitionszeit erfolgen können. Das heisst, in der CreateGraph-Methode der Netzklasse muss es die Möglichkeit geben, den gewünschten Elementen einen Animator zuzuweisen, der dann zum geeigneten Zeitpunkt Einfluss nehmen kann. Diese Assoziation erfolgt wiederum über eine Registry in der zentralen Verwaltung der Instanz von SP_Core. Dabei erfolgt die Verknüpfung symbolisch, das heisst für eine eineindeutige Identifikation eines Elements wird ein Animator registriert. In der vorliegenden Version erfolgt diese Registrierung über die Bildung eines Schlüssels aus dem Namen des Graphen und dem Namen der Partition, für deren Elemente der Animator seinen Dienst tun soll. Dieser Schlüssel ist Key in einer Map, deren Value-Feld aus einer konkreten Instanz eines Animators besteht. Um zur Laufzeit dann für alle konkreten Elemente der benannten Partition des aktuellen Graphen eine identische Kopie dieses einen Animators erzeugen zu können, folgen die Animatoren der Vorgabe, eine Clone-Methode zu implementieren, wie wir sie schon aus den Elementen des Graphen kennen.

SP_DS_Animation ist nun die übergeordnete Struktur, die zur Steuerung und zum Aufrufen der drei definierten Funktionen all ihrer Animatoren dient. Eine Instanz von ihr oder einer ihrer Ableitungen wird dem Graphen selbst zugeordnet und die Aufnahme der Arbeit kann über die Oberfläche gesteuert werden. Dabei ist die Initialisierung immer dieselbe, indem als erstes durch Iteration über alle Elemente des Graphen in seinem aktuellen Zustand, für jedes Element, für dessen Partition ein Animator registriert

wurde, eine Kopie der Animatorvorlage erzeugt und in einer Liste der Animation gespeichert wird. Um eine Änderung des Graphen nach dieser Initialisierung zu verhindern, sind fast alle Möglichkeiten der Einflussnahme ab dem Zeitpunkt der Aktivierung einer Animation in der Oberfläche gesperrt.



Zeichnung 15 - SP_DS_Animation

Die Ableitung von der Eventhandler-Klasse von *wxWidgets* dient einzig dem Zweck, das Verhalten der Klasse `SP_DS_Animation` durch einen Timer steuern zu können. `SP_DS_Animation` entspricht im weitesten Sinne der Qualität von `SP_DS_Graph`. Das heisst, sie aggregiert alle verfügbaren Animatoren und steuert deren Verhalten, wie `SP_DS_Graph` alle Elemente eines Graphen kontrolliert und deren Verhalten steuern kann.

Da ein Animator, wie beschrieben, in einer Iteration der drei Funktionen arbeitet, ist es Aufgabe der Animation, diese drei Funktionen für seine Animatoren anzusprechen. Dabei hat sich gezeigt, dass die im `PreStep` getroffene Entscheidung eines Animators, ob er überhaupt den `Step` ausführen will, als Kriterium zur Reduzierung der Menge der Animatoren dienen kann. Also hält die Animation nicht nur die Liste aller möglichen Animatoren, sondern fügt eine Auswahl davon, nämlich die, die im `PreStep` aktiviert wurden in eine zweite Liste ein, die dann für die `Step`- und die `PostStep`-Phasen die einzig relevante ist.

Wird die Animation gestartet, so startet sie ihrerseits einen Timer, dessen Frequenz und maximale Anzahl von Schritten steuerbar ist und führt in jedem Tick eine der drei Funktionen ihrer Animatoren aus. Dabei ist der immer erste Tick der Zeitpunkt, in dem für alle Animatoren der `PreStep` ausgeführt wird. Anschliessend folgt eine lange Anzahl von Ticks zum Auf-

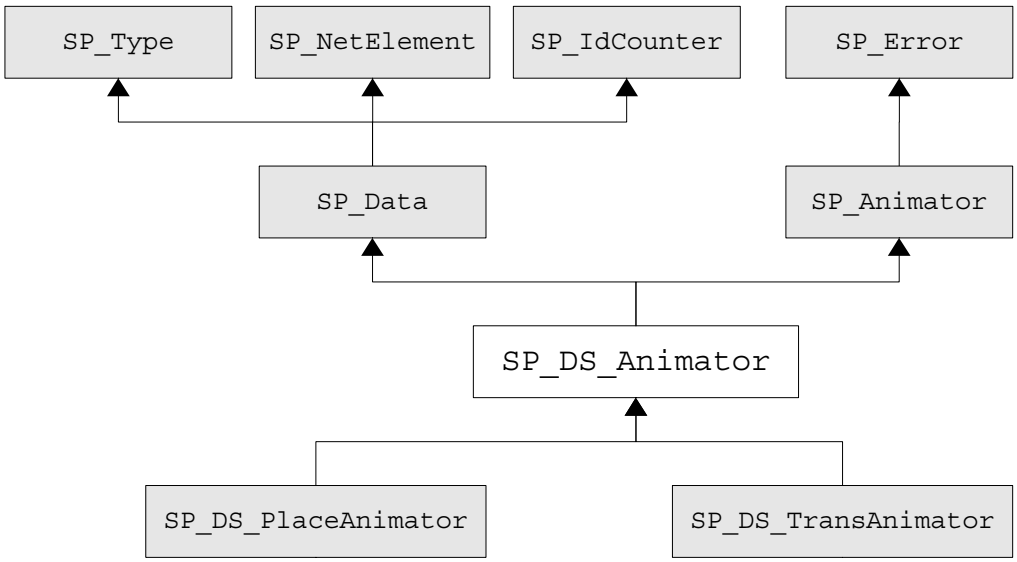
ruf der `Step`-Funktionen aller aktivierten Animatoren. Das hat den Grund, dass eine „Animation“ im Wortsinn eine Bewegung ausführen sollte und dies geschieht, gesteuert durch jeden Animator selbst, in dessen `Step`-Methode. Zum Zeitpunkt des letzten Ticks (bevor der Timer wider von vorn beginnt) werden für alle aktivierten Animatoren die `PostStep`-Methoden aufgerufen und die temporären Listen der Animatoren für den eben ausgeführten Schritt geleert.

Die Ticks des Timers hängen dabei von zwei Werten ab, die der Nutzer festlegen kann. Einmal die Frequenz mit der der Timer arbeiten soll, üblicherweise ein Wert im Bereich von hundert Millisekunden. Dieser dient vor allem dazu, die eigentliche Bewegung flüssig aussehen zu lassen. Zum Anderen kann man die Gesamtdauer eines „Schritts“ der Animation festlegen. Das ist nicht mit der Step-Methode zu verwechseln, sondern legt fest, wie lange eine Bewegung dauern soll und liegt üblicherweise im Sekundenbereich. Aus dem Quotienten von Dauer und Frequenz ergibt sich ein Wert, der beschreibt, wie oft der Timer mit dieser Frequenz für die angestrebte Dauer „ticken“ kann. Eine Frequenz von 100 Millisekunden für eine Dauer von 3 Sekunden führt zum Beispiel zu 30 Ticks,wobei beim 1. Tick die Animation den `PreStep` ihrer Animatoren auswertet, anschliessend 30 mal die ausgewählten Animatoren ihre `Step`-Methode ausführen dürfen und im 30. Tick zusätzlich die `PostStep`-Methoden aufgerufen und die temporären Listen geleert werden.

Diese Abläufe wiederholen sich immer wieder, bis die Animation (und damit der Timer) gestoppt wird. Die Animatoren müssen nun dafür sorgen, dass sie sich richtig für ihre Aktivierung entscheiden, eventuell Werte in der Datenstruktur zu Beginn ihrer Aktivität setzen, in ihrer `Step`-Methode irgendeine sinnvolle Bewegung ausführen und im `PreStep` unter Umständen noch einmal die Werte in der Datenstruktur setzen, je nach dem welche Semantik mit der Animation verbunden sein soll.

Sämtliche Änderungen von, zum Beispiel, Attributwerten der Datenstruktur lassen sich einfach visualisieren, ähnlich wie bei direkter Bearbeitung eines Attributs mit Hilfe eines Dialogs. Ein Problem stellte die Handhabung der eigentlichen Bewegung dar, die ein Animator ausführen soll. Die Lösung lag in den schon so lange verwendeten und hier auch schon vorgestellten, eigenen Strukturen. Animation bedeutet bei uns die Bewegung graphischer Objekte über den Bildschirm. Dabei ist jedes der graphischen Objekte genau einem Objekt einer Verwaltungsstruktur zugeordnet und wird von ihm gesteuert. Es lag also nahe, die Animatoren wie Datenstrukturelemente eines Graphen und deren graphische Ausprägungen zu organisieren. Also wurde die konkrete Klasse `SP_DS_Animator` eine Ableitung von `SP_Data` und

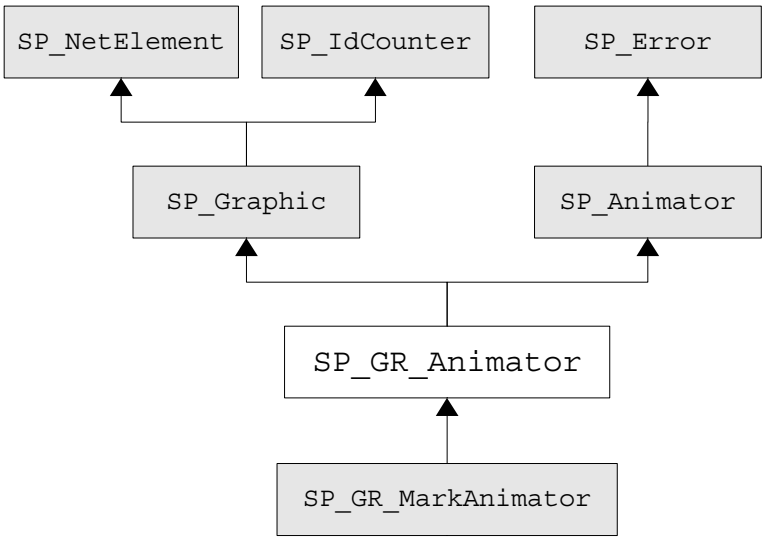
verfügte somit automatisch über alle Fähigkeiten zur Verwaltung graphischer Ausprägungen und darüber hinaus auch noch über den Zwang, eine Methode zum Generieren exakter Kopien bereit zu stellen.



Zeichnung 16 - SP_DS_Animator als Ableitung von SP_Data

Daraus folgte der logische Schluss, dass es auch Objekte in der Menge der graphischen Ausprägungen geben sollte, die der bestehenden Hierarchie folgen und dennoch als zur Animation gehörig gekennzeichnet sind.

Die Lösung lag in der Erweiterung der Vererbungshierarchie auf der Ebene von `SP_GR_Node`, `SP_GR_Edge` und `SP_GR_Attribute` durch Hinzufügen einer neuen, definierten graphischen Ausprägung als Vorlage und Schnittstellendefinition für eigene Implementierungen.



Zeichnung 17 - SP_GR_Animator

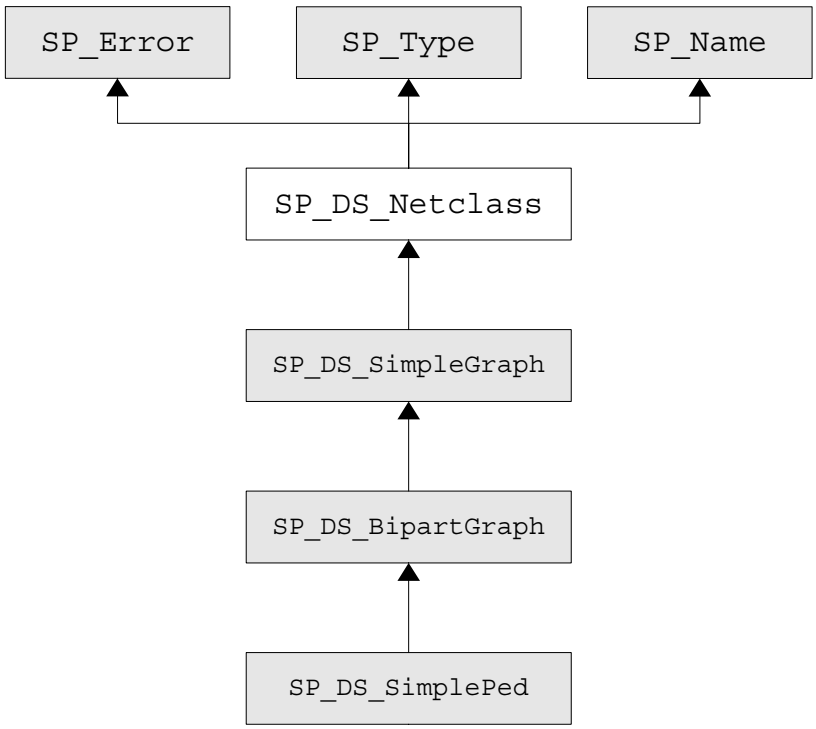
Sowohl die Animatoren der Datenstruktur, als auch die der Graphik folgen den beschriebenen Gesetzmässigkeiten. Sie sind durch einen eigenen Typen identifizierbar (`SP_ELEMENT_ANIMATOR` für die Datenstruktur und `SP_GRAPHIC_ANIMATOR` für die Graphik) und implementieren alle erforderlichen Schnittstellen. Darüber hinaus erweitern sie die Forderung der zu implementierenden Schnittstellen um die drei essentiellen Funktionen, in deren Rahmen sich eine Animation bewegt (`PreStep`, `Step` und `PostStep`).

Alle an der Animation beteiligten Klassen sind per Definition Vorlagen für eigene Implementierungen. Wie in den Klassendiagrammen ersichtlich, liegt eine konkrete Anwendung bereits vor. Auf diese `SP_DS_PedAnimation` und ihre beteiligten Klassen gehe ich im Abschnitt Beispielimplementierungen ab Seite 76 näher ein.

5 Beispielimplementierungen

Zur Verdeutlichung des Einsatzes der in den vorangegangenen Abschnitten vorgestellten Klassen stelle ich hier drei der beispielhaft implementierten Netzklassen aus der Software mit ihren Eigenheiten der Implementierung vor.

Zum besseren Verständnis hier noch einmal das Klassendiagramm für die Basis aller Netzklassen `SP_DS_Netclass`, wie es auch schon im entsprechenden Abschnitt auf Seite 43 verwendet wurde:



Zeichnung 18 - Vererbungshierarchie von SP_DS_Netclass

5.1 SP_DS_SimpleGraph

Diese erste Ableitung von `SP_DS_Netclass` stellt die einfachste Art eines Graphen dar, der nur über genau eine Knotenpartition und genau eine Kantenpartition verfügt. Dabei besitzt kein Element der beiden Mengen ein Attribut sondern nur eine einfache graphische Ausprägung.

Im Folgenden sehen wir die beiden Konstruktoren der Klasse, die sich in einem Parameter unterscheiden, der als Name für Graphen dieser Klasse verwendet werden. In der vorliegenden Version wird ausschliesslich der Standardkonstruktor verwendet, das Vergeben einer Bezeichnung obliegt also dem Designer der Netzklasse. Das Attribut `m_sExtension` ist ein Member der Basisklasse `SP_DS_Netclass` und speichert eine Dateinamenerweiterung, wie sie zum Laden und Speichern verwendet wird. Diese wird zur Unterscheidung der verschiedenen Graphen in den entsprechenden Dialogen verwendet.

```
1    SP_DS_SimpleGraph::SP_DS_SimpleGraph()
2    :SP_DS_Netclass("Simple Graph")
3    {
4        m_sExtension = "spgraph";
5    }
6
7    SP_DS_SimpleGraph::SP_DS_SimpleGraph(const char* p_pchName)
8    :SP_DS_Netclass(p_pchName)
9    {
10       m_sExtension = "spgraph";
11   }
```

Der nächste Ausschnitt zeigt die zentrale `CreateGraph`-Methode, in der zwei neue Objekte, nämlich Instanzen von `SP_DS_Nodeclass` und `SP_DS_Edgeclass` mit je einem Namen und dem aktuellen Graphen als Elternelement, angelegt werden. Ausserdem bekommen beide Objekte über `SetGraphic` eine graphische Ausprägung zugewiesen. Diese Methode delegiert den Aufruf zum Hinzufügen der graphischen Ausprägung an den entsprechenden Prototypen, löscht aber vorher alle eventuell bereits vermerkten anderen Referenzen auf Graphiken. Dieses Vorgehen stellt sicher, dass alle Objekte zur Definitionszeit, wo es sich nur um Prototypen handelt, nur maximal eine graphische Ausprägung referenzieren.

```
1    SP_DS_Graph*
2    SP_DS_SimpleGraph::CreateGraph(SP_DS_Graph* p_pcGraph)
3    {
4        if (!SP_DS_Netclass::CreateGraph(p_pcGraph))
5            return NULL;
6
7        SP_DS_Nodeclass* l_pcNC;
8        SP_DS_Edgeclass* l_pcEC;
9
10       ////////////////////////////////////////////////////////////////////
11       l_pcNC = new SP_DS_Nodeclass(p_pcGraph, "Node");
12       l_pcNC->SetGraphic(new SP_GR_Circle(l_pcNC->GetPrototype(), 20.0));
13       p_pcGraph->AddNodeclass(l_pcNC);
14
15       ////////////////////////////////////////////////////////////////////
16       l_pcEC = new SP_DS_Edgeclass(p_pcGraph, "Edge");
17       l_pcEC->SetGraphic(new SP_GR_ArrowEdge(l_pcEC->GetPrototype()));
18       p_pcGraph->AddEdgeclass(l_pcEC);
19
20       return p_pcGraph;
21   }
```

Die Konstruktoren der graphischen Objekte (Zeilen 12 und 17) sind von Klasse zu Klasse verschieden, erhalten jedoch immer eine Referenz auf ihr Elternelement. Der `SP_GR_Circle`, der einen einfachen Kreis darstellen soll, wird darüber hinaus noch mit einem Radius initialisiert, wohingegen die `SP_GR_ArrowEdge` keine weiteren graphischen Attribute zu initialisieren braucht. Da an diesen Stellen aber immer mit den konkreten Klassen für die graphischen Objekte gearbeitet wird sind die Notwendigkeiten vom Programmierer der jeweiligen Klasse frei festlegbar.

In `EdgeRequirement` und `NodeRequirement` wird in dieser Netzklasse einfach die Bearbeitung an die Basisklasse `SP_DS_Netclass` delegiert. Das bedeutet genau genommen, dass beide Methoden nicht überschrieben und selbst implementiert werden. Zur Vervollständigung der Betrachtung, reiche ich an dieser Stelle die Implementierung dieser beiden Methoden von `SP_DS_Netclass` nach, die beide, bis auf die Überprüfung der Parameter nichts tun und so nur für den Fall, dass mindestens eine der Adressen den Wert `NULL` trägt, widersprechen.

```
1    bool
2    SP_DS_Netclass::NodeRequirement(SP_DS_Node* p_pcNode)
3    {
4        CHECK_POINTER(p_pcNode, return FALSE);
5
6        return TRUE;
7    }
8
9    bool
10   SP_DS_Netclass::EdgeRequirement(SP_DS_Edgeclass* p_pcClass,
11                                   SP_Data* p_pcSrc,
12                                   SP_Data* p_pcTrg)
13   {
14       CHECK_POINTER(p_pcClass, return FALSE);
15       CHECK_POINTER(p_pcSrc, return FALSE);
16       CHECK_POINTER(p_pcTrg, return FALSE);
17
18       return TRUE;
19   }
```

Die verwendeten CHECK_POINTER-Makros wurden im Abschnitt „Basisklassen, Templates und Makros“ auf Seite 25 vorgestellt und überprüfen die Identität des ersten Parameters mit NULL. Für den Fall, dass es keine Beanstandungen aus dieser Richtung gibt, sind in dieser Implementierung jeder Knoten und jede Kante zugelassen.

5.2 SP_DS_BipartGraph

Wie in der Vererbung, anhand des Eingangs dieses Kapitels gezeigten Klassendiagramms, ersichtlich ist, ist `SP_DS_BipartGraph` eine Ableitung unseres eben vorgestellten einfachen Graphen mit einer Knoten- und einer Kantenpartition. Im Unterschied zu dieser Vorlage, wie der Name schon andeutet, soll ein Graph durch diese Netzklasse um eine Knotenpartition erweitert werden.

Die Konstruktoren unterscheiden sich kaum von denen der Basisklasse, abgesehen davon, dass der Standardkonstruktor mit einem anderen Namen initialisiert wird und sich die zu verwendende Dateinamenserweiterung für Graphen dieser Klasse unterscheidet.

```
1    SP_DS_BipartGraph::SP_DS_BipartGraph()
2    :SP_DS_SimpleGraph("Bipart Graph")
3    {
4        m_sExtension = "spbipart";
5    }
6
7    SP_DS_BipartGraph::SP_DS_BipartGraph(const char* p_pchName)
8    :SP_DS_SimpleGraph(p_pchName)
9    {
10       m_sExtension = "spbipart";
11   }
```

Die `CreateGraph`-Methode ist etwas umfangreicher, werden hier doch beispielhaft allen Prototypen, egal ob neu oder modifiziert, Attribute eingeschrieben.

```
1    SP_DS_Graph*
2    SP_DS_BipartGraph::CreateGraph(SP_DS_Graph* p_pcGraph)
3    {
4        if (!SP_DS_SimpleGraph::CreateGraph(p_pcGraph))
5            return NULL;
6
7        SP_DS_Attribute* l_pcAttr;
8        SP_Graphic* l_pcGr;
9        SP_DS_Nodeclass* l_pcNC;
10       SP_DS_Edgeclass* l_pcEC;
11       SP_GR_Rectangle* l_pcRect;
12
13       ////////////////////////////////////////////////////////////////////
14       l_pcNC = p_pcGraph->RenameNodeclass("Node", "Circle");
15       l_pcAttr = l_pcNC->AddAttribute(new SP_DS_TextAttribute("Name", ""));
16       l_pcGr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
17       l_pcGr->SetOffsetY(20);
18       l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogText("General"));
19
20       ////////////////////////////////////////////////////////////////////
21       l_pcNC = new SP_DS_Nodeclass(p_pcGraph, "Rectangle");
22       l_pcAttr = l_pcNC->AddAttribute(new SP_DS_TextAttribute("Name", ""));
23       l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogText("General"));
24       l_pcGr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
25       l_pcGr->SetOffsetY(20);
26       l_pcRect = new SP_GR_Rectangle(l_pcNC->GetPrototype(), 20.0, 20.0);
27       l_pcNC->SetGraphic(l_pcRect);
28       p_pcGraph->AddNodeclass(l_pcNC);
29
30       ////////////////////////////////////////////////////////////////////
31       l_pcEC = p_pcGraph->GetEdgeclass("Edge");
32       l_pcAttr = l_pcEC->AddAttribute(new SP_DS_TextAttribute("Name"));
33       l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogText("General"));
34       l_pcGr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
35
36       return p_pcGraph;
37   }
```

Wie es die Vererbung nahe legt überlässt die Methode als erstes ihrer Basisklasse die Bearbeitung des Parameters, woraufhin der Graph dem von `SP_DS_SimpleGraph` Modell entspricht. Darauf aufbauend wird dieses Modell verändert, indem zum Beispiel in Zeile 14 die bestehende Knotenpartition namens „Node“ in „Circle“ umbenannt wird. Da diese Beispiele der Anschauung dienen, habe ich der besseren Lesbarkeit wegen auf die notwendigen Tests auf gültige Rückgabewerte verzichtet. Stattdessen wird diese bestehende Knotenpartition in Zeile 15 sofort um ein Textattribut mit der Bezeichnung „Name“ und dem Standardwert „“ (dem leeren String) erweitert. Dieses neue Attribut erhält in der folgenden Zeile eine graphische Ausprägung zugewiesen, wodurch sein Inhalt im weiteren Verlauf jederzeit angezeigt werden kann. Die Methode `SetOffsetY` setzt die graphische Ausprägung des Attributs initial um 20 Einheiten in der vertikalen vom Zentrum der Graphik des Elternelements entfernt. Das bezieht sich auf die Graphik unseres „Circle“, so wie sie in `SP_DS_SimpleGraph` als Graphik der Partiti-

on „Node“ definiert wurde. An dieser Definition wird nichts verändert, es bleibt also bei einer 20 Einheiten im Radius messenden Instanz von `SP_GR_Circle`.

Neu ist der Aufruf von `RegisterDialogWidget` mit einem neuen Objekt der Klasse `SP_WDG_DialogText`, wodurch dem Attribut „Name“ der Knotenpartition „Circle“ vom Typ `SP_ATRIBUTE_TEXT` (durch die Instanziierung der Klasse `SP_DS_TextAttribute`) ein Widget zum Eingeben von Text in einem Dialog zugeordnet wird. Diese Registrierung erfolgt, wie im Abschnitt „SP_Core“ auf Seite 49 ausgeführt, in der zentralen Registry für Dialogelemente. Der an den Konstruktor des Widgets übergebene String dient der Definition einer Seite im Notebook des Dialogs.

Damit ist die von `SP_DS_SimpleGraph` übernommene Definition einer Knotenpartition verändert und erweitert worden.

Als nächstes erfolgt die Definition einer zweiten Knotenpartition, die, im Gegensatz zur ersten, völlig neu erstellt wird. Die graphische Ausprägung dieser Partition übernimmt ein Rechteck, welches mit je einem Wert für Höhe und Breite initialisiert wird. Die in den Zeilen 26 und 27 verwendete Art der Zuweisung soll verdeutlichen, dass man alle Objekte typgenau konstruieren und unter Umständen noch weitere, nicht in der Basisklasse ausgeführte Initialisierungen vornehmen kann, während die Methoden zum Setzen einer graphischen Ausprägung oder zum Hinzufügen eines Attributs immer nur Typen der jeweiligen Basisklasse zurück liefern.

Zum Abschluss wird noch den Elementen der bestehenden Kantenpartition Namens „Edge“ ebenfalls ein Text-Attribut einbeschrieben. Damit ist die Definition des neuen Graphen abgeschlossen und was bleibt ist der Blick auf `EdgeRequirement` und `NodeRequirement`. NodeRequirement bleibt wieder einmal unangetastet, für EdgeRequirement wollen wir diesmal aber das Ziehen von Kanten zwischen Knoten der gleichen Partition verbieten.

```
1    bool
2    SP_DS_BipartGraph::EdgeRequirement(SP_DS_Edgeclass* p_pcClass,
3                                       SP_Data* p_pcSrc,
4                                       SP_Data* p_pcTrg)
5    {
6        if (!SP_DS_SimpleGraph::EdgeRequirement(p_pcClass, p_pcSrc, p_pcTrg))
7            return FALSE;
8
9        return (p_pcSrc->GetClassObject() != p_pcTrg->GetClassObject());
10   }
```

Durch das Delegieren der ersten Entscheidung an die entsprechende Methode der Basisklasse bekommen wir den Test der Gültigkeit der Parameter geschenkt. Falls dieser Test erfolgreich ist, testen wir die Instanzen der `SP_DS_Elementclass`-Referenzen der beiden Knotenparameter auf Identität. Nur für den Fall, dass beide Knoten aus verschiedenen Partitionen stammen, wird die Kante gestattet.

5.3 SP_DS_SimplePed

Diese Klasse stellt die bislang umfangreichste Netzklasse dar und baut, zu Demonstrationszwecken, auf die zuvor beschriebenen Netzklassen auf. Ihre Konstruktoren sind immer noch so einfach wie die der vorangegangenen Klassen.

```
1    SP_DS_SimplePed::SP_DS_SimplePed()
2    :SP_DS_BipartGraph("Simple Ped")
3    {
4        m_sExtension = "spped";
5    }
6
7    SP_DS_SimplePed::SP_DS_SimplePed(const char* p_pchName)
8    :SP_DS_BipartGraph(p_pchName)
9    {
10       m_sExtension = "spped";
11   }
```

Einzig die Wahl des Namens im Standardkonstruktor und die Auswahl einer Dateinamenerweiterung machen auch hier den Unterschied aus.

Dafür ist die Implementierung der `CreateGraph`-Methode wesentlich umfangreicher, weshalb ich im folgenden Abschnitt die Implementierung immer in Teilen vorstellen und kommentieren werde.

```
1    SP_DS_Graph*
2    SP_DS_SimplePed::CreateGraph(SP_DS_Graph* p_pcGraph)
3    {
4        if (!SP_DS_BipartGraph::CreateGraph(p_pcGraph))
5            return NULL;
```

Hier wird wieder die `CreateGraph`-Methode der zugrunde liegenden Netzklasse als erster Schritt in der Erstellung herangezogen. Schlägt diese fehl, wird kein gültiges Ergebnis zurückgegeben.

```
6         SP_DS_Attribute* l_pcAttr;
7         SP_DS_Coarse* l_pcCoarse;
8         SP_GR_Node* l_pcGr;
9         SP_Graphic* l_pcGrAttr;
10        SP_DS_Nodeclass* l_pcNS;
11
12        ////////////////////////////////////////////////////////////////////
13        l_pcNS = p_pcGraph->RenameNodeclass("Circle", "Place");
14        l_pcAttr = l_pcNS->GetPrototype()->GetAttribute("Name");
15        l_pcAttr->SetGlobalShow();
16
17        l_pcAttr = l_pcNS->AddAttribute(new SP_DS_IdAttribute("ID"));
18        l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogShowOnly("General"));
19        l_pcGrAttr = new SP_GR_NumberAttribute(l_pcAttr, "_p%_");
20        l_pcGrAttr->SetOffsetX(20);
21        l_pcGrAttr->SetShow(FALSE);
22        l_pcAttr->AddGraphic(l_pcGrAttr);
23        l_pcAttr->SetGlobalShow();
24
25        l_pcAttr = new SP_DS_NumberAttribute("Marking", 0);
26        l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogNumber("General"));
27        l_pcAttr->AddGraphic(new SP_GR_MarkAttribute(l_pcAttr, 0));
28        l_pcNS->AddAttribute(l_pcAttr);
29
30        l_pcAttr = new SP_DS_LogicAttribute("Logic", "Name");
31        l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogBool("General"));
32        l_pcNS->AddAttribute(l_pcAttr);
33
34        l_pcAttr = new SP_DS_TextAttribute("Comment", "");
35        l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogMultiline("Other"));
36        l_pcGrAttr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
37        l_pcGrAttr->SetOffsetY(40);
38        l_pcNS->AddAttribute(l_pcAttr);
39
40        l_pcGr = new SP_GR_Circle(l_pcNS->GetPrototype(), 20.0);
41        l_pcNS->SetGraphic(l_pcGr);
42        l_pcGr->SetFixedSize(TRUE);
43        l_pcNS->RegisterGraphicWidget(new SP_WDG_DialogGraphic("Graphic"));
44
45        // animator
46        l_pcNS->AddAnimator(new SP_DS_PlaceAnimator("Marking"));
```

In diesem Abschnitt wird die Veränderung der bereits vorhandenen Knotenpartition namens „Circle“ aus `SP_DS_BipartGraph` vorgenommen. Zum Einen wird die Bezeichnung noch einmal verändert und zwar heisst diese ab jetzt „Place“ und steht für die Partition der Plätze in einem Platz/Transitions-Netz. In den Zeilen 14 und 15 wird dem bestehenden Attribut „Name“ durch `SetGlobalShow` mitgeteilt, dass es über einen globalen Schalter zur Anzeige gebracht bzw. dass seine Anzeige abgeschaltet werden kann.

Anschliessend werden der Partition weiter Attribute eingeschrieben. Das Attribut namens „ID“ erhält durch die Instanziierung von `SP_DS_IdAttribute` die Fähigkeit, seinen Wert mit allen anderen Attributen derselben Art und derselben Partition zu synchronisieren. Der Wert ist ein numerischer Typ, der durch ein `SP_GR_NumberAttribute` visualisiert werden soll. Der zweite Parameter im Konstruktor dieser graphischen Ausprägung definiert eine Formatvorlage, wobei das „%“ Zeichen als Platzhalter für den konkre-

ten Wert des Attributs steht. Das Widget, über das dieses Attribut im Dialog repräsentiert wird, bietet nicht die Möglichkeit, den Wert zu ändern, sondern, wie der Name `SP_WDG_DialogShowOnly` ausdrücken soll, nur die Anzeige an oder ab zu schalten. Die Initialisierungen des graphischen Attributs stehen für die Position sowie die Tatsache, dass es standardmässig nicht angezeigt werden soll. Und mit `SetGlobalShow` wird auch die Anzeige der graphischen Ausprägungen dieses Attributs von zentraler Stelle aus steuerbar.

Die Zeilen 25-28 definieren ein numerisches Attribut, das über ein Widget zur Eingabe von Zahlenwerten im Dialog editierbar ist, sich allerdings nicht einer einfachen Anzeige über das `SP_GR_NumberAttribute` bedient. Stattdessen wurde eine neue graphische Ausprägung definiert, die das Darstellen der Werte dieses Attributs individuelle gestaltet. Dabei werden Werte kleiner als 4 durch besonders angeordnete, gefüllte Kreise dargestellt und erst Werte ab 4 und darüber durch die entsprechende Zahl.

Das Attribut „Logic“ steht für die Möglichkeit, die Elemente dieser Partition so zu organisieren, dass es mehrere graphische Ausprägungen zu ein und demselben Element in der Datenstruktur geben kann. Dabei gibt der zweite Parameter des Konstruktors von `SP_DS_LogicAttribute` an, dass die Entscheidung, ob zwei Elemente bei Aktivierung dieser Option ihre graphischen Ausprägungen zusammenführen sollen, von der Gleichheit der Werte in den Attributen mit dem Bezeichner „Name“ abhängig gemacht wird.

Das Attribut „Comment“ zeigt die Verwendung eines weiteren Widgets für die Darstellung in einem Dialog. Dabei steht `SP_WDG_DialogMultiline` für ein mehrzeiliges Eingabefeld, was zudem noch auf einer separaten Seite im Dialog (nämlich auf einer Seite namens „Other“) angezeigt werden soll.

Dass die graphische Ausprägung für die Elemente dieser Partition neu festgelegt wird (durch `SetGraphic`) geschieht nur aus Bequemlichkeit. Eigentlich würde es ausreichen, der bereits assoziierten graphischen Ausprägung vom selben Typ über `SetFixedSize(TRUE)` die Möglichkeit der freien Grössenänderung zu entziehen.

Über `SetGraphicWidget` wird den Elementen dieser Partition die Möglichkeit gegeben, sich zur Änderung der Farbattribute im Dialog darzustellen. Diese gesonderte Möglichkeit wird über die Definition des Widgets auf eine Seite namens „Graphic“ verbannt.

Zum Abschluss wird in Zeile 46 noch eine individuelle Ableitung von `SP_DS_Animator`, nämlich ein `SP_DS_PlaceAnimator`, mit dieser Partition verknüpft. Der Parameter „Marking“ steht dabei für den Namen des Attributs, dessen Wert zur Entscheidung der Aktivierung und zur Modifikation während der Animation verwendet werden soll.

```
47      ////////////////////////////////////////////////////////////////////
48      l_pcNS = p_pcGraph->RenameNodeclass("Rectangle", "Transition");
49      l_pcAttr = l_pcNS->GetPrototype()->GetAttribute("Name");
50      l_pcAttr->SetGlobalShow();
51
52      l_pcAttr = l_pcNS->AddAttribute(new SP_DS_IdAttribute("ID"));
53      l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogShowOnly("General"));
54      l_pcGrAttr = new SP_GR_NumberAttribute(l_pcAttr, "_t%_");
55      l_pcGrAttr->SetOffsetX(20);
56      l_pcGrAttr->SetShow(FALSE);
57      l_pcAttr->AddGraphic(l_pcGrAttr);
58      l_pcAttr->SetGlobalShow();
59
60      l_pcAttr = new SP_DS_LogicAttribute("Logic", "Name");
61      l_pcNS->AddAttribute(l_pcAttr);
62      l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogBool("General"));
63
64      l_pcAttr = new SP_DS_TextAttribute("Comment", "");
65      l_pcNS->AddAttribute(l_pcAttr);
66      l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogMultiline("Other"));
67      l_pcGrAttr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
68      l_pcGrAttr->SetOffsetX(40);
69
70      l_pcGr = new SP_GR_Rectangle(l_pcNS->GetPrototype(), 20, 20);
71      l_pcGr->SetFixedSize(TRUE);
72      l_pcNS->SetGraphic(l_pcGr);
73      l_pcNS->RegisterGraphicWidget(new SP_WDG_DialogGraphic("Graphic"));
74
75      // animator
76      l_pcNS->AddAnimator(new SP_DS_TransAnimator());
```

Die Modifikation der bestehenden Knotenpartition namens „Rectangle“ erfolgt nach einem ganz ähnlichen Muster, wie die eben besprochene. Sie erhält sogar denselben Satz von Attributen, mit einem Unterschied in der Formatierung der Instanz von `SP_DS_IdAttribute`. Ausserdem Ist der assoziierte Animator vom eigens für diese Netzklasse implementierten Typ `SP_DS_TransAnimator`.

```
77      ////////////////////////////////////////////////////////////////
78      l_pcNS = new SP_DS_Nodeclass(p_pcGraph, "Coarse Place");
79      p_pcGraph->AddNodeclass(l_pcNS);
80      l_pcAttr = l_pcNS->AddAttribute(new SP_DS_TextAttribute("Name", ""));
81      l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogText("General"));
82      l_pcGrAttr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
83      l_pcGrAttr->SetOffsetY(20);
84      l_pcGrAttr->SetShow(TRUE);
85      l_pcAttr->SetGlobalShow();
86
87      // subnet property for this class
88      l_pcCoarse = new SP_DS_Coarse("Sub", "Place", p_pcGraph, FALSE);
89      l_pcCoarse->SetLabelAttribute("Name");
90      l_pcNS->SetCoarse(l_pcCoarse);
91      l_pcGr = new SP_GR_DoubleCircle(l_pcNS->GetPrototype(), 20.0);
92      l_pcGr->SetFixedSize(TRUE);
93      l_pcNS->SetGraphic(l_pcGr);
94      l_pcNS->RegisterGraphicWidget(new SP_WDG_DialogGraphic("Graphic"));
```

Die Knotenpartition „Coarse Place“ ist neu in dieser Netzklasse. Neben dem bekannten Vorgehen zum Hinzufügen eines Textattributs mit der Bezeichnung „Name“ und einem Widget zur Manipulation von textuellen Werten fällt die Definition von `SP_DS_Coarse` auf (siehe auch Abschnitt „SP_DS_Coarse“ auf Seite 30).

Diese Klasse wird über 4 Parameter initialisiert, wobei der erste dem Objekt einen Namen gibt. Der zweite Parameter „Place“ steht für den Namen der Knotenpartition, von deren Typ die Elemente „am inneren Rand“ der Unternetze sein sollen. Der dritte Parameter steht für den Graphen, aus dessen Partitionen die im Inneren darstellbaren Elemente bestehen und da es derselbe Graph ist, den wir aktuell auch bearbeiten (denkbar wäre auch die Instanziierung eines neuen Objekts, das mit einer anderen Netzklasse initialisiert wird) steht der vierte Parameter für die Tatsache, dass dieses Coarse-Netz seinen Graphen nicht selbst verwaltet. `SetLabelAttribute` definiert eines der für diese Partition definierten Attribute, dass sein Wert an diversen Stellen in der Oberfläche als Anzeige verwendet werden soll.

Die Festlegung einer graphischen Ausprägung und die Erlaubnis, sich im Dialog zur Änderung seiner Farbattribute zu melden, schliesst die Definition dieser Partition ab.

```
95      ////////////////////////////////////////////////////////////////////
96      l_pcNS = new SP_DS_Nodeclass(p_pcGraph, "Coarse Transition");
97      p_pcGraph->AddNodeclass(l_pcNS);
98      l_pcAttr = l_pcNS->AddAttribute(new SP_DS_TextAttribute("Name"));
99      l_pcAttr->RegisterDialogWidget(new SP_WDG_DialogText("General"));
100     l_pcGrAttr = l_pcAttr->AddGraphic(new SP_GR_TextAttribute(l_pcAttr));
101     l_pcGrAttr->SetOffsetY(20);
102     l_pcGrAttr->SetShow(TRUE);
103     l_pcAttr->SetGlobalShow();
104
105     // subnet property for this class
106     l_pcCoarse = new SP_DS_Coarse("Sub", "Transition", p_pcGraph, FALSE);
107     l_pcCoarse->SetLabelAttribute("Name");
108     l_pcNS->SetCoarse(l_pcCoarse);
109
110     l_pcGr = new SP_GR_DoubleRectangle(l_pcNS->GetPrototype(), 20, 20);
111     l_pcGr->SetFixedSize(TRUE);
112     l_pcNS->SetGraphic(l_pcGr);
113     l_pcNS->RegisterGraphicWidget(new SP_WDG_DialogGraphic("Graphic"));
```

Als Äquivalent zur Partition von Elementen, die „platzberandete“ (die Randknoten sind Elemente der Partition „Place“) Unternetze repräsentieren, wird noch eine weitere Partition für die „transitionsberandeten“ Unternetze eingeführt. Diese Definition erfolgt analog und es bleibt nur noch die Modifikation der schon existierenden Kantenpartition des Ursprungsgraphen.

```
114     ////////////////////////////////////////////////////////////////////
115     SP_DS_Edgeclass* l_pcEC = p_pcGraph->GetEdgeclass("Edge");
116     l_pcAttr = l_pcEC->GetPrototype()->GetAttribute("Name");
117     l_pcAttr->SetGlobalShow();
118     l_pcEC->RegisterGraphicWidget(new SP_WDG_DialogGraphic("Graphic"));
119
120     p_pcGraph->SetAnimation(new SP_DS_PedAnimation(200, 5000));
121
122     return p_pcGraph;
123 }
```

Da die Bezeichnung der Partition beibehalten werden soll, wird diese nur zur Modifikation eines Attributs vom Graphen erfragt (`GetEdgeclass`) und daraufhin das Attribut mit dem Bezeichner „Name“ auch der Liste der Attribute, die über einen zentralen Dialog an- und abgeschaltet werden können, hinzugefügt. Ausserdem kann der Anwender auch die Attribute der graphischen Ausprägungen dieser Partitionen auf einer Seite im Dialog bearbeiten. In Zeile 120 wird dem Graphen lediglich noch mitgeteilt, welche Animationsklasse für die Steuerung der Animatoren zuständig ist. Die Initialisierung gibt die Timerfrequenz (200 ms) und die Dauer eines Zyklus' (5s) an.

5.4 SP_DS_PedAnimation

Diese Klasse stellt die bislang einzige Ableitung von SP_DS_Animation dar und wird im Zusammenhang mit der zuletzt beschriebenen Netzklasse SP_DS_SimplePed auch aktiv benutzt. Um ihre Arbeit leisten zu können wurden ausserdem zwei Datenstruktur-Animatoren (`SP_DS_PlaceAnimator` und `SP_DS_TransAnimator`) von `SP_DS_Animator` sowie `SP_GR_MarkAnimator` von `SP_GR_Animator` abgeleitet.

Wie im Abschnitt „SP_DS_Animation“ auf Seite 53 beschrieben, wird das Animationsobjekt zum Zeitpunkt des Umschaltens in den Animations-Modus über die Methode `Initialise` initialisiert. Dabei überlässt die Implementierung von `SP_DS_PedAnimation::Initialise` als erstes der Basisklasse die Aufgabe, alle vorhandenen Objekte mit je einer Instanz der registrierten Animatoren zu assoziieren. Anschliessend wird die Liste der Animatoren der Basisklasse iteriert und je nach Typ des Animators der Eintrag in eine von zwei neuen Listen aufgenommen. Diese sind Attribute von `SP_DS_PedAnimation` und enthalten einmal alle Animatoren für die Partition der Plätze und zum Anderen die der Partition der Transitionen.

```
1    bool
2    SP_DS_PedAnimation::Initialise(SP_DS_Graph* p_pcGraph)
3    {
4       bool l_bReturn = SP_DS_Animation::Initialise(p_pcGraph);
5       SP_List<SP_DS_Animator*>::iterator l_Iter;
6       m_lAllPlaceAnimators.clear();
7       m_lAllTransAnimators.clear();
8       if (l_bReturn)
9       {
10           for (l_Iter = m_lAllAnimators.begin();
11                l_Iter != m_lAllAnimators.end();
12                ++l_Iter)
13           {
14               if ((*l_Iter)->GetAnimatorType() == SP_DS_ANIMATOR_PLACE)
15               {
16                   m_lAllPlaceAnimators.push_back(
17                       static_cast<SP_DS_PlaceAnimator*>((*l_Iter)));
18               }
19               else
20                   if ((*l_Iter)->GetAnimatorType() == SP_DS_ANIMATOR_TRANS)
21                   {
22                       m_lAllTransAnimators.push_back(
23                           static_cast<SP_DS_TransAnimator*>((*l_Iter)));
24                   }
25       }
26
27       return l_bReturn;
28   }
```

Der Aufruf der, die Animation definierenden, Methoden erfolgt weiterhin aus der Timer-Methode von `SP_DS_Animation`, allerdings wurden alle drei Methoden (`PreStep`, `Step` und `PostStep`) in dieser Ableitung überschrieben, arbeiten sie doch immer auf den eben erstellten, getrennten Listen von Animatoren.

Die beiden neuen Klassen SP_DS_TransAnimator und SP_DS_PlaceAnimator definieren, neben den geforderten, eine Vielzahl neuer Methoden, die hauptsächlich der Entscheidung der Aktivierbarkeit dienen und die Kenntnis des „Markenspiels“ im Zusammenhang mit Platz/Transitions-Netzen voraussetzt.

In SP_DS_PedAnimation::PreStep läuft die Feststellung der Menge der schaltfähigen Transitionen und damit der Menge der Plätze, von denen Marken abgezogen werden, folgendermassen ab:

für alle Trans-Animatoren `:: InformPrePlaces`

Die Transitionen „informieren“ ihre Vorplätze (genauer die Animatoren der Vorplätze) über die Tatsache, dass sie an Marken von ihnen interessiert sind. Dabei spielt die Anzahl der Marken (bedingt durch die Anzahl der verbundenen Kanten) eine Rolle. Nach Abschluss dieses Schritts

sind die Animatoren der Plätze über die Anzahl an Marken informiert, die sie pro Nachtransition zur Verfügung haben müssten um jeder Konzession erteilen zu können.

für alle Place-Animatoren `:: TestMarking`

Jeder Nachtransition (die sich ja für Marken gemeldet hat) wird mitgeteilt, ob der Wert im Attribut „Marking“ für die geforderte Anzahl Marken ausreicht. Ist das der Fall, registriert sich der Platzanimator bei dem Animator der entsprechenden Nachtransition.

für alle Trans-Animatoren `:: TestConcession`

Die Animatoren der Transitionen überprüfen die Anzahl der Meldungen ihrer Vorplätze über das zur-Verfügung-stellen von Marken. Nur, wenn die Anzahl der im 1. Schritt informierten Vorplätze kleiner/gleich der Anzahl der eingetroffenen Bestätigungen ist, wird dieser Schritt als erfolgreich gewertet. Andernfalls schaltet sich der Animator dieser Transition von selbst aus und zieht im gleichen Schritt die Markenanforderung an all seine Vorplätze wieder zurück. Somit sind Transitionen, die nicht genügend Marken auf allen Vorplätzen haben, automatisch als „tot“ markiert und andererseits hat sich die Anzahl der Forderungen an den Animatoren der Plätze unter Umständen reduziert.

für alle Place-Animatoren `:: ResolveConflicts`

Alle jetzt noch mit Markenforderungen belegten Animatoren der Plätze können entweder alle Forderungen erfüllen, oder es besteht ein Konflikt über den Nachtransitionen. Nun wird, per fairem Zufall, die Menge der fordernden Nachtransitionen so lange reduziert, bis der Wert im Attribut „Marking“ grösser/gleich der Summe der Forderungen aller Nachtransitionen ist. Nach dieser Reduktion werden alle übrig gebliebenen Animatoren der Nachtransitionen „aktiviert“.

für alle Trans-Animatoren `:: PreStep`

Das ist der eigentliche PreStep, wie er allen Animatoren eigen sein muss. Für die Transitionen wird hier nur überprüft, ob sie „aktiviert“ worden sind. Sollte das der Fall sein, aktiviert der Animator all seine Vorplätze und erzeugt für die Menge der graphischen Ausprägungen all seiner ausgehenden Kanten je eine eigene graphische Ausprägung (Instanz von

SP_GR_MarkAnimator). Wenn das passiert ist, wird PreStep mit TRUE verlassen und dieser Animator der Liste der aktivierten Transitions-Animatoren für diesen aktuellen Schritt in SP_DS_PedAnimation hinzugefügt.

für alle Place-Animatoren `::PreStep`

Dieser Schritt entspricht dem eigentlichen PreStep. Ist der Animator „aktiviert“ worden, erzeugt er für alle graphischen Ausprägungen all seiner ausgehenden Kanten je eine Instanz von SP_GR_Markattribute und fügt sie seiner eigenen Liste der graphischen Ausprägungen hinzu. Ist das der Fall, wird ausserdem der Wert des zugehörigen „Marking“-Attributs entsprechend der abgegebenen Marken decrementiert und anschliessend die Methode mit TRUE verlassen und dieser Animator der Liste der für diesen Schritt aktivierten Platz-Animatoren in SP_DS_PedAnimation hinzugefügt.

Nach dieser Abfolge ist der PreStep der Animation abgeschlossen und in den beiden Listen der Animatoren für den aktuellen Schritt sind unter Umständen einige Einträge enthalten. Für alle diese Einträge wird für die erste Hälfte der Ticks die Methode `Step` der Platzanimatoren ausgeführt, was die Marken von den Plätzen auf den ausgehenden Kanten entlang wandern lässt. Nach der Hälfte der maximalen Ticks für diesen Zyklus werden die `PostStep`-Methoden der Platzanimatoren aufgerufen, woraufhin sie ihre Liste der graphischen Ausprägungen leeren, dafür angelegten Speicher wieder freigeben und ihren internen Status zurücksetzen.

Anschliessend folgt für den Rest der Ticks der Aufruf der `Step`-Methode der Transitionsanimatoren, was die generierten Instanzen von `SP_GR_MarkAnimator` über die ausgehenden Kanten wandern lässt, und beim letzten Tick des Timers der Aufruf von `PostStep` für eben diese verbleibenden Animatoren. Dabei informieren diese jeden der nachfolgenden Platzanimatoren, wodurch diese den Wert in ihrem entsprechenden „Marking“ Attribut inkrementieren.

`SP_GR_MarkAnimator` selbst ist dabei nicht viel mehr als eine besonders bewegliche Ausgabe von `SP_GR_Circle`. Genau wie diese Klasse aggregiert sie eine Instanz von `SP_GR_BaseCircle` als Primitive, die sie allerdings in Rot und Schwarz einfärbt und gegen sämtliche Einflussnahme von aussen (durch Mausklicks zum Beispiel) isoliert. Ausserdem besitzt `SP_GR_MarkAnimator` die Fähigkeit, der graphischen Ausprägung einer Kante (Instanz von `SP_GR_Edge`) zu folgen, indem die maximale Anzahl der Schritte, die

erlaubt sind auf die Länge der zurückzulegenden Strecke umgerechnet und anschliessend die zu durchlaufenden Koordinaten vorab berechnet und gespeichert werden. Das führt dazu, dass in der `Step`-Methode nur noch der nächste gespeicherte Wegpunkt angesprungen werden muss, was den Rechenaufwand für die meiste Zeit eines Animationszyklus' minimieren helfen soll. Dieses Ausrichten des MarkAnimators auf die Kante erfolgt jeweils in den beiden PreStep-Methoden von `SP_DS_PlaceAnimator` und `SP_DS_TransAnimator`, für den Fall, dass sie jeweils „aktiviert“ worden sind.

6 Weiterführende Arbeiten

An weiterführenden Arbeiten bleibt hauptsächlich die Implementierung neuer Attribute, graphischer Ausprägungen, Widgets und Animatoren sowie Animationen – je nach Anforderung der zu erfüllenden Aufgaben. Zu diesem Zweck findet sich im Anhang ab Seite 89 das „Handbuch des Netzadministrators“, das die Klassen und Funktionen, die im Zusammenhang mit der Erweiterung der Funktionalität wichtig sind, in einer übersichtlichen Form beschreibt.

Notwendig ist darüber hinaus ein Review des Quelltextes mit dem Ziel, die Portierung auf Systeme unter Linux, Solaris oder auch MacOS durchführen zu können. Prinzipiell ist jede der verwendeten Bibliotheken schon auf diese Portierbarkeit ausgelegt und sie hat in der ersten Version auch funktioniert. Leider fehlten mir die Ressourcen und vor allem die Zeit, die Machbarkeit im Rahmen dieser Arbeit nachzuweisen. Einige der verwendeten Makros könnten sich als Stolpersteine entpuppen und müssten im Rahmen der Portierung neu definiert werden.

Die, unter Verwendung der Software *doxygen* [4], begonnene Dokumentation des Quelltextes hat noch nicht den Umfang erreicht, der zur erschöpfenden Erklärung aller Zusammenhänge wünschenswert ist. Ausserdem auf der Wunschliste finden sich noch der Entwurf und die Umsetzung eines tragfähigen Konzepts zur „externen“ Definition von Netzklassen, also ohne den Quelltext jedesmal neu übersetzen zu müssen, sowie die Idee eines Satzes von graphischen Ausprägungen, der als Bibliothek zur Erweiterung zusammengestellt und verteilt werden kann.

Anhang

Anhang A Quellcodehierarchie

Der Quelltext der Software ist nach einer ähnlichen Hierarchie sortiert, wie sie durch die Vererbung impliziert ist. Insgesamt umfasst die Anwendung zum jetzigen Zeitpunkt 95 Header- (Dateiendung .h) und 90 Implementierungs-Dateien (Dateiendung .cpp) der Sprache C++.

Die folgende Tabelle stellt alle Dateien des Systems dar:

Pfad	Name	LOC	Code LOC	Code %
/	snoopy.cpp	119	79	66%
/	snoopy.h	66	42	64%
/	sp_defines.h	128	84	66%
/	sp_utilities.cpp	42	28	67%
/	sp_utilities.h	37	9	24%
/	wx_intellisense.h	165	90	55%
/core	sp_core.cpp	572	440	77%
/core	sp_core.h	352	82	23%
/core	sp_list.cpp	10	1	10%
/core	sp_list.h	38	24	63%
/core	sp_map.cpp	10	1	10%
/core	sp_map.h	25	11	44%
/core	sp_set.cpp	11	1	9%
/core	sp_set.h	25	11	44%
/core	sp_vector.cpp	11	1	9%
/core	sp_vector.h	25	11	44%
/core	sp_xmlreader.cpp	941	753	80%
/core	sp_xmlreader.h	79	54	68%
/core	sp_xmlwriter.cpp	444	352	79%
/core	sp_xmlwriter.h	53	36	68%
/core/base	sp_animator.cpp	18	8	44%
/core/base	sp_animator.h	38	22	58%
/core/base	sp_data.cpp	513	385	75%
/core/base	sp_data.h	428	75	18%
/core/base	sp_elementclass.cpp	42	25	60%
/core/base	sp_elementclass.h	52	29	56%
/core/base	sp_error.h	64	16	25%
/core/base	sp_graphic.cpp	559	414	74%
/core/base	sp_graphic.h	604	131	22%
/core/base	sp_name.h	36	13	36%
/core/base	sp_netelement.cpp	23	11	48%
/core/base	sp_netelement.h	62	18	29%

Pfad	Name	LOC	Code LOC	Code %
/core/base	sp_type.h	56	36	64%
/core/tools	sp_xmltools.cpp	53	29	55%
/core/tools	sp_xmltools.h	184	92	50%
/sp_ds	sp_ds_animator.cpp	27	17	63%
/sp_ds	sp_ds_animator.h	59	36	61%
/sp_ds	sp_ds_attribute.cpp	197	153	78%
/sp_ds	sp_ds_attribute.h	82	46	56%
/sp_ds	sp_ds_edge.cpp	652	489	75%
/sp_ds	sp_ds_edge.h	121	57	47%
/sp_ds	sp_ds_edgeclass.cpp	446	327	73%
/sp_ds	sp_ds_edgeclass.h	105	49	47%
/sp_ds	sp_ds_graph.cpp	695	533	77%
/sp_ds	sp_ds_graph.h	139	80	58%
/sp_ds	sp_ds_netclass.cpp	98	70	71%
/sp_ds	sp_ds_netclass.h	49	30	61%
/sp_ds	sp_ds_node.cpp	742	562	76%
/sp_ds	sp_ds_node.h	298	64	21%
/sp_ds	sp_ds_nodeclass.cpp	452	338	75%
/sp_ds	sp_ds_nodeclass.h	125	47	38%
/sp_ds/animators	sp_ds_placeanimator.cpp	252	197	78%
/sp_ds/animators	sp_ds_placeanimator.h	60	36	60%
/sp_ds/animators	sp_ds_transanimator.cpp	193	152	79%
/sp_ds/animators	sp_ds_transanimator.h	46	28	61%
/sp_ds/attributes	sp_ds_boolattribute.cpp	24	12	50%
/sp_ds/attributes	sp_ds_boolattribute.h	33	18	55%
/sp_ds/attributes	sp_ds_idattribute.cpp	60	42	70%
/sp_ds/attributes	sp_ds_idattribute.h	33	16	48%
/sp_ds/attributes	sp_ds_logicattribute.cpp	115	78	68%
/sp_ds/attributes	sp_ds_logicattribute.h	42	22	52%
/sp_ds/attributes	sp_ds_numberattribute.cpp	47	29	62%
/sp_ds/attributes	sp_ds_numberattribute.h	34	18	53%
/sp_ds/attributes	sp_ds_textattribute.cpp	39	21	54%
/sp_ds/attributes	sp_ds_textattribute.h	34	19	56%
/sp_ds/extensions	sp_ds_animation.cpp	338	266	79%
/sp_ds/extensions	sp_ds_animation.h	106	72	68%
/sp_ds/extensions	sp_ds_coarse.cpp	388	294	76%
/sp_ds/extensions	sp_ds_coarse.h	121	58	48%
/sp_ds/extensions	sp_ds_pedanimation.cpp	269	221	82%
/sp_ds/extensions	sp_ds_pedanimation.h	63	40	63%
/sp_ds/netclasses	sp_ds_bipartgraph.cpp	81	55	68%
/sp_ds/netclasses	sp_ds_bipartgraph.h	27	14	52%
/sp_ds/netclasses	sp_ds_simplegraph.cpp	55	33	60%
/sp_ds/netclasses	sp_ds_simplegraph.h	25	13	52%
/sp_ds/netclasses	sp_ds_simpleped.cpp	205	150	73%
/sp_ds/netclasses	sp_ds_simpleped.h	28	15	54%
/sp_gr	sp_gr_animator.cpp	19	9	47%
/sp_gr	sp_gr_animator.h	31	17	55%
/sp_gr	sp_gr_attribute.cpp	43	27	63%
/sp_gr	sp_gr_attribute.h	44	26	59%
/sp_gr	sp_gr_edge.cpp	852	647	76%

Pfad	Name	LOC	Code LOC	Code %
/sp_gr	sp_gr_edge.h	188	51	27%
/sp_gr	sp_gr_node.cpp	509	362	71%
/sp_gr	sp_gr_node.h	91	43	47%
/sp_gr/animators	sp_gr_markanimator.cpp	189	137	72%
/sp_gr/animators	sp_gr_markanimator.h	50	28	56%
/sp_gr/attributes	sp_gr_markattribute.cpp	188	129	69%
/sp_gr/attributes	sp_gr_markattribute.h	48	29	60%
/sp_gr/attributes	sp_gr_numberattribute.cpp	149	103	69%
/sp_gr/attributes	sp_gr_numberattribute.h	44	27	61%
/sp_gr/attributes	sp_gr_textattribute.cpp	149	103	69%
/sp_gr/attributes	sp_gr_textattribute.h	43	26	60%
/sp_gr/base	sp_gr_basecircle.cpp	105	76	72%
/sp_gr/base	sp_gr_basecircle.h	34	20	59%
/sp_gr/base	sp_gr_basedrawn.cpp	134	100	75%
/sp_gr/base	sp_gr_basedrawn.h	45	26	58%
/sp_gr/base	sp_gr_baseedge.cpp	158	104	66%
/sp_gr/base	sp_gr_baseedge.h	35	19	54%
/sp_gr/base	sp_gr_baserectangle.cpp	104	76	73%
/sp_gr/base	sp_gr_baserectangle.h	34	20	59%
/sp_gr/base	sp_gr_basetext.cpp	110	82	75%
/sp_gr/base	sp_gr_basetext.h	36	20	56%
/sp_gr/edges	sp_gr_arrowedge.cpp	71	42	59%
/sp_gr/edges	sp_gr_arrowedge.h	30	16	53%
/sp_gr/eventhandler	sp_grm_attributehandler.cpp	107	79	74%
/sp_gr/eventhandler	sp_grm_attributehandler.h	31	16	52%
/sp_gr/eventhandler	sp_grm_edgehandler.cpp	269	195	72%
/sp_gr/eventhandler	sp_grm_edgehandler.h	39	22	56%
/sp_gr/eventhandler	sp_grm_eventhandler.cpp	412	305	74%
/sp_gr/eventhandler	sp_grm_eventhandler.h	57	33	58%
/sp_gr/eventhandler	sp_grm_shapehandler.cpp	71	51	72%
/sp_gr/eventhandler	sp_grm_shapehandler.h	27	14	52%
/sp_gr/eventhandler	sp_grm_upwardhandler.cpp	152	122	80%
/sp_gr/eventhandler	sp_grm_upwardhandler.h	44	25	57%
/sp_gr/shapes	sp_gr_circle.cpp	79	48	61%
/sp_gr/shapes	sp_gr_circle.h	33	18	55%
/sp_gr/shapes	sp_gr_doublecircle.cpp	77	39	51%
/sp_gr/shapes	sp_gr_doublecircle.h	31	16	52%
/sp_gr/shapes	sp_gr_doublerectangle.cpp	82	43	52%
/sp_gr/shapes	sp_gr_doublerectangle.h	31	16	52%
/sp_gr/shapes	sp_gr_drawnshape.cpp	119	73	61%
/sp_gr/shapes	sp_gr_drawnshape.h	39	22	56%
/sp_gr/shapes	sp_gr_rectangle.cpp	65	41	63%
/sp_gr/shapes	sp_gr_rectangle.h	31	17	55%
/sp_gui/dialogs	sp_dlg_animation.cpp	129	98	76%
/sp_gui/dialogs	sp_dlg_animation.h	56	25	45%
/sp_gui/dialogs	sp_dlg_animationproperties.cpp	90	63	70%
/sp_gui/dialogs	sp_dlg_animationproperties.h	51	22	43%
/sp_gui/dialogs	sp_dlg_chosecoarse.cpp	105	74	70%
/sp_gui/dialogs	sp_dlg_chosecoarse.h	54	24	44%
/sp_gui/dialogs	sp_dlg_graphproperties.cpp	122	80	66%

Pfad	Name	LOC	Code LOC	Code %
/sp_gui/dialogs	sp_dlg_graphproperties.h	58	27	47%
/sp_gui/dialogs	sp_dlg_shapeproperties.cpp	179	123	69%
/sp_gui/dialogs	sp_dlg_shapeproperties.h	99	33	33%
/sp_gui/dialogs	sp_dlg_transformgraphic.cpp	138	104	75%
/sp_gui/dialogs	sp_dlg_transformgraphic.h	64	33	52%
/sp_gui/management	sp_gm_docmanager.cpp	196	144	73%
/sp_gui/management	sp_gm_docmanager.h	45	28	62%
/sp_gui/management	sp_gm_doctemplate.cpp	22	19	86%
/sp_gui/management	sp_gm_doctemplate.h	38	25	66%
/sp_gui/mdi	sp_mdi_coarsedoc.cpp	142	104	73%
/sp_gui/mdi	sp_mdi_coarsedoc.h	39	21	54%
/sp_gui/mdi	sp_mdi_coarseview.cpp	69	42	61%
/sp_gui/mdi	sp_mdi_coarseview.h	27	15	56%
/sp_gui/mdi	sp_mdi_diagram.cpp	109	85	78%
/sp_gui/mdi	sp_mdi_diagram.h	43	23	53%
/sp_gui/mdi	sp_mdi_doc.cpp	304	237	78%
/sp_gui/mdi	sp_mdi_doc.h	98	67	68%
/sp_gui/mdi	sp_mdi_view.cpp	1.119	877	78%
/sp_gui/mdi	sp_mdi_view.h	125	85	68%
/sp_gui/widgets	sp_wdg_coarsetreectrl.cpp	178	135	76%
/sp_gui/widgets	sp_wdg_coarsetreectrl.h	86	37	43%
/sp_gui/widgets	sp_wdg_graphtreectrl.cpp	135	98	73%
/sp_gui/widgets	sp_wdg_graphtreectrl.h	92	43	47%
/sp_gui/widgets	sp_wdg_notebook.cpp	58	45	78%
/sp_gui/widgets	sp_wdg_notebook.h	35	21	60%
/sp_gui/widgets	sp_wdg_notebookpage.cpp	41	28	68%
/sp_gui/widgets	sp_wdg_notebookpage.h	46	20	43%
/sp_gui/widgets	sp_wdg_toolbar.cpp	24	13	54%
/sp_gui/widgets	sp_wdg_toolbar.h	37	22	59%
/sp_gui/widgets	sp_wdg_treectrl.cpp	227	172	76%
/sp_gui/widgets	sp_wdg_treectrl.h	144	80	56%
/sp_gui/widgets/dialogs	sp_wdg_dialogbase.cpp	93	70	75%
/sp_gui/widgets/dialogs	sp_wdg_dialogbase.h	56	30	54%
/sp_gui/widgets/dialogs	sp_wdg_dialogbool.cpp	76	47	62%
/sp_gui/widgets/dialogs	sp_wdg_dialogbool.h	39	18	46%
/sp_gui/widgets/dialogs	sp_wdg_dialoggraphic.cpp	234	186	79%
/sp_gui/widgets/dialogs	sp_wdg_dialoggraphic.h	67	39	58%
/sp_gui/widgets/dialogs	sp_wdg_dialogmultiline.cpp	87	60	69%
/sp_gui/widgets/dialogs	sp_wdg_dialogmultiline.h	32	15	47%
/sp_gui/widgets/dialogs	sp_wdg_dialognumber.cpp	84	54	64%
/sp_gui/widgets/dialogs	sp_wdg_dialognumber.h	39	19	49%
/sp_gui/widgets/dialogs	sp_wdg_dialogshowonly.cpp	52	31	60%
/sp_gui/widgets/dialogs	sp_wdg_dialogshowonly.h	34	16	47%
/sp_gui/widgets/dialogs	sp_wdg_dialogtext.cpp	83	55	66%
/sp_gui/widgets/dialogs	sp_wdg_dialogtext.h	37	18	49%
/sp_gui/windows	sp_gui_canvas.cpp	698	542	78%
/sp_gui/windows	sp_gui_canvas.h	112	72	64%
/sp_gui/windows	sp_gui_childframe.cpp	280	208	74%
/sp_gui/windows	sp_gui_childframe.h	102	42	41%
/sp_gui/windows	sp_gui_devbarcontainer.cpp	102	77	75%

Pfad	Name	LOC	Code LOC	Code %
/sp_gui/windows	sp_gui_devbarcontainer.h	47	27	57%
/sp_gui/windows	sp_gui_mainframe.cpp	407	323	79%
/sp_gui/windows	sp_gui_mainframe.h	254	58	23%
Summe:		25.858	16.990	

Tabelle 4 - Quellcodedateien der Anwendung

Anhang B Handbuch des Netzadministrators

Hier stelle ich die zentralen Klassen zur Definition eigener Netzklassen alphabetisch mit ihren Funktionen im Detail und mit dem Fokus auf den Softwareentwickler vor.

In fast allen Klassenbeschreibungen ist der Vorstellung der Methoden eine Auflistung von Attributen vorangestellt. Diese Attribute sind immer mindestens als `protected`-Members der entsprechenden Klasse definiert, es gibt keine `public` Attribute. Der Zugriff auf diese sollte, auch aus abgeleiteten Klassen immer über die Zugriffsmethoden (die zumeist als `inline` deklariert sind) erfolgen!

1 SP_Data

Basis für alle Datenstrukturelemente eines Graphen. In dieser Klasse ist hauptsächlich die Verbindung von Elementen der Datenstruktur zu ihren graphischen Ausprägungen gekapselt.

Die meisten der hier implementierten Methoden sind als `virtual` deklariert und sollten trotzdem in aller Regel in den entsprechenden Ableitungen aufgerufen werden, da sie grundlegende Arbeiten erledigen und so den Aufwand reduzieren können.

Abgeleitet von

`SP_IdCounter`
`SP_NetElement`
`SP_Type`

Include

`core/base/SP_Data.h`

SP_Data::m_bLogical

```
bool m_bLogical
```

Statusvariable, die anzeigt, dass dieses Element der Datenstruktur in der Liste seiner graphischen Ausprägungen, vom Nutzer explizit gewünscht, eine Mehrdeutigkeit anzeigt.

SP_Data::m_pcCoarse

```
SP_DS_Coarse* m_pcCoarse
```

Instanz der *Coarse*-Klasse, die angibt, dass dieses Element der Datenstruktur ein Unternetz repräsentiert. Wenn vorhanden, wird diese Instanz an den wichtigen Abläufen stets beteiligt.

SP_Data::SP_Data

```
SP_Data(SP_ELEMENT_TYPE p_eType)
```

Konstruktor. Der Wert des Parameters ist einer aus denen, die im Abschnitt „SP_Type“ auf Seite 117 vorgestellt werden.

SP_Data::AddGraphic

```
virtual SP_Graphic* AddGraphic(SP_Graphic* p_pcGraphic)
```

Fügt der Liste der graphischen Ausprägungen dieses Objekts einen neuen Eintrag hinzu. Dabei wird ausschliesslich überprüft, ob dieses Objekt nicht schon Element der Liste ist.

Rückgabe

Liefert in jedem Fall den Wert des Parameters zurück.

SP_Data::AddGraphicInSubnet

```
virtual SP_Graphic* AddGraphicInSubnet(unsigned int p_nNetnumber)
```

Fügt eine neue graphische Ausprägung nach der Vorlage des ersten Eintrags der Liste der graphischen Ausprägungen (wenn vorhanden) im angegebenen Netz hinzu, oder liefert ein bereits vorhandenes Element im angegebenen Netz zurück.

Parameter

p_nNetnumber

Nummer des Netzes, für das eine Graphik gesucht oder erzeugt werden soll.

Rückgabe

Ein neues Objekt in der Liste der graphischen Ausprägungen, oder ein bereits vorhandenes in dem angegebenen Netz. Oder `NULL` für den Fall, dass es keine Einträge in der Liste der graphischen Ausprägungen gibt.

SP_Data::GetClassName

```
virtual const char* GetClassName() = 0
```

Schnittstellendefinition, die in abgeleiteten Klassen implementiert werden muss und für Elemente vom Typ SP_ELEMENT_NODE oder SP_ELEMENT_EDGE den Namen der zugehörigen Partition liefert.

SP_Data::GetClassObject

```
virtual SP_ElementClass* GetClassObject()
```

Schnittstellendefinition, die in dieser Klasse immer NULL liefert und in Ableitungen vom Typ SP_ELEMENT_NODE oder SP_ELEMENT_EDGE die Instanz der zugehörigen Partition liefert.

SP_Data::GetCoarse

```
SP_DS_Coarse* GetCoarse() const
```

Zugriffsmethode auf m_pcCoarse.

SP_Data::GetGraphicInSubnet

```
virtual SP_Graphic* GetGraphicInSubnet(unsigned int p_nNetnumber)
```

Liefert die erste graphische Ausprägung im angegebenen Netz oder NULL, wenn es keinen entsprechenden Eintrag gibt.

SP_Data::GetGraphics

```
SP_List<SP_Graphic*>* GetGraphics()
```

Liefert einen Zeiger auf die Liste der graphischen Ausprägungen.

SP_Data::GetLogical

```
bool GetLogical() const
```

Zugriffsmethode auf m_bLogical.

SP_Data::RemoveAllGraphics

```
bool RemoveAllGraphics(bool p_bDelete = FALSE)
```

Entfernt alle Einträge aus der Liste der graphischen Ausprägungen.

Parameter

p_bDelete

Gibt den Speicher durch den Aufruf des Destruktors frei, wenn TRUE übergeben wird. Sonst wird nur die Liste geleert.

SP_Data::RemoveGraphic

```
virtual SP_DELETE_STATE RemoveGraphic(unsigned int p_nNetnumber)
virtual SP_DELETE_STATE RemoveGraphic(SP_List<SP_Graphic*>* p_plGraphic,
                                      bool p_bDelete = TRUE)
virtual bool RemoveGraphic(SP_Graphic* p_pcGraphic, bool p_bDelete = TRUE)
```

Löscht graphische Ausprägungen.

Parameter

p_nNetnumber

Netznummer, deren Entsprechungen entfernt werden sollen.

p_plGraphic

Liste von graphischen Ausprägungen, die entfernt werden sollen.

p_pcGraphic

Graphische Ausprägung, die entfernt werden soll.

p_bDelete

TRUE, wenn die zu entfernenden Objekte auch zerstört werden sollen.

Rückgabe

SP_DELETE_STATE ist entweder SP_NO_MORE_OBJECTS, wenn die Liste der graphischen Ausprägungen anschliessend keine Objekte mehr enthält, SP_MORE_OBJECTS wenn nach dem Löschen noch Einträge vorhanden sind oder SP_DELETE_ERROR, falls ein Fehler aufgetreten ist.

Die dritte Implementierung liefert TRUE im Erfolgsfall und FALSE sonst.

SP_Data::SetCoarse

```
SP_DS_Coarse* SetCoarse(SP_DS_Coarse* p_pcVal)
```

Assoziiert den Parameter als Wert für m_pcCoarse. Im Debug-Modus wird ausserdem eine Meldung generiert, wenn es schon einen Eintrag für m_pc-Coarse geben sollte.

SP_Data::SetLogical

```
bool SetLogical(bool p_bVal)
```

Setzt den Wert von `m_bLogical` auf den des Parameters und liefert immer `TRUE` zurück.

SP_Data::SetNetnumber

```
bool SetNetnumber(unsigned int p_nNewVal, unsigned int p_nOldVal = 0)
```

Reimplementierung aus `SP_NetElement`, die `SetNetnumber` mit denselben Parametern für alle graphischen Ausprägungen, sowie, soweit vorhanden, für `m_pcCoarse`, aufruft und anschliessend die Methode mit dem Aufruf der Basisimplementierung verlässt.

Parameter

siehe Abschnitt „SP_NetElement“ auf Seite 115.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

2 SP_Elementclass

Das ist die Basis für SP_DS_Nodeclass und SP_DS_Edgeclass und implementiert unter anderem die Verwaltung der Beziehung zu einem Graphobjekt, sowie die Vorlage für Attribute, die den Elementen der Partition eine fortlaufende Nummerierung erlauben (siehe auch SP_DS_IdAttribute).

Abgeleitet von

SP_Error
SP_Type
SP_Name

Include

core/base/SP_ElementClass.h

SP_Elementclass::m_pcGraph

```
SP_DS_Graph* m_pcGraph
```

Graph, als dessen Teil diese Partition definiert wurde.

SP_Elementclass::m_nIdCounter

```
long m_nIdCounter
```

Interner Zähler für die Elemente dieser Partition. Wird in Verbindung mit GetNewIdCount und SP_DS_IdAttribute eingesetzt.

SP_Elementclass::SP_Elementclass

```
SP_ElementClass(SP_ELEMENT_TYPE p_eType, const char* p_pchName)
```

Konstruktor.

Parameter

p_eType

SP_ELEMENT_NODECLASS oder SP_ELEMENT_EDGECLASS

p_pchName

Name der Partition

SP_Elementclass::HasAttributeType

```
virtual bool HasAttributeType(SP_ATTRIBUTE_TYPE p_eVal)
```

Virtuelle Methode, die von den abgeleiteten Klassen implementiert wird.

Parameter

p_eVal

Eintrag aus dem Aufzählungstypen aus SP_DS_Attribute.h

SP_Elementclass::GetParentGraph

```
SP_DS_Graph* GetParentGraph() const
```

Liefert den Graphen als dessen Teil diese Partition definiert wurde.

SP_Elementclass::SetParentGraph

```
SP_DS_Graph* SetParentGraph(SP_DS_Graph* p_pcVal)
```

Setzt die Variable m_pcGraph auf den Wert des Parameters und liefert diesen zurück.

SP_Elementclass::GetNewIdCount

```
long GetNewIdCount()
```

Erhöht den Wert von m_nIdCounter und liefert diesen neuen Wert zurück.

SP_Elementclass::GetIdCount

```
long GetIdCount() const
```

Liefert den Wert von m_nIdCounter.

SP_Elementclass::TestIdCount

```
bool TestIdCount(long p_nVal)
```

Setzt, falls p_nVal > m_nIdCounter den internen Zähler auf p_nVal. Wird beim Laden von Graphen und beim Kopieren/Einfügen verwendet.

SP_Elementclass::AddAnimator

```
bool AddAnimator(SP_DS_Animator* p_pcAnimator)
```

Fügt einen Animator für die Elemente dieser Partition in die Animator-Registry ein.

Parameter

p_pcAnimator

Das Animatorobjekt.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

3 SP_Error

Klasse zur Kapselung der Verwaltung von Fehlerinformationen und damit Basis für viele Klassen der Software. Die Idee ist, dass Objekte über ihre Ableitung von `SP_Error` Statusinformationen während ihrer Ausführung schreiben können, die von den aufrufenden Klassen im Falle einer Fehleranzeige durch, zum Beispiel, den Rückgabewert einer Funktion, ausgelesen und zur Anzeige gebracht werden können.

Include

`core/base/SP_Error.h`

SP_Error::m_bError

```
bool m_bError
```

Boolsche Variable, in der vermerkt wird, ob ein Fehler aufgetreten ist.

SP_Error::m_sMessage

```
wxString m_sMessage
```

Menschen lesbare Fehlermeldung, die auch zur Anzeige in Dialogen verwendet wird.

SP_Error::SP_Error

```
SP_Error()
```

Konstruktor. Initialisiert die beiden Attribute `m_bError` und `m_sMessage`

SP_Error::GetErrorSet

```
const bool GetErrorSet() const
```

Liefert den Wert der Variablen `m_bError`.

SP_Error::GetLastError

```
const wxString& GetLastError() const
```

Liefert den Wert der Variablen `m_sMessage`.

SP_Error::SetLastError

```
void SetLastError(const wxString& p_sMsg)
```

Als `protected` definierte Methode, die in Ableitungen statt des direkten Zugriffs auf die Variablen verwendet werden soll um später einfach, eventuelle Funktionalitätserweiterungen hinzufügen zu können.

4 SP_Graphic

Basisklasse für alle graphischen Ausprägungen. Diese sind den Elementen der Datenstruktur in der globalen Map von Listen

```
SP_Map<SP_Data*, SP_List<SP_Graphic*> > g_mlData2Graphic
```

zugeordnet, deren Zugriff in SP_Data transparent gemacht wird. Neben dieser Beziehung zwischen Datenstruktur und Graphik existiert eine erweiterte Beziehung zwischen graphischen Objekten. Genau wie Datenstruktur-Attribute zum Beispiel einem Datenstruktur-Knoten als Kinder zugeordnet sind, wird eine graphische Ausprägung des Attributs auch einer graphischen Ausprägung des Knotens als Kind zugeordnet. Die Speicherung dieser Beziehung erfolgt in

```
SP_Map<SP_Graphic*, SP_List<SP_Graphic*> > g_mlGraphicChildren
```

wobei der Key in der Map die graphische Ausprägung des Knotens ist (um bei dem Beispiel zu bleiben) und die Liste die graphischen Ausprägungen der Attribute enthält. Ausserdem wird in dem Klassenattribut SP_Graphic::m_pcGraphicParent der Attributgraphiken die Graphik des entsprechende Knotens als Elternelement referenziert.

Abgeleitet von

SP_IdCounter
SP_NetElement

Include

core/base/SP_Graphic.h

Typen

```
enum SP_GRAPHIC_STATE
{
    SP_STATE_NORMAL = 1,
    SP_STATE_MASTER = 2,
    SP_STATE_COARSEBORDERTOP = 4,
    SP_STATE_COARSEBORDERDOWN = 8,
    SP_STATE_COARSECOARSE = 16,
};
```

NORMAL steht für alle einfachen Objekte ohne besondere Bedeutung. MASTER wird in der vorliegenden Version nicht verwendet. COARSEBORDERTOP steht für graphische Objekte, die als Randknoten eine Kante zu einer graphischen Ausprägung besitzen (oder selbst eine solche Kante sind), dessen Datenstrukturelement ein Objekt in m_pcCoarse enthält. Zu diesen existiert immer eine weitere Graphik des States COARSEBORDERDOWN im entspre-

chenden Unternetz. COARSECOARSE steht für die graphischen Ausprägungen von Datenstrukturelementen, die in ihrem m_pcCoarse Attribut ein Objekt enthalten.

```
enum SP_GRAPHIC_TYPE
{
    SP_GRAPHIC_NODE,
    SP_GRAPHIC_EDGE,
    SP_GRAPHIC_ATTRIBUTE,
    SP_GRAPHIC_ANIMATOR,
};
```

SP_Graphic::m_bShow

```
bool m_bShow
```

Flag, über welches die Anzeige dieser Graphik an- oder abgeschaltet werden kann. Diese Variable ist eine Erweiterung zu dem in der OGL und seinen Shapes sowieso schon vorhandenen Anzeige-Flag für wxShape.

SP_Graphic::m_eGraphicState

```
SP_GRAPHIC_STATE m_eGraphicState
```

Zustandsbeschreibung, die im Zusammenhang mit der Beziehung zu Unternetzen eine Rolle spielt.

SP_Graphic::m_eGraphicType

```
SP_GRAPHIC_TYPE m_eGraphicType
```

Identifikator der angibt, ob die Ableitungen vom Typ SP_GR_Node, SP_GR_Edge, SP_GR_Attribute oder SP_GR_Animator sind.

SP_Graphic::m_pcGraphicParent

```
SP_Graphic* m_pcGraphicParent
```

Die Graphik eines Attributs ist als Eintrag in der Liste der graphischen Kinder des entsprechenden Knotens oder der Kante zugeordnet, um schneller gemeinsame Auswahl entscheiden zu können. In diesem Fall ist in m_pcGraphicParent dieses Attributs die Graphik des Knotens referenziert. Für Objekte, die nicht in einer Kind-Beziehung zu einem Elternelement stehen, ist der Wert dieses Attributs das Objekt selbst (also this).

SP_Graphic::SP_Graphic

```
SP_Graphic(SP_GRAPHIC_TYPE p_eType)
```

Konstruktor, initialisiert den Wert von `m_eGraphicType` mit dem Wert des Parameters `p_eType`.

SP_Graphic::AddGraphicChildren

```
bool AddGraphicChildren(SP_Graphic* p_pcChild)
```

Fügt der Liste der mit dieser Graphik in Beziehung stehenden Graphiken einen neuen Eintrag hinzu, wenn dieser noch nicht existiert.

Diese Eltern-Kind-Beziehung bezieht sich auf das Verhältnis von graphischen Ausprägungen von Knoten oder Kanten zu den graphischen Ausführungen von den, den dazugehörigen Datenstrukturelementen zugeordneten Graphiken.

Parameter

p_pcChild

Die Graphik, die ein Kind dieses Objekts sein soll.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_Graphic::AddToCanvas

```
virtual bool AddToCanvas(SP_GUI_Canvas* p_pcCanvas,
                         double p_nX = -1.0, double p_nY = -1.0,
                         int p_nKeys = 0)
```

Schnittstellendefinition, die in dieser Klasse immer `TRUE` liefert und in ihren Ableitungen zum erstmaligen Anzeigen im Fenster verwendet wird.

Parameter

p_pcCanvas

Fensterinhalt, zu dem dieses Graphik hinzugefügt werden soll.

p_nX

Horizontale Position, an der das Element angezeigt werden soll. Ein Wert von `-1` steht für „nicht verändern“ also den Wert verwenden, der der Graphik schon eigenen ist.

p_nY

Vertikale Position, an der das Element angezeigt werden soll. Ein Wert von `-1` steht für „nicht verändern“ also den Wert verwenden, der der Graphik schon eigenen ist.

p_nKeys

Eventuelle Modifikatoren durch gehaltene Tasten, die beim Klicken in der Oberfläche aktiv waren.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_Graphic::Clone

```
virtual SP_Graphic* Clone(SP_Data* p_pcParent) = 0
```

Rein virtuelle Schnittstellendefinition, die von allen Ableitungen implementiert werden muss und sicher stellt, dass diese in der Lage sind, Kopien von sich selbst zu erzeugen.

Parameter

p_pcParent

Das neue Elternelement in der Datenstruktur, zu dem die zu erzeugende Kopie gehört.

Rückgabe

Das neue Objekt oder `NULL` im Fehlerfall.

SP_Graphic::CloneBase

```
virtual bool CloneBase(SP_Graphic* p_pcCopy)
```

Hilfsmethode zum Aufruf aus `Clone` in den Ableitungen, die die in `SP_Graphic` gespeicherten Attribute, wie `m_bShow`, etc. kopiert.

Parameter

p_pcCopy

Neues Objekt, in dem die Attribute der Basisklasse gesetzt werden sollen.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_Graphic::Flip

```
virtual bool Flip(double p_nX)
```

Positionsveränderung durch Spiegelung an der durch den Parameter p_nX erzeugten, vertikal verlaufenden, Spiegelachse. Diese Methode wird aus der Oberfläche heraus bei Anwahl des entsprechenden Menüpunkts für alle selektierten Objekte aufgerufen.

SP_Graphic::GetCanvas

```
SP_GUI_Canvas* GetCanvas()
```

Zugriffsmethode auf den Fensterteil, in dem das konkrete graphische Element angezeigt wird.

SP_Graphic::GetGraphicChildren

```
SP_List<SP_Graphic*>* GetGraphicChildren()
```

Liefert die Adresse der Liste mit den, zu diesem Element gehörenden, graphischen Kindern. Das sind in der Regel die graphischen Ausprägungen der Attribute eines Knotens oder einer Kante, die jeweils der graphischen Ausprägung des Knotens oder der Kante als Kinder zugeordnet sind.

SP_Graphic::GetGraphicParent

```
virtual SP_Graphic* GetGraphicParent()
```

Zugriffsmethode auf den Wert in `m_pcGraphicParent`.

SP_Graphic::GetGraphicState

```
SP_GRAPHIC_STATE GetGraphicState() const
```

Zugriffsmethode auf den Wert in `m_eGraphicState`.

SP_Graphic::GetGraphicType

```
SP_GRAPHIC_TYPE GetGraphicType() const
```

Zugriffsmethode auf den Wert in `m_eGraphicType`.

SP_Graphic::GetOffsetX

```
virtual double GetOffsetX() { return 0.0; }
```

Schnittstellendefinition, die in den Ableitungen für Attribute den Wert der Verschiebung zum graphischen Elternelement in horizontaler Richtung angibt.

SP_Graphic::GetOffsetY

```
virtual double GetOffsetY() { return 0.0; }
```

Schnittstellendefinition, die in den Ableitungen für Attribute den Wert der Verschiebung zum graphischen Elternelement in vertikaler Richtung angibt.

SP_Graphic::GetParent

```
virtual SP_Data* GetParent()
```

Liefert das Datenstrukturelement zu dem diese Graphik gehört oder `NULL`, wenn es keine solche Assoziation gibt.

SP_Graphic::GetPosX

```
virtual double GetPosX()
```

Position des graphischen Objekts in horizontaler Richtung.

SP_Graphic::GetPosY

```
virtual double GetPosY()
```

Position des graphischen Objekts in vertikaler Richtung

SP_Graphic::GetPrimitive

```
virtual wxShape* GetPrimitive() = 0
```

Schnittstellendefinition, die von allen Ableitungen implementiert werden muss und einen Zeiger auf ein Objekt vom Typ `wxShape`, der Basisklasse aller Objekte in der verwendeten *OGL*, liefert.

SP_Graphic::GetShow

```
bool GetShow() const
```

Zugriffsmethode auf den Wert in `m_bShow`.

SP_Graphic::Mirror

```
virtual bool Mirror(double p_nY)
```

Positionsveränderung durch Spiegelung an der durch den Parameter p_nY erzeugten, horizontal verlaufenden, Spiegelachse. Diese Methode wird aus der Oberfläche heraus bei Anwahl des entsprechenden Menüpunkts für alle selektierten Objekte aufgerufen.

SP_Graphic::RemoveFromCanvas

```
virtual bool RemoveFromCanvas()
```

Entfernen des Objekts aus der Liste der sichtbaren Elemente des Fensters. In dieser Methode wird hauptsächlich die Arbeit geleistet, die von der *OGL* gefordert wird. Das Graphikobjekt, von dem diese Methode aufgerufen wurde ist anschliessend weiterhin ein gültiges Element der SNOOPY-Struktur, kann aber erst dann einer anderen Ansicht zugeordnet werden.

SP_Graphic::Rotate

```
virtual bool Rotate(double p_nX, double p_nY, double p_nAngle)
```

Positionsveränderung, um den durch p_nX und p_nY definierten Punkt und den durch p_nAngle definierten Winkel im Gradmass (0° – 360°). Diese Methode wird aus der Oberfläche heraus bei Anwahl des entsprechenden Menüpunkts für alle selektierten Objekte aufgerufen.

Dabei wird in den momentan implementierten Objekten wirklich nur die Position anhand des Rotationszentrums und des Winkels neu gesetzt. Es erfolgt keine Rotation des Objekts selbst.

SP_Graphic::SetDataParent

```
virtual SP_Data* SetDataParent(SP_Data* p_pcParent)
```

Zugriffsmethode zum Setzen des zu dieser Graphik gehörenden Datenstrukturelements.

SP_Graphic::SetGraphicParent

```
virtual SP_Graphic* SetGraphicParent(SP_Graphic* p_pcVal)
```

Setzen des Wertes in m_pcGraphicParent.

SP_Graphic::SetGraphicState

```
bool SetGraphicState(SP_GRAPHIC_STATE p_eVal)
```

Setzen des Wertes in `m_eGraphicState`. Diese Methode liefert in der vorliegenden Version immer `TRUE`.

SP_Graphic::SetNetnumber

```
virtual bool SetNetnumber(unsigned int p_nNewVal, unsigned int p_nOldVal = 0)
```

Methode zum Setzen der Nummer des Netzes, zu dem diese Graphik gehört. Hier wird für alle Elemente in der Liste der graphischen Kinder ebenfalls `SetNetnumber` und am Ende noch die Basisimplementierung in `SP_NetElement::SetNetnumber`, jeweils mit den übergebenen Parametern, aufgerufen.

SP_Graphic::SetOffsetX

```
virtual bool SetOffsetX(double p_nVal)
```

Schnittstellendefinition, die in den Ableitungen vom Typ `SP_GRAPHIC_ATTRIBUTE` den Wert für die horizontale Verschiebung relativ zum Elternelement setzt.

SP_Graphic::SetOffsetY

```
virtual bool SetOffsetY(double p_nVal)
```

Schnittstellendefinition, die in den Ableitungen vom Typ `SP_GRAPHIC_ATTRIBUTE` den Wert für die vertikale Verschiebung relativ zum Elternelement setzt.

SP_Graphic::SetPosX

```
virtual bool SetPosX(double p_nVal)
```

Zugriffsmethode zum Setzen der horizontalen Position des konkreten graphischen Elements.

SP_Graphic::SetPosY

```
virtual bool SetPosY(double p_nVal)
```

Zugriffsmethode zum Setzen der vertikalen Position des konkreten graphischen Elements.

SP_Graphic::SetShow

```
bool SetShow(bool p_bVal)
```

Zugriffsmethode zum Setzen des Wertes in `m_bShow`.

SP_Graphic::Translate

```
virtual bool Translate(double p_nX, double p_nY)
```

Methode zum relativen Verschieben der Position dieser Graphik um den in `p_nX` und `p_nY` definierten Wert.

5 SP_IdCounter

Basisklasse, die allen abgeleiteten Klassen einen über die gesamte Anwendung hinweg eineindeutigen numerischen Identifikator zuordnet. Dieser Identifikation spielt hauptsächlich beim Laden und Speichern eine wichtige Rolle, um die Beziehungen zwischen den Knoten und Kanten ohne die Kenntnis von konkreten Objekten zu sichern und wiederherstellen zu können.

Dabei wird zur Vergabe des Wertes bei der Erzeugung neuer Instanzen eine anwendungsglobale Variable namens `g_nIdCounter` (Instanziiert in `core/base/SP_Data.cpp`) verwendet, die zur Laufzeit nur incrementiert wird.

Include

`core/base/SP_Type.h`

SP_IdCounter::m_nId

```
unsigned long m_nId
```

Identifikator für das aktuelle Objekt.

SP_IdCounter::SP_IdCounter

```
SP_IdCounter():m_nId(g_nIdCounter++) { }
```

Konstruktor. Initialisiert das einzige Attribut `m_nId` mit dem inkrementierten Wert des globalen Zählers.

SP_IdCounter::GetId

```
const unsigned long GetId() const
```

Zugriffsmethode auf das Attribut `m_nId`.

SP_IdCounter::SetId

```
void SetId(unsigned long p_nVal)
```

Zugriffsmethode, die im Zusammenhang mit dem Laden von Netzen eine Rolle spielt, da in diesem Fall die Original- „Nummerierung“ benötigt wird, um die Verbindungen wiederherstellen zu können, ohne auf ein Mapping der alten zu neuen Werten zurückgreifen zu müssen.

Parameter

p_nVal

Neuer Identifikator dieses Objekts.

Bemerkung

Dabei wird in der Implementierung darauf geachtet, dass der Wert des globalen Zählers `g_nIdCounter` stets grösser ist als jeder Wert, der durch diese Methode in ein Objekt geschrieben wird. Das soll vermeiden, dass nach dem Laden bei nachfolgenden Instanziierungen doppelte Identifikatoren auftreten.

6 SP_Name

Basisklasse, die allen Ableitungen die Fähigkeit zur Verwaltung eines textuellen Identifikators verleiht. Mit dieser Klasse als Basis kann ein Objekt also einen Namen haben.

Include

`core/base/SP_Name.h`

SP_Name::m_sNameVal

```
wxString m_sNameVal
```

Wert des zugewiesenen Namens.

SP_Name::SP_Name

```
SP_Name(const char* p_pchValue):m_sNameVal(p_pchValue) { }
```

Konstruktor. Initialisiert das einzige Attribut `m_sNameVal` mit dem Wert des Parameters.

SP_Name::GetName

```
const char* GetName()
```

Zugriffsmethode, liefert den Inhalt von `m_sNameVal`.

SP_Name::SetName

```
wxString& SetName(const char* p_pchName)
```

Setzt den Wert von `m_sNameVal` auf den des Parameters und liefert eine Referenz auf `m_sNameVal` zurück.

7 SP_NetElement

Diese Klasse ist Basis für `SP_Data` und `SP_Graphic` und kapselt die Eigenschaft eines Objekts in einem Graphen, zu einem bestimmten Netz zu gehören, wie es im Abschnitt „Hierarchien und Mehrdeutigkeiten“ auf Seite 18 dargelegt wurde.

Neben der Speicherung und Verwaltung dieses Netzidentifikators kann man alle abgeleiteten Objekte zudem als „gelöscht“ markieren. Diese Markierung dient beispielsweise dem Zweck, während der zahlreichen Iterationen über die Elemente der verschiedenen Partitionen beim Löschen von Knoten auch das Löschen der Kanten einzuleiten – ohne dies jedoch wirklich durch den Aufruf des Destruktors des dazugehörigen Objekts zu diesem Zeitpunkt auszuführen - unter Umständen wird dieses Objekt während der Iteration nämlich noch benötigt.

Include

`core/base/SP_NetElement.h`

SP_NetElement::m_bDelete

```
bool m_bDelete
```

Attribut zur Speicherung der „zum Löschen“ -Markierung.

SP_NetElement::m_nNetnumber

```
unsigned int m_nNetnumber
```

Numerischer Identifikator, der dieses Objekt als zu einem bestimmten Netz zugehörig auszeichnet.

SP_NetElement::SP_NetElement

```
SP_NetElement(unsigned int p_nVal = 0)
```

Konstruktor, initialisiert `m_nNetnumber` mit dem Wert des Parameters `p_nVal` und `m_bDelete` mit `FALSE`.

SP_NetElement::GetDelete

```
bool GetDelete() const
```

Zugriffsmethode, liefert den Wert von `m_bDelete`.

SP_NetElement::GetNetnumber

```
unsigned int GetNetnumber() const
```

Zugriffsmethode, liefert den Wert von m_nNetnumber.

SP_NetElement::SetDelete

```
bool SetDelete(bool p_bVal = TRUE)
```

Markiert dieses Objekt zum Löschen, also zum Entfernen aus der Menge der Datenstruktur- oder Graphikobjekte zu einem späteren Zeitpunkt, wenn p_bVal gleich TRUE ist.

SP_NetElement::SetNetnumber

```
virtual bool SetNetnumber(unsigned int p_nNewVal, unsigned int p_nOldVal = 0)
```

Setzt den Wert von m_nNetnumber und wird in SP_Data und SP_Graphic überschrieben. Beide Klassen beenden ihre SetNetnumber-Methoden aber immer mit dem Aufruf dieser Basisimplementierung.

Parameter

p_nNewVal

neuer Wert für m_nNetnumber

p_nOldVal

Wenn p_nOldVal ungleich 0 ist, wird nur dann der Wert von p_nNewVal in m_nNetnumber gespeichert, wenn zuvor gilt: p_nOldVal == m_nNetnumber.

Rückgabe

Diese Methode liefert in der vorliegenden Version immer TRUE, unabhängig davon, ob sie den Wert wirklich verändert hat, oder nicht.

Bemerkung

Diese Definition der Zugriffsmethode zum Setzen von m_nNetnumber hat folgenden Grund: Da alle Teile eines Graphen die anfallenden Aufgaben hauptsächlich an ihre untergeordneten Strukturen delegieren, wurde das hier implementierte Vorgehen „weiter oben“ notwendig. So ist es durch diese Definition zum Beispiel einfach möglich, in einer Iteration über die Elemente einer Partition nur denjenigen Einträgen eine neue Netznummer zuzuweisen, die zuvor in einem bekannten Netz einer bestimmten Nummer existierten.

8 SP_Type

Klasse, die als Basis für alle Elemente der Datenstruktur verwendet wird und alle Objekte ihrer Ableitungen anhand eines Eintrags aus einem Aufzählungstypen unterscheidbar, beziehungsweise einer bestimmten Klasse zuordbar macht.

Abgesehen von der Verwaltung des Wertes aus dem Aufzählungstypen hat diese Klasse keine weiteren Methoden. Von ihr existieren Ableitungen, bei denen der direkte Vergleich anhand dieses Typen gewollt ist, bei anderen wiederum sollte sich die Notwendigkeit eines solchen Vergleichs auf ein akademisches Minimum reduzieren lassen. Objekte der Ableitungen über `SP_Data` (`SP_DS_Node`, `SP_DS_Edge`, `SP_DS_Attribute`) zum Beispiel sind sinnvollerweise vergleichbar – es sollte aber nicht zu der Notwendigkeit kommen, ein Objekt von `SP_DS_Graph` mit einem von `SP_DS_Nodeclass` vergleichen zu müssen.

Include

`core/base/SP_Type.h`

Typen

```
enum SP_ELEMENT_TYPE
{
    SP_ELEMENT_NULL = 0,
    SP_ELEMENT_GRAPH = 1,
    SP_ELEMENT_NETCLASS = 2,
    SP_ELEMENT_NODECLASS = 4,
    SP_ELEMENT_EDGECLASS = 8,
    SP_ELEMENT_NODE = 16,
    SP_ELEMENT_EDGE = 32,
    SP_ELEMENT_ATTRIBUTE = 64,
    // special
    SP_ELEMENT_ANIMATOR = 128,
};
```

Bemerkung

Für diese Klasse existiert nur eine lesende Zugriffsmethode, da die Ableitungen nicht zur Laufzeit ihren Typ ändern können. Es kann also nicht die in dieser Klasse gekapselte grundlegende Unterscheidung in, zum Beispiel, Element der Knoten- oder Kantenpartition, plötzlich wechseln.

SP_Type::m_eElementType

```
SP_ELEMENT_TYPE m_eElementType
```

Wert aus dem Aufzählungstypen.

SP_Type::SP_Type

```
SP_Type(SP_ELEMENT_TYPE p_eType):m_eElementType(p_eType) { }
```

Konstruktor. Initialisiert das einzige Attribut `m_eElementType` mit dem Wert des Parameters.

SP_Type::GetElementType

```
SP_ELEMENT_TYPE GetElementType() const
```

Zugriffsmethode auf den Wert von `m_eElementType`.

9 SP_DS_Animation

SP_DS_Animation ist die Basisklasse für individuelle Graphanimationen, welche die zentrale Steuerung (über einen Timer) implementiert und die Schnittstellen festlegt. Eine Animation bedient sich so genannter Animatoren für die Elemente der Datenstruktur (siehe SP_DS_Animator). Diese können durch graphische Objekte (abgeleitet von SP_GR_Animator) um Elemente für die Bewegung ergänzt werden. Zwei Parameter sind für jede Animation entscheidend, zum Einen die Frequenz des Timers (m_nRefreshFrequ) und zum Anderen die Dauer eines Zyklus' (m_nStepDuration). Generell läuft eine Animation pro Zyklus in drei Schritten ab:

PreStep

Pro Zyklus einmalige Auswahl der Animatoren für den nächsten Schritt. Dazu werden aus der Menge aller Animatoren über deren PreStep Methode diejenigen Elemente extrahiert, die im aktuellen Zyklus aktiv sein sollen.

Step

Ausführen der Step-Methoden der, in der vorangegangenen Auswahl bestimmten, Animatoren für die gesamte Dauer des Zyklus. Dabei erfolgt der Aufruf der Step-Methoden genau m_nStepDuration / m_nRefreshFrequ mal.

PostStep

Pro Zyklus einmaliger Aufruf für die Möglichkeit des Setzens von Werten durch die beteiligten Animatoren und das Aufräumen von aggregiertem Speicher.

Abgeleitet von

SP_IdCounter
SP_Error
wxEvtHandler

Include

sp_ds/extensions/SP_DS_Animation.h

SP_DS_Animation::m_cTimer

```
wxTimer m_cTimer
```

Timerobjekt, das im Konstruktor mit der Instanz der Animation verbunden wird. Durch die Vererbung von `SP_IdCounter` wird für jede Animationsinstanz ein Identifikator generiert, der dynamisch, unter Anwendung von `wxEvtHandler::ConnectEvent` und `::DisconnectEvent`) als Event-ID für die Timerbehandlung verwendet wird.

SP_DS_Animation::m_lAllAnimators

```
SP_List<SP_DS_Animator*> m_lAllAnimators
```

Liste aller Animatoren dieser Animation. Diese Liste wird in `Initialise`, ausgehend von der, in der Netzklasse definierten, Menge der Animatoren für alle konkreten Elemente aller Partitionen gefüllt.

SP_DS_Animation::m_lStepAnimators

```
SP_List<SP_DS_Animator*> m_lStepAnimators
```

Aktive Animatoren für den aktuellen Zyklus. Durch Iteration über die Liste aller Animatoren in `PreStep` wird diese Liste gefüllt und nach dem Ende eines Zyklus' wieder geleert. Dabei wird kein neuer Speicher angefordert, es werden nur einige oder alle Elemente aus `m_lAllAnimators` ebenfalls in diese Liste aufgenommen.

SP_DS_Animation::m_mlData2Animator

```
SP_Map<SP_Data*, SP_List<SP_DS_Animator*> > m_mlData2Animator
```

Mapping von Objekten der Datenstruktur auf eine Liste von Animatoren. Dieses Mapping wird in `Initialise` für den aktiven Graphen erzeugt und ermöglicht den Zugriff auf die Animatoren über ein Objekt der Datenstruktur während der gesamten Laufzeit der Animation.

SP_DS_Animation::m_nActStep

```
unsigned int m_nActStep
```

Nummer des aktuellen Schritts, beginnend bei 0 bis zur Maximalzahl, die sich aus dem Quotienten der Dauer eines Zyklus (`m_nStepDuration`) und der Frequenz des Timers (`m_nRefreshFrequ`) ergibt. Das Inkrementieren erfolgt in der Timerbehandlungsmethode für jeden Tick und wird in `PostStep` wieder zurück gesetzt.

SP_DS_Animation::m_nRefreshFrequ

```
unsigned int m_nRefreshFrequ
```

Frequenz in Millisekunden, die angibt, aller wie viel Zeiteinheiten die Timerbehandlungsmethode angesprochen werden soll. Üblicherweise in der Grössenordnung von 100 Millisekunden.

SP_DS_Animation::m_nStepDuration

```
unsigned int m_nStepDuration
```

Wert in Millisekunden, der angibt, wie lange ein Animationszyklus dauern soll. Üblicherweise in der Grössenordnung von mehreren 1000 Millisekunden.

SP_DS_Animation::m_nSteps

```
unsigned int m_nSteps
```

Maximalzahl der Schritte pro Animationszyklus, die sich aus dem Quotienten der Dauer eines Zyklus (`m_nStepDuration`) und der Frequenz des Timers (`m_nRefreshFrequ`) ergibt.

SP_DS_Animation::m_tsCanvas

```
SP_Set<SP_GUI_Canvas*> m_tsCanvas
```

Liste der sichtbaren Fenster in der Oberfläche, die für jeden ausgeführten Schritt eines Zyklus neu gezeichnet werden müssen. Diese Menge wird in der `PreStep`-Phase von jedem Animator, der im folgenden Zyklus aktiv sein wird, mit dem Zeiger auf sein aktuelles Fenster gefüllt. Die Verwendung eines Sets (von `std::set`) stellt sicher, dass alle Einträge einmalig sind, ohne beim Hinzufügen auf Existenz prüfen zu müssen. Diese Menge ist immer nur für einen Zyklus aktuell und wird in `PostStep` wieder geleert.

SP_DS_Animation::SP_DS_Animation

```
SP_DS_Animation(unsigned int p_nRefresh, unsigned int p_nDuration)
```

Konstruktor, erzeugt ein neues Animationsobjekt.

Parameter

p_nRefresh

Timerfrequenz in Millisekunden. Üblicherweise in der Grösse von 100 ms.

p_nDuration

Schrittdauer in Millisekunden, also die Zeit, für die in einem Animationszyklus aller `p_nRefresh` Millisekunden die `Step`-Methoden der Animatoren ausgeführt werden.

SP_DS_Animation::AddAnimator

```
bool AddAnimator(SP_Data* p_pcParent, SP_DS_Animator* p_pcAnimator)
```

Fügt einen Animator für ein Element der Datenstruktur hinzu. Die Datenhaltung erfolgt in einer Map von `SP_Data`-Objekten zu einer Liste von `SP_DS_Animator`-Objekten und wird automatisch beim Aufruf des Konstruktors von `SP_DS_Animator` organisiert. So kann man zu einem späteren Zeitpunkt zu einem beliebigen Element der Datenstruktur den Animator erfragen.

Parameter

p_pcParent

Datenstrukturelement.

p_pcAnimator

Animator-Objekt.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_DS_Animation::AddCanvas

```
void AddCanvas(SP_GUI_Canvas* p_pcCanvas)
```

Methode zum Füllen von `m_tsCanvas`, der Menge der Fenster, deren Inhalte für jeden Schritt neu gezeichnet werden müssen. Der Aufruf sollte aus der `PreStep`-Methode der aktivierten Animatoren erfolgen, wenn sie ihre graphischen Ausprägungen erzeugen.

SP_DS_Animation::AddToControl

```
virtual bool AddToControl(SP_DLG_Animation* p_pcCtrl, wxSizer* p_pcSizer)
```

Methode zum Hinzufügen individueller Fensterelemente zum Steuerungspanel der Animation. Kann in Ableitungen überschrieben werden und wird automatisch aufgerufen. Dieses Steuerungspanel enthält, bei Aufruf der Basisimplementierung immer einen Start/Stopp-Button und einen Button zum Aufruf eines Optionsdialogs.

Parameter

p_pcCtrl

Kontrolldialog-Objekt. Das ist das Panel, welches im aktivierten Animations-Modus sichtbar ist und mindestens die beiden Buttons „Play/Stop“ und „Options“ enthält.

p_pcSizer

Vertikales Sizer-Objekt des Dialogs, dem die selbst definierten Fensterelemente hinzugefügt werden müssen (siehe `wxSizer`).

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_DS_Animation::AddToDialog

```
virtual bool AddToDialog(SP_DLG_AnimationProperties* p_pcDlg,
                         wxSizer* p_pcSizer)
```

Methode zum Hinzufügen individueller Einstellungselemente im Optionsdialog der Animation. Kann in Ableitungen überschrieben werden und wird automatisch aufgerufen. Der Optionsdialog enthält, bei Aufruf dieser Basisimplementierung in den Ableitungen, mindestens zwei Eingabefelder für die Timerfrequenz und die Dauer eines Zyklus'.

Parameter

p_pcDlg

Dialogobjekt, zum Verwenden als Parent in den Fensterelementen (von `wxDialog` abgeleitete Klasse).

p_pcSizer

Vertikales Sizer-Objekt des Dialogs, dem die selbst definierten Fensterelemente hinzugefügt werden müssen (siehe `wxSizer`).

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_DS_Animation::Clone

```
virtual SP_DS_Animation* Clone()
```

Methode zum Erzeugen einer exakten Kopie eines Animationsobjekts. Muss von jeder Ableitung implementiert werden und stellt sicher, dass aus der, für eine Netzklasse registrierten, Animation immer eine Option für die konkreten Graphen dieser Netzklasse erzeugt werden kann.

Rückgabe

Das neue Animationsobjekt.

SP_DS_Animation::GetAnimator

```
SP_DS_Animator* GetAnimator(SP_Data* p_pcParent, SP_DS_ANIMATOR_TYPE p_eType)
```

Liefert den ersten gefunden Animator des Datenstrukturobjekts `p_pcParent` der dem gesuchten Typ in `p_eType` entspricht.

Parameter

p_pcParent

Datenstrukturobjekt, dessen Animatoren durchsucht werden sollen.

p_eType

Eintrag des Aufzählungstypen aus `sp_ds/SP_DS_Animator.h` der explizit für eigene Erweiterungen vorgesehen ist.

SP_DS_Animation::GetAnimators

```
SP_List<SP_DS_Animator*>* GetAnimators(SP_Data* p_pcParent)
```

Liefert die Liste der Animatoren zu einem Datenstrukturobjekt.

Parameter

p_pcParent

Datenstrukturelement, dessen Animatoren gesucht werden.

Rückgabe

Adresse der Liste der Animatoren oder `NULL`, wenn keine existieren.

SP_DS_Animation::GetRefreshFrequ

```
unsigned int GetRefreshFrequ() const
```

Liefert die Anzahl der Millisekunden für den Timer, also den Wert in `m_nRefreshFrequ`.

SP_DS_Animation::GetRunning

```
bool GetRunning() const
```

Liefert `TRUE` wenn die Animation läuft, `FALSE` sonst.

SP_DS_Animation::GetStepDuration

```
unsigned int GetStepDuration() const
```

Liefert die Anzahl der Millisekunden, die ein Zyklus dauert, also den Wert in `m_nStepDuration`.

SP_DS_Animation::Initialise

```
virtual bool Initialise(SP_DS_Graph* p_pcGraph)
```

Instanziierung der Gesamtmenge aller Animatoren nach den Vorgaben aus der Animator-Registry, wie sie im Entwurf der Netzklasse festgelegt wurden. In dieser Methode werden die Liste `m_lAllAnimators` und die Map `m_mlData2Animator` gefüllt.

Parameter

p_pcGraph

Objekt des aktuellen Graphen, zu dessen Bestandteilen die Animatoren festgestellt werden sollen.

Rückgabe

`TRUE` im Erfolgsfall, `FALSE` sonst.

SP_DS_Animation::OnDlgOk

```
virtual bool OnDialogOk()
```

Automatisch aufgerufene Methode, sobald der Optionsdialog mit „OK“ bestätigt wurde. Kann in Ableitungen überschrieben werden, diese sollten aber immer die Implementierung der Basisklasse aufrufen!

Für den Fall, dass diese Methode FALSE liefert, wird der Dialog nicht geschlossen und der per `SetLastError` unter Umständen gesetzte Fehlertext wird dem Nutzer angezeigt.

SP_DS_Animation::PostStep

```
virtual bool PostStep()
```

Methode, die einmalig am Ende jedes Animationszyklus' aufgerufen wird und die aktivierten Animatoren zurücksetzt. Der Aufruf erfolgt automatisch aus der Timerbehandlungsroutine für den ersten Schritt.

Liefert `TRUE`, wenn alle aktivierten Animatoren in ihrer `PostStep`-Methode ebenfalls `TRUE` liefern, sonst `FALSE`.

SP_DS_Animation::PreStep

```
virtual bool PreStep()
```

Methode, die einmalig pro Animationszyklus aufgerufen wird und die aktiven Animatoren bestimmt. Der Aufruf erfolgt automatisch aus der Timerbehandlungsroutine für den ersten Schritt.

Liefert TRUE, wenn es mindestens einen Animator gibt, der im momentanen Zyklus aktiv ist, FALSE sonst.

SP_DS_Animation::RestartTimer

```
virtual bool RestartTimer()
```

Liefert die boolesche AND Verknüpfung der Aufrufe von StopTimer und StartTimer.

SP_DS_Animation::SetRefreshFrequ

```
void SetRefreshFrequ(unsigned int p_nVal)
```

Setzt die Anzahl der Millisekunden für den Timer, also den Wert in m_nRefreshFrequ.

Parameter

p_nVal

Wert in Millisekunden, üblicherweise in der Grössenordnung von 100.

SP_DS_Animation::SetRunning

```
void SetRunning(bool p_bVal)
```

Setzt den internen Status auf den Wert des Parameters.

Parameter

p_bVal

TRUE oder FALSE

SP_DS_Animation::SetStepDuration

```
void SetStepDuration(unsigned int p_nVal)
```

Setzt den Wert für die Dauer eines Animationszyklus', also den Wert in m_nStepDuration.

Parameter

p_nVal

Wert in Millisekunden, üblicherweise in der Grössenordnung von 1000.

SP_DS_Animation::StartTimer

```
virtual bool StartTimer()
```

Startet den Timer und liefert `TRUE`, wenn der Timer auch wirklich läuft. Es besteht eigentlich keine Notwendigkeit, diese Methode zu überschreiben.

SP_DS_Animation::Step

```
virtual bool Step()
```

Methode, die für jeden Tick des Timers für die festgelegte Anzahl von Millisekunden aufgerufen wird. Der Aufruf erfolgt automatisch aus der Timerbehandlungsroutine.

Liefert `TRUE`, wenn alle aktiven Animatoren für ihre Step-Methode `TRUE` zurück liefern.

SP_DS_Animation::StopTimer

```
virtual bool StopTimer()
```

Stoppt den Timer und ruft ausserdem die `PostStep`-Methode der Animation auf. Es besteht eigentlich keine Notwendigkeit, diese Methode zu überschreiben.

10 SP_DS_Animator

Animatorklasse, die als Vorlage für alle Animatoren einer bestimmten Partition dieser zur Definitionszeit zugeordnet wird und aus der im Animationsmodus Kopien für alle konkreten Datenstrukturelemente dieser Partition generiert werden.

Diese Klasse definiert eine abstrakte Schnittstelle und es können keine Objekte von ihr instanziiert werden. Ableitungen müssen mindestens die Methoden `PreStep`, `Step`, `PostStep` und `Clone` implementieren!

Abgeleitet von

`SP_Animator`
`SP_Data`

Include

`sp_ds/SP_DS_Animator.h`

Typen

```
enum SP_DS_ANIMATOR_TYPE
{
    SP_DS_ANIMATOR_NULL,
    SP_DS_ANIMATOR_PLACE,
    SP_DS_ANIMATOR_TRANS,
};
```

Diese Aufzählung kann für jeden individuell implementierten Animator um Einträge erweitert werden und dient dazu, die Animatoren auch als Objekt vom Typ dieser Basisklasse unterscheiden zu können.

SP_DS_Animator::m_eAnimatorType

```
SP_DS_ANIMATOR_TYPE m_eAnimatorType
```

Wert des Aufzählungstypen der explizit für eigene Erweiterungen vorgesehen ist.

SP_DS_Animator::m_pcParent

```
SP_Data* m_pcParent
```

Datenstrukturobjekt, für das dieser Animator zuständig ist.

SP_DS_Animator::m_pcAnimation

```
SP_DS_Animation* m_pcAnimation
```

Animation in deren Instanz dieser Animator definiert ist.

SP_DS_Animator::m_bEnabled

```
bool m_bEnabled
```

Statusvariable, die anzeigt, ob der Animator „aktiviert“ ist.

SP_DS_Animator::m_nSteps

```
unsigned int m_nSteps
```

Maximale Anzahl der Schritte, für die die `Step`-Methode dieses Animators aufgerufen werden wird. Diese Angabe dient der Feststellung der Möglichkeiten, die dieser Animator im aktuellen Zyklus haben wird, auch wirklich Bewegungen auszuführen und wird jedem Animator, aus der `PreStep`-Methode der Animation heraus, mitgeteilt.

SP_DS_Animator::SP_DS_Animator

```
SP_DS_Animator::SP_DS_Animator(SP_Data* p_pcParent, SP_DS_Animation* p_pcAnim)
```

Konstruktor, erzeugt ein neues Animatorobjekt und fügt es über `SP_DS_Animation::AddAnimator` in die Map der Animation ein.

Parameter

p_pcParent

Datenstrukturobjekt dieses Animators.

p_pcAnim

Animation, in deren Instanz dieser Animator definiert wurde.

SP_DS_Animator::Clone

```
virtual SP_DS_Animator* Clone(SP_Data* p_pcParent, SP_DS_Animation* p_pcAnim) = 0
```

Rein virtuelle Methode, muss von allen Ableitungen implementiert werden.

Parameter

p_pcParent

Neues Datenstrutkurobjekt als Elternelement der Kopie.

p_pcAnim

Animation, in deren Instanz der neue Animator definiert wird.

SP_DS_Animator::SetAnimatorType

```
void SetAnimatorType(SP_DS_ANIMATOR_TYPE p_eVal)
```

Setzt den Typen des Animators um die Ableitungen dieser Klasse identifizieren zu können.

Parameter

p_eVal

Eintrag des Aufzählungstypen der explizit für eigene Erweiterungen vorgesehen ist.

SP_DS_Animator::GetAnimatorType

```
SP_DS_ANIMATOR_TYPE GetAnimatorType() const
```

Zugriffsmethode auf den Typen des Animatorobjekts, also den Wert in `m_eAnimatorType`.

SP_DS_Animator::PreStep

```
virtual bool PreStep(unsigned int p_nSteps) = 0;
```

Rein virtuelle Methode, muss von allen Ableitungen implementiert werden.

Parameter

p_nSteps

Anzahl der Aufrufe der `Step`-Methode die dieser Animator, für den Fall, dass er aktiv wird, durchlaufen kann. Entspricht in der Regel der maximalen Anzahl der Schritte, die eine Animation pro Zyklus durchläuft und ergibt sich aus dem Quotienten von Dauer der Animation und Timerfrequenz. Sollte in `m_nSteps` gespeichert werden.

SP_DS_Animator::Step

```
virtual bool Step(unsigned int p_nStep) = 0;
```

Rein virtuelle Methode, muss von allen Ableitungen implementiert werden.

Parameter

p_nStep

Nummer des aktuellen Schritts. Sollte zwischen 0 und dem Wert der maximalen Anzahl Schritte für diesen Animator, also `m_nSteps`, liegen und wird aus der `Step`-Methode der Animation für alle aktiven Animatoren aufgerufen.

SP_DS_Animator::PostStep

```
virtual bool PostStep() = 0;
```

Rein virtuelle Methode, muss von allen Ableitungen implementiert werden und dient dazu, angeforderten Speicher wieder frei zu geben und die internen Statusvariablen zurück zu setzen.

SP_DS_Animator::SetEnabled

```
void SetEnabled(bool p_bVal)
```

Setzt den „aktiviert“-Status des Animatorobjekts, also `m_bEnabled`, auf den Wert des Parameters.

SP_DS_Animator::GetEnabled

```
bool GetEnabled() const
```

Liefert den Wert der internen Variable, also `m_bEnabled`, für den Status ob dieser Animator im aktuellen Schritt „aktiviert“ ist.

11 SP_DS_Attribute

Basisklasse für alle Datenstrukturattribute. Diese Klasse definiert die erweiterten Schnittstellen und es können von ihr keine Objekte instanziiert werden.

Abgeleitet von

SP_Error
SP_Data
SP_Name

Include

sp_ds/SP_DS_Attribute.h

Typen

```
enum SP_ATTRIBUTE_TYPE
{
    SP_ATTRIBUTE_NULL,
    SP_ATTRIBUTE_TEXT,
    SP_ATTRIBUTE_BOOL,
    SP_ATTRIBUTE_LOGIC,
    // pointless, signed number
    SP_ATTRIBUTE_NUMBER,
    // pointless, signed number to be used as unique ID
    SP_ATTRIBUTE_ID,
};
```

Aufzählung der vorhandenen Attribute in der Datenstruktur, die explizit zur individuellen Erweiterung vorgesehen ist.

SP_DS_Attribute::m_bShowInGlobalDialog

```
bool m_bShowInGlobalDialog
```

Flag, das angibt, ob die Sichtbarkeit der graphischen Ausprägungen dieses Attributs über einen globalen Dialog steuerbar ist.

SP_DS_Attribute::m_eAttributeType

```
SP_ATTRIBUTE_TYPE m_eAttributeType
```

Eintrag aus dem Aufzählunstypen, der das Objekt als Instanz einer ganz bestimmten Ableitung identifiziert.

SP_DS_Attribute::m_pcParent

```
SP_Data* m_pcParent
```

Datenstrukturelement, mit dem dieses Attribut assoziiert ist.

SP_DS_Attribute::SP_DS_Attribute

```
SP_DS_Attribute(const char* p_pchName,
                SP_ATTRIBUTE_TYPE p_eType = SP_ATTRIBUTE_NULL)
```

Konstruktor, der von allen Ableitungen, hauptsächlich wegen der Initialisierung des Typen, aufgerufen werden muss.

Parameter

p_pchName

Bezeichner dieses Attributs

p_eType

Eintrag aus dem Aufzählungstypen, der für eigene Erweiterungen vorgesehen ist.

SP_DS_Attribute::Clone

```
virtual SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE) = 0
```

Rein virtuelle Methode, die von allen abgeleiteten Klassen implementiert werden muss. Das Kopieren der graphischen Ausprägungen kann in jedem Fall der Implementierung der Basisklasse in `CloneBase` überlassen werden.

Parameter

p_bCloneGr

Flag, das angibt ob die graphischen Ausprägungen auch kopiert werden sollen.

SP_DS_Attribute::CloneBase

```
SP_DS_Attribute* CloneBase(SP_DS_Attribute* p_pcAttr, bool p_bCloneGr = TRUE)
```

Implementierung, über welche alle Ableitungen ihren Kopien die Attribute und Werte der Basisklasse mitgeben können, ohne diese selbst setzen zu müssen. Wird hauptsächlich für das Kopieren der graphischen Ausprägungen verwendet.

Parameter

p_pcAttr

Das in der Ableitung neu angelegte Attribut, welches die Eigenschaften der Basisklasse erhalten soll.

p_bCloneGr

Flag ob die graphischen Ausprägungen kopiert werden sollen. Sollte dem Wert des Parameters im Aufruf von `Clone` entsprechen.

SP_DS_Attribute::GetDataParent

```
SP_Data* GetParent() const
```

Zugriffsmethode auf das Element der Datenstruktur, also m_pcParent, zu dem dieses Attribut gehört. Sollte eine Instanz von `SP_DS_Node` oder `SP_DS_Edge` sein.

Rückgabe

Das Datenstrukturobjekt oder `NULL`, wenn noch keine solche Assoziation existiert.

SP_DS_Attribute::GetGlobalShow

```
bool GetGlobalShow() const
```

Liefert den Status von `m_bShowInGlobalDialog`.

SP_DS_Attribute::GetValueString

```
virtual const char* GetValueString() = 0
```

Rein virtuelle Methode, die von allen Ableitungen implementiert werden muss und den von Ableitung zu Ableitung verschiedenen eigentlichen Typen des Attributs in einen String serialisiert.

Rückgabe

Eine Zeichenkette, die dem Wert des Attributs entspricht und zwar in der Form, dass diese Zeichenkette zumindest von einem Attribut desselben Typs wieder deserialisiert werden kann.

Bemerkung

Der in dieser Methode erzeugte (lokale) Wert muss durch den Einsatz von `strdup` vor der Rückgabe dupliziert werden. Alle bislang implementierten Methoden, die `GetValueString` eines Attributs aufrufen, geben den erhaltenen Speicher auch immer durch den Aufruf von `free` wieder frei!

SP_DS_Attribute::GetAttributeType

```
SP_ATTRIBUTE_TYPE GetAttributeType() const
```

Zugriffsmethode auf den Typen des Attributs.

SP_DS_Attribute::Remove

```
bool Remove()
```

Entfernt das Attribut aus der Liste der Attribute des Elternelements. Dabei wird kein Speicher frei gegeben!

Rückgabe

Diese Methode liefert im Moment immer `TRUE`, egal ob dem Attribut ein Elternelement zugeordnet wurde oder ob an diesem überhaupt das Attribut registriert ist. In jedem Fall ist nach dem Aufruf dieser Methode das aktuelle Objekt nicht mehr Attribut eines Elternelements, weshalb die Aktion als erfolgreich angesehen werden kann. Ausserdem wird der Wert von `m_pcParent` auf `NULL` gesetzt.

SP_DS_Attribute::SetAttributeType

```
bool SetAttributeType(SP_ATTRIBUTE_TYPE p_eType)
```

Setzen des Attributtypen in `m_eAttributeType`. Liefert in der aktuellen Version immer `TRUE`.

SP_DS_Attribute::SetDataParent

```
bool SetDataParent(SP_Data* p_pcParent)
```

Setzen des Elements aus der Datenstruktur in `m_pcParent`, zu dem dieses Attribut gehört. Also in der Regel eine Instanz von `SP_DS_Node` oder `SP_DS_Edge`. Liefert in der aktuellen Version immer `TRUE`.

Parameter

p_pcParent

Das Elternelement in der Datenstruktur.

SP_DS_Attribute::SetGlobalShow

```
bool SetGlobalShow(bool p_bVal = TRUE)
```

Setzt den Wert von `m_bShowInGlobalDialog` auf den Wert des Parameters.

Parameter

p_bVal

TRUE wenn die Sichtbarkeit des Attributs aus einem globalen Dialog heraus an- und abschaltbar sein soll.

Rückgabe

TRUE im Erfolgsfall, FALSE sonst.

Bemerkung

Neben dem Setzen der Variable m_bShowInGlobalDialog wird ausserdem für den Fall, dass der Parameter TRUE ist, im Graph-Objekt, von dessen Partition dieses Attribut einem Element zugeordnet wurde, das Flag gesetzt, dass der Graph einen Dialog zur Steuerung dieser globalen Show-Flags in der Oberfläche zur Verfügung stellen soll.

Diese Methode sollte nur zur Definitionszeit verwendet werden.

SP_DS_Attribute::SetValueString

```
virtual bool SetValueString(const char* p_pchVal) = 0
```

Rein virtuelle Methode, die von allen Ableitungen implementiert werden muss und dazu dient, Zeichenketten, die durch den Aufruf von GetValue-String erzeugt wurden wieder zu deserialisieren.

Parameter

p_pchVal

Wert des Attributs, wie er durch GetValueString erzeugt wurde.

Rückgabe

TRUE im Erfolgsfall, FALSE sonst.

12 SP_DS_BoolAttribute

Datenstrukturattribut, das einen boolschen Wahrheitswert repräsentiert, also die Werte Wahr (TRUE) oder Falsch (FALSE) annehmen kann.

Abgeleitet von

SP_DS_Attribute

Include

sp_ds/attributes/SP_DS_BoolAttribute.h

SP_DS_BoolAttribute::m_bValue

```
bool m_bValue
```

Attribut, das den Wert speichert.

SP_DS_BoolAttribute::SP_DS_BoolAttribute

```
SP_DS_BoolAttribute(const char* p_pchName, bool p_bVal = TRUE)
```

Konstruktor, erzeugt ein neues Attribut und initialisiert die Basisklasse SP_DS_Attribute mit dem Typen SP_ATTRIBUTE_BOOL und dem übergebenen Namen p_pchName.

SP_DS_BoolAttribute::Clone

```
virtual SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE)
```

Implementierung der geforderten Schnittstelle aus SP_DS_Attribute. Erzeugt eine neue Instanz von SP_DS_BoolAttribute mit dem Wert aus m_bValue.

SP_DS_BoolAttribute::GetValue

```
virtual bool GetValue()
```

Zugriffsmethode auf den Wert von m_bValue.

SP_DS_BoolAttribute::GetValueString

```
const char* GetValueString()
```

Implementierung der geforderten Schnittstelle aus SP_DS_Attribute. Liefert „1“ wenn m_bValue == TRUE, sonst „0“.

Bemerkung

Der zurückgegebene Character-Pointer wurde durch den Aufruf von `strdup` erzeugt, was die aufrufende Methode dafür verantwortlich macht, den angeforderten Speicher auch wieder freizugeben!

SP_DS_BoolAttribute::SetValue

```
virtual bool SetValue(bool p_bVal)
```

Zugriffsmethode zum Setzen des Wertes in `m_bValue`.

SP_DS_BoolAttribute::SetValueString

```
bool SetValueString(const char* p_pchVal)
```

Implementierung der geforderten Schnittstelle aus `SP_DS_Attribute`. Interpretiert den Wert von `p_pchVal` und setzt `m_bValue = TRUE`, wenn `p_pchVal == „1“`, sonst `FALSE`.

13 SP_DS_Edge

Klasse, die die Objekte der Kantenmenge des Graphen repräsentiert. Von dieser Klasse existieren keine Ableitungen.

Abgeleitet von

```
SP_Error
SP_Data
```

Include

```
sp_ds/SP_DS_Edge.h
```

SP_DS_Edge::m_lAttributes

```
SP_List<SP_DS_Attribute*> m_lAttributes
```

Liste der Attribute, die zu dieser Kante definiert wurden.

SP_DS_Edge::m_pcEdgeclass

```
SP_DS_Edgeclass* m_pcEdgeclass
```

Partition als deren Teil das Objekt definiert ist.

SP_DS_Edge::m_pcSource

```
SP_Data* m_pcSource
```

Datenstrukturelement, an dem diese Kante beginnt.

SP_DS_Edge::m_pcTarget

```
SP_Data* m_pcTarget
```

Datenstrukturelement, an dem diese Kante endet.

SP_DS_Edge::SP_DS_Edge

```
SP_DS_Edge(SP_DS_Edgeclass* p_pcEdgeclass, unsigned int p_nNetnumber = 0)
```

Konstruktor. Erzeugt ein neues Objekt ohne Attribute.

Parameter

p_pcEdgeclass

Partition, zu der dieser Knoten gehört.

p_nNetnumber

Netznummer, in der dieser Knoten liegt.

SP_DS_Edge::AddAttribute

```
SP_DS_Attribute* AddAttribute(SP_DS_Attribute* p_pcAttr)
```

Fügt das übergebene Objekt der Liste der Attribute dieser Kante hinzu. Die einzige Überprüfung erfolgt dabei gegen NULL für den Parameter und ob das übergebene Attribut nicht schon zu einem anderen Element gehört.

Parameter

p_pcAttr

hinzuzufügendes Attribut

Rückgabe

Der Wert des Parameters oder NULL, im Fehlerfall.

SP_DS_Edge::Clone

```
SP_DS_Edge* Clone(bool p_bCloneGr = TRUE)
```

Erzeugt eine exakte Kopie des Objekts, allerdings ohne die Werte in m_pcSource und m_pcTarget.

Parameter

p_bCloneGr

Flag, das anzeigt, ob die die graphischen Ausprägungen mit kopiert werden sollen.

Rückgabe

Die neue Kante, oder NULL im Fehlerfall.

SP_DS_Edge::GetAttribute

```
SP_DS_Attribute* GetAttribute(const char* p_pchName)
```

Liefert das erste Attribut des angegebenen Namens aus m_lAttributes oder NULL, wenn kein solches Attribut existiert.

SP_DS_Edge::GetAttributes

```
SP_List<SP_DS_Attribute*>* GetAttributes()
```

Liefert einen Zeiger auf m_lAttributes.

SP_DS_Edge::GetEdgeclass

```
SP_DS_Edgeclass* GetEdgeclass()
```

Zugriffsmethode auf das Objekt der Partition, zu der diese Kante gehört.

SP_DS_Edge::GetSource

```
SP_Data* GetSource() const
```

Liefert `m_pcSource`.

SP_DS_Edge::GetTarget

```
SP_Data* GetTarget() const
```

Liefert `m_pcTarget`.

SP_DS_Edge::RemoveAttribute

```
SP_DS_Attribute* RemoveAttribute(SP_DS_Attribute* p_pcVal)
```

Entfernt das angegebene Attribut aus `m_lAttributes`, gibt dabei aber keinen Speicher frei, sondern liefert das Objekt wieder zurück, falls es in der Liste existierte, sonst `NULL`.

Bemerkung

Ruft für den Parameter, wenn er Element von `m_lAttributes` war, `SP_DS_Attribute::SetDataParent(NULL)` auf.

SP_DS_Edge::SetEdgeclass

```
SP_DS_Edgeclass* SetEdgeclass(SP_DS_Edgeclass* p_pcVal)
```

Setzt den Wert von `m_pcEdgeclass` auf den des Parameters und liefert diesen zurück. Diese Methode wird nur im Zusammenhang mit Kopieren/Einfügen-Operationen verwendet.

SP_DS_Edge::SetNetnumber

```
bool SetNetnumber(unsigned int p_nNewVal, unsigned int p_nOldVal = 0)
```

Ruft `SP_DS_Attribute::SetNetnumber` mit denselben Parametern für alle Elemente von `m_lAttributes` auf und liefert anschliessend den Rückgabewert für `SP_Data::SetNetnumber`, ebenfalls mit den erhaltenen Parametern.

SP_DS_Edge::SetNodes

```
SP_DS_Edge* SetNodes(SP_Data* p_pcSource, SP_Data* p_pcTarget)
```

Setzen der Werte für `m_pcSource` und `m_pcTarget`. Dabei wird für die Parameter automatisch `AddSourceEdge` bzw. `AddTargetEdge` aufgerufen.

Parameter

p_pcSource

Datenstrukturobjekt am Anfang der Kante.

p_pcTarget

Datenstrukturobjekt am Ende der Kante.

Rückgabe

Das Objekt selbst, also `this` oder `NULL` im Fehlerfall.

14 SP_DS_Edgeclass

Klasse für die Kantenpartitionen eines Graphen. Von dieser Klasse besteht keine Notwendigkeit, Ableitungen zu definieren.

Abgeleitet von

`SP_Elementclass`

Include

`sp_ds/SP_DS_Edgeclass.h`

SP_DS_Edgeclass::m_pcPrototype

```
SP_DS_Edge* m_pcPrototype
```

Prototypisches Element, das zur Definitionszeit manipuliert wird und während der Benutzung als Vorlage für neue Elemente dient.

SP_DS_Edgeclass::m_lElements

```
SP_List<SP_DS_Edge*> m_lElements
```

Liste der konkreten Elemente dieser Partition.

SP_DS_Edgeclass::SP_DS_Edgeclass

```
SP_DS_Edgeclass(SP_DS_Graph* p_pcGraph, const char* p_pchName)
```

Konstruktor.

Parameter

p_pcGraph

Graphobjekt, als dessen Teil diese Partition definiert wird.

p_pchName

Name der Partition.

SP_DS_Edgeclass::AddAttribute

```
SP_DS_Attribute* AddAttribute(SP_DS_Attribute* p_pcAttr)
```

Hilfsfunktion für die Definition, die den Aufruf an `m_pcPrototype` weiterleitet, vorher aber überprüft, ob es schon ein Attribut dieses Namens an `m_pcPrototype` gibt.

Parameter

p_pcAttr

Attribut, das dem Prototypen hinzugefügt werden soll.

Rückgabe

Liefert den Wert des Aufrufs von `SP_DS_Edge::AddAttribute` oder `NULL`, falls ein solches Attribut schon existiert. Für diesen Fall wird eine Fehlermeldung durch `SetLastError` vermerkt.

SP_DS_Edgeclass:AddElement

```
bool AddElement(SP_DS_Edge* p_pcVal)
```

Fügt das im Parameter übergebene Objekt der Liste der Elemente hinzu.

Rückgabe

Liefert in der vorliegenden Version immer `TRUE`, unabhängig davon, ob der Parameter `NULL` ist oder sich dieser schon in `m_lElements` befand.

SP_DS_Edgeclass::GetElements

```
SP_List<SP_DS_Edge*>* GetElements()
```

Zugriff auf die Liste der Elemente dieser Partition.

Rückgabe

Adresse von `m_lElements`.

SP_DS_Edgeclass::GetPrototype

```
SP_DS_Edge* GetPrototype()
```

Zugriffsmethode auf das prototypische Element.

SP_DS_Edgeclass::HasAttributeType

```
bool HasAttributeType(SP_ATTRIBUTE_TYPE p_eVal)
```

Liefert `TRUE` für den Fall, dass ein Attribut des angegebenen Typs für den Prototypen definiert ist, `FALSE` sonst.

SP_DS_Edgeclass::NewElement

```
SP_DS_Edge* NewElement(unsigned int p_nNetnumber,
                       SP_Graphic* p_pcSource = NULL,
                       SP_Graphic* p_pcTarget = NULL)
```

Erzeugt ein neues Element aus der Kopie des Prototypen für die angegebene Netznummer.

Parameter

p_nNetnumber

Netznummer, in der das neue Element angelegt werden soll.

p_pcSource

Graphisches Objekt, dessen Datenstrukturentsprechung die Quelle ist.

p_pcTarget

Graphisches Objekt, desssen Datenstrukturentsprechung das Ziel ist.

Rückgabe

Das neue Objekt der Partition oder `NULL` im Fall, dass die Erstellung verweigert wurde.

Bemerkung

In dieser Methode wird die `EdgeRequirement`-Methode der Netzklasse aufgerufen, über welche die Möglichkeit besteht, die Erstellung zu verweigern.

SP_DS_Edgeclass::RemoveElement

```
bool RemoveElement(SP_DS_Edge* p_pcVal);
```

Entfernt den Parameter aus `m_lElements`.

Parameter

p_pcVal

Konkretes Objekt, das entfernt werden soll.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst.

Bemerkung

Diese Methode gibt keinen Speicher wieder frei, das muss mit dem erfolgreich entfernten Objekt durch die aufrufende Methode gemacht werden.

SP_DS_Edgeclass::RemoveElements

```
bool RemoveElements(unsigned int p_nNetnumber, SP_List<SP_Data*>* p_plCollect)
```

Entfernt mehrere Elemente aus `m_lElements`.

Parameter

p_nNetnumber

Alle Elemente, mit dieser Netznummer werden aus der Liste entfernt.

p_plCollect

Pointer zu einer Liste von Datenstrukturobjekten, die beim erfolgreichen Entfernen mit den entfernten Objekten gefüllt wird.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst. `FALSE` wird nur zurück gegeben, wenn die Liste nicht existiert.

Bemerkung

Diese Methode gibt keinen Speicher frei, das muss von der aufrufenden Funktion für alle Objekte, die in die Liste eingetragen wurden, gemacht werden.

SP_DS_Edgeclass::SetGraphic

```
SP_Graphic* SetGraphic(SP_Graphic* p_pcGraphic)
```

Vermerkt `p_pcGraphic` als neue graphische Ausprägung des Prototypen, indem es zuvor alle graphischen Ausprägungen löscht und anschliessend `SP_DS_Edge::AddGraphic` aufruft.

Parameter

p_pcGraphic

Die neue graphische Ausprägung.

Rückgabe

Das Objekt des Parameters im Erfolgsfall, sonst `NULL`.

SP_DS_Edgeclass::SetNetnumbers

```
bool SetNetnumbers(unsigned int p_nNewVal, unsigned int p_nOldVal)
```

Ruft `SP_DS_Edge::SetNetnumber`, mit denselben Parametern, für alle Elemente aus `m_lElements` auf und liefert `TRUE`, wenn all diese Aufrufe `TRUE` liefern, sonst `FALSE`.

15 SP_DS_Graph

Der Graph ist die zentrale Verwaltungsstruktur aller Elemente und Zentrum der meisten Aktionen während der Arbeit mit der Software. Ausserdem ist eine Instanz von `SP_DS_Graph` auch das Objekt, welches zur Definitionszeit einer Netzklasse bearbeitet wird.

Abgeleitet von

`SP_Error`
`SP_Type`
`SP_Name`

Include

`sp_ds/SP_DS_Graph.h`

SP_DS_Graph::m_bHasAnimators

```
bool m_bHasAnimators
```

Flag, das angibt, ob für mindestens eins der Elemente des Graphen Animatoren definiert wurden. Ist dieses Flag `TRUE`, wird ein Dialog zum Aktivieren des Animationsmodus' in der Oberfläche angezeigt.

SP_DS_Graph::m_bShowInGlobalDialog

```
bool m_bShowInGlobalDialog
```

Korrespondiert mit `SP_DS_Attribute::m_bShowInGlobalDialog`, indem diese Variable des Graphen auf `TRUE` gesetzt wird, sobald auch nur ein Attribut eines seiner Elemente dieses Flag ebenfalls auf `TRUE` gesetzt bekommt. Ist diese Variable `TRUE`, wird in der Oberfläche ein Dialog erzeugt, über den die Sichtbarkeit der gewählten Attribute global an- oder abgeschaltet werden kann.

SP_DS_Graph::m_lEdgeclass

```
SP_List<SP_DS_Edgeclass*> m_lEdgeclass
```

Liste der Kantenpartitionen.

SP_DS_Graph::m_lNodeclass

```
SP_List<SP_DS_Nodeclass*> m_lNodeclass
```

Liste der Knotenpartitionen.

SP_DS_Graph::m_nNetnumber

```
unsigned int m_nNetnumber
```

Zähler, über den für Elemente von Partitionen, die für Unternetze stehen, eine eindeutige Nummer für die Elemente in diesen Unternetzen generiert werden kann.

SP_DS_Graph::m_pcNetclass

```
SP_DS_Netclass* m_pcNetclass
```

SP_DS_Graph::SP_DS_Graph

```
SP_DS_Graph(SP_DS_Netclass* p_pcNetclass = NULL);
```

Konstruktor, erzeugt ein neues Graphobjekt.

Parameter

p_pcNetclass

Die Netzklasse, von der dieser Graph definiert werden soll. Wenn der Parameter nicht `NULL` ist, übernimmt der Graph den Namen der Netzklasse als seinen eigenen Namen und ruft ausserdem `SP_DS_Netclass::CreateGraph` mit sich selbst als Argument auf.

SP_DS_Graph::AddEdgeclass

```
SP_DS_Edgeclass* AddEdgeclass(SP_DS_Edgeclass* p_pcClass)
```

Fügt eine neue Kantenpartition hinzu.

Parameter

p_pcClass

Instanz der neuen Kantenpartition.

Rückgabe

Gibt den Wert des Parameters zurück. Das ist im Zusammenhang mit der Konstruktion der neuen Partition direkt im Parameter des Aufrufs sinnvoll.

SP_DS_Graph::AddNodeclass

```
SP_DS_Nodeclass* AddNodeclass(SP_DS_Nodeclass* p_pcClass)
```

Fügt eine neue Knotenpartition hinzu.

Parameter

p_pcClass

Instanz der neuen Knotenpartition.

Rückgabe

Gibt den Wert des Parameters zurück. Das ist im Zusammenhang mit der Konstruktion der neuen Partition direkt im Parameter des Aufrufs sinnvoll.

SP_DS_Graph::GetEdgeclass

```
SP_DS_Edgeclass* GetEdgeclass(const char* p_pchName)
```

Liefert die Kantenpartition des angegebenen Namens, oder NULL.

Parameter

p_pchName

Name der gesuchten Partition.

Rückgabe

Die erste Partition des angegebenen Namens oder NULL, wenn keine gefunden wurde.

SP_DS_Graph::GetNetclass

```
SP_DS_Netclass* GetNetclass()
```

Liefert den Pointer zur aktuellen Netzklasse.

SP_DS_Graph::GetNodeclass

```
SP_DS_Nodeclass* GetNodeclass(const char* p_pchName)
```

Liefert die Knotenpartition des angegebenen Namens, oder NULL.

Parameter

p_pchName

Name der gesuchten Partition.

Rückgabe

Die erste Partition des angegebenen Namens oder NULL, wenn keine gefunden wurde.

SP_DS_Graph::RemoveEdgeclass

```
SP_DS_Edgeclass* RemoveEdgeclass(const char* p_pchName)
```

Entfernt eine eine bestehende Partition aus der Liste der Partitionen.

Parameter

p_pchName

Name der zu entfernenden Partition

Rückgabe

Gibt die Partition zurück, die erfolgreich aus der Liste entfernt wurde, oder NULL, wenn keine Partition dieses Namens gefunden wurde.

Bemerkung

Da in der Regel der Speicher für neue Partitionen in der CreateGraph-Methode der Netzklasse angelegt wird und der Aufruf dieser Methode auch nur im Zusammenhang mit der Modifikation der Definition Sinn macht, muss der Destruktor der Netzklasse, für den Fall, dass sie erfolgreich entfernt wurde, auch wieder vom Nutzer aufgerufen werden.

SP_DS_Graph::RemoveNodeclass

```
SP_DS_Nodeclass* RemoveNodeclass(const char* p_pchName)
```

Entfernt eine eine bestehende Partition aus der Liste der Partitionen.

Parameter

p_pchName

Name der zu entfernenden Partition

Rückgabe

Gibt die Partition zurück, die erfolgreich aus der Liste entfernt wurde, oder NULL, wenn keine Partition dieses Namens gefunden wurde.

Bemerkung

Da in der Regel der Speicher für neue Partitionen in der CreateGraph-Methode der Netzklasse angelegt wird und der Aufruf dieser Methode auch nur im Zusammenhang mit der Modifikation der Definition Sinn macht, muss der Destruktor der Netzklasse, für den Fall, dass sie erfolgreich entfernt wurde, auch wieder vom Nutzer aufgerufen werden.

SP_DS_Graph::RenameEdgeclass

```
SP_DS_Edgeclass* RenameEdgeclass(const char* p_pchFrom, const char* p_pchTo)
```

Benennt eine bestehende Partition um.

Parameter

p_pchFrom

Name der umzubenennenden Partition

p_pchTo

Neuer Name

Rückgabe

Gibt die Partition zurück, die erfolgreich umbenannte wurde, oder NULL, wenn keine Partition mit dem Namen im ersten Parameter gefunden wurde.

SP_DS_Graph::RenameNodeclass

```
SP_DS_Nodeclass* RenameNodeclass(const char* p_pchFrom, const char* p_pchTo)
```

Benennt eine bestehende Partition um.

Parameter

p_pchFrom

Name der umzubenennenden Partition

p_pchTo

Neuer Name

Rückgabe

Gibt die Partition zurück, die erfolgreich umbenannte wurde, oder NULL, wenn keine Partition mit dem Namen im ersten Parameter gefunden wurde.

SP_DS_Graph::SetAnimation

```
bool SetAnimation(SP_DS_Animation* p_pcAnim)
```

Assoziiert eine Animation mit der Netzklasse des Graphen.

Parameter

p_pcAnim

Instanz der Animation, die als Vorlage verwendet werden soll.

Rückgabe

Liefert `TRUE` im Erfolgsfall, sonst `FALSE`.

Bemerkung

Diese Methode liefert im Moment immer `TRUE`, sie löscht allerdings bereits registrierte Animationen für Netzklassen desselben Namens. Das soll verhindern, dass Speicherlöcher entstehen, wenn durch das Durchlaufen von `CreateGraph` mehrere male `SetAnimation` aufgerufen wird.

SP_DS_Graph::SetNetclass

```
SP_DS_Netclass* SetNetclass(SP_DS_Netclass* p_pcVal)
```

Setzt den Pointer zur aktuellen Netzklasse. Dabei wird nicht die `CreateGraph`-Methode der Netzklasse aufgerufen!

Parameter

p_pcVal

Die neue Netzklasse.

Rückgabe

Gibt den Wert des Parameters zurück.

16 SP_DS_IdAttribute

Verwaltet keinen eigenen Wert, überschreibt aber die SetValue und GetValue Methoden der Basisklasse.

Abgeleitet von

`SP_DS_NumberAttribute`

Include

`sp_ds/attributes/SP_DS_IdAttribute.h`

SP_DS_IdAttribute::SP_DS_IdAttribute

```
SP_DS_IdAttribute(const char* p_pchName)
```

Konstruktor, erzeugt ein neues `SP_DS_IdAttribute`, initialisiert die Basisklasse `SP_DS_NumberAttribute` mit dem übergebenen Namen und „-1“ im Wertfeld und setzt den Typen auf `SP_ATTRIBUTE_ID`.

SP_DS_IdAttribute::Clone

```
SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE)
```

Erzeugt eine Kopie.

SP_DS_IdAttribute::GetValue

```
long GetValue()
```

Liefert den Wert des in der Basisklasse gespeicherten Attributs für den Zähler. Überprüft aber zuvor, ob der dort eingeschriebene Wert -1 entspricht. Ist das der Fall, wird durch den Aufruf der Methode der Partition des Elternelements SP_Elementclass::GetNewIdCount ein neuer Wert generiert.

SP_DS_IdAttribute::SetValue

```
bool SetValue(long p_nVal)
```

Setzt den Wert des Parameters in das in der Basisklasse vorgesehene Wertfeld, stellt aber zuvor in der Partition des Elternelements durch Aufruf von `SP_Elementclass::TestIdCount` sicher, dass der dort implementierte Zähler weiterhin gültige Werte liefert.

SP_DS_IdAttribute::Squeeze

```
bool Squeeze()
```

Sondermethode, die aus der Oberfläche heraus aufgerufen wird und die Werte aller `SP_DS_IdAttribute`-Instanzen jeweils für die Elemente der entsprechenden Partition kontinuierlich numeriert.

17 SP_DS_LogicAttribute

Attribut, das es den zugeordneten Elementen erlaubt, durch Nutzereingriff mehrere „logische“ graphische Ausprägungen zu besitzen.

Abgeleitet von

`SP_DS_BoolAttribute`

Include

`sp_ds/attributes/SP_DS_LogicAttribute.h`

SP_DS_LogicAttribute::m_bOneShot

```
bool m_bOneShot
```

Flag, das anzeigt, ob durch das Setzen des boolschen Werts dieses Attributs auch die Umorganisation der Elemente angestossen werden soll. Ist m_bOneShot == TRUE wird in SetValue nur der Wert gesetzt, andernfalls auch die Suche nach „Verwandten“ angestossen.

SP_DS_LogicAttribute::m_sReferenceAttribute

```
wxString m_sReferenceAttribute
```

Name eines Attributs der Elemente derselben Partitionen zu der auch dieses Attribut gehört. Dieses Attribut legt fest, in welchem Wert sich die Elemente gleichen müssen, um als zueinander logisch angesehen und vereinigt zu werden.

SP_DS_LogicAttribute::SP_DS_LogicAttribute

```
SP_DS_LogicAttribute(const char* p_pchName,
                     const char* p_pchRef,
                     bool p_bVal = FALSE)
```

Konstruktor, der den Konstruktor der Basisklasse mit dem Namen und dem Standardwert initialisiert und `SP_ATTRIBUTE_LOGIC` als Typen setzt.

Parameter

p_pchName

Name des Attributs

p_pchRef

Name des Referenzattributs, mit dem m_sReferenceAttribute initialisiert wird.

p_bVal

Initialwert des Attributs.

SP_DS_LogicAttribute::Clone

```
SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE)
```

Erzeugt eine Kopie.

SP_DS_LogicAttribute::GetReferenceAttribute

```
const char* GetReferenceAttribute()
```

Liefert das Ergebnis des Aufrufs von `m_sReferenceAttribute::c_str`.

SP_DS_LogicAttribute::JoinElements

```
bool JoinElements()
```

Wird aus der `OnChangeState`-Methode heraus aufgerufen, wenn `m_bOneShot` nicht `TRUE` ist und stösst die Suche nach ähnlichen Elementen derselben Partition im Objekt des Graphen an, zu das zu diesem Attribut gehörende Element logisch sein soll.

SP_DS_LogicAttribute:: OnChangeState

```
bool OnChangeState()
```

Ruft, je nach Wert des boolschen Attributs in der Basisklasse, nach dem Aufruf von `SetValue` eine der Methoden `JoinElements` oder `SplitElements` auf. Das geschieht nur, wenn `m_bOneShot == FALSE`.

SP_DS_LogicAttribute:: SetOneShot

```
void SetOneShot(bool p_bVal = TRUE)
```

Zugriffsmethode zum Setzen des Werts von `m_bOneShot`.

SP_DS_LogicAttribute::SetValue

```
bool SetValue(bool p_bVal)
```

Neuimplementierung der Methode aus `SP_DS_BoolAttribute` um nach dem Setzen des Attributs in der Basisklasse `OnChangeState` aufrufen zu können.

SP_DS_LogicAttribute::SplitElements

```
bool SplitElements()
```

Implementierung des Verhaltens, wenn ein einmal aktiviertes „logisch“-Attribut wieder deaktiviert wird. Dabei erfolgt die eigentliche Trennung der Elemente über einen Aufruf aus dem Graph-Objekt.

18 SP_DS_Node

Klasse, die die Objekte der Knotenmenge des Graphen repräsentiert. Von dieser Klasse existieren keine Ableitungen.

Abgeleitet von

SP_Error
SP_Data

Include

sp_ds/SP_DS_Node.h

SP_DS_Node::m_lAttributes

```
SP_List<SP_DS_Attribute*> m_lAttributes
```

Liste der Attribute, die zu diesem Knoten definiert wurden.

SP_DS_Node::m_lSourceEdges

```
SP_List<SP_DS_Edge*> m_lSourceEdges
```

Liste der ausgehenden Kanten dieses Knotens. Also alle Kanten, die hier beginnen.

SP_DS_Node::m_lTargetEdges

```
SP_List<SP_DS_Edge*> m_lTargetEdges
```

Liste der eingehenden Kanten dieses Knotens. Also alle Kanten, die hier enden.

SP_DS_Node::m_pcNodeclass

```
SP_DS_Nodeclass* m_pcNodeclass
```

Partition als deren Teil das Objekt definiert ist.

SP_DS_Node::SP_DS_Node

```
SP_DS_Node(SP_DS_Nodeclass* p_pcNodeclass, unsigned int p_nNetnumber = 0)
```

Konstruktor. Erzeugt ein neues Objekt ohne Attribute und mit leeren Kantenlisten.

Parameter

p_pcNodeclass

Partition, zu der dieser Knoten gehört.

p_nNetnumber

Netznummer, in der dieser Knoten liegt.

SP_DS_Node::AddAttribute

```
SP_DS_Attribute* AddAttribute(SP_DS_Attribute* p_pcAttr)
```

Fügt das übergebene Objekt der Liste der Attribute dieses Knotens hinzu. Die einzige Überprüfung erfolgt dabei gegen `NULL` für den Parameter und ob das übergebene Attribut nicht schon zu einem anderen Element gehört.

Parameter

p_pcAttr

hinzuzufügendes Attribut

Rückgabe

Der Wert des Parameters oder `NULL`, im Fehlerfall.

SP_DS_Node::AddSourceEdge

```
SP_DS_Node* AddSourceEdge(SP_DS_Edge* p_pcEdge)
```

Fügt den Parameter der Liste der ausgehenden Kanten hinzu, falls er noch nicht in der Liste existiert.

Parameter

p_pcEdge

hinzuzufügende Kante

Rückgabe

Gibt das Objekt des Knotens selbst zurück, also `this`.

Bemerkung

Wenn diese Kante als Element von `m_lSourceEdges` akzeptiert wurde, wird automatisch auch für den Parameter `SP_DS_Edge::SetSource (this)` aufgerufen!

SP_DS_Node::AddTargetEdge

```
SP_DS_Node* AddTargetEdge(SP_DS_Edge* p_pcEdge)
```

Fügt den Parameter der Liste der eingehenden Kanten hinzu, falls er noch nicht in der Liste existiert.

Parameter

p_pcEdge

hinzuzufügende Kante

Rückgabe

Gibt das Objekt des Knotens selbst zurück, also `this`.

Bemerkung

Wenn diese Kante als Element von `m_lTargetEdges` akzeptiert wurde, wird automatisch auch für den Parameter `SP_DS_Edge::SetTarget (this)` aufgerufen!

SP_DS_Node::Clone

```
SP_DS_Node* Clone(bool p_bCloneGr = TRUE)
```

Erzeugt eine exakte Kopie des Objekts, allerdings mit leeren Kantenlisten.

Parameter

p_bCloneGr

Flag, das anzeigt, ob die die graphischen Ausprägungen mitkopiert werden sollen.

Rückgabe

Der neue Knoten, oder `NULL` im Fehlerfall.

SP_DS_Node::GetAttribute

```
SP_DS_Attribute* GetAttribute(const char* p_pchName)
```

Liefert das erste Attribut des angegebenen Namens aus `m_lAttributes` oder `NULL`, wenn kein solches Attribut existiert.

SP_DS_Node::GetAttributes

```
SP_List<SP_DS_Attribute*>* GetAttributes()
```

Liefert einen Zeiger auf `m_lAttributes`.

SP_DS_Node::GetNodeclass

```
SP_DS_Nodeclass* GetNodeclass()
```

Zugriffsmethode auf das Objekt der Partition, zu der dieser Knoten gehört.

SP_DS_Node::GetSourceEdges

```
SP_List<SP_DS_Edge*>* GetSourceEdges()
```

Liefert einen Zeiger auf `m_lSourceEdges`.

SP_DS_Node::GetTargetEdges

```
SP_List<SP_DS_Edge*>* GetTargetEdges()
```

Liefert einen Zeiger auf `m_lTargetEdges`.

SP_DS_Node::RemoveAttribute

```
SP_DS_Attribute* RemoveAttribute(SP_DS_Attribute* p_pcVal)
```

Entfernt das angegebene Attribut aus `m_lAttributes`, gibt dabei aber keinen Speicher frei, sondern liefert das Objekt wieder zurück, falls es in der Liste existierte, sonst `NULL`.

Bemerkung

Ruft für den Parameter, wenn er Element von `m_lAttributes` war, `SP_DS_Attribute::SetDataParent(NULL)` auf.

SP_DS_Node::RemoveEdge

```
bool RemoveEdge(SP_DS_Edge* p_pcEdge)
```

Ruft, je nach Lage der Kante, `RemoveSourceEdge` oder `RemoveTargetEdge` auf und liefert deren Ergebnis zurück. Sollte die Kante weder ein- noch ausgehende Kante sein, liefert diese Funktion `FALSE`.

SP_DS_Node::RemoveSourceEdge

```
bool RemoveSourceEdge(SP_DS_Edge* p_pcEdge)
```

Entfernt den Parameter aus `m_lSourceEdges` und ruft ausserdem `SP_DS_Edge::SetSource(NULL)` für den Parameter auf.

SP_DS_Node::RemoveTargetEdge

```
bool RemoveTargetEdge(SP_DS_Edge* p_pcEdge)
```

Entfernt den Parameter aus `m_lTargetEdges` und ruft ausserdem `SP_DS_Edge::SetTarget(NULL)` für den Parameter auf.

SP_DS_Node::SetNetnumber

```
bool SetNetnumber(unsigned int p_nNewVal, unsigned int p_nOldVal = 0)
```

Ruft `SP_DS_Attribute::SetNetnumber` mit denselben Parametern für alle Elemente von `m_lAttributes` auf und liefert anschliessend den Rückgabewert für `SP_Data::SetNetnumber`, ebenfalls mit den erhaltenen Parametern.

SP_DS_Node::SetNodeclass

```
SP_DS_Nodeclass* SetNodeclass(SP_DS_Nodeclass* p_pcVal)
```

Setzt den Wert von `m_pcNodeclass` auf den des Parameters und liefert diesen zurück. Diese Methode wird nur im Zusammenhang mit Kopieren/Einfügen-Operationen verwendet.

19 SP_DS_Nodeclass

Klasse für die Knotenpartitionen eines Graphen. Von dieser Klasse besteht keine Notwendigkeit, Ableitungen zu definieren.

Abgeleitet von

`SP_Elementclass`

Include

`sp_ds/SP_DS_Nodeclass.h`

SP_DS_Nodeclass::m_lElements

```
SP_List<SP_DS_Node*> m_lElements
```

Liste der konkreten Elemente dieser Partition.

SP_DS_Nodeclass::m_pcPrototype

```
SP_DS_Node* m_pcPrototype
```

Prototypisches Element, das zur Definitionszeit manipuliert wird und während der Benutzung als Vorlage für neue Elemente dient.

SP_DS_Nodeclass::SP_DS_Nodeclass

```
SP_DS_Nodeclass(SP_DS_Graph* p_pcGraph, const char* p_pchName)
```

Konstruktor.

Parameter

p_pcGraph

Graphobjekt, als dessen Teil diese Partition definiert wird.

p_pchName

Name der Partition.

SP_DS_Nodeclass::AddAttribute

```
SP_DS_Attribute* AddAttribute(SP_DS_Attribute* p_pcAttr)
```

Hilfsfunktion für die Definition, die den Aufruf an `m_pcPrototype` weiterleitet, vorher aber überprüft, ob es schon ein Attribut dieses Namens an `m_pcPrototype` gibt.

Parameter

p_pcAttr

Attribut, das dem Prototypen hinzugefügt werden soll.

Rückgabe

Liefert den Wert des Aufrufs von `SP_DS_Node::AddAttribute` oder `NULL`, falls ein solches Attribut schon existiert oder der Parameter ungültig war. Für diesen Fall wird eine Fehlermeldung durch `SetLastError` vermerkt.

SP_DS_Nodeclass::AddElement

```
bool AddElement(SP_DS_Node* p_pcVal)
```

Fügt das im Parameter übergebene Objekt der Liste der Elemente hinzu.

Rückgabe

Liefert in der vorliegenden Version immer `TRUE`, unabhängig davon, ob der Parameter `NULL` ist oder sich dieser schon in `m_lElements` befand, wofür dann aber das Hinzufügen unterbunden wird.

SP_DS_Nodeclass::GetElements

```
SP_List<SP_DS_Node*>* GetElements()
```

Zugriff auf die Liste der Elemente dieser Partition.

Rückgabe

Adresse von `m_lElements`.

SP_DS_Nodeclass::GetPrototype

```
SP_DS_Node* GetPrototype()
```

Zugriffsmethode auf das prototypische Element `m_pcPrototype`.

SP_DS_Nodeclass::HasAttributeType

```
bool HasAttributeType(SP_ATTRIBUTE_TYPE p_eVal)
```

Liefert `TRUE` für den Fall, dass ein Attribut des angegebenen Typs für den Prototypen definiert ist, `FALSE` sonst.

SP_DS_Nodeclass::NewElement

```
SP_DS_Node* NewElement(unsigned int p_nNetnumber, SP_DS_Node* p_pcCloneRef = NULL)
```

Erzeugt ein neues Element aus der Kopie des Prototypen für die angegebene Netznummer.

Parameter

p_nNetnumber

Netznummer, in der das neue Element angelegt werden soll.

p_pcCloneRef

Referenzobjekt, das statt `m_pcPrototype` zur Vorlage beim Kopieren verwendet werden soll. Wird nur im Zusammenhang mit Kopieren/Einfügen verwendet.

Rückgabe

Das neue Objekt der Partition oder `NULL` im Fall, dass die Erstellung verweigert wurde.

Bemerkung

In dieser Methode wird die `NodeRequirement`-Methode der Netzklasse aufgerufen, durch die der Nutzer Einfluss auf die Erstellung nehmen kann.

SP_DS_Nodeclass::RemoveElement

```
bool RemoveElement(SP_DS_Node* p_pcVal);
```

Entfernt den Parameter aus `m_lElements`.

Parameter

p_pcVal

Konkretes Objekt, das entfernt werden soll.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst.

Bemerkung

Diese Methode gibt keinen Speicher wieder frei, das muss mit dem erfolgreich entfernten Objekt durch die aufrufende Methode gemacht werden.

SP_DS_Nodeclass::RemoveElements

```
bool RemoveElements(unsigned int p_nNetnumber, SP_List<SP_Data*>* p_plCollect)
```

Entfernt mehrere Elemente aus `m_lElements`.

Parameter

p_nNetnumber

Alle Elemente, mit dieser Netznummer werden aus der Liste entfernt.

p_plCollect

Pointer zu einer Liste von Datenstrukturobjekten, die beim erfolgreichen Entfernen mit den entfernten Objekten gefüllt wird.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst. `FALSE` wird nur zurück gegeben, wenn die Liste nicht existiert.

Bemerkung

Diese Methode gibt keinen Speicher frei, das muss von der aufrufenden Funktion für alle Objekte, die in die Liste eingetragen wurden, gemacht werden.

SP_DS_Nodeclass::SetGraphic

```
SP_Graphic* SetGraphic(SP_Graphic* p_pcGraphic)
```

Vermerkt `p_pcGraphic` als neue graphische Ausprägung des Prototypen, indem es zuvor alle graphischen Ausprägungen löscht und anschliessend `SP_DS_Node::AddGraphic` aufruft.

Parameter

p_pcGraphic

Die neue graphische Ausprägung.

Rückgabe

Das Objekt des Parameters im Erfolgsfall, sonst `NULL`.

SP_DS_Nodeclass::SetNetnumbers

```
bool SetNetnumbers(unsigned int p_nNewVal, unsigned int p_nOldVal)
```

Ruft `SP_DS_Node::SetNetnumber`, mit denselben Parametern, für alle Elemente aus `m_lElements` auf und liefert `TRUE`, wenn all diese Aufrufe `TRUE` liefern, sonst `FALSE`.

20 SP_DS_NumberAttribute

Attribut zur Repräsentation von numerischen Werten. Dieses Attribut verwendet SP_ATTRIBUTE_NUMBER als Typen.

Abgeleitet von

SP_DS_Attribute

Include

sp_ds/attributes/SP_DS_NumberAttribute.h

SP_DS_NumberAttribute::m_nValue

```
long m_nValue
```

Wert des Attributs.

SP_DS_NumberAttribute::SP_DS_NumberAttribute

```
SP_DS_NumberAttribute(const char* p_pchName, long p_nVal = 0)
```

Konstruktor. Initialisiert die Basisklasse mit dem übergebenen Namen, dem Typen SP_ATTRIBUTE_NUMBER und setzt m_nValue auf den Wert von p_nVal.

SP_DS_NumberAttribute::Clone

```
virtual SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE)
```

Implementierung nach der Vorgabe der Basisklasse, die ein neues Attribut desselben Typs, Namens und Wertes erzeugt.

SP_DS_NumberAttribute::GetValue

```
virtual long GetValue()
```

Zugriffsmethode auf den Wert in m_nValue.

SP_DS_NumberAttribute::GetValueString

```
const char* GetValueString()
```

Umwandlung des Wertes in eine textuelle Repräsentation unter Verwendung von wxString::Printf.

Bemerkung

Der zurückgegebene Character-Pointer wurde durch den Aufruf von `strdup` erzeugt, was die aufrufende Methode dafür verantwortlich macht, den angeforderten Speicher auch wieder freizugeben!

SP_DS_NumberAttribute::SetValue

```
virtual bool SetValue(long p_nVal)
```

Zugriffsmethode zum Setzen des Werts in `m_nValue`.

SP_DS_NumberAttribute:: SetValueString

```
bool SetValueString(const char* p_pchVal)
```

Deserialisierung einer durch `GetValueString` erzeugten, textuellen Repräsentation dieses Attributs. Die Konvertierung erfolgt mittels `wxString::ToLong`.

Rückgabe

`TRUE` für den Fall, dass die Umwandlung erfolgreich war, `FALSE` sonst.

21 SP_DS_TextAttribute

Attribut, das einen textuellen Wert repräsentiert.

Abgeleitet von

`SP_DS_Attribute`

Include

`sp_ds/attributes/SP_DS_TextAttribute.h`

SP_DS_TextAttribute::m_sValue

```
wxString m_sValue
```

Wert des Attributs.

SP_DS_TextAttribute::SP_DS_TextAttribute

```
SP_DS_TextAttribute(const char* p_pchName, const char* p_pchVal = "")
```

Konstruktor, initialisiert die Basisklasse mit dem übergebenen Namen und dem Typen `SP_ATTRIBUTE_TEXT`.

SP_DS_TextAttribute::Clone

```
SP_DS_Attribute* Clone(bool p_bCloneGr = TRUE)
```

Implementierung der geforderten Schnittstelle der Basisklasse. Liefert ein neues Text-Attribut.

SP_DS_TextAttribute::GetValue

```
const char* GetValue()
```

Zugriffsmethode auf den Wert von `m_sValue.c_str`.

SP_DS_TextAttribute::GetValueString

```
const char* GetValueString()
```

Liefert den Wert von `GetValue`.

Bemerkung

Der zurückgegebene Character-Pointer wurde durch den Aufruf von `strdup` erzeugt, was die aufrufende Methode dafür verantwortlich macht, den angeforderten Speicher auch wieder freizugeben!

SP_DS_TextAttribute::SetValue

```
bool SetValue(const char* p_pchVal)
```

Zugriffsmethode zum Setzen des Wertes in `m_sValue`.

SP_DS_TextAttribute::SetValueString

```
bool SetValueString(const char* p_pchVal)
```

Liefert das Ergebnis des Aufrufs von `SetValue(p_pchVal)`.

22 SP_GR_Animator

Basisklasse für alle graphischen Objekte, im Zusammenhang mit der Netz-Animation. Von dieser Klasse können keine Objekte instanziiert werden, sie erweitert lediglich die in SP_Graphic definierte Schnittstelle um Funktionalitäten im Hinblick auf Animationen.

Abgeleitet von

`SP_Animator`
`SP_Graphic`

Include

`sp_gr/SP_GR_Animator.h`

SP_GR_Animator::SP_GR_Animator

```
SP_GR_Animator()
```

Konstruktor, initialisiert SP_Graphic mit dem Typen `SP_GRAPHIC_ANIMATOR`.

SP_GR_Animator::PostStep

```
virtual bool PostStep() = 0
```

Festlegung der Schnittstelle für alle abgeleiteten Klassen.

SP_GR_Animator::PreStep

```
virtual bool PreStep(unsigned int p_nSteps) = 0
```

Festlegung der Schnittstelle für alle abgeleiteten Klassen.

Parameter

p_nSteps

Anzahl der für diesen Animator zur Verfügung stehenden Schritte, also die Aufrufe seiner `Step`-Methode, wie sie vom Datenstruktur-Animator festgelegt wird.

SP_GR_Animator::Step

```
virtual bool Step(unsigned int p_nStep) = 0
```

Festlegung der Schnittstelle für alle abgeleiteten Klassen.

Parameter

p_nStep

Nummer des aktuellen Schritts, wie sie vom Datenstruktur-Animator festgelegt wird.

23 SP_GR_ArrowEdge

Graphische Kante, an deren einem Ende eine einfache Pfeilspitze angezeigt wird.

Abgeleitet von

`SP_GR_Edge`

Include

`sp_gr/edges/SP_GR_ArrowEdge.h`

SP_GR_ArrowEdge::m_pcPrimitive

```
SP_GR_BaseEdge* m_pcPrimitive
```

Primitiv für die konkrete graphische Ausprägung. `SP_GR_BaseEdge` ist dabei eine Ableitung von `wxLineShape`, wie sie in der *OGL* definiert wird.

SP_GR_ArrowEdge::SP_GR_ArrowEdge

```
SP_GR_ArrowEdge(SP_DS_Edge* p_pcParent)
```

Konstruktor, initialisiert SP_GR_Edge, instanziiert ein Objekt von `SP_GR_BaseEdge` als Darstellungselement und vermerkt sich selbst und sein *wx*-Objekt im Mapping der externen Objekte aus der *OGL* zu den SNOOPY-Objekten.

```
1   SP_GR_ArrowEdge::SP_GR_ArrowEdge(SP_DS_Edge* p_pcParent)
2   :SP_GR_Edge(p_pcParent)
3   {
4       m_pcPrimitive = new SP_GR_BaseEdge();
5       SP_Core::Instance()->AssociateExtern(m_pcPrimitive, this);
6
7       m_pcPrimitive->AddArrow(ARROW_ARROW);
8       m_pcPrimitive->SetBrush(wxBLACK_BRUSH);
9   }
```

SP_GR_ArrowEdge::Clone

```
SP_Graphic* Clone(SP_Data* p_pcParent)
```

Erzeugt ein neues Objekt als Kopie seiner selbst und weist ihm als Elternelement den Parameter zu.

SP_GR_ArrowEdge::GetPrimitive

```
wxShape* GetPrimitive()
```

Zugriffsmethode auf den Wert in `m_pcPrimitive` als Objekt vom Basis-Typen der *OGL*.

24 SP_GR_Attribute

Basisklasse für alle graphischen Attribute. Von dieser Klasse können keine Objekte instanziiert werden, es handelt sich lediglich um die Erweiterung der in SP_Graphic definierten Schnittstelle mit Blick auf die Besonderheiten graphischer Attribute.

Abgeleitet von

`SP_Error`
`SP_Graphic`

Include

`sp_gr/SP_GR_Attribute.h`

SP_GR_Attribute::m_nOffsetX

```
double m_nOffsetX
```

Horizontale Verschiebung des Attributs, relativ zum Mittelpunkt des Elternelements, wie dieser durch dessen `GetPosX` Methode definiert wird.

SP_GR_Attribute::m_nOffsetY

```
double m_nOffsetY
```

Vertikale Verschiebung des Attributs, relativ zum Mittelpunkt des Elternelements, wie dieser durch dessen `GetPosY` Methode definiert wird.

SP_GR_Attribute::SP_GR_Attribute

```
SP_GR_Attribute(SP_DS_Attribute* p_pcDataParent)
```

Konstruktor, assoziiert diese Graphik mit dem Element der Datenstruktur und initialisiert `SP_Graphic` mit dem Typen `SP_GRAPHIC_ATTRIBUTE`.

SP_GR_Attribute::Clone

```
virtual SP_Graphic* Clone(SP_Data* p_pcParent) = 0
```

Übernahme der rein virtuellen Schnittstelle aus `SP_Graphic` zur Implementierung innerhalb der Ableitungen.

SP_GR_Attribute::GetOffsetX

```
double GetOffsetX()
```

Zugriffsmethode auf den Wert in `m_nOffsetX`.

SP_GR_Attribute::GetOffsetY

```
double GetOffsetY()
```

Zugriffsmethode auf den Wert in `m_nOffsetY`.

SP_GR_Attribute::GetPrimitive

```
virtual wxShape* GetPrimitive() = 0
```

Übernahme der rein virtuellen Schnittstelle aus `SP_Graphic` zur Implementierung innerhalb der Ableitungen.

SP_GR_Attribute::SetOffsetX

```
bool SetOffsetX(double p_nVal)
```

Zugriffsmethode zum Setzen des Werts in `m_nOffsetX`.

SP_GR_Attribute::SetOffsetY

```
bool SetOffsetY(double p_nVal)
```

Zugriffsmethode zum Setzen des Werts in `m_nOffsetY`.

25 SP_GR_Circle

Einfacher Kreis zur Verwendung als graphische Ausprägung eines Knotens.

Abgeleitet von

`SP_GR_Node`

Include

`sp_gr/shapes/SP_GR_Circle.h`

SP_GR_Circle::m_pcPrimitive

```
SP_GR_BaseCircle* m_pcPrimitive
```

Konkretes Objekt zur Anzeige. `SP_GR_BaseCircle` ist dabei eine Ableitung von *wxCircleShape*, wie sie in der *OGL* definiert wird.

SP_GR_Circle::SP_GR_Circle

```
SP_GR_Circle(SP_DS_Node* p_pcParent, double p_nR = 20.0)
```

Konstruktor, erzeugt die Instanz für `m_pcPrimitive`, initialisiert diese mit dem angegebenen Radius in `p_nR` und vermerkt sich selbst und sein *wx*-Objekt im Mapping der externen Objekte aus der *OGL* zu den SNOOPY-Objekten.

SP_GR_Circle::Clone

```
SP_Graphic* Clone(SP_Data* p_pcParent)
```

Erzeugt ein neues Objekt als Kopie seiner selbst und weist ihm als Elternelement den Parameter zu.

SP_GR_Circle::GetPrimitive

```
wxShape* GetPrimitive()
```

Zugriffsmethode auf den Wert in `m_pcPrimitive` als Objekt vom Basis-Typen der *OGL*.

SP_GR_Circle::GetRadius

```
double GetRadius()
```

Zugriffsmethode auf den Radius, wie er im Objekt von `SP_GR_BaseCircle` gespeichert ist.

SP_GR_Circle::SetRadius

```
bool SetRadius(double p_nVal)
```

Zugriffsmethode zum Setzen des Radius', wie er im Objekt von `SP_GR_BaseCircle` gespeichert ist.

26 SP_GR_DoubleCircle

Klasse, die einen doppelwandigen Kreis repräsentiert, wird in `SP_DS_SimplePed` für *Coarse*-Places verwendet.

Abgeleitet von

`SP_GR_DrawnShape`

Include

`sp_gr/shapes/SP_GR_DoubleCircle.h`

SP_GR_DoubleCircle::m_nRadius

```
double m_nRadius
```

Wert des Radius' mit dem der äussere Kreis gezeichnet werden soll.

SP_GR_DoubleCircle::SP_GR_DoubleCircle

```
SP_GR_DoubleCircle(SP_DS_Node* p_pcParent, double p_nRadius = 10.0)
```

Erzeugt ein neues Objekt, instanziiert aber selbst aktiv kein Objekt für `m_pcPrimitive`, da das im Konstruktor der Basisklasse `SP_GR_DrawnShape` passiert.

SP_GR_DoubleCircle::Clone

```
SP_Graphic* Clone(SP_Data* p_pcParent)
```

Erzeugt eine Kopie seiner selbst die dem im Parameter übergebenen Datenstruktur-Objekt zugeordnet wird.

SP_GR_DoubleCircle::Draw

```
bool Draw()
```

Zeichnet im angegebenen Radius einen Kreis und einen, mit einem um die Hälfte kleineren Radius, in dessen Mitte.

27 SP_GR_DrawnShape

Graphisches Objekt, in dem mit einfachen graphischen Befehlen jederzeit „gezeichnet“ werden kann, zur Verwendung als graphische Ausprägung eines Knotens.

Diese Klasse ist eine Erweiterung der Schnittstelle aus SP_GR_Node und es können keine Objekte von ihr instanziiert werden.

Abgeleitet von

`SP_GR_Node`

Include

`sp_gr/shapes/SP_GR_DrawnShape.h`

SP_GR_DrawnShape::m_pcPrimitive

```
SP_GR_BaseCircle* m_pcPrimitive
```

Konkretes Objekt zur Anzeige. `SP_GR_BaseDrawn` ist dabei eine Ableitung von *wxDrawnShape*, wie sie in der *OGL* definiert wird.

SP_GR_DrawnShape::SP_GR_ DrawnShape

```
SP_GR_DrawnShape(SP_DS_Node* p_pcParent)
```

Konstruktor, erzeugt die Instanz für `m_pcPrimitive` und vermerkt sich selbst und sein *wx*-Objekt im Mapping der externen Objekte aus der *OGL* zu den SNOOPY-Objekten.

SP_GR_DrawnShape::Clone

```
SP_Graphic* Clone(SP_Data* p_pcParent) = 0
```

Weiterleitung der Schnittstelle.

SP_GR_DrawnShape::Draw

```
virtual bool Draw()
```

Methode, die von allen Ableitungen überschrieben werden sollte und in der in das Objekt selbst gezeichnet werden kann:

```
1       m_pcPrimitive->DrawRectangle(wxRect(wxPoint(-5, -5), wxSize(10, 10)));
2       m_pcPrimitive->DrawLine(wxPoint(-5, -5), wxPoint( 5, 5));
3       m_pcPrimitive->DrawLine(wxPoint( 5, -5), wxPoint(-5, 5));
```

SP_GR_DrawnShape::GetPrimitive

```
wxShape* GetPrimitive()
```

Zugriffsmethode auf den Wert in `m_pcPrimitive` als Objekt vom Basis-Typen der *OGL*.

28 SP_GR_Edge

Basisklasse für alle graphischen Kanten. Von dieser Klasse können keine Objekte instanziiert werden, sie erweitert lediglich das in SP_Graphic definierte Interface und bildet die Schnittstelle.

Abgeleitet von

SP_Error
SP_Graphic

Include

sp_gr/SP_GR_Edge.h

SP_GR_Edge::m_pcSource

```
SP_Graphic* m_pcSource
```

Graphik, an der diese Kante beginnt.

SP_GR_Edge::m_pcTarget

```
SP_Graphic* m_pcTarget
```

Graphik, an der diese Kante endet.

SP_GR_Edge::SP_GR_Edge

```
SP_GR_Edge(SP_DS_Edge* p_pcDataParent)
```

Konstruktor, assoziiert dieses Objekt zu einem Objekt in der Datenstruktur und initialisiert SP_Graphic mit dem Typen SP_GRAPHIC_EDGE.

SP_GR_Edge::AddToCanvas

```
bool AddToCanvas(SP_GUI_Canvas* p_pcCanvas,
                 double p_nX = -1,
                 double p_nY = -1,
                 int p_nKeys = 0)
```

Neuimplementierung aus SP_Graphic, die im wesentlichen darauf achtet, dass nach den von der *OGL* geforderten Arbeitsschritten zum Anzeigen der Graphik sowohl Quell-, als auch Zielknoten über die neue Kante informiert werden, um die sogenannte Attachement-Position, also den Auftreffpunkt der Kante auf den Knoten zu berechnen.

SP_GR_Edge::Clone

```
virtual SP_Graphic* Clone(SP_Data* p_pcParent) = 0
```

Übernahme der rein virtuellen Schnittstelle aus SP_Graphic zur Implementierung innerhalb der Ableitungen.

SP_GR_Edge::GetLength

```
virtual double GetLength()
```

Liefert die gesamte Länge der Kante auf dem Weg über all ihre Kontrollpunkte und wird im Zusammenhang mit der PedAnimation verwendet.

SP_GR_Edge::GetPrimitive

```
virtual wxShape* GetPrimitive() = 0
```

Übernahme der rein virtuellen Schnittstelle aus SP_Graphic zur Implementierung innerhalb der Ableitungen.

SP_GR_Edge::GetSource

```
SP_Graphic* GetSource() const
```

Zugriffsmethode auf den Wert von m_pcSource.

SP_GR_Edge::GetTarget

```
SP_Graphic* GetTarget() const
```

Zugriffsmethode auf den Wert von m_pcTarget.

SP_GR_Edge::ReleaseSource

```
bool ReleaseSource()
```

Gibt den Knoten am Beginn der Kante frei, so dass diese im Anschluss mit diesem Ende „in der Luft hängt". Dabei erfolgt das Loslösen durch den Aufruf von SP_GR_Node::RemoveSourceEdge, was die Kante auch aus der Assoziation in der Datenstruktur entfernt.

Rückgabe

Liefert in der aktuellen Version in jedem Fall TRUE, da auch eine nicht gesetzte Variable in m_pcSource, zum Beispiel, als erfolgreicher Abschluss dieser Operation gesehen wird.

SP_GR_Edge::ReleaseTarget

```
bool ReleaseTarget()
```

Gibt den Knoten am Ende der Kante frei, so dass diese im Anschluss mit diesem Ende „in der Luft hängt". Dabei erfolgt das Loslösen durch den Aufruf von `SP_GR_Node::RemoveTargetEdge`, was die Kante auch aus der Assoziation in der Datenstruktur entfernt.

Rückgabe

Liefert in der aktuellen Version in jedem Fall `TRUE`, da auch eine nicht gesetzte Variable in `m_pcTarget`, zum Beispiel, als erfolgreicher Abschluss dieser Operation gesehen wird.

SP_GR_Edge::SetSource

```
SP_GR_Edge* SetSource(SP_Graphic* p_pcSource)
```

Zugriffsmethode zum Setzen des Wertes in `m_pcSource`. Liefert in jedem Fall das Objekt der Kante selbst zurück, also `this`.

SP_GR_Edge::SetTarget

```
SP_GR_Edge* SetTarget(SP_Graphic* p_pcTarget)
```

Zugriffsmethode zum Setzen des Wertes in `m_pcTarget`. Liefert in jedem Fall das Objekt der Kante selbst zurück, also `this`.

29 SP_GR_Node

Erweiterung der Schnittstelle aus SP_Graphic zur Verwendung für graphische Objekte, die für die Elemente der Knotenmenge stehen. Von dieser Klasse können keine Objekte instanziiert werden.

Abgeleitet von

`SP_Error`
`SP_Graphic`

Include

`sp_gr/SP_GR_Node.h`

SP_GR_Node::m_bFixedSize

```
bool m_bFixedSize
```

Flag, das festlegt, ob dieser Knoten in der Oberfläche vom Nutzer in der Grösse verändert werden kann.

SP_GR_Node::SP_GR_Node

```
SP_GR_Node(SP_DS_Node* p_pcDataParent)
```

Konstruktor, weisst diese Graphik einem Knoten in der Datenstrutkur zu und initialisiert `SP_Graphic` mit dem Typen `SP_GRAPHIC_NODE`.

SP_GR_Node::AddSourceEdge

```
virtual SP_Graphic* AddSourceEdge(SP_Graphic* p_pcEdge)
```

Fügt eine Kante hinzu, die an dieser Graphik beginnt.

Parameter

p_pcEdge

Graphische Ausprägung der hinzuzufügenden Kante.

Rückgabe

Liefert das Ergebnis des Aufrufs von `SP_GR_Edge::SetSource(this)` für den übergebenen Parameter oder `NULL`, wenn der Parameter ungültig oder zum Beispiel nicht vom Typ `SP_GRAPHIC_EDGE` ist.

SP_GR_Node::AddTargetEdge

```
virtual SP_Graphic* AddTargetEdge(SP_Graphic* p_pcEdge, SP_Data* p_pcTarget)
```

Fügt eine Kante hinzu, die an dieser Graphik endet.

Parameter

p_pcEdge

Graphische Ausprägung der hinzuzufügenden Kante.

Rückgabe

Liefert das Ergebnis des Aufrufs von `SP_GR_Edge::SetTarget(this)` für den übergebenen Parameter oder `NULL`, wenn der Parameter ungültig oder zum Beispiel nicht vom Typ `SP_GRAPHIC_EDGE` ist.

SP_GR_Node::Clone

```
virtual SP_Graphic* Clone(SP_Data* p_pcParent) = 0
```

Übernahme der rein virtuellen Schnittstelle aus `SP_Graphic` zur Implementierung innerhalb der Ableitungen.

SP_GR_Node::GetPrimitive

```
virtual wxShape* GetPrimitive() = 0
```

Übernahme der rein virtuellen Schnittstelle aus `SP_Graphic` zur Implementierung innerhalb der Ableitungen.

SP_GR_Node::RemoveSourceEdge

```
virtual bool RemoveSourceEdge(SP_Graphic* p_pcEdge, bool p_bInDs = TRUE)
```

Entfernt eine Kante, die an diesem Knoten beginnt. Dabei geht es hauptsächlich um die notwendigen Arbeitsschritte, um die Strukturen innerhalb der *OGL* zu bereinigen.

Parameter

p_pcEdge

Graphik der Kante, die entfernt werden soll.

p_bInDs

Gibt an, ob die Assoziationen innerhalb der Datenstruktur ebenfalls gelöst werden sollen.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst.

SP_GR_Node::RemoveTargetEdge

```
virtual bool RemoveTargetEdge(SP_Graphic* p_pcEdge, bool p_bInDs = TRUE)
```

Entfernt eine Kante, die an diesem Knoten endet. Dabei geht es hauptsächlich um die notwendigen Arbeitsschritte, um die Strukturen innerhalb der *OGL* zu bereinigen.

Parameter

p_pcEdge

Graphik der Kante, die entfernt werden soll.

p_bInDs

Gibt an, ob die Assoziationen innerhalb der Datenstruktur ebenfalls gelöst werden sollen.

Rückgabe

Liefert `TRUE` im Erfolgsfall, `FALSE` sonst.

SP_GR_Node::SetFixedSize

```
void SetFixedSize(bool p_bVal)
```

Zugriffsmethode zum Setzen des Wertes in `m_bFixedSize`.

30 SP_GR_TextAttribute

Graphische Ausprägung zur Verwendung für Textattribute. Genau genommen kann jedes Attribut mit dieser graphischen Ausprägung dargestellt werden, da sie die zwingend vorgeschriebene Implementierung von `SP_DS_Attribute::GetValueString` zur Erfragung des Wertes benutzt.

Abgeleitet von

`SP_GR_Attribute`

Include

`sp_gr/attributes/SP_GR_TextAttribute.h`

SP_GR_TextAttribute::m_pcPrimitive

```
SP_GR_BaseText* m_pcPrimitive
```

Konkretes Objekt zur Darstellung. `SP_GR_BaseText` ist dabei eine Ableitung von `wxTextShape` aus der *OGL*.

SP_GR_TextAttribute::m_sFormat

```
wxString m_sFormat
```

Formatdefinition. Im Format können für den eigentlichen Wert des darzustellenden Attributs weitere Zeichen definiert werden, die in die Darstellung einfliessen. Das Zeichen „`%`“ (Prozent) steht dabei für die Stellen, an denen der konkrete Wert des Attributs stehen soll. So erzeugt zum Beispiel eine Formatangabe von „`[ % ]`“ bei einem Attribut, das den Wert „`Text`“ enthält die Ausgabe „`[ Text ]`“.

SP_GR_TextAttribute::SP_GR_TextAttribute

```
SP_GR_TextAttribute(SP_DS_Attribute* p_pcParent,
                    const char* p_pchFormat = "%",
                    double p_nWidth = 0.0,
                    double p_nHeight = 0.0)
```

Konstruktor, erzeugt ein neues graphisches Attribut, Instanziiert m_pcPrimitive, speichert den zweiten Parameter in m_sFormat und vermerkt sich selbst und sein *wx*-Objekt im Mapping der externen Objekte aus der *OGL* zu den SNOOPY-Objekten.

SP_GR_TextAttribute::Clone

```
SP_Graphic* Clone(SP_Data* p_pcParent)
```

Erzeugt ein neues Objekt als Kopie seiner selbst und weist ihm als Elternelement den Parameter zu.

SP_GR_TextAttribute::FormatText

```
wxString FormatText()
```

Methode, die bei Änderungen des Wertes im Datenstrukturattribut verwendet wird, um die Anzeige zu synchroniesieren. Hier wird auch die Ersetzung des Platzhalters in der Formatdefinition vorgenommen.

SP_GR_TextAttribute::GetPrimitive

```
wxShape* GetPrimitive()
```

Zugriffsmethode auf den Wert in `m_pcPrimitive` als Objekt vom Basis-Typen der *OGL*.

31 SP_WDG_DialogBase

Basisklasse für alle Elemente, die zur Manipulation von Attributwerten im *Properties*-Dialog verwendet werden. Ableitung müssn mindestens die `Clone`-, die `AddToDialog`- und die `OnDlgOk`-Methode implementieren.

Der Aufbau des Dialogs erfolgt automatisch.

Abgeleitet von

`SP_Error`
`SP_Name`
`wxEvtHandler`

Include

`sp_gui/widgets/dialogs/SP_WDG_DialogBase.h`

SP_WDG_DialogBase::m_bMultiple

```
bool m_bMultiple
```

Flag, das angibt, ob mehrere Attribute mit diesem Element angezeigt und manipuliert werden. Das passiert, wenn mehrere Elemente derselben Partition selektiert wurden.

SP_WDG_DialogBase::m_pcShow

```
wxCheckBox* m_pcShow
```

Anzeigeelement zum an- und abschalten der Sichtbarkeit von Attributen.

SP_WDG_DialogBase::m_tlAttributes

```
SP_List<SP_DS_Attribute*> m_tlAttributes
```

Liste der konkreten Attribute, deren Werte mit diesem Element manipuliert werden sollen. Wenn die Anzahl der Elemente in dieser Liste grösser als eins ist, ist auch `m_bMultiple == TRUE`.

SP_WDG_DialogBase::SP_WDG_DialogBase

```
SP_WDG_DialogBase(const char* p_sPage = "Page 1")
```

Konstruktor, erzeugt ein neues Darstellungselement und speichert den Parameter in der Basisklasse `SP_Name`. Dieser Name gibt an, auf welcher Seite des Dialogs dieses Element dargestellt werden soll.

SP_WDG_DialogBase::AddShowFlag

```
bool AddShowFlag(SP_WDG_NotebookPage* p_pcPage,
                 wxBoxSizer* p_pcSizer,
                 SP_DS_Attribute* p_pcAttr)
```

Erzeugt eine Instanz für m_pcShow rechts neben dem Eingabefeld für den Wert des Attributs zur an- und abschaltung der Anzeige. Kann aus den Implementierungen der `AddToDialog`-Methoden der Ableitungen aufgerufen werden.

Parameter

p_pcPage

Seite des Dialogs auf der auch die Elemente zur Darstellung definiert wurden.

p_pcSizer

Konstrukt für das Layout des Dialogs aus *wxWidgets*. Diesem Element wird die Show-Checkbox mittels `wxSizer::Add` hinzugefügt.

p_pcAttr

Attribut der Datenstruktur, dessen Wert für die Anzeige zur Initialisierung der Checkbox verwendet wird.

SP_WDG_DialogBase::AddToDialog

```
virtual bool AddToDialog(SP_List<SP_DS_Attribute*>* p_ptlAttributes,
                         SP_DLG_ShapeProperties* p_pcDlg)
virtual bool AddToDialog(SP_List<SP_Graphic*>* p_plGraphics,
                         SP_DLG_ShapeProperties* p_pcDlg)
```

Methode, die bei der Konstruktion des Dialogs automatisch für alle Elemente zur Darstellung aufgerufen wird und in der die konkreten Fensterelemente zur Bearbeitung instanziiert werden sollen.

Nicht alle Ableitungen müssen beide Methoden implementieren.

Parameter

p_ptlAttributes

Liste der Attribute, für die dieses Widget verwendet wird. Der Inhalt wird in `m_tlAttributes` gespeichert um die vorgenommenen Änderungen beim Schliessen des Dialogs zurückschreiben zu können.

p_plGraphics

Liste der graphischen Ausprägungen, die für dieses Element ausgewählt wurden. Diese Methode wird nur aufgerufen, wenn es sich um ein Element zur Bearbeitung der graphischen Attribute (Vorder- und Hintergrund-Farbe) handelt.

p_pcDlg

Instanz des Dialogs, der die Anzeige der einzelnen Seiten verwaltet und jedem Widget, je nach gespeichertem Namen, entweder eine vorhandene Seite zuweist oder eine neue, des entsprechenden Namens erzeugt.

SP_WDG_DialogBase:: CleanUpAttributes

```
bool CleanUpAttributes()
```

Leert die Liste `m_tlAttributes` und überprüft ausserdem, ob einige Attribute gelöscht werden können, was zum Beispiel bei der Aktivierung des *Logic*-Flags notwendig wird, wenn einige Datenstrukturelement durch Vereinigung obsolet werden.

SP_WDG_DialogBase::Clone

```
virtual SP_WDG_DialogBase* Clone()
```

Erzeugt eine Kopie seiner selbst und sollte in allen Ableitungen implementiert werden, um die Instanziierung des richtigen Typen sicherzustellen.

SP_WDG_DialogBase::OnDlgOk

```
virtual bool OnDlgOk()
```

Methode, die beim Bestätigen des Dialogs mit „OK“ durch den Nutzer automatisch für alle Widgets aufgerufen wird und in der es die Aufgabe des Designers des Widgets ist, die aus seinen individuellen Fensterelementen gewonnenen Informationen in die Attribute zurückzuschreiben, die in `m_tlAttributes` gespeichert sind.

Jede abgeleitete Klasse kann durch Aufruf dieser Implementierung das Zurückschreiben des Wertes aus der Anzeige-Checkbox (`m_pcShow`) in alle graphischen Ausprägungen sicherstellen.

Wird diese Methode nicht mit `TRUE` beendet, wird auch der Dialog nicht geschlossen, und eventuell in `SP_Error` gespeicherte Fehlermeldungen werden zur Anzeige gebracht.

32 SP_WDG_DialogMultiline

Widget zur Bearbeitung von mehrzeiligen Textattributen in einem Dialog. Hier wird nur `AddToDialog` überschrieben, zur Erzeugung eines anderen Eingabefeldes als in der Basisklasse. `OnDlgOk` wird aus der Basisklasse verwendet.

Abgeleitet von

`SP_WDG_DialogText`

Include

`sp_gui/widgets/dialogs/SP_WDG_DialogMultiline.h`

SP_WDG_DialogMultiline::SP_WDG_DialogMultiline

```
SP_WDG_DialogMultiline(const char* p_sPage = "Page 1")
```

Konstruktor, initialisiert die Basisklasse mit dem Parameter für den Namen der Seite, auf der dieses Widget dargestellt werden soll.

SP_WDG_DialogMultiline::AddToDialog

```
virtual bool AddToDialog(SP_List<SP_DS_Attribute*>* p_ptlAttributes,
                         SP_DLG_ShapeProperties* p_pcDlg)
```

Implementierung nach den Vorgaben der Basisklasse. Hier wird `m_pcTextCtrl` der Basisklasse so instanziiert, dass es die Eingabe mehrzeiliger Texte erlaubt.

SP_WDG_DialogMultiline::Clone

```
virtual SP_WDG_DialogBase* Clone()
```

Erzeugt eine identische Kopie.

33 SP_WDG_DialogText

Widget zur Bearbeitung von einzeiligen Textattributen in einem Dialog.

Abgeleitet von

`SP_WDG_DialogBase`

Include

`sp_gui/widgets/dialogs/SP_WDG_DialogText.h`

SP_WDG_DialogText::m_pcTextCtrl

```
wxTextCtrl* m_pcTextCtrl
```

Element zur Eingabe von Text, wie es in *wxWidgets* verwendet wird.

SP_WDG_DialogText::SP_WDG_DialogText

```
SP_WDG_DialogText(const char* p_sPage = "Page 1")
```

Konstruktor, initialisiert die Basisklasse mit dem Parameter für den Namen der Seite, auf der dieses Widget dargestellt werden soll.

SP_WDG_DialogText::AddToDialog

```
virtual bool AddToDialog(SP_List<SP_DS_Attribute*>* p_ptlAttributes,
                         SP_DLG_ShapeProperties* p_pcDlg)
```

Implementierung nach den Vorgaben der Basisklasse. Hier wird eine Instanz für `m_pcTextCtrl` der vom Dialog `p_pcDlg` gelieferten Seite für den gespeicherten Namen hinzugefügt.

SP_WDG_DialogText::Clone

```
virtual SP_WDG_DialogBase* Clone()
```

Erzeugt eine identische Kopie.

SP_WDG_DialogText::OnDlgOk

```
virtual bool OnDlgOk()
```

Implementierung nach den Vorgaben der Basisklasse. Hier wird die Liste der Attribute, für die dieses Widget verwendet wird traversiert und alle Einträge bekommen mittels `SetValueString` den Inhalt von `m_pcTextCtrl` zugewiesen.

Die Behandlung des Anzeigeflags erfolgt durch Aufruf der Basisimplementierung von `OnDlgOk`.

Literaturverzeichnis

[1] - Balzert, Heide, *Methoden der objektorientierten Systemanalyse*, Spektrum Akademischer Verlag, 1999

[2] - Breymann, Ulrich, *Komponenten entwerfen mit der C++ STL*, Addison-Wesley-Longman, 1999

[3] - Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, M, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996

[4] - Heesch, Dimitri van, *Doxygen*, www.doxygen.org [URL]

[5] - Jungnickel, Dieter, *Graphen, Netzwerke und Algorithmen*, Spektrum Akademischer Verlag, 1994

[6] - Menzel, Thomas, *Entwurf und Impl. eines Frameworks zur Petri-Netz orientierten Modellierung*, 1997 [Diplomarbeit]

[7] - Microsoft Corporation, *Microsoft Developer Network*, msdn.microsoft.com [URL]

[8] - Reisig, Wolfgang, *Petrinetze: Eine Einführung*, Springer, 1986

[9] - Rumbaugh, James R.; Blaha, Michael R.; Lorensen, William; Eddy, Frederick;, *Objektorientiertes Modellieren und Entwerfen*, Hanser, 1993

[10] - Smart, Julian; Roebling, Robert; Zeitlin, Vadim; Dunn, Robin; et al, *wxWidgets: A portable C++ and Python GUI toolkit*, www.wxwidgets.org [URL]

[11] - Starke, Peter H., *Analyse von Petri-Netz-Modellen*, Teubner, 1990

[12] - Starke, Peter H., *INA Integrated Net Analyzer*, www.informatik.hu-berlin.de/~starke/ina.html [URL]

[13] - Stroustrup, Bjarne, *The C++-programming language*, Addison-Wesley-Longman, 1998

[14] - The Apache XML Project, *Xerces-C++, a validating XML Parser in a portable subset of C++*, xml.apache.org [URL]

[15] - Tittmann, Peter, *Graphentheorie, eine anwendungsorientierte Einführung*, Hanser, 2003