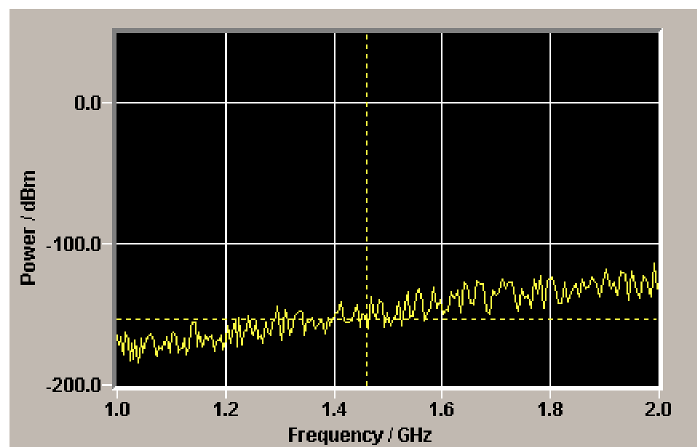
ZSDiagram



Das Subsystem *ZSDiagram* ist eine C++ Klassenbibliothek zur Ausgabe von Messwertverläufen innerhalb eines Diagramm-Fensters implementiert auf Basis der Qt Version 3.3 der Firma Trolltech.

Inhaltsverzeichnis

1. Einführung 3
2. Beispiele 11
2.1. Implementierung eines Diagramms 11
2.1.1. Diagramm-Widget 11
2.1.2. Skalierung der Achsen 12
2.1.3. Trace-Objekte 13
2.1.4. Sichtbare Diagramm-Objekte 14
2.1.5. Skalieren des Diagrams und Ausgabe von Messwerten zur Laufzeit 16
2.2. Implementierung einer Diagram Objekt Klasse 23
2.2.1. Anforderungen (Pflichtenheft) 23
2.2.2. Implementierung 24
3. Klassen- und Schnittstellenbeschreibungen *(not up to date)* 32
3.1. Klasse *CWdgtDiagram* 32
3.1.1. Übersicht 32
3.1.2. Konstruktor 33
3.1.3. Öffentliche Methoden 34

1. Einführung

Das Subsystem *ZSDiagram* ist eine C++ Bibliothek zur Ausgabe von Messwertverläufen innerhalb eines Diagramm-Fensters implementiert auf Basis der Qt Version 3.3 der Firma Trolltech.

Ein Diagramm im Sinne dieser C++ Bibliothek besteht immer aus einer Instanz der Klasse *CDataDiagram*, *CPixmapDiagram* oder *CWdgtDiagram,* zu der verschiedene Diagramm-Objekte zur Laufzeit hinzugefügt und entfernt werden können.

CDataDiagram ist die Basisklasse der Diagramm-Klassen und ist dann zu instanziieren, wenn das Diagramm keine sichtbaren Ausgaben vornehmen soll sondern lediglich dazu benutzt werden soll, um Messwertverläufe zu analysieren.

CPixmapDiagram ist von *CDataDiagram* abgeleitet und ist dann zu instanziieren, wenn das Diagramm zwar sichtbare Ausgaben vornehmen soll, diese aber auf das Speichern in einer Pximap beschränkt ist.

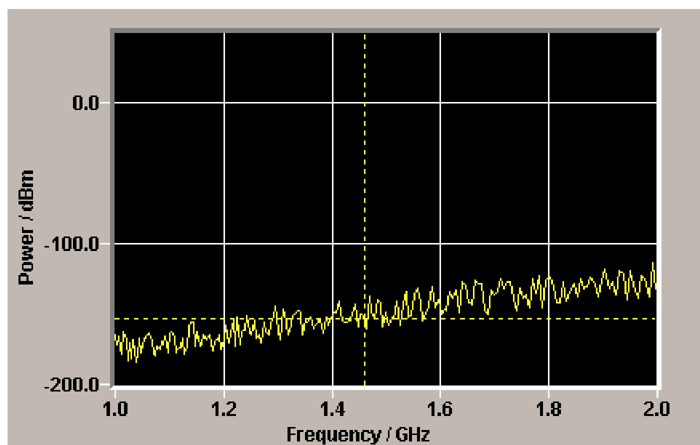
CWdgtDiagram ist von *CPixmapDiagram* und von *QWidget* abgeleitet und ist dann zu instanziieren, wenn das Diagramm sichtbare Ausgaben auf den Bildschirm vornehmen soll und der Benutzer mit dem Diagramm interagieren will (wie z.B. Marker mit der Maus packen und verschieben).

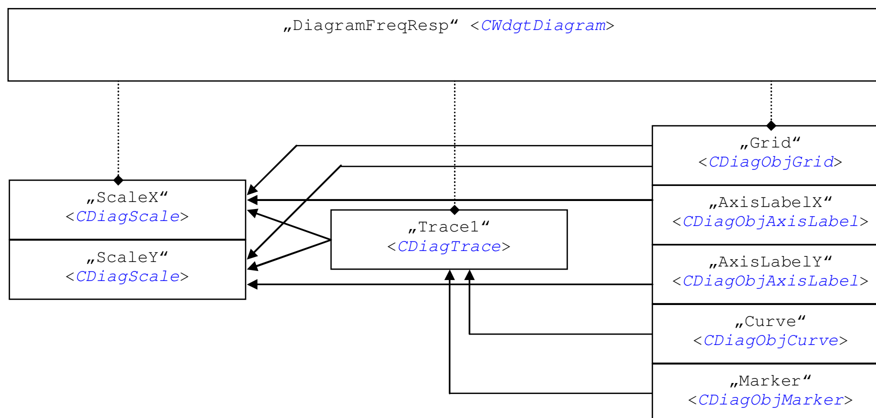
ZSDiagram kennt drei Gruppen von Diagram-Objekten:

1. Die erste Objekt-Gruppe sind die „Scale“-Objekte, die nicht sichtbare Skalierungen darstellen und im Wesentlichen aus einer Methodensammlung für kalkulatorische Aufgaben besteht, die in Diagrammen immer wieder benötigt werden. Unter diese „Dienste“, die „Scale“-Objekte zur Verfügung stellen, gehört u. a. die Unterteilung der Achsen in Trennlinien und die Umrechnung von Welt- in Bildschirmkoordinaten – also die Umrechnung physikalischer Werte in die entsprechenden Pixel-Werte.
2. Die zweite Objekt-Gruppe sind die „Trace“-Objekte, die ähnlich wie die „Scale“-Objekte für den Benutzer nicht sichtbar sind und auch im Wesentlichen aus einer Methodensammlung kalkulatorischer Aufgaben besteht. Unter diese „Dienste“ gehört u. a. die Analyse der Trace Daten wie z.B. die Ermittlung eines zu einem vorgegebenen X-Wertes (entweder als Welt- oder Bildschirmkoordinate vorliegend) gehörigen Y-Wertes. Für jeden sichtbaren „Trace“ (darzustellenden Messwertverlauf) eines Diagramms muss ein „Trace“-Objekt instanziiert werden. Jedes dieser „Trace“-Objekte ist mit einem X- als auch einem Y-„Scale“-Objekt zu verbinden, da die „Trace“-Objekte wiederum Dienste der „Scale“-Objekte verwenden.
3. Bei der dritten Objekt-Gruppe handelt es sich um die sichtbaren Diagram-Objekte, die zur Ausgabe der Achsenbeschriftungen, Ausgabe eines Grids, Wiedergabe der Messwerte und Abtasten des Messwertverlaufs dienen. Je nach Typ des sichtbaren Objekts ist es entweder mit „Scale“-Objekten oder mit „Trace“-Objekten zu verbinden, wobei eine Verbindung mit einem „Trace“-Objekt auch immer eine „indirekte“ Verbindung zu einem X- und Y-„Scale“-Objekt bedeutet. Alle Diagramm Objekte sind von der Klasse *CDiagObj* abgeleitet. Diese Basis-Klasse definiert im Wesentlichen die Schnittstelle, die von den Diagram-Klassen angesprochen wird, um die Events an die Objekte weiterzuleiten. Die Klassen *CDiagObjGrid*, *CDiagObjCurve*, *CDiagObjAxisLabel* und *CDiagObjMarker*

sind Spezialisierungen dieser Basis-Klasse und definieren erst das eigentliche Verhalten und optische Erscheinungsbild der Objekte. Da die Schnittstelle der Basisklasse *CDiagObj* ausreichend dokumentiert ist und ferner ein konkretes Beispiel für die Implementierung einer Diagram-Objekt-Klasse in diesem Dokument zu finden ist, ist es ohne weiteres möglich, eine anwendungsspezifischen, von *CDiagObj* abgeleitete Klasse implementieren und so die Gruppe der Diagramm-Objekt-Klassen zu erweitern.

Um die Zusammenhänge zu verdeutlichen zeigen die folgenden Bilder zunächst ein typisches Diagramm, das mit dem Subsystem ZSDiagram erzeugt wurde und anschließend eine Übersicht über die entsprechenden Diagramm-Objekte, die hierfür instanzieert wurden.





Wie obiges Bild zeigt verwaltet die Diagramm-Instanz eine Liste von *CDiagScale*, eine Liste von *CDiagTrace* und eine Liste von *CDiagObj* Instanzen. Jede *CDiagObj* Instanz ist mit wenigstens einem Scale Objekt verbunden. Dies muss aber nicht zwingend so sein, sondern hängt von der Spezialisierung des Diagram-Objekts ab. So beansprucht das Grid z.B. sowohl Dienste des X- als auch des Y-Scale-Objekts, da Grid-Linien an die Unterteilungsstriche beider Achsen gebunden sind, die von den Scale-Objekten berechnet werden. Ebenso delegiert das Curve- und das Marker-Objekt die Umrechnung von X/Y Messwerten in Bildschirmkoordinaten über das angebundene Trace-Objekt an das jeweilige Scale-Objekt. Die Axis-Label Objekte benötigen jeweils nur eines der beiden Scale-Objekte. Denkbar sind auch Objekte, die vielleicht gar keine Dienste von Scale- oder Trace-Objekten benötigen (wie z.B. Objekte zur Beschriftung des Diagramms).

Die von *CDataDiagram* abgeleiteten Klassen übernehmen die wichtigsten, administrativen Aufgaben. Die Diagramm Klasse verteilt empfangene Events (wie z.B. Mouse, Paint und Update Events) an die Scale-, Trace- und Diagramm-Objekte. Aber auch für die „Verteilung“ der Messwerte oder auch für die Änderung der Skalierungen zeichnen die von *CDataDiagram* abgeleiteten Klassen verantwortlich.

Die Abarbeitung der Events erfolgt dabei in den vier Prozesstiefen *Layout*, *Data*, *Pixmap* und *Widget*. Werden z.B. neue Skalierungs- und Messwerte empfangen und diese an das Diagramm übergeben, werden zunächst die neuen Skalierungs- und Messwerte an die Scale-, Trace- und Diagram-Objekte lediglich weitergegeben. Großartige Berechnungen soll das Diagramm (und damit die Objekte des Diagramms) zu diesem Zeitpunkt noch nicht durchführen. Durch *Invalidieren* sogenannter *Update-Flags* merken sich die Objekte nur, welche Prozesstiefe bei einem nachfolgenden *update* Aufruf zu aktualisieren ist.

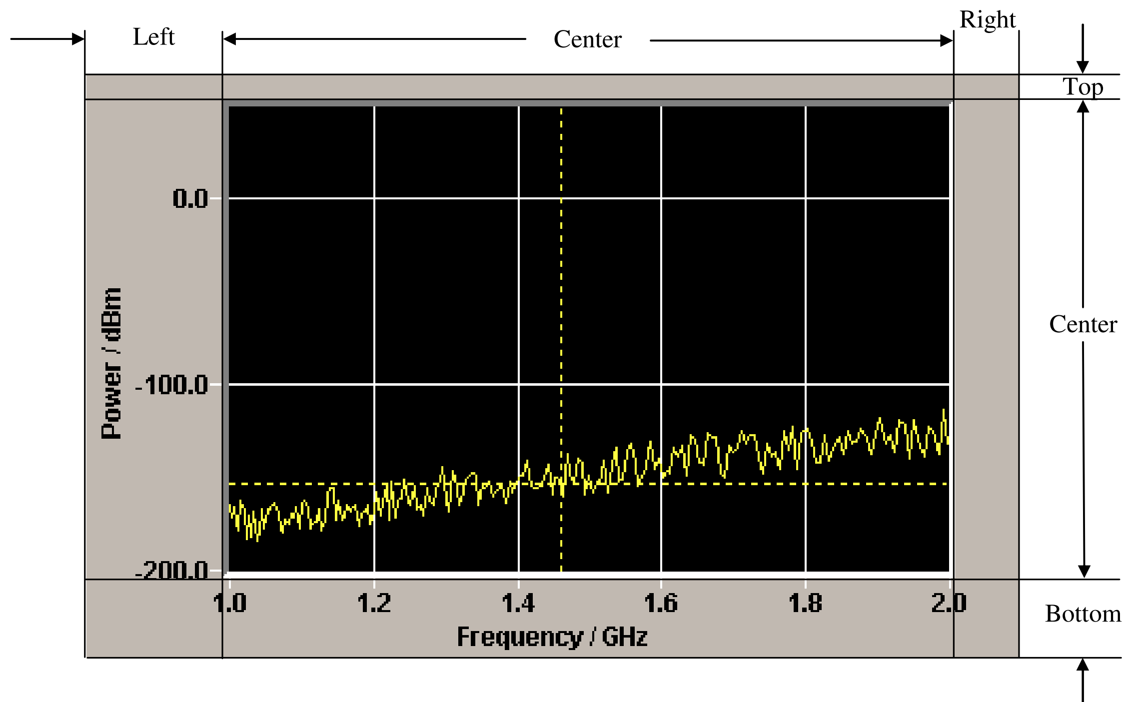
Wurden alle empfangenen Daten an das Diagramm übergeben und von dort an die Objekte verteilt, wird das Diagramm per *update* Aufruf dazu aufgefordert, seinen Inhalt zu aktualisieren. Durch diese „externe Update-Triggerung“ des Diagramms wird erreicht, dass das Diagramm nicht bereits unnötige Berechnungen durchführt, bevor nicht alle anliegenden Parameter-Änderungen an das Diagramm übermittelt wurden.

Den „externen Update-Request“ delegiert das Diagramm an seine einzelnen Objekte und geht dabei die einzelnen Prozesstiefen der Reihe nach durch. Innerhalb der Prozesstiefen *Layout* und *Data* geht das Diagramm die Objekte in der Reihenfolge durch, wie sie auch per *addDiagObj* Aufruf dem Diagramm hinzugefügt wurden.

Layout-Processing:

Bei Änderung der Skalierung, Ändern des Sichtbarkeitszustands eines Objekts oder nach einem „resizeEvent“ muss das Diagramm ggf. seine sichtbaren Objekte neu anordnen.

Ein Diagram-Fenster unterteilt sich in fünf verschieden Layout Bereiche. Jedes Diagram-Objekt kann nur innerhalb eines dieser fünf Layout-Bereiche positioniert werden, wobei sich die äußeren Bereiche überlappen. Das folgende Bild stellt die verschiedenen Layout-Bereiche dar.



Den *Center* Bereich betrachtet das Diagramm als in der Größe veränderbar und passt die Größe des Bereichs anhand der Wunschgröße der in den Rand-Bereichen liegenden Objekte an. Hierzu müssen zunächst die Scale-Objekte anhand der Abmessung des *Center* Bereichs und der gewählten Min/Max-Skalierung ihre internen Datenstrukturen aktualisieren und alle notwendigen Berechnungen durchführen, damit im weiteren Verlauf die Trace- und Diagramm-Objekte auf die Dienste der Scale-Objekte zurückgreifen können. Beim Layout-Processing berechnen die Scale-Objekte anhand der Skalierungswerte und der Geometrie-Daten die Unterteilung der Achsen in Achsen-Trennlinien (Division-Lines). Im folgenden Schritt werden die Trace-Objekte aufgefordert, ihre internen Datenstrukturen zu aktualisieren. Nachdem sowohl die Scale- als auch die Trace-Objekte ihre Daten berechnet haben, möchte das Diagramm von den sichtbaren Objekten deren „Wunschgröße" wissen und ruft hierfür die *sizeHint* Methode der Objekte auf. Diagramm-Objekte, die in den Randbereichen (*Left, Top, Right, Bottom*) des Diagrams liegen, müssen ihre internen Datenstrukturen beim Layout-Processing so weit aktualisieren, dass sie eine vernünftige Antwort auf den *sizeHint* Aufruf liefern können. Denn nur wenn das Diagramm die „Wunsch-Höhe" bzw. „Wunsch-Breite" der Objekte kennt, kann das Diagramm die Höhe und Breite der Randbereich bestimmen. Insbesondere für das Diagramm-Objekt *AxisLabel*, das die Beschriftung der Achsen vornimmt, bedeutet das, dass bereits beim Layout-Processing komplexe Berechnungen durchzuführen sind.

Muss das Diagramm aufgrund eines *sizeHint* Aufrufs die Größe wenigstens eines Randbereiches ändern, muss das *Layout*-Processing von neuem beginnen, da damit ja auch die Größe des *Center* Bereichs anzupassen ist und sich damit der Bereich für die Skalierung geändert haben kann, was wiederum eine Neu-Anordnung der Achsen-Trennlinien zur Folge haben könnte. Das *Layout*-Processing ist somit ein iterativer Vorgang, wobei die Anzahl der Iterationen auf eine sinnvolle Maximalzahl begrenzt ist, um ggf. bei Programmierfehlern innerhalb der Diagramm-Objekte einer Endlos-Schleife vorzubeugen.

Data-Processing:

Weder innerhalb des *Layout-Processings* noch innerhalb des *Data-Processings* kann und darf keines der Objekte irgendeine Zeichenroutine ausführen. Innerhalb des *Data-Processings* sollen die Objekte alle notwendigen Berechnungen und Datentransformationen durchführen, so daß nachfolgende Zeichenroutinen möglichst schnell durchgeführt werden können. So sollte z.B. ein Diagramm-Objekt zur Ausgabe des Messwertverlaufs (*Curve*) die in Welt-Koordinaten vorliegenden X/Y-Messwerte in die entsprechenden X/Y-Bildschirmkoorindaten konvertieren und für eine spätere Ausgabe zwischenspeichern.

Es ist denkbar, dass beim Ausführen des *Data-Processings* ein Objekt wiederum ein anderes invalidiert. Nur mal angenommen, Marker-Werte sollen in einer Tabelle innerhalb des Diagramms ausgegeben werden. Erst nach Ausführen des *Data-Processings* kennt der Marker seine resultierende Position. Die Tabelle würde nun erst nach Ausführen des *Data-Processings* von neuen Marker-Werten in Kenntnis gesetzt werden. Das könnte nun wiederum zur Folge haben, dass die Tabelle ihre Größe ändern möchte, weil der Wert sonst nicht mit der gewünschten Stellenzahl dargestellt werden kann. Damit ist auch das *Data-Processing* ein iterativer Vorgang, wobei auch hier die Anzahl der Iterationen auf eine sinnvolle Maximalzahl begrenzt ist, um ggf. bei Programmierfehlern innerhalb der Diagramm-Objekte einer Endlos-Schleife vorzubeugen. Mehr noch, falls ein Objekt innerhalb des *Data-Processings* die Prozesstiefe Layout invalidiert, wird erneut zum *Layout-Processing* zurückgesprungen.

Pixmap-Processing:

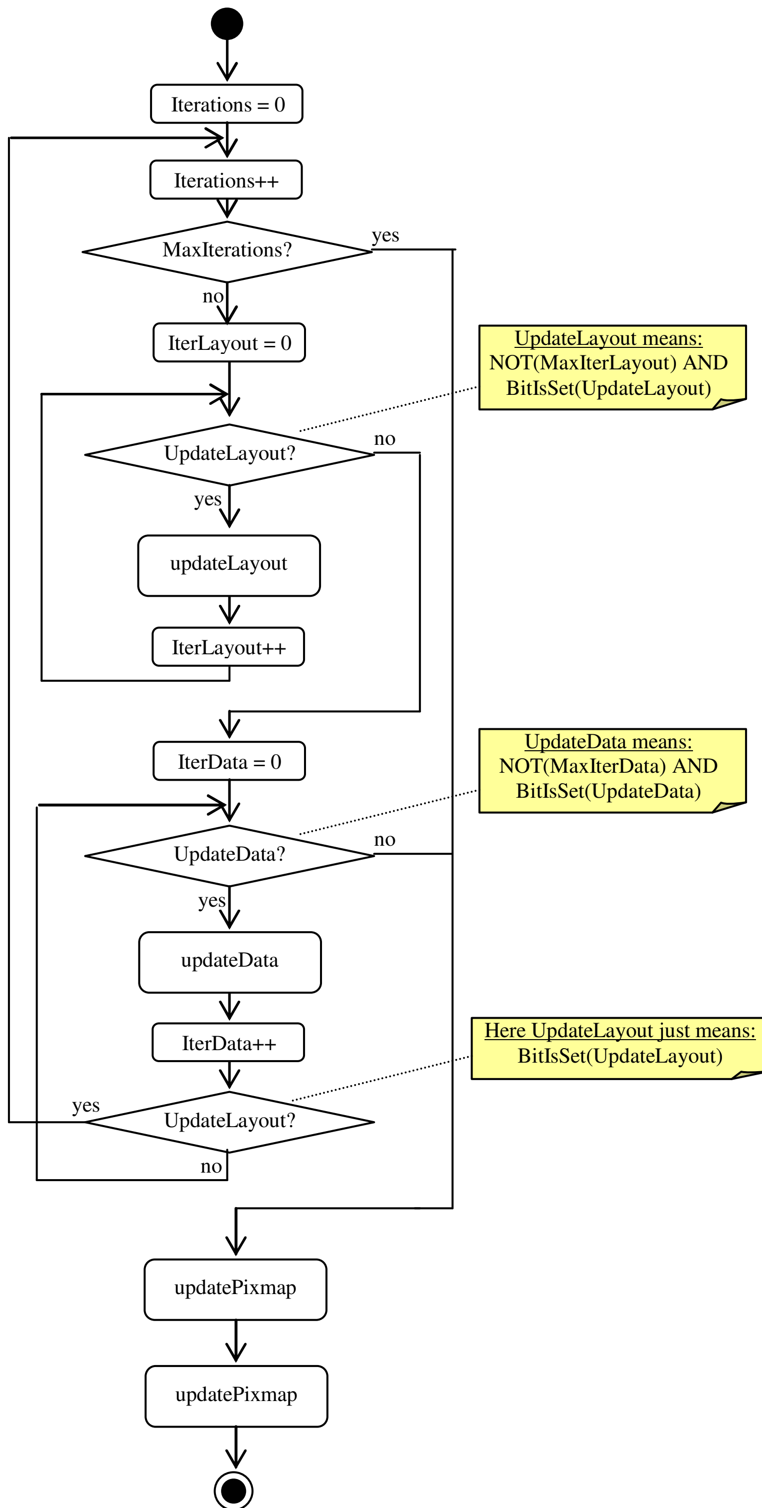
Für die Prozesstiefe stellt Diagramm den Objekten eine Pixmap zur Verfügung und fordert seine sichtbaren Diagramm-Objekte per *update* Aufruf auf, in diese Pixmap zu zeichnen.

Für das endgültige Erscheinungsbild des Diagrams ist es wichtig, welches Objekt sich zuerst in die Pixmap ausgibt, denn alle Objekte, die später malen, „überschreiben“ unter Umständen die grafischen Ausgaben des Objekts, das sich zuvor ausgegeben hat. Die Reihenfolge, wann welches Objekt gezeichnet und aber auch dazu aufgefordert wird, die Daten zu berechnen, entnimmt das Diagramm seiner Objekt-Liste. Das unterste (erste) Objekt der Liste wird immer auch als erstes behandelt. Natürlich kann es aber auch erforderlich sein, die Reihenfolge der Objekte zur Laufzeit ändern zu müssen. Wären z.B. mehrere Marker sichtbar, so soll immer der aktuell „aktive“ Marker zuletzt ausgegeben werden. Zur Änderung der Abarbeitungsreihenfolge innerhalb des *Pixmap-Prozessings* stellen die Diagramm-Klassen die notwendigen Schnittstellen zur Verfügung. Im Falle des beschriebenen Marker-Szenarios ist das Diagramm aber selbst intern in der Lage, das „editierte“ Objekt innerhalb der „Paint-Liste“ ans Ende zu setzen.

Widget-Processing:

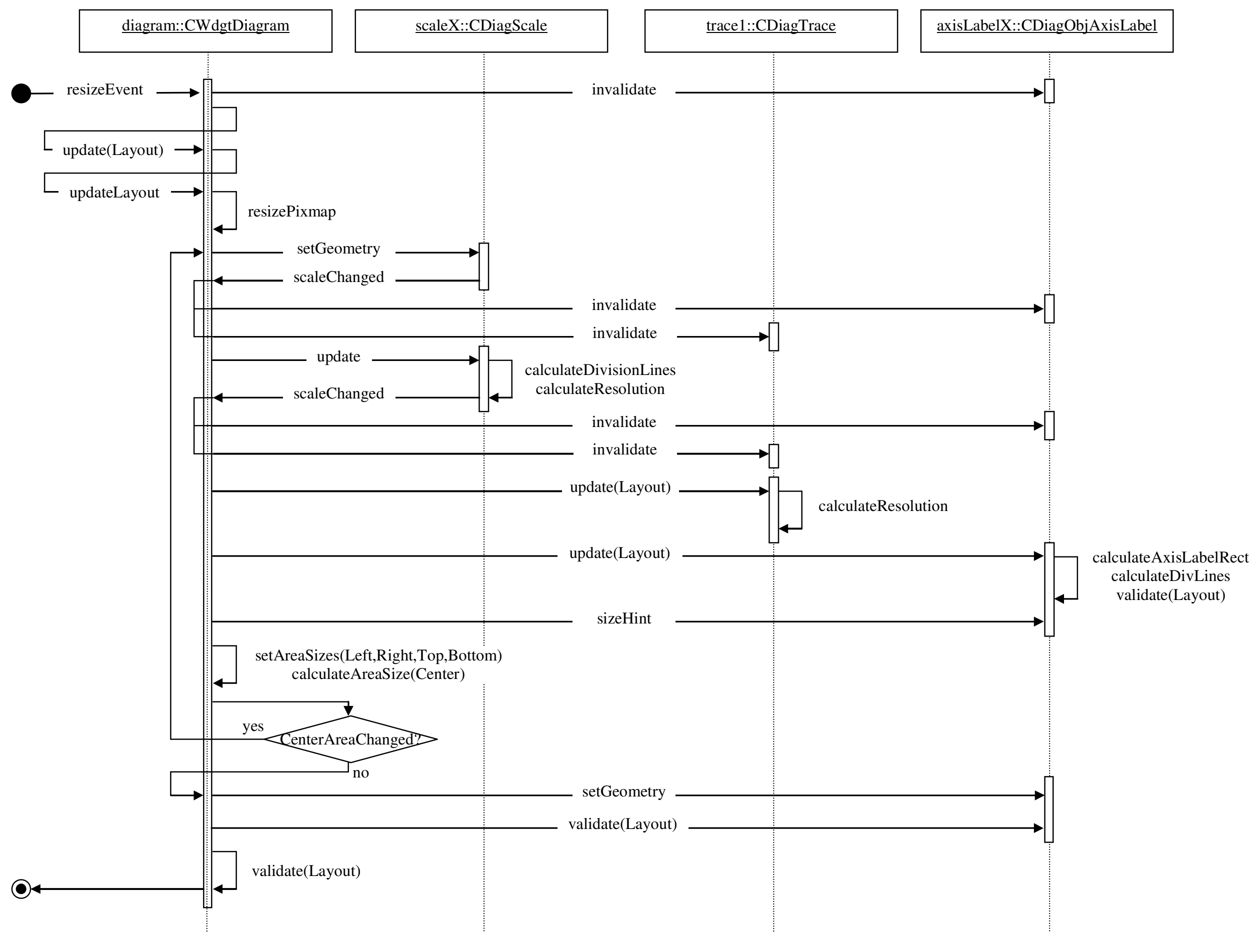
Im Falle einer von *CWdgtDiagram* abgeleiteten Klasse wird nach dem *Pixmap-Processing* zu einem späteren Zeitpunkt (wann, bestimmt die Event-Queue von Qt) die „paintEvent“ Methode von *CWdgtDiagram* aufgerufen. Das Widget-Processing beschränkt sich jedoch darauf, dass die einzelnen Diagramm-Objekte die von ihnen geänderten Rechtecksbereiche invalidieren. Die während des *Pixmap-Processings* erzeugte Pixmap wird vom Diagram per Bit-Blit Funktion in den sichtbaren Bildschirmbereich geschoben.

Ablaufdiagramm Diagramm Update

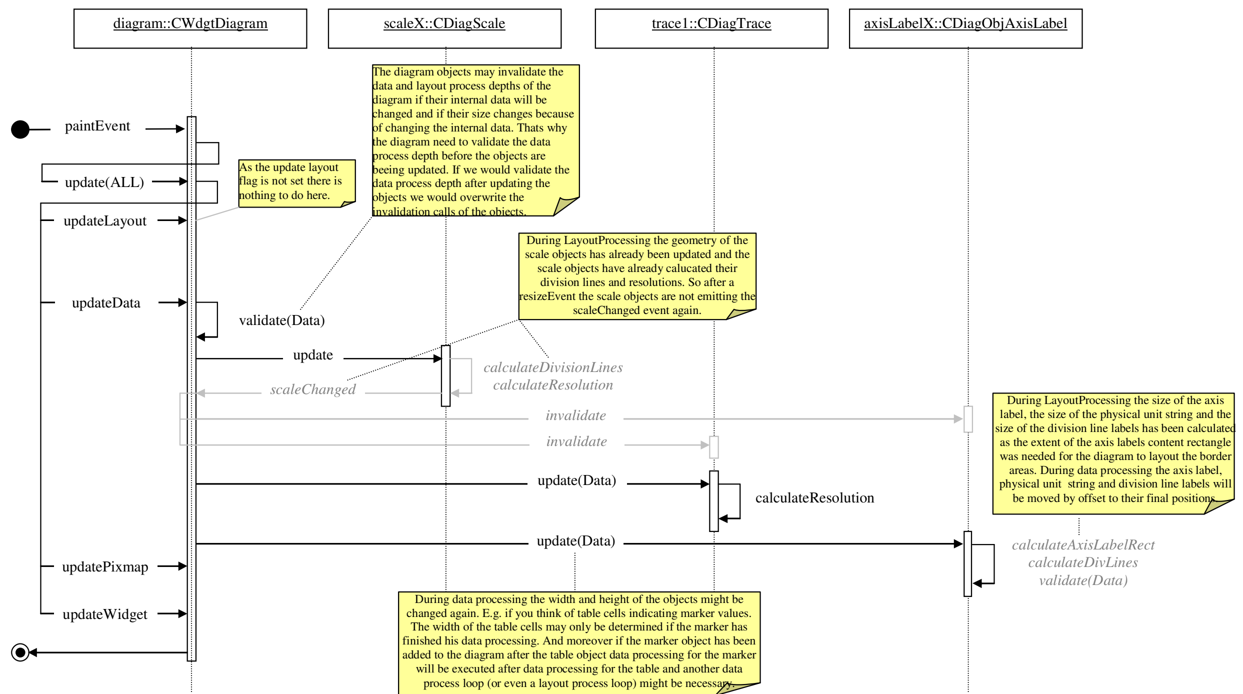


Sequenzdiagramm für *LayoutProcessing* nach „*resizeEvent*“

Unmittelbar auf den *resizeEvent* wird nur *LayoutProcessing* durchgeführt, da nach dem *resizeEvent* immer ein *paintEvent* auf das Diagram-Widget folgen wird. Da der *resizeEvent* unter Umständen mehrmals aufgerufen wird, bevor der *paintEvent* folgt (wegen interner Optimierungsmechanismen innerhalb Qt) ist es effizienter, beim *resizeEvent* lediglich die neuen Positionen und Größen der Diagramm-Objekte zu ermitteln und erst beim späteren *paintEvent* die internen Datenstrukturen aller Objekte berechnen und die Objekte in die neu erstellte Pixmap zeichnen zu lassen.



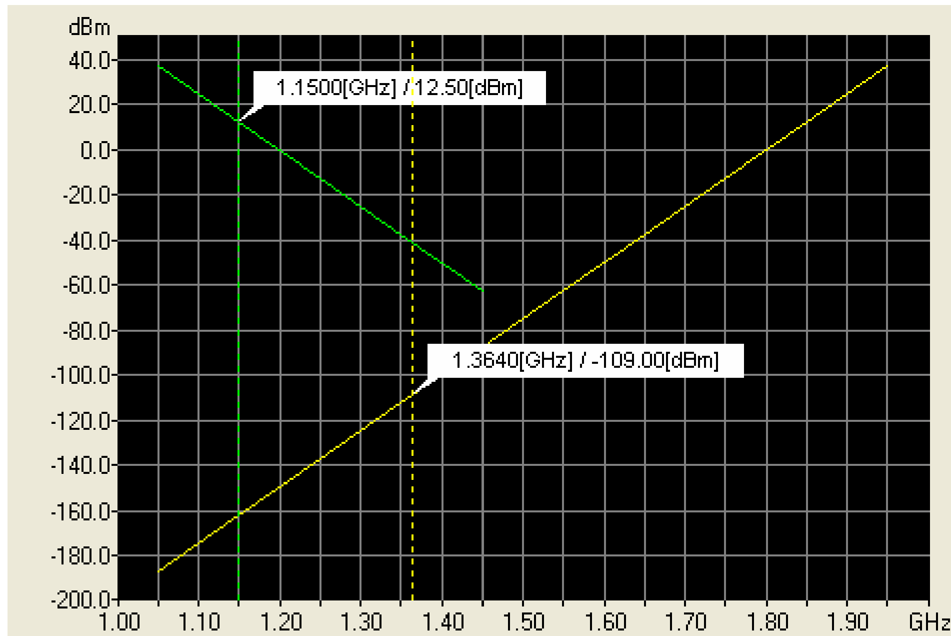
Sequenzdiagramm für *DataProcessing* nach „*paintEvent*“ (nach obigem „*resizeEvent*“)



2. Beispiele

2.1. Implementierung eines Diagramms

Innerhalb dieses Beispiels soll ein Diagramm erzeugt werden das zwei Traces (Kurvenverläufe) wiedergibt und deren Kurven mit Markern abgetastet werden können:



2.1.1. Diagramm-Widget

Die erste Aufgabe besteht darin, eine Instanz der Klasse *CWdgtDiagram* zu erzeugen:

```
m_pDiagram = new CWdgtDiagram(
    /* pWdgtParent */ this,
    /* strObjName  */ "Diagram" );
```

Mit obigem Konstruktor-Aufruf würde ein Diagram-Fenster erzeugt, dessen oberer Rand in der Höhe variabel wäre und von der Y-Achsenbeschriftung abhängt. Auch für den rechten Rang gilt, dass seine Breite von der X-Achsenbeschriftung abhängt. Das ist meist unerwünscht und in den meisten Fällen sollen diese Randbereiche konstant hoch bzw. breit gehalten werden. Aber auch für den linken und unteren Bereich ist meist nicht erwünscht, dass sich bei Änderung der Skalierung deren Breite und/oder Höhe ändert, weil nun mehr oder weniger gültige Stellen für die korrekte und eindeute Skalierung der Achsen erforderlich ist.

Deshalb bietet die Klasse Methoden an, um eine minimale Höhe für den oberen und unteren sowie eine minimale Breite für den linken und rechten Randbereich festzulegen:

```
m_pDiagram->setMinimumHeightPartTop(12);
m_pDiagram->setMinimumHeightPartBottom(12);
m_pDiagram->setMinimumWidthPartLeft(64);
m_pDiagram->setMinimumWidthPartRight(30);
```

Da Marker einen Fokus-Rahmen zeigen sollen, wenn man die Maus darüber hinwegbewegt, ist mit nachfolgendem Aufruf das *Mouse-Tracking* zu aktivieren:

```
m_pDiagram->setMouseTracking(true);
```

Mit nachfolgendem Aufruf wird eine schwarze Hintergrundfarbe für das Ausgabefenster der Messwertverläufe (*Center*-Bereich) gewählt:

```
m_pDiagram->setColBgPartCenter(Qt::black);
```

2.1.2. Skalierung der Achsen

Als nächstes sind die Scale-Objekte zu erzeugen und dem Diagramm hinzuzufügen.

```
SScale scaleX(
    /* physSize */ EPhysSizeFrequency,
    /* physUnit */ IdEUnitHz,
    /* fMin     */ 1.0e9,
    /* fMax     */ 2.0e9,
    /* fRes     */ 1.0);
SScale scaleY(
    /* physSize */ EPhysSizeElectricalPower,
    /* physUnit */ IdEUnitDbm,
    /* fMin     */ -200.0,
    /* fMax     */ 50.0,
    /* fRes     */ 0.01);

m_pDiagScaleX = new CDiagScale(
    /* strObjName */ "DiagScaleX",
    /* iScaleId   */ -1,
    /* scaleDir   */ EScaleDirX,
    /* scale      */ scaleX );
m_pDiagScaleX->setMainDivLineDistMinPix(20);
m_pDiagram->addDiagScale(m_pDiagScaleX);

m_pDiagScaleY = new CDiagScale(
    /* strObjName */ "DiagScaleY",
    /* iScaleId   */ -1,
    /* scaleDir   */ EScaleDirY,
    /* scale      */ scaleY );
m_pDiagScaleY->setMainDivLineDistMinPix(20);
m_pDiagram->addDiagScale(m_pDiagScaleY);
```

In der Regel sind immer genau zwei Scale-Objekte erforderlich. Eines für die X und eines für die Y Achse. Diagram unterstützt derzeit lineare und eine logarithmische Skalierungen von Achsen. Allerdings gilt die Einstellung ob logarithmisch oder linear immer für alle Skalierungen innerhalb einer *ScaleDir*. Deshalb ist das *Spacing* nicht eine Eigenschaft der Scale-Objekte sondern des Diagramms.

Innerhalb einer *ScaleDir* muss die *ScaleId* für jedes Scale-Objekt, das dem Diagramm hinzugefügt wird, eindeutig sein da u.a. die Zuordnung der Trace-Objekte als auch der Diagramm-Objekte zu den Scale-Objekten anhand der *ScaleDir* in Verbindung mit der *ScaleId* geschieht.

Es besteht die Möglichkeit, die Achsen in Hauptlinien und Sublinien zu unterteilen. Welchen Abstand die Haupt- und Sublinien mindestens voneinander haben sollen, wird über *setMainDivLineDistMinPix* bzw. *setSubDivLineDistMinPix* Aufrufe festgelegt.

Im obigen Beispiel ist lediglich eine Untereilung in Hauptlinien gewünscht, weshalb der entsprechende Methoden-Aufruf für den Minimal-Abstand der Sublinien entfällt (für weitere Informationen zur Unterteilung der Achsen in Haupt- und Sublinien siehe Beschreibung der Klasse *CDiagScale*).

Beim Konstruieren der Scale-Objekte muss eine gültige Default-Skalierung übergeben werden, in der zumindest die voreingestellte physikalische Größe und Einheit für die Achsenskalierungen festgelegt werden muss. Falls die Skalierungswerte nicht bereits zum Zeitpunkt des Konstruierens des Diagrams bekannt sind sondern erst später z.B. von einer Datenbank eingelesen werden, muss Diagram aber wenigstens bis dahin in der Lage sein, ein einigermaßen sinnvolles Diagram wiederzugeben. Hierzu gehört die Beschriftung der Achsen mit den jeweiligen Einheiten.

Denkbar sind auch Anwendungen, in denen mehr als nur eine Skalierung pro Achse erwünscht ist. Während dies bei der Y-Achse häufiger der Fall ist, sind Anwendungen mit mehr als einer X-Achsenskalierung exotisch zu nennen. Folgende Anwendung wäre aber denkbar und könnte mit dem Subsystem *ZSDiagram* verwirklicht werden:

Nehmen wir an, es besteht die Notwendigkeit, den Verlauf von Wetterwerten über mehrere Jahre hinweg pro Jahr miteinander zu vergleichen. Hierzu wäre für jedes Jahr ein eigenes X-Scale Objekt sowie für jedes Jahr eine eigenes *DiagObjCurve*-Objekt zu erzeugen und die Curve Objekte sind mit dem entsprechenden X-Scale Objekt zu verbinden. Zusätzlich müsste man ein X-Scale Objekt anlegen, das die X-Achse vom 1. Jan. bis 31. Dez. skaliert sowie eine Achsenbeschriftung implementieren, die das Datum ohne die Jahreszahlen ausgibt. Trägt man dafür Sorge, dass der „Range“ der Scale-Objekte immer gleich ist, so werden die jährlichen Verläufe der Wetterwerte „übereinander“ gelegt und wären somit einfach miteinander zu vergleichen.

2.1.3. Trace-Objekte

Im nächsten Schritt gilt es, zwei Trace-Objekte zu instanziieren und dem Diagram hinzuzufügen.

```
m_pDiagTrace0 = new CDiagTrace(
    /* strObjName  */ "DiagTrace0",
    /* iTraceId    */ 0,
    /* pDiagScaleX */ m_pDiagScaleX,
    /* pDiagScaleY */ m_pDiagScaleY );
m_pDiagram->addDiagTrace(m_pDiagTrace0);

m_pDiagTrace1 = new CDiagTrace(
    /* strObjName  */ "DiagTrace1",
    /* iTraceId    */ 1,
    /* pDiagScaleX */ m_pDiagScaleX,
    /* pDiagScaleY */ m_pDiagScaleY );
m_pDiagram->addDiagTrace(m_pDiagTrace1);
```

Die *TraceId* muss für jedes Trace-Objekt, das dem Diagramm hinzugefügt wird, eindeutig sein. Werden Daten an das Diagramm z.B. über einen Speicherblock übergeben, erfolgt die Zuordnung der Daten an die Trace-Objekte anhand dieser *TraceId*.

2.1.4. Sichtbare Diagramm-Objekte

2.1.4.1. Grid

Normalerweise ist das „unterste“ Objekt eines Diagramms ein Grid, dessen Raster-Linien von den anderen Diagram-Objekten überlagert werden. Eine Instanz der Klasse *CDiagObjGrid* ist somit in der Regel immer das erste sichtbare Diagramm-Objekt, das einem Diagramm hinzugefügt wird.

```
m_pDiagObjGrid = new CDiagObjGrid(
    /* strObjName  */ "DiagObjGrid",
    /* pDiagScaleX */ m_pDiagScaleX,
    /* pDiagScaleY */ m_pDiagScaleY );
m_pDiagram->addDiagObj(m_pDiagObjGrid);
```

Da das Grid zum Erzeugen und Malen der Rasterlinien die Messwerte nicht benötigt, sondern vielmehr die Unterteilung der X- und Y-Achsen kennen muss, ist die Grid-Instanz an die beiden X- und Y-Skalierungsobjekte und nicht mit dem *Trace-Objekt* verbunden. Die *Scale-Objekte* berechnen die Unterteilung der Achsen in „Division-Lines“. Diesen „Dienst“ nimmt das Grid in Anspruch, weshalb es eine Referenz sowohl auf das X-Scale-Objekt als auch eine Referenz auf das Y-Scale-Objekt benötigt. Das vorliegende Grid gibt nur „Haupt“-Grid-Linien aus.

2.1.4.2. AxisLabel

Um die X- und Y-Achse mit den Trennlinien sowie der Achsenbeschriftung auszugeben, sind Instanzen der Klasse *CDiagObjAxisLabel* zu instanziieren.

```
m_pDiagObjAxisLabelX = new CDiagObjAxisLabel(
    /* strObjName  */ "DiagObjAxisLabelX",
    /* pDiagScaleX */ m_pDiagScaleX,
    /* layoutPos   */ ELayoutPosBottom );
m_pDiagram->addDiagObj(m_pDiagObjAxisLabelX);

m_pDiagObjAxisLabelY = new CDiagObjAxisLabel(
    /* strObjName  */ "DiagObjAxisLabelY",
    /* pDiagScaleX */ m_pDiagScaleY,
    /* layoutPos   */ ELayoutPosLeft );
m_pDiagram->addDiagObj(m_pDiagObjAxisLabelY);
```

Um das Erscheinungsbild der Achsen-Objekte zu modifizieren, stellt die Klasse CDiagObjAxisLabel einen Satz Methoden zur Verfügung. Es können z.B. der Font und die Schriftgröße, die Farben für die Trennlinien oder aber die Position der Achsenlinien sowie der Texte für Achsenskalierung relativ zum Diagramm-Ausgabebereich festgelegt werden.

Zur Positionierung der Trennlinien sowie der Achsenbeschriftungen benötigt das X-Axis-Label Objekt eine Referenz auf das X-Scale-Objekt und das Y-Axis-Label Objekt eine Referenz auf das Y-Scale-Objekt. Dabei verwenden die Label-Objekte denselben Dienst wie auch das Grid zum Positionieren der Rasterung. Auf diese Weise wird gewährleistet, dass die Trennlinien der Achsenbeschriftung exakt mit den Rasterlinien des Grids übereinstimmen und nicht etwa durch Rundungsfehler auseinander laufen.

Mit *ELayoutPosBottom* wird für die X-Achse festgelegt, dass diese unterhalb des Diagrams ausgegeben werden soll. Die Y-Achse wird dagegen mit *ELayoutPosLeft* auf der linken Seite des Diagramms positioniert.

2.1.4.3. Curve

Das „zentrale“ Diagram-Objekt ist das Curve-Objekt, das die Werte innerhalb des Diagrams als Kurvenverlauf darstellt. Da zwei *Traces* auszugeben sind, brauchen wir auch zwei *Curve*-Objekte:

```
m_pDiagObjCurve0 = new CDiagObjCurve(
    /* strObjName */ "DiagObjCurve0",
    /* pDiagTrace */ m_pDiagTrace0 );
m_pDiagram->addDiagObj(m_pDiagObjCurve0);

m_pDiagObjCurve1 = new CDiagObjCurve(
    /* strObjName */ "DiagObjCurve1",
    /* pDiagTrace */ m_pDiagTrace1 );
m_pDiagram->addDiagObj(m_pDiagObjCurve1);
```

Da das *Curve-Objekt* den Messwertverlauf wiedergeben soll, benötigt das *Curve-Objekt* eine Referenz auf das *Trace-Objekt*.

2.1.4.4. Marker

Das Objekt in einem Diagram, das in der Regel als letztes Objekt ausgeben wird und dabei die optischen Ausgaben aller anderen Objekte überlagert, ist der sog. *Marker*, über den der wiedergegebene Kurvenverlauf abgetastet werden kann. Da beide *Traces* abzutasten sind, werden auch zwei *Marker* benötigt, die mit dem abzutastenden *Trace* zu verbinden sind.

```
m_pDiagObjMarker0 = new CDiagObjMarker(
    /* strObjName */ "DiagObjMarker0",
    /* pDiagTrace */ m_pDiagTrace0 );

m_pDiagObjMarker1 = new CDiagObjMarker(
    /* strObjName */ "DiagObjMarker1",
    /* pDiagTrace */ m_pDiagTrace1 );
```

Ein wie in obigem Beispiel erzeugter Marker ist jedoch nicht sichtbar, selbst wenn er dem Diagramm hinzugefügt wird. Ein Marker kann aus verschiedenen graphischen Elementen zusammengesetzt werden, wie einer vertikalen Linie, einem Image zur Anzeige seiner Position oder einem Tool-Tip, in dem der aktuelle Wert ausgegeben wird. Per Default ist keines dieser Elemente vorhanden. Die Elemente müssen erst erzeugt, dem Marker übergeben und mit einem *showElement* Aufruf sichtbar geschaltet werden. In unserem Beispiel bestehen beiden Marker aus einer gepunkteten, vertikalen Linie sowie einem Element, um den aktuellen Wert in Form eines Tool-Tips auszugeben:

```
pLineStyle = new SLineStyle(Qt::yellow,Qt::DotLine,1);
m_pDiagObjMarker0->setLineStyleVer(EDiagObjStateCount,pLineStyle);
pLineStyle = NULL; //lint !e423
m_pDiagObjMarker0->showElement(EDiagObjStateCount,CDiagObjMarker::EElementLineVer);

pToolTipStyle = new CToolTipStyle(
    /* colFg         */ Qt::black,
    /* colBg         */ Qt::white,
    /* fnt           */ fntToolTip,
    /* physUnitXVal  */ IdEUnitNone, // IdEUnitNone means "use best unit"
    /* physUnitYVal  */ IdEUnitNone, // IdEUnitNone means "use best unit"
    /* cxOffs        */ 8,
    /* cyOffs        */ 8,
    /* pFrameStyle   */ NULL,
    /* iMarginTop    */ 0,
    /* iMarginBottom */ 0,
    /* iMarginLeft   */ 0,
    /* iMarginRight  */ 0,
```

```
    /* cxArrowWidth  */ 6 );
m_pDiagObjMarker0->setToolTipStyle(EDiagObjStateCount,pToolTipStyle);
pToolTipStyle = NULL;
m_pDiagObjMarker0->showElement(EDiagObjStateCount,CDiagObjMarker::EElementToolTip);

pLineStyle = new SLineStyle(Qt::yellow,Qt::DotLine,1);
m_pDiagObjMarker1->setLineStyleVer(EDiagObjStateCount,pLineStyle);
pLineStyle = NULL;
m_pDiagObjMarker1->showElement(EDiagObjStateCount,CDiagObjMarker::EElementLineVer);

pToolTipStyle = new CToolTipStyle(
    /* colFg         */ Qt::black,
    /* colBg         */ Qt::white,
    /* fnt           */ fntToolTip,
    /* physUnitXVal  */ IdEUnitNone, // IdEUnitNone means "use best unit"
    /* physUnitYVal  */ IdEUnitNone, // IdEUnitNone means "use best unit"
    /* cxOffs        */ 8,
    /* cyOffs        */ 8,
    /* pFrameStyle   */ NULL,
    /* iMarginTop    */ 0,
    /* iMarginBottom */ 0,
    /* iMarginLeft   */ 0,
    /* iMarginRight  */ 0,
    /* cxArrowWidth  */ 6 );
m_pDiagObjMarker1->setToolTipStyle(EDiagObjStateCount,pToolTipStyle);
pToolTipStyle = NULL;
m_pDiagObjMarker1->showElement(EDiagObjStateCount,CDiagObjMarker::EElementToolTip);
```

Die *set..Style* und *showElement* Methoden verlangen einen Übergabeparameter vom Typ *EDiagObjState*. Dieser Parameter legt fest, in welchem Zustand das Element sichtbar ausgegeben werden soll. So wäre es z.B. denkbar, dass der Tool-Tip nur dann ausgegeben wird, wenn der Marker fokussiert oder editiert, als mit der Maus bewegt wird. Mit *EDiagObjStateCount* wurde festgelegt, dass die Elemente in allen Marker-Zuständen sichtbar sein sollen.

Ein Marker ist in der Regel immer direkt mit einem Kurvenverlauf verbunden. Deshalb wäre intuitiv anzunehmen, dass das *Marker-Objekt* eine Referenz auf das *Curve-Objekt* erhält. In Wirklichkeit aber „arbeitet" das *Marker-Objekt* aber nur auf denselben Daten, wie auch das Curve-Objekt - nämlich den auszugebenden X/Y-Wertepaaren. Und diese werden zentral durch das Trace-Objekt gehalten.

Wurden die graphischen Elemente des Markers konfiguriert, muss dieser noch dem Diagramm hinzugefügt werden:

```
m_pDiagram->addDiagObj(m_pDiagObjMarker0);
m_pDiagram->addDiagObj(m_pDiagObjMarker1);
```

Falls einem die aktuell bereitgestellten Attribute und Gestaltungsmöglichkeiten des Markers nicht ausreichen, spricht eigentlich nichts dagegen – außer dem zusätzlichen Aufwand – selbst eine Spezialisierung der Klasse *CDiagObjMarker* zu implementieren.

2.1.5. Skalieren des Diagrams und Ausgabe von Messwerten zur Laufzeit

Fertig ist das Diagram. Aber es wird noch "leer" und mit einer ungültigen Skalierung erscheinen. Wie das Diagramm mit Werten „gefüllt" und zur Laufzeit skaliert wird, wird im Folgenden beschrieben.

Um Messwerte zu aktualisieren, die Skalierung zu ändern oder Parameter des Diagramms zu ändern, bieten sowohl die Diagramm-Klasse als auch die Objekt-Klassen geeignete Schnittstellen an. So können z.B. X/Y-Werte durch Aufruf der Methode *copyValues* an die Trace-Objekte übergeben werden, wie in nachfolgendem Code-Fragment verdeutlicht:

```
double fXScaleRange = scaleX.m_fMax - scaleX.m_fMin;
double fXScaleOffs  = fXScaleRange/10.0;
double fYScaleRange = scaleY.m_fMax - scaleY.m_fMin;
double fYScaleOffs  = fYScaleRange/10.0;
int    idxVal;

double arfXValuesTrace0[10];
double arfYValuesTrace0[10];
double arfXValuesTrace1[5];
double arfYValuesTrace1[5];

for( idxVal = 0; idxVal < 10; idxVal++ )
{
    arfXValuesTrace0[idxVal] = scaleX.m_fMin + (idxVal+0.5)*fXScaleOffs;
    arfYValuesTrace0[idxVal] = scaleY.m_fMin + (idxVal+0.5)*fYScaleOffs;
}
for( idxVal = 0; idxVal < 5; idxVal++ )
{
    arfXValuesTrace1[idxVal] = scaleX.m_fMin + (idxVal+0.5)*fXScaleOffs;
    arfYValuesTrace1[idxVal] = scaleY.m_fMax - (idxVal+0.5)*fYScaleOffs;
}

m_pDiagTrace0->copyValues(EScaleDirX,10,arfXValuesTrace0);
m_pDiagTrace0->copyValues(EScaleDirY,10,arfYValuesTrace0);

m_pDiagTrace1->copyValues(EScaleDirX,5,arfXValuesTrace1);
m_pDiagTrace1->copyValues(EScaleDirY,5,arfYValuesTrace1);
```

Wie bereits erwähnt sind Datenerfassung und Benutzeroberfläche häufig voneinander entkoppelt und die Daten müssen oft über Thread-, Prozess- oder gar Netzwerkgrenzen hinweg übertragen werden. Damit die Daten nicht „seriell“ am Diagram eintreffen und so zu unnötigen Rechenschritten und „Zwischen-Aktualisierungen“ am Bildschirm führen, sollten immer alle geänderten Daten auf einmal übertragen werden.

Speziell für diesen Anwendungsfall besteht die Möglichkeit, einen Speicherblock vom Typ *SMemBlockDiagram* zu reservieren, diesen Speicherblock entsprechend zu füllen und dann den Zeiger auf den Speicherblock an das Diagram zu übergeben.

Das Reservieren und Füllen eines Diagram-Speicherblocks ist nicht ganz trivial und deshalb wird dieser Vorgang weiter unten separat durch einen Beispielcode verdeutlicht.

Rin solcher Diagram Speicherblock kann sowohl die Achsenskalierungen als auch die auszugebenden Messwerte und noch zusätzliche Informationen enthalten. Die Verwendung eines solchen Speicherblocks bietet sich z.B. dann an, wenn man eine Art „Auto-Ranging“ implementieren will. Beim „Auto-Ranging“ sind Minimal- und Maximalwert der Skalierung von den Messwerten abhängig. Mit anderen Worten, ändern sich die Messwerte, ändert sich ggf. auch die Skalierung. Es wäre ein Unding, wenn die Datenbank alle Einzelparameter getrennt an das Diagramm übertragen würde. Dann müsste das Diagramm für jeden Einzelparameter komplett neu aufgebaut werden.

Einen Speicherblock übergibt man an das Diagramm über den Methodenaufruf *setMemBlockValues*:

```
SMemBlockDiagram* pMemBlockDiagram = RECEIVE_DATA();
m_pDiagram->setMemBlockValues(pMemBlockDiagram);
m_pDiagram->update(EUpdateAll);
```

Die Code-Zeile mit *RECEIVE_DATA()* ist nicht wirklich ein Methodenaufruf sondern soll nur darstellen, dass an dieser Stelle von irgendwoher ein Speicherblock empfangen wird, der die Daten enthält, die das Diagram ausgeben soll. Das Diagram und die Diagram-Objekte aktualisieren nicht implizit ihre Datenstrukturen und geben sich am Bildschirm aus. Dies erfolgt immer erst nach einem expliziten *update* Aufruf. Dies deshalb, weil man auf diese Weise dem Diagramm Schritt für Schritt neue Attribute und Parameterwerte übergeben kann, ohne dass jedes Mal komplexe Berechnungen durchgeführt werden. Diese Berechnungen werden erst mit dem *update* Aufruf „getriggert“.

Zu Beachten ist, dass die X/Y-Wertepaare durch den *setMemBlockValues* Aufruf kopiert werden. Der für den Speicherblock reservierte Speicher muss durch den Aufrufer selbst wieder freigegeben werden. Es kann ja sein, dass ein- und derselbe Speicherblock an mehrere Diagramm-Instanzen gleichzeitig übergeben wird.

Reservieren und Füllen eines Diagram-Speicherblocks

Hierzu dient der Diagram-Speicherblock, dessen Speicherbedarf im Wesentlichen von der Anzahl der zu übertragenden Traces und Anzahl der Wertepaare pro Trace abhängt und wie folgt aufgebaut ist:

```
struct ZSDIAGRAM_API SMemBlockDiagram:
    unsigned int   m_uMemBlockSize;
    int            m_iBlockId;
    EMeasState     m_measState;
    EMeasMode      m_measMode;
    int            m_iMeasType;
    unsigned int   m_uScaleCount;
    unsigned int   m_uTraceCount;
    SMemBlockScale m_arMemBlockScales[1];
    SMemBlockTrace m_arMemBlockTraces[1];
```

Da jeder Trace individuell skaliert werden kann oder aber mehrere Traces die gleiche Skalierung verwenden, andere ggf. aber wiederum separat skaliert werden sollen oder aber alle Traces immer gleich skaliert werden sollen, kann der Speicherblock eine variable Anzahl von Skalierungen aufnehmen.

```
struct ZSDIAGRAM_API SMemBlockScale:
    SScaleContent m_scaleContent;

struct ZSDIAGRAM_API SScaleContent:
    int       m_iScaleId;
    EValidity m_arvalidity[EScaleDirCount];
    ESpacing  m_arspacing[EScaleDirCount];
    SScale    m_arscale[EScaleDirCount];
```

Die Zuordnung zum Diagram Scale-Objekte erfolgt über das Member-Element „m_iScaleId“. Da die Diagram Trace-Objekte wiederum an Diagram Scale-Objecte gebunden sind, kann die Diagram-Klasse die Skalierungen so den Traces und damit den einzelnen Diagram-Objekten zuordnen.

Für jeden Trace soll ferner die Möglichkeit bestehen, charakteristische Kenngrößen des Messverlaufs mit anzuzeigen. Diese Kenngrößen müssen deshalb auch zusammen mit den

Messwerten innerhalb des Diagram Speicherblocks an die Diagram-Klasse übertragen werden, damit diese gleichzeitig mit den Messwerten am Bildschirm aktualisiert werden können.

```
struct ZSDIAGRAM_API SMemBlockTrace:
    STraceContent m_traceContent;
    CPhysVal      m_arphysValCharacteristics[1];
    double        m_arfMemBlockValues[EScaleDirCount];

struct ZSDIAGRAM_API STraceContent:
    int          m_iTraceId;
    unsigned int m_uCharacteristicsCount;
    double       m_arfValRes[EScaleDirCount];
    EValidity    m_arvalidity[EScaleDirCount];
    unsigned int m_aruValCount[EScaleDirCount];
```

Um die Zusammenhänge zu verdeutlichen wird nachfolgend ein Speicherblock reserviert, der Werte für die X- und Y-Achse sowie Messwerte für 2 Traces aufnehmen soll. Die Werte für die X- und Y-Achse ordnet das Diagramm über die *ScaleDir* und der *ScaleId* den Scale-Objekten zu. Über die *TraceId* bestimmt das Diagramm das *Trace-Objekt*, an das die X/Y-Messwerte zu übergeben sind. Für den ersten Trace wurden 2 Kenngrößen ermittelt, die mit zu übertragen sind (allerdings werden diese Kenngrößen im Beispiel nicht innerhalb des Diagramms ausgegeben). Für Trace 1 werden 10, für Trace 2 werden 5 X/Y Wertepaare übertragen.

```
SScale scaleX(
    /* physSize */ EPhysSizeFrequency,
    /* physUnit */ IdEUnitHz,
    /* fMin     */ 1.0e9,
    /* fMax     */ 2.0e9,
    /* fRes     */ 1.0);
SScale scaleY(
    /* physSize */ EPhysSizeElectricalPower,
    /* physUnit */ IdEUnitDbm,
    /* fMin     */ -200.0,
    /* fMax     */ 50.0,
    /* fRes     */ 0.01);

SScaleContent     scaleContent;
STraceContent     artraceContent[2];
SMemBlockTrace*   pMemBlockTrace;
SMemBlockDiagram* pMemBlockDiagram;

CPhysVal physValTrace0Min(
    /* physSizeValue */ EPhysSizeElectricalPower,
    /* physUnitValue */ IdEUnitDbm,
    /* fValue        */ -20.0,
    /* fResolution   */ 0.01 );
CPhysVal physValTrace0Max(
    /* physSizeValue */ EPhysSizeElectricalPower,
    /* physUnitValue */ IdEUnitDbm,
    /* fValue        */ 20.0,
    /* fResolution   */ 0.01 );
```

Soll das Diagramm eine neue Skalierung erhalten, müssen diese entsprechend im Speicherblock gesetzt werden. Falls der *setScale* Aufruf nicht erfolgt, ist die Skalierung im Speicherblock auf *invalid* gesetzt und das Diagramm wird die Struktur innerhalb des Speicherblocks nicht berücksichtigen:

```
scaleContent.setScaleId(-1);
scaleContent.setScale(EScaleDirX,scaleX);
```

```
scaleContent.setScale(EScaleDirY,scaleY);

artraceContent[0].setTraceId(0);
artraceContent[0].setValCount(EScaleDirX,10);
artraceContent[0].setValCount(EScaleDirY,10);
artraceContent[0].setCharacteristicsCount(2);
artraceContent[1].setTraceId(1);
artraceContent[1].setValCount(EScaleDirX,5);
artraceContent[1].setValCount(EScaleDirY,5);

pMemBlockDiagram = SMemBlockDiagram::alloc(1,&scaleContent,2,artraceContent);
```

Der *alloc* Aufruf reserviert den Speicherblock und übernimmt gleichzeitig die X/Y Skalierungswerte aus den übergebenen *ScaleContent* Strukturen. Die X/Y-Werte sowie die Kenngrößen sind separat in den Speicherblock einzutragen:

```
pMemBlockDiagram->setMeasState(EMeasStateOn);
pMemBlockDiagram->setMeasMode(EMeasModeSingle);
```

Wie bei der Skalierung gilt auch für die Messwerte, dass diese nur dann auf valid gesetzt werden, wenn der entsprechende *copyValues* Methodenaufruf erfolgt. Solange dem Speicherblock keine gültigen Messwerte übergeben werden, sind die Werte im Speicherblock auf *invalid* gesetzt und das Diagramm wird die Struktur innerhalb des Speicherblocks nicht berücksichtigen:

```
pMemBlockTrace = pMemBlockDiagram->getMemBlockTrace(0);
pMemBlockTrace->copyValues(EScaleDirX,10,arfXValuesTrace0);
pMemBlockTrace->copyValues(EScaleDirY,10,arfYValuesTrace0);
pMemBlockTrace->setCharacteristicsVal(0,physValTrace0Min);
pMemBlockTrace->setCharacteristicsVal(1,physValTrace0Max);

pMemBlockTrace = pMemBlockDiagram->getMemBlockTrace(1);
pMemBlockTrace->copyValues(EScaleDirX,5,arfXValuesTrace1);
pMemBlockTrace->copyValues(EScaleDirY,5,arfYValuesTrace1);
```

Die nachfolgende Tabelle, gibt den Inhalt des so reservierten Diagram Speicherblocks wieder.

Name	Datentyp	Bytes
m_uMemBlockSize	unsigned int	4
m_iBlockId	int	4
m_measState	int	4
m_measMode	int	4
m_iMeasType	int	4
m_uScaleCount	unsigned int	4
m_uTraceCount	unsigned int	4
m_arMemBlockScales[0]		
m_iScaleId	int	4
m_arvalidity[EScaleDirX]	int	4
m_arvalidity[EScaleDirY]	int	4
m_arspacing[EScaleDirX]	int	4
m_arspacing[EScaleDirY]	int	4
m_arscale[EScaleDirX]		
m_physSize	int	4
m_physUnit	int	4
m_fMin	double	8
m_fMax	double	8
m_fRes	double	8
m_fResMin	double	8
m_fResMax	double	8
m_arscale[EScaleDirY]		
m_physSize	int	4
m_physUnit	int	4
m_fMin	double	8
m_fMax	double	8
m_fRes	double	8
m_fResMin	double	8
m_fResMax	double	8

m_arMemBlockTraces[0]		
m_traceContent		
m_iTraceId	int	4
m_uCharacteristicsCount	unsigned int	4
m_arfValRes[EscaleDirX]	double	8
m_arfValRes[EscaleDirY]	double	8
m_arvalidity[EScaleDirX]	int	4
m_arvalidity[EScaleDirY]	int	4
m_aruValCount[EscaleDirX]	unsigned int	4
m_aruValCount[EscaleDirY]	unsigned int	4
m_arphysValCharacteristics[0]		
m_validity	int	4
m_physSizeValue	int	4
m_physUnitValue	int	4
m_fValue	double	8
m_physSizeAccuracy	int	4
m_physUnitAccuracy	int	4
m_fAccuracy	double	8
m_arphysValCharacteristics[1]		
m_validity	int	4
m_physSizeValue	int	4
m_physUnitValue	int	4
m_fValue	double	8
m_physSizeAccuracy	int	4
m_physUnitAccuracy	int	4
m_fAccuracy	double	8
m_arfMemBlockValues		
X-Values		
m_arfMemBlockValues[0]	double	8
m_arfMemBlockValues[1]	double	8
m_arfMemBlockValues[2]	double	8
m_arfMemBlockValues[3]	double	8
m_arfMemBlockValues[4]	double	8
m_arfMemBlockValues[5]	double	8
m_arfMemBlockValues[6]	double	8
m_arfMemBlockValues[7]	double	8
m_arfMemBlockValues[8]	double	8
m_arfMemBlockValues[9]	double	8
Y-Values		
m_arfMemBlockValues[0]	double	8
m_arfMemBlockValues[1]	double	8
m_arfMemBlockValues[2]	double	8
m_arfMemBlockValues[3]	double	8
m_arfMemBlockValues[4]	double	8
m_arfMemBlockValues[5]	double	8
m_arfMemBlockValues[6]	double	8
m_arfMemBlockValues[7]	double	8
m_arfMemBlockValues[8]	double	8
m_arMemBlockTraces[1]		
m_traceContent		
m_iTraceId	int	4
m_uCharacteristicsCount	unsigned int	4
m_arfValRes[EscaleDirX]	double	8
m_arfValRes[EscaleDirY]	double	8
m_arvalidity[EScaleDirX]	int	4
m_arvalidity[EScaleDirY]	int	4
m_aruValCount[EscaleDirX]	unsigned int	4
m_aruValCount[EscaleDirY]	unsigned int	4
m_arphysValCharacteristics[0]		
m_validity	int	4
m_physSizeValue	int	4
m_physUnitValue	int	4
m_fValue	double	8
m_physSizeAccuracy	int	4
m_physUnitAccuracy	int	4
m_fAccuracy	double	8
m_arfMemBlockValues		
X-Values		
m_arfMemBlockValues[0]	double	8
m_arfMemBlockValues[1]	double	8
m_arfMemBlockValues[2]	double	8
m_arfMemBlockValues[3]	double	8
m_arfMemBlockValues[4]	double	8
Y-Values		
m_arfMemBlockValues[0]	double	8
m_arfMemBlockValues[1]	double	8
m_arfMemBlockValues[2]	double	8
m_arfMemBlockValues[3]	double	8
m_arfMemBlockValues[4]	double	8
Summe Bytes (MemBlockSize)		**564**

Zu beachten ist, dass auch für Trace 2 ein Element „m_arphysValCharacteristics[0]" angelegt wurde, das aber nicht benutzt wird und dessen Elemente mit Nullen initialisiert wurden. Generell wird im Speicherblock für Datenblöcke, deren Größe erst zur Laufzeit ermittelt werden kann, immer wenigstens ein Datenelement reserviert. Falls z.B. keine Skalierung übertragen werden soll, wird trotzdem ein Skale Block allokiert, dessen Inhalt jedoch nur Nullen und damit keine gültigen Werte enthält. Falls weder X- noch Y-Werte übertragen werden sollen, werden trotzdem im Trace-Memory-Block zwei double Werte reserviert.

2.2. Implementierung einer Diagram Objekt Klasse

Die Erfahrung hat gezeigt, dass man noch so viele Attribute und Gestaltungsmöglichkeiten vorsehen und in die Klassen implementieren kann, es wird immer einen Anwendungsfall geben, den man nicht berücksichtigt hat. Damit man sein eigenes Diagramm gestalten und implementieren kann, wurden die meisten Methoden der Diagramm-Klassen als *virtuelle* Methoden deklariert und es wurde möglichst auf *private* Elemente verzichtet.

Sicherlich müssen auch die von *CDataDiagram* abgeleiteten Diagramm-Klassen selbst noch um einige Funktionen erweitert bzw. an manchen Stellen opitimiert werden. So fehlt z.B. noch die Möglichkeit, einen Kurvenverlauf durch Anhängen von Werten schrittweise aufzubauen, so dass der Kurvenverlauf langsam nach rechts anwächst, ohne dass die Kurve jedesmal neu gezeichnet wird. Darüberhinaus fehlt noch die dritte Dimension – also die Z-Achse. Auch fehlt noch die Möglichkeit, praktisch unendlich viele X/Y-Wertepaare in das Diagramm aufzunehmen, indem ggf. die Trace-Objekte nicht die X/Y-Werte sondern eine Bitmap der Werte speichert. Aber diese „Features" werden dann implementiert, wenn sie wirklich gebraucht werden.

Um einen möglichst hohen Grad an Flexibilität zu erreichen, besteht die Möglichkeit, sich selbst Diagramm-Objekt-Klassen zu erzeugen und zur Laufzeit an das Diagramm zu übergeben. Hierzu muss man eine Klasse implementieren die von der abstrakten Basisklasse *CDiagObj* abgeleitet ist und entsprechende Schnittstellen-Methoden implementieren.

Da die Notwendigkeit hierzu recht oft bestehen wird – besonders während der „jungen" Zeit des ZSDiagram Subsystems – wird im Folgenden beispielhaft eine Diagramm-Objekt-Klasse implementiert. Ein weiterer Nebeneffekt des Implementierens einer eigenen Diagramm-Objekt Klasse ist, dass man dadurch das Diagramm-Subsystem besser kennen und verstehen lernt.

Im Folgenden soll eine Diagramm-Objekt-Klasse entworfen und implementiert werden, die X/Y Werte als Balkendiagramm ausgibt.

2.2.1. Anforderungen (Pflichtenheft)

Das folgende Bild soll ein Diagramm mit einem Histogramm zeigen, wie wir es implementieren wollen:

Folgende Eigenschaften soll das Histogramm bekommen:

1. Festlegen der Position und Größe der Balken soll möglich sein:
   a. durch „herkömmliche" X/Y-Wertepaare. Optional Angabe der horizontalen Breite der Balken und ob YMin auf der Null-Linie oder beim Minimalwert der Y-Achse liegen soll.
   b. durch Angabe von YMin und YMax sowie XMin und XMax.

2. Das optische Erscheinungsbild jedes einzelnen Balkens im Histogramm soll separat definiert werden können. So soll es möglich sein, die Balken mit einem Rahmen zu versehen, die Füllfarbe zu definieren und ggf. ein Füllmuster festzulegen.
3. Jeder Balken soll selektierbar (fokussierbar) sein. Für fokussierte Balken sollen per Tool Tip weitere Informationen abgefragt werden können (z.B. aktueller Wert und/oder Bezug auf Datenquelle wie „Bananen: 34,2 %“).

2.2.2. Implementierung

Unsere Arbeit beginnt damit, sowohl ein C++ Header File als auch ein C++ Source File anzulegen, in der die von *CDiagObj* abgeleitete Klasse *CDiagObjHistogram* implementiert werden soll. Zunächst beschränken wir uns darauf, nur die für die Schnittstelle von *CDiagObj* erforderlichen Methoden zu definieren und zu implementieren. Damit könnte die Deklaration der Klasse CDiagObjHistogram im Header-File wie folgt aussehen:

```
namespace ZS
{
namespace Diagram
{
//*******************************************************************************
class ZSDIAGRAM_API CDiagObjHistogram : public CDiagObj
//*******************************************************************************
{
public: // ctors and dtor
    CDiagObjHistogram(
        const QString& i_strObjName,
        CDiagTrace*    i_pDiagTrace );
    virtual ~CDiagObjHistogram();
public: // instance methods
    void setCol( const QColor&  i_col );
    QColor getCol() const;
public: // must overridables of base class CDiagObj
    virtual CDiagObj* clone( CDataDiagram* i_pDiagramTrg ) const;
    virtual void update( unsigned int i_uUpdateFlags, QPaintDevice* i_pPaintDevice = NULL );
private: // copy ctor not allowed
    CDiagObjHistogram( const CDiagObjHistogram& );
private: // assignment operator not allowed
    void operator=( const CDiagObjHistogram& );
protected: // instance members
    QColor       m_col;
    QPointArray* m_pPtArr;

}; // class CDiagObjHistogram

} // namespace Diagram

} // namespace ZS
```

Eine “*clone*”-Methode gehört zur Schnittstelle von *CDiagObj*. Mit Hilfe dieser Methode soll es möglich, das Diagramm-Objekt aus einem Diagramm in ein anderes Objekt mit den aktuellen eingestellten Eigenschaften zu kopieren.

Ebenfalls zur Schnittstelle von *CDiagObj* gehört die „*update*“-Methode, die vom Diagramm mit unterschiedlichen Prozesstiefen aufgerufen wird, damit das Objekt seine interne Datenstruktur aktualisiert, sich in die Pixmap des Diagramms ausgibt und dem Widget mitteilt, welcher Rechtecksbereich geändert und aktualisiert werden muss.

Mit „setCol“ erlauben wir es, die Füllfarbe der Balken festzulegen (für den ersten Entwurf sollte uns diese Möglichkeit der optischen Gestaltungsmöglichkeiten zunächst ausreichen).

In einer ersten Version implementieren wir das Histogramm in der einfachsten Form, in dem X/Y-Werte als Linie ausgehend vom Minimalwert der Achse bis zum Y-Wert gezeichnet

werden. Hierzu reicht uns zunächst die Verwendung eines *QPointArrays*. In diesem Array werden wir bei Ausführung des *Data-Processings* die Pixel-Koordinaten der Y-Werte speichern. Bei Ausführung des *Pixmap-Processing* wird anschliessend dieses Point-Array über *drawLine* Befehle in die Pixelmap des Diagramms ausgegeben.

Haben wir die erste Version der Klassendefinition im Header-File untergebracht, können wir dazu übergehen, die Methoden der Klasse zu implementieren. Beginnen wir mit dem Konstruktur und dem Destruktor.

Ein Histogram Objekt liegt immer im Center-Bereich des Diagramms, weshalb es nicht nötig ist, die Layout-Position über den Konstruktor vorgeben zu lassen. Stattdessen wird die Basisklasse *CDiagObj* mit *ELayoutPosCenter* konstruiert:

```
//------------------------------------------------------------------------------
CDiagObjHistogram::CDiagObjHistogram(
    const QString& i_strObjName,
    CDiagTrace*    i_pDiagTrace ) :
//------------------------------------------------------------------------------
    CDiagObj(
        /* strObjName */ i_strObjName,
        /* pDiagTrace */ i_pDiagTrace,
        /* layoutPos  */ ELayoutPosCenter ),
    m_col(Qt::yellow),
    m_pPtArr(NULL)
{
} // ctor
```

Der Destruktor muss das Point-Array, das ggf. innerhalb der *update*-Methode allokiert wurde, wieder vom Heap entfernen:

```
//------------------------------------------------------------------------------
CDiagObjHistogram::~CDiagObjHistogram()
//------------------------------------------------------------------------------
{
    try
    {
        delete m_pPtArr;
    }
    catch(...)
    {
    }
    m_pPtArr = NULL;

} // dtor
```

Um zur Laufzeit die Farbe der Histogramm-Balken zu setzen bzw. deren aktuelle Farbe auszulesen, werden folgende set und get-Methoden implementiert:

```
//------------------------------------------------------------------------------
void CDiagObjHistogram::setCol( const QColor&  i_col )
//------------------------------------------------------------------------------
{
    m_col = i_col;
    invalidate(EUpdatePixmapWidget,true);
}

//------------------------------------------------------------------------------
QColor CDiagObjHistogram::getCol() const
//------------------------------------------------------------------------------
{
    return m_col;
}
```

Die clone-Methode, mit der das Histogramm Objekt in ein anderes Diagramm kopiert werden können soll, kann wie folgt implementiert werden:

```
//------------------------------------------------------------------------------
CDiagObj* CDiagObjHistogram::clone( CDataDiagram* i_pDiagramTrg ) const
//------------------------------------------------------------------------------
{
    if( i_pDiagramTrg == NULL || m_pDiagTrace == NULL )
    {
        return NULL;
    }

    CDiagTrace* pDiagTrace = i_pDiagramTrg->getDiagTrace( m_pDiagTrace->getTraceId() );

    if( pDiagTrace == NULL )
    {
        return NULL;
    }

    CDiagObjHistogram* pDiagObj = new CDiagObjHistogram(
        /* strObjName */ m_strObjName,
        /* pDiagTrace */ pDiagTrace );

    // Members from base class CDiagObj:
    pDiagObj->m_layoutPos = m_layoutPos;
    pDiagObj->m_rectContent = m_rectContent;
    pDiagObj->m_bAdjustContentRect2DiagPartCenter = m_bAdjustContentRect2DiagPartCenter;
    pDiagObj->m_bVisible = m_bVisible;
    pDiagObj->m_state = m_state;
    pDiagObj->m_bIsFocusable = m_bIsFocusable;
    pDiagObj->m_bIsEditable = m_bIsEditable;

    // Members from this class:
    pDiagObj->m_col = m_col;

    i_pDiagramTrg->addDiagObj(pDiagObj);

    return pDiagObj;

} // clone
```

Um das Objekt zu clonen, wird zunächst ein neues Histogramm Objekt auf dem Heap konstruiert. Hierfür wird eine Referenz auf das Trace-Objekt des **neuen** Diagramms benötigt. Da das Diagramm geclont wird, müssen die Id's der Trace-Objekte des zu kopierenden Diagramms mit den Id's der Trace-Objekte im geclonten Diagramm übereinstimmen. Damit kann die Trace-Id verwendet werden, um vom geclonten Target-Diagramm eine Referenz auf das dort befindliche Trace-Objekt zu erhalten. Danach werden die Eigenschaften aus der Basisklasse *CDiagObj* und die speziellen Eigenschaften des Histogramm-Objekts in das neu geschaffene Histogramm Objekt übernommen. Abschliessend wird das neue Histogramm-Objekt dem Target-Diagramm hinzugefügt.

Die umfangreichste zu implemtierende Methode ist die update-Methode, auf die wir im Folgenden etwas näher eingehen wollen.

```
//------------------------------------------------------------------------------
void CDiagObjHistogram::update( unsigned int i_uUpdateFlags, QPaintDevice* i_pPaintDevice )
//------------------------------------------------------------------------------
{
```

Jedes Diagramm-Objekt verwaltet für sich *UpdateFlags*. Jedes Bit von *UpdateFlags* entspricht dabei einer Prozesstiefe. Ist das Bit in den UpdateFlags des Diagramm-Objekts gesetzt, bedeutet dies, dass die Daten des Objekts bzgl. dieser Prozesstiefe nicht auf aktuellem Stand sind. Wird z.B. die Skalierung geändert, werden alle vier Prozesstiefen (*Layout*, *Data*, *Pixmap* und Widget) der Objekte vom Diagramm *invalidiert*, die direkt oder indirekt (über die Trace-Objekte) mit der Skalierung verbunden sind. Wird aber nur die Position eines Markers verdreht, werden für das Marker-Objekt die Prozesstiefen *Data, Pixmap* und *Widget invalidiert*. Für alle anderen Objekte wird die Prozesstiefe *Data* in diesem Fall nicht *invalidiert*. Allerdings werden daraufhin die Prozesstiefen *Pixmap* und *Widget* auch aller

anderen Diagramm-Objekte invalidiert. Die Diagramm-Objekte wissen nämlich nicht, was „unter" ihnen in die Pixelmap ausgegeben wurde. Und wird ein Marker verdreht, muss ja die Pixelmap an seiner vorherigen Stelle aktualisiert werden.

Erhält nun das Diagramm die Aufforderung auf einen *update*, geht das Diagramm alle Scale-, Trace- und Diagramm-Objekte durch und ruft deren *update* Methode auf. Wie bereits erwähnt, werden diese *update*'s in hirarchischer Reihenfolge entsprechend der Prozesstiefen durchgeführt. Welche Prozesstiefe aktuell abzuarbeiten ist, ist im Übergabeparameter UpdateFlags kodiert (ebenfalls in Bits kodiert). Mit anderen Worten, um festzustellen, welche Prozesstiefe neu zu berechnen ist, muss sowohl das entsprechende Prozesstiefen-Bit in den *UpdateFlags* des Objekts gesetzt sein als auch das entspechende Prozesstiefen-Bit des vom Diagramm an die *update*-Methode übergebenen Parameters *UpdateFlags*.

Die Prozesstiefe *Layout* nimmt eine Sonderstellung ein, denn die Diagramm-Objekte führen selbst keine aktiven Layout-Berechnungen durch. Dies ist Aufgabe des Diagramms, das für die Positionierung der Objekte im Diagramm zunächst die Scale- und Trace-Objekte auffordert, ihre internen Datenstrukturen zu aktualisieren (z.B. die Unterteilung der Achsen in die Division-Lines vorzunehmen) und anschliessend die *sizeHint* Methoden der Diagramm-Objekte aufruft. Für das Histogramm-Objekt ist die sizeHint Methode denkbar einfach, denn Histogramme werden im Center-Bereich des Diagramms ausgegeben. Der Center-Bereich ist „Expandable" und wird vom Diagramm als Ergebnis der Layout-Berechnungen gesetzt. Bedeutet, dass für die Histogramm-Klasse die *sizeHint*-Implementierung der Basis-Klasse ausreicht. Aber für z.B. eine *AxisScale-DiagObj* Klasse wäre diese Basis-Implementierung nicht ausreichend. Denn damit ein *AxisScale*-Objekt für die X-Achsen-Skalierung seine aktuelle Breite über den *sizeHint* Aufruf zurückgeben kann, muss das Objekt erst einmal wissen, wie viele gültige Stellen zur Beschriftung der Achse für den Minimal- und Maximalwert notwendig sind. Hier müsste die *sizeHint*-Methode des Diagramm-Objekts dafür sorgen, dass alle hierfür notwendigen Berechnungen durchgeführt werden. Idealerweise ruft die *sizeHint* Methode selbst die *update*-Methode des Objekts auf.

Wie bereits erwähnt ist das *Layout Processing* einzig Aufgabe des Diagramms. Sollte (aufgrund eines Programmierfehlers) die Methode trotzdem aus irgendeinem Grund mit dieser Prozesstiefe aufgerufen werden, wird die Methode einfach beendet.

```
if( i_uUpdateFlags & EUpdateLayout && m_uUpdateFlags & EUpdateLayout )
{
    validate(EUpdateLayout);
}
```

Falls für das Objekt *Data Processing* angefordert wurde und auch notwendig ist …

```
if( i_uUpdateFlags & EUpdateData && m_uUpdateFlags & EUpdateData )
{
```

Wie bereits mehrfach erwähnt werden die X/Y-Wertepaare von den Trace-Objekten verwaltet. Damit der nachfolgende Code einfacher zu lesen ist, wird eine Referenz auf die notwendigen Daten vom Trace-Objekt geholt und auf dem Stack abgelegt.

```
    unsigned int  uXValCount = m_pDiagTrace->getValCount(EScaleDirX);
    const double* pfXValues  = m_pDiagTrace->getValues(EScaleDirX);
    unsigned int  uYValCount = m_pDiagTrace->getValCount(EScaleDirY);
    const double* pfYValues  = m_pDiagTrace->getValues(EScaleDirY);
```

Eigentlich müssen die Anzahl der X- und Y-Werte übereinstimmen. Aber sicher ist sicher …

```
    unsigned int uValCount;
```

```
uValCount = uXValCount;
if( uValCount > uYValCount )
{
    uValCount = uYValCount;
}
```

Falls das Histogramm-Objekt nicht sichtbar ist, wird das allokierte Point-Array freigegeben und es ist nichts weiter zu tun.

```
if( !isVisible() || uValCount <= 1 || pfXValues == NULL || pfYValues == NULL )
{
    delete m_pPtArr;
    m_pPtArr = NULL;
    return;
}
```

Falls das Histogramm-Objekt sichtbar ist, wird zunächst geprüft, wie viele X/Y-Werte innerhalb der aktuellen Achsenskalierung liegen (Diagramm können gezoomt werden und/oder die Skalierung kann geändert worden sein).

```
int           idxValMinPrev;
int           idxValMaxNext;
unsigned int  udxVal;
unsigned int  uPtCount;
QPoint*       pPt;
const double* pfX = pfXValues;
const double* pfY;
double        fXMin = m_pDiagTrace->getScale(EScaleDirX).m_fMin;
double        fXMax = m_pDiagTrace->getScale(EScaleDirX).m_fMax;
double        fx, fy;
int           xPix, yPix;

pfX = pfXValues;
for( udxVal = 0, idxValMinPrev = -1, idxValMaxNext = -1, uPtCount = 0;
     udxVal < uValCount;
     udxVal++, pfX++ )
{
    if( *pfX < fXMin )
    {
        idxValMinPrev = static_cast<int>(udxVal);
    }
    else if( *pfX > fXMax )
    {
        idxValMaxNext = static_cast<int>(udxVal);
        break;
    }
    else
    {
        uPtCount++;
    }
}
```

Falls keiner der X/Y-Werte innerhalb der sichtbaren Skalierung liegt, hat das Histogramm-Objekt auch nichts auszugeben. Es ist nichts weiter zu tun.

```
// If all of the points are left of XScaleMin (incl. XScaleMin) ...
if( uPtCount <= 1 && idxValMaxNext == -1 )
{
    // ... none of the points are visible.
    delete m_pPtArr;
    m_pPtArr = NULL;
    return;
}
// If all of the points are right of XScaleMax (incl. XScaleMax) ...
else if( uPtCount <= 1 && idxValMinPrev == -1 )
{
    // ... none of the points are visible.
    delete m_pPtArr;
    m_pPtArr = NULL;
    return;
}
```

Das Point-Array wird so dimensioniert, dass es nur alle wirklich sichtbaren Werte aufnehmen kann.

```
        // The number of the points to be calculated are known now and the
        // point array will be allocated:
        if( m_pPtArr != NULL )
        {
            if( m_pPtArr->size() != uPtCount )
            {
                delete m_pPtArr;
                m_pPtArr = new QPointArray(static_cast<int>(uPtCount));
            }
        }
        else
        {
            m_pPtArr = new QPointArray(static_cast<int>(uPtCount));
        }
        pPt = m_pPtArr->data();
```

Mit Hilfe der Methode getValPix des Trace-Objekts werden die X/Y-Werte in Pixelkoordinaten umgerechnet und im Point-Array abgelegt. Es ist davon abzuraten, selbst einen Algorithmus zur Umrechnung von Welt- in Bildschirmkoordinaten zu implementieren. Um Pixel-Fehler durch Rundungsfehler zu vermeiden sollten alle Diagramm-Objekte die Methoden der Trace- und Scale-Objekte verwenden. Nur so kann sichergestellt werden, dass sowohl für Grid-Linien, Trennlinien der Achsen, Marker-Positionen etc. für die gleichen X/Y-Werte auch immer die gleichen Bildschirmkoordinaten ermittelt werden.

```
        // Calculate the points between XScaleMin and XScaleMax ...
        pfX = &pfXValues[idxValMinPrev+1];
        pfY = &pfYValues[idxValMinPrev+1];
        for( udxVal = static_cast<unsigned int>(idxValMinPrev+1);
             udxVal < uValCount;
             udxVal++, pfX++, pfY++, pPt++ )
        {
            fx = *pfX;
            if( fx > fXMax )
            {
                break;
            }
            xPix = m_pDiagTrace->getValPix(EScaleDirX,fx);
            pPt->setX(xPix);

            fy = *pfY;
            yPix = m_pDiagTrace->getValPix(EScaleDirY,fy);
            pPt->setY(yPix);
        }
```

Das *Data Processing* Flag wird *validiert* (auf Null zurückgesetzt). Wird bis zum nächsten *update*-Aufruf mit *Data Processing* das Flag nicht wieder *invalidiert* (was während des Layout-Processings durchaus der Fall sein kann), muss das Point-Array nicht erneut berechnet werden.

```
        // Mark current process depth as executed (reset bit):
        validate(EUpdateData);

    } // if( EUpdateData )
```

Falls für das Objekt *Pixmap Processing* angefordert wurde und auch notwendig ist …

```
    if( i_uUpdateFlags & EUpdatePixmap && m_uUpdateFlags & EUpdatePixmap )
    {
```

Falls das Histogramm-Objekt sichtbar ist …

```
        if( isVisible() )
        {
```

Das Diagramm übergibt eine Referenz auf ein Paint-Device. Es kann nämlich auch sein, dass das Diagramm auf einen Drucker ausgegeben werden soll. In diesem Fall steckt hinter dem Paint-Device nicht eine Pixmap sondern ein QPrinter Objekt. Es wird ein QPointer Objekt erzeugt, mit dessen Hilfe das Histogramm die grafischen Ausgaben vornimmt. Im vorliegenden Fall werden lediglich Linien von Y-Min bis zum Y-Wert gezeichnet.

```
            QPainter painter(i_pPaintDevice);

            unsigned int udxVal;

            painter.setClipRect(m_rectContent);
            painter.setClipping(true);
            painter.setPen(m_col);

            if( m_pPtArr != NULL )
            {
                for( udxVal = 0; udxVal < m_pPtArr->count(); udxVal++ )
                {
                    painter.drawLine(
                        /* x1 */ m_pPtArr->at(udxVal).x(),
                        /* y1 */ m_pDiagTrace->getScaleMinValPix(EScaleDirY),
                        /* x2 */ m_pPtArr->at(udxVal).x(),
                        /* y2 */ m_pPtArr->at(udxVal).y() );
                }
            }
        } // if( isVisible() )
```

Das *Pixmap Processing* Flag wird *validiert* (auf Null zurückgesetzt).

```
        validate(EUpdatePixmap);

    } // if( EUpdatePixmap )
```

Falls für das Objekt *Widget Processing* angefordert wurde und auch notwendig ist …

```
    if( i_uUpdateFlags & EUpdateWidget && m_uUpdateFlags & EUpdateWidget )
    {
        CWdgtDiagram* pWdgtDiagram = dynamic_cast<CWdgtDiagram*>(m_pDataDiagram);
```

Sicher ist sicher (und lint will es auch so) …

```
        if( pWdgtDiagram != NULL )
        {
```

Das Histogramm liegt im Center-Bereich des Diagramms. Wurde das Histogramm neu gezeichnet (ansonsten würde wir hier nicht landen), muss der komplette Center-Bereich neu am Bildschirm ausgegeben werden und ist entsprechend am Widget zu invalidieren. Es wurde absichtlich darauf verzichtet, umständlich berechnen zu lassen, welche Rechtecksbereiche von den Histogramm-Balken tatsächlich verändert wurden, denn diese Berechnung incl. der nachfolgenden Invalidierung dieser Rechtecke würde mit Sicherheit länger dauern, als einfach den kompletten Center-Bereich am Bildschirm zu aktualisieren.

```
            pWdgtDiagram->update(m_rectContent);

        } // if( pWdgtDiagram != NULL )
```

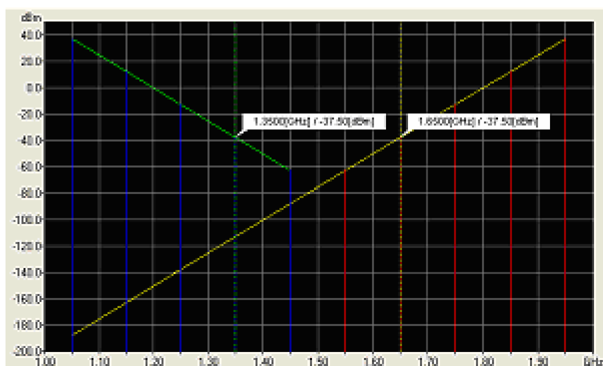
Das *Widget Processing* Flag wird *validiert* (auf Null zurückgesetzt).

```
        // Mark current process depth as executed (reset bit):
        validate(EUpdateWidget);

    } // if( EUpdateWidget )

} // update
```

Hat man die Klasse (z.B.) mit der ZSDiagram-Dll übersetzt, kann sie unserem Beispiel-Diagramm wie gewohnt hinzugefügt werden:

```
m_pDiagObjHistogram0 = new CDiagObjHistogram(
    /* strObjName */ "Example::0",
    /* pDiagTrace */ m_pDiagTrace0 );
m_pDiagObjHistogram0->setCol(Qt::red);
m_pDiagram->addDiagObj(m_pDiagObjHistogram0);

m_pDiagObjHistogram1 = new CDiagObjHistogram(
    /* strObjName */ "Example::1",
    /* pDiagTrace */ m_pDiagTrace1 );
m_pDiagObjHistogram1->setCol(Qt::blue);
m_pDiagram->addDiagObj(m_pDiagObjHistogram1);
```

Das resultierende Diagramm, incl. unserer neuen Histogramm-Objekte, sieht dann in etwa wie folgt aus:



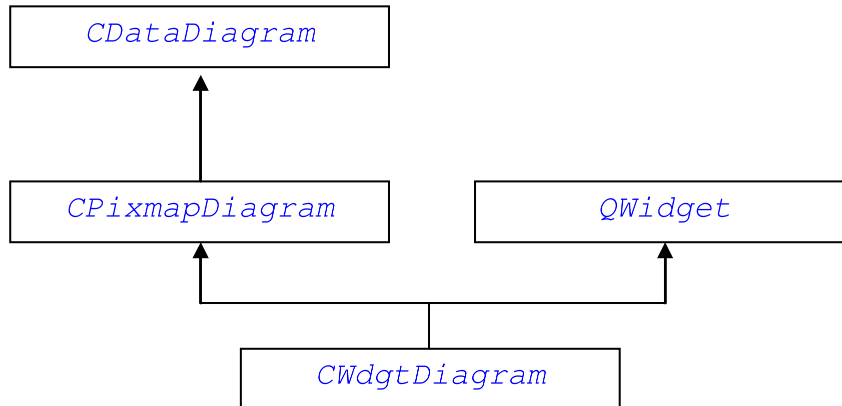
Zu beachten ist, dass die ersten X-Werte für beide Traces übereinstimmen. Da das „blaue Histogramm" nach dem „roten Histogramm" dem Diagramm hinzugefügt wurde, überdecken die ersten fünf blauen „Balken" die ersten fünf roten „Balken".

Bisher ist unsere Diagramm-Objekt-Klasse lediglich in der Lage, die Messwerte als ein Linien-Diagramm wiederzugeben (*LineChart*). Was aber eher erwünscht ist, ist die Anzeige als Balken, also als *BarChart*. Durch Hinzufügen des Attributes *ChartType* wollen wir im nachfolgenden die Diagramm-Objekt-Klasse dahingehend erweitern, dass sie die Messwerte entweder als Linien oder als Balken wiedergibt. Gleichzeitig brauchen wir aber noch weitere Attribute, um z.B. Rahmenfarbe und Füllmuster der Balken festzulegen. Ferner wollen wir einen Freiheitsgrad schaffen, indem wir auch die Breite der Balken vom Diagramm-Objekt entweder automatisch selbst berechnen lassen oder in dem wir die Breite der Balken „von außen" vorgeben lassen wollen. Und wenn wir schon dabei sind, erweitern wir die Klasse gleich so weit, dass die Balken „von außen" komplett frei positioniert werden können.

3. Klassen- und Schnittstellenbeschreibungen *(not up to date)*

3.1. Klasse *CWdgtDiagram*

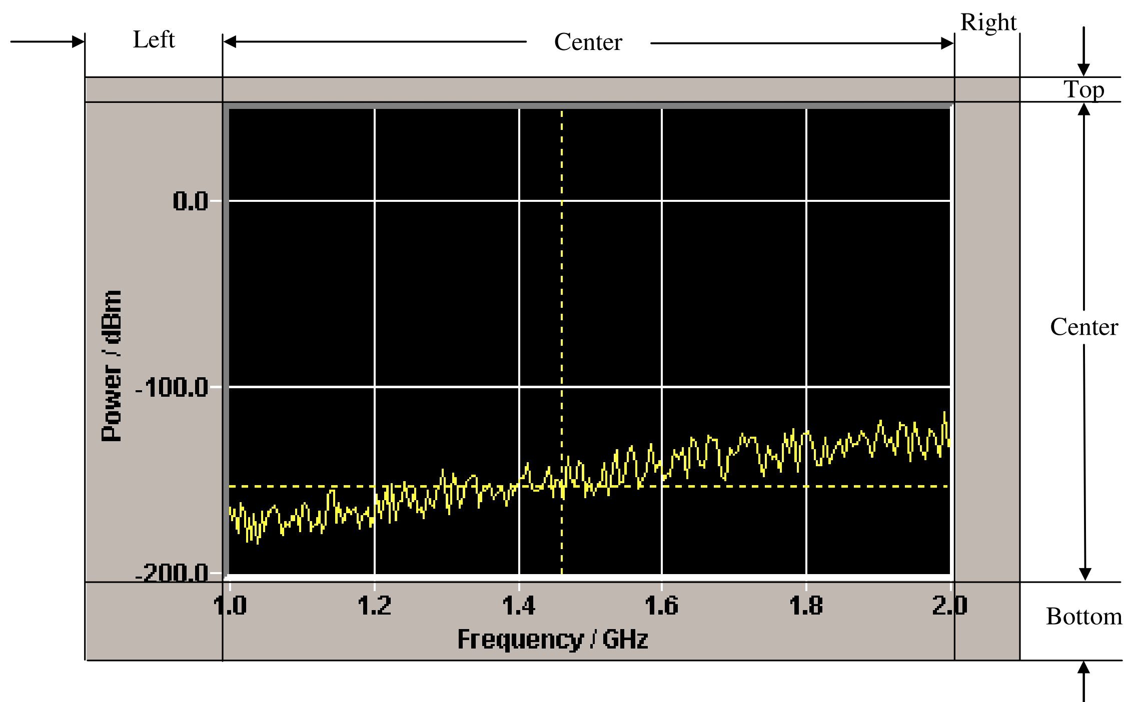
CWdgtDiagram ist eine Spezialisierung der Klasse *QWidget* und direkt von dieser Klasse abgeleitet.



3.1.1. Übersicht

3.1.1.1. Positionierung der Diagram-Objekte (Layout)

Ein Diagram-Fenster unterteilt sich in fünf verschieden Layout Bereiche. Jedes Diagram-Objekt kann nur innerhalb eines dieser fünf Layout-Bereiche positioniert werden, wobei sich die äußeren Bereiche überlappen. Das folgende Bild stellt die verschiedenen Layout-Bereiche dar.



Jedem dieser Bereiche ist eine Konstante zugeordnet, die in der Klasse *CWdgtDiagram* über folgende Enumeration definiert ist:

```
typedef enum {
    ELayoutPosMin    = 0,
    ELayoutPosTop    = 0,
    ELayoutPosBottom = 1,
    ELayoutPosLeft   = 2,
    ELayoutPosRight  = 3,
    ELayoutPosCenter = 4,
    ELayoutPosMax    = 4,
    ELayoutPosCount
}   ELayoutPos;
```

Jedes mal, wenn die Größe des Diagrams geändert wird (so z.B. auch bei erstmaliger Anzeige des Diagrams) bestimmt das Diagram die maximalen Abmessungen der äußeren Bereiche anhand der *sizeHint* Methoden der Diagram Objekte, die im jeweiligen Bereich liegen. Den „Center" Bereich passt das Diagram in der Größe an die jeweiligen maximalen Abmessungen der äußeren Bereiche an.

In welchem Layout-Bereich ein Diagram-Objekt liegt, muss bereits im Konstruktor des Objekts festgelegt werden und wird vom Diagram über die Methode *getLayoutPos* erfragt. Nicht jede Layout-Position macht für die verschiedenen Arten von Diagram-Objekten Sinn. So werden Grid-, und Curve- und Marker-Objekte immer im Center Bereich liegen, während Achsenbeschriftungen der X-Achse wohl meist im Bottom-Bereich und Achsenbeschriftungen der Y-Achse wohl meist im Left-Bereich liegen werden.

Die Achsenbeschriftungen ermitteln ihre Abmessungen für den *sizeHint* Aufruf über den Platz, den die Skalierungswerte und die Benennung der Achse in Anspruch nimmt. Da nun die Anzahl gültiger Stellen von der Skalierung abhängig ist und damit auch die horizontale Ausdehnung der Y-Achsenbeschriftung ändern könnte, würde in diesem Fall das Diagram in Abhängigkeit von der Skalierung den linken Rand mal vergrößern und mal verkleinern. Das ist ein unerwünschter Effekt, weshalb es für jeden Randbereich möglich ist, eine minimale Ausdehnung anzugeben.

3.1.1.2. Aktualisieren der Datenstrukturen der Diagram-Objekte (Update)

3.1.1.3. Ausgabe der Diagram-Objekte am Bildschirm (Paint)

3.1.2. Konstruktor

Deklaration:

```
CWdgtDiagram(
    QWidget*        i_pWdgtParent,
    const QString& i_strObjName,
    int             i_cyPartTopMinHeight,
    int             i_cyPartBottomMinHeight,
    int             i_cxPartLeftMinWidth,
    int             i_cxPartRightMinWidth,
    const QColor&  i_colBgPartCenter,
```

```
    unsigned int    i_uTraceCount = 1 );
```

Erzeugt ein Diagram-Widget als „Child“ des Parent-Widgets (für weitere Information bzgl. Parent-Child-Widgets siehe Qt-Dokumentation).

Parameters:

i_pWdgtParent (IN)
Wertebereich [valid address]
Parent-Widget (siehe Qt Dokumentation). Darf nicht NULL sein.
i_strObjName (IN)
Wertebereich [QString::null, valid string]
Name des Diagrams (für Debugging Zwecke)
i_cyPartTopMinHeight (IN)
Wertebereich [0..N]
Minimale vertikale Ausdehnung des oberen Diagram-Bereichs.
i_cyPartBottomMinHeight (IN)
Wertebereich [0..N]
Minimale vertikale Ausdehnung des unteren Diagram-Bereichs.
i_cyPartLeftMinWidth (IN)
Wertebereich [0..N]
Minimale horizontale Ausdehnung des linken Diagram-Bereichs.
i_cyPartRightMinWidth (IN)
Wertebereich [0..N]
Minimale horizontale Ausdehnung des rechten Diagram-Bereichs.
i_colBgPartCenter (IN)
Wertebereich [valid color]
Hintergrundfarbe für den Ausgabebereich des Diagrams (die äußeren Bereiche erhalten die Hintergrundfarbe des Parent-Widgets).
i_uTraceCount (IN)
Wertebereich [1..N]
Anzahl der Traces (Kurvenverläufe), die maximal im Diagram auszugeben sind. Da das Diagram für jeden möglichen Trace dynamisch eine Verwaltungsstruktur anlegt, muss die maximale Anzahl von Traces bereits beim Konstruieren des Diagrams bekannt sein.

3.1.3. Öffentliche Methoden

3.1.3.1. getObjName

Gibt den Namen des Objekts zurück.

Deklaration:
```
QString getObjName() const;
```

Rückgabewert:
Name des Objekts.

3.1.3.2. getRectPartTop

Gibt die aktuelle Größe und Position (relativ zum Diagram-Widget) des oberen Diagram-Bereichs zurück.

Deklaration:
```
QRect getRectPartTop() const;
```

Rückgabewert:
Aktuelle Größe und Position (relativ zum Diagram-Widget) des oberen Diagram-Bereichs.

3.1.3.3. getRectPartBottom

Gibt die aktuelle Größe und Position (relativ zum Diagram-Widget) des unteren Diagram-Bereichs zurück.

Deklaration:
```
QRect getRectPartBottom() const;
```

Rückgabewert:
Aktuelle Größe und Position (relativ zum Diagram-Widget) des unteren Diagram-Bereichs.

3.1.3.4. getRectPartLeft

Gibt die aktuelle Größe und Position (relativ zum Diagram-Widget) des linken Diagram-Bereichs zurück.

Deklaration:
```
QRect getRectPartLeft() const;
```

Rückgabewert:
Aktuelle Größe und Position (relativ zum Diagram-Widget) des linken Diagram-Bereichs.

3.1.3.5. getRectPartRight

Gibt die aktuelle Größe und Position (relativ zum Diagram-Widget) des rechten Diagram-Bereichs zurück.

Deklaration:
```
QRect getRectPartRight() const;
```

Rückgabewert:
Aktuelle Größe und Position (relativ zum Diagram-Widget) des rechten Diagram-Bereichs.

3.1.3.6. getRectPartCenter()

Gibt die aktuelle Größe und Position (relativ zum Diagram-Widget) des mittleren (zentralen) Diagram-Bereichs zurück. Der zentrale Bereich ist der Ausgabebereich für die Traces.

Deklaration:
```
QRect getRectPartCenter() const;
```

Rückgabewert:
Aktuelle Größe und Position (relativ zum Diagram-Widget) des mittleren (zentralen) Diagram-Bereichs.

3.1.3.7. getColBgPartCenter

Gibt die aktuelle Hintergrundfarbe für den zentralen Ausgabebereich des Diagrams zurück.

Deklaration:
```
QColor getColBgPartCenter() const;
```

Rückgabewert:
Aktuelle Hintergrundfarbe des zentralen Diagram-Bereichs.