UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich Informatik

Entwurf und Implementierung grundlegender Funktionen eines CAD-Systems und Vorstellung weiterführender Algorithmen

Studienarbeit

vorgelegt von

Hanno Binder

Mat.-Nr. 119820226

Betreuer:

Prof. Dr. Manfred Rosendahl,

Fachbereich Informatik, Institut für Softwaretechnik

Koblenz, im Juli 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, 03.07.2006

Hanno Binder

Inhalt

Abstract

Die Planung in Form eines visuellen Entwurfs ist schon seit jeher Bestandteil des Prozesses der Entwicklung von Artefakten aller Art. Heutzutage wird der visuelle Entwurf zukünftiger Produkte mithilfe von CAD-Systemen umgesetzt. Im ersten Teil der Arbeit wird daher ein einfaches System implementiert und erläutert, das die grundlegenden Funktionen umsetzt, die auch in professionellen CAD-Systemen unerlässlich sind. Im zweiten Teil werden über die Grundfunktionen hinaus ausgewählte Algorithmen der parametrischen Modellierung vorgestellt, die die Grundlage für die leistungsfähigen, flexiblen und produktiven Systeme im modernen computer-unterstützten Entwurf bilden.

Einleitung

Sowie seit jeher die Planung der Herstellung von Gegenständen, Werkzeugen und Maschinen vorausgeht, so geht mit der Planung seit jeher der visuelle Entwurf einher. Dieser visuelle Entwurf hat seine Ursprünge in einfachen handgefertigten Skizzen und reicht bis hin zum fotorealistischen Modell moderner CAD Werkzeuge.
Hierbei ist das Kürzel „CAD“ durchaus mit unterschiedlichen, aber ähnlichen, Bedeutungen belegt. Während „CA“ in der Regel für „Computer Aided“ oder „Computer Assisted“, also durch den Computer unterstützt steht, gibt es für den Buchstaben „D“ Bedeutungen wie „Design“, die Entwicklung eines Gegenstandes einschließlich des Aussehens und ggf. auch von Funktionalität, oder auch „Drafting“ („technisches Zeichnen“), also eine Art Zeichnen mit Annotation von z.B. Maßangaben.
Heutige CAD-Systeme finden in vielen Bereichen Anwendung, in denen im Laufe eines größeren Prozesses geometrische Modelle zu erstellen sind, wie zum Beispiel in der Computergrafik und der computer-unterstützten Fertigung (CAM). In diesen Bereichen stellen sie nicht nur eine wesentliche Arbeitserleichterung für die mit der Erstellung der Modelle befassten Personen dar – verglichen z.B. mit dem Entwurf an einem Zeichenbrett – sondern ermöglichen durch neue Funktionen und Fähigkeiten vor allem die Vereinfachung von Fertigungsprozessen sowie die einfache Wiederverwendung von einmal erstellten Problemlösungen.
Ersteres wird überhaupt erst möglich durch die exakte numerische Darstellung geometrischer Modelle und unterstützt durch die Integration mit anderen numerischen Systemen, wie z.B. CNC (Computer Numerical Controlled)-Systemen zur Herstellung von Werkstücken.
Nicht nur durch diese Art von Funktionalität wird das Erstellen von Entwürfen und Modellen erleichtert, sondern natürlich auch durch die Weiterentwicklung der Benutzer-Schnittstellen von CAD-Systemen, die mit zunehmend verfügbarer Rechenleistung sowohl einfacher zu benutzen als auch zugleich mächtiger werden (im Sinne der Anzahl und des Umfanges der verfügbaren Werkzeuge in einem CAD-System).

Im Folgenden soll nun der Entwurf eines einfachen Zeichen-Systems entwickelt werden, der die grundlegenden Operationen jedes CAD-Systems veranschaulichen soll. Des

Weiteren werden im zweiten Teil unter dem Oberbegriff „Parametrische Modellierung“ Algorithmen zur Varianten-Erstellung mittels Constraints vorgestellt, die unter anderem eine wichtige Rolle spielen, wenn es um die Wiederverwendbarkeit von einmal erstellten Modellen geht.

Teil 1: Entwurf eines Zeichensystems

Im Folgenden soll beschrieben werden, welche Funktionen ein übliches CAD-System i.d.R. zur Verfügung stellt und welche Anforderungen an ein CAD-System generell gestellt werden. Anschließend wird auf die Umsetzung eines Zeichensystems eingegangen, das die grundlegenden Funktionen eines CAD-Systems implementiert.
Es wird erläutert werden, wie bestimmte Teil-Probleme im Umfeld von CAD-Systemen gelöst werden können, und dies durch Beispiele aus dem implementierten Zeichensystem veranschaulicht. Als Sprache für die Implementierung wird Borland Delphi verwendet, so dass alle Möglichkeiten einer objektorientierten Programmierung zur Verfügung stehen.

1 Persistenz

In der Regel sollen die erstellten geometrischen Objekte – und damit Objekt-Instanzen – in irgendeiner Form weiterverarbeitet werden. Dies kann im selben Programm zu anderer Zeit oder an anderem Ort sein, oder die Verarbeitung erfolgt in einem anderen System, wie z.B. einem System zur automatischen Fertigung.
In jedem Fall muss eine Schnittstelle vorgesehen sein, um die betreffenden Objekt-Instanzen im Arbeitsspeicher so auszugeben, dass sie aus dieser Ausgabe wieder vollständig rekonstruiert werden können. Der implementierte Entwurf stellt als einzige Schnittstelle eine auf Dateien basierende zur Verfügung, was jedoch keine große Einschränkung bedeutet, da die in den Dateien abgelegten Informationen auch unverändert über andere Medien wie Netzwerke o.ä. übertragen werden könnten – mit dem gleichen Ergebnis.

Als Datenformat zur Ein- und Ausgabe wurde XML gewählt. Neben den bekannten Vorteilen der „Universalität“ von XML bietet es sich unter anderem durch seine inhärente Eigenschaft an, bereits einen Mechanismus zum Abbilden von Verschachtelungen zu besitzen. Damit lässt sich ein Baum von Objekten, wie er in diesem Fall in einer Zeichnung (als Teilbaum) häufig vorkommt, relativ bequem in XML abbilden und wiederherstellen.

1.1 Implementation

1.1.1 Zugriff auf XML-Daten

Bei der vorliegenden Implementation wird der Zugriff auf XML-Daten mithilfe der Unit cXmlData von Sergey Kucherov durchgeführt. Diese Unit definiert im wesentlichen die Klasse TxDataNode, die die Funktionalität zur Verfügung stellt, um XML-Daten aus einer Datei oder einem String zu parsen und um einen XML-String aus einer TxDataNode-Instanz zu erzeugen.

Ein TxDataNode entspricht dabei genau einem Element einer XML-Datei, so dass ein ganzes Dokument in einem TxDataNode abgelegt wird, das den Root-Knoten des Dokuments enthält. Insofern handelt es sich hierbei um einen sog. DOM-Parser.
Ein TxDataNode hat als wichtigste Eigenschaften zum einen eine Liste von Attributen, die in dem entsprechenden XML-Knoten aufgeführt sind, und zum anderen eine Liste von TxDataNodes, die den Child-Knoten des aktuellen XML-Elements entsprechen.

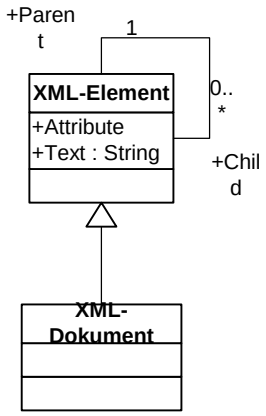

Abbildung 1: Struktur eines XML-Dokuments (vereinfacht)

Damit wird die (rekursive) Struktur eines XML-Dokuments mit XML-Elementen, die andere XML-Elemente enthalten können, in einen Baum von Objekten vom Typ TxDataNode abgebildet, der komplett im Speicher gehalten und über den der Inhalt des XML-Dokuments beliebig traversiert werden kann.

Daten werden somit in ein XML-Dokument geschrieben, indem die Eigenschaften eines TxDataNodes gesetzt werden. Das Lesen von Daten wird zum Lesen von Eigenschaftswerten.

1.1.2 Abbildung Objekte in XML

Zur Serialisierung, also zur Abbildung der Objekte in ein XML-Dokument, wird ein Mechanismus verwendet, der den verschiedenen Objekten die Aufgabe zukommen lässt, ihren eigenen Inhalt in einem XML-Knoten (Node) abzulegen.

Diese Funktionalität muss dann natürlich von den Objekten selber über eine entsprechende Schnittstellen-Signatur zur Verfügung gestellt werden. Diese Signatur wird in der Klasse TCadXMLItem definiert, von der alle anderen Klassen abgeleitet werden, deren Instanzen als XML abgelegt werden sollen.

Die Verwaltung eines XML-Dokuments wird von einer Instanz der Klasse TCadXMLManager übernommen. Die Methoden in TCadXMLManager arbeiten mit Instanzen von TCadXMLItem und XML-Nodes vom Typ TxDataNode. Hierbei werden beim Serialisieren über die Methode ToXMLNode aus TCadXMLItem TxDataNodes mit Inhalt gefüllt und beim Deserialisieren aus TxDataNodes Objekt-Instanzen erzeugt und mithilfe von FromXMLNode in TCadXMLItem mit Inhalt befüllt.

Diese XML-Funktionalität ist in der Unit uCadXMLManager enthalten.

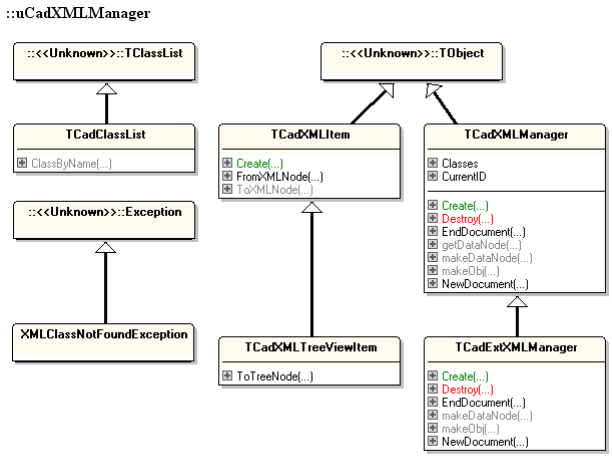

Abbildung 2: Klassen in der Unit uCadXMLManager

1.1.3 Koordination

Die Zentrale Klasse bei der Handhabung von XML-Dokument-Strukturen ist hier TCadXMLManager.

Diese Klasse wird zum einen benutzt, um aus einem vorgegebenen Objekt einen ‚minimalen' XML-Node zu erzeugen. In diesen Knoten werden dann die Instanz-Daten des Objektes abgelegt.

Zum anderen können aus XML-Nodes ‚leere' Objekte erzeugt werden, die dann wieder Instanzdaten aus dem Node übernehmen.

Ein ‚minimaler' XML-Knoten im Sinne des XMLManagers ist dabei ein XML-Element, das

- einen Bezeichner,
- das Attribut ‚id' mit einem eindeutigen Wert und
- das Attribut ‚classname' mit einem Klassennamen als Wert

hat. Diese minimalen Knoten werden in der Regel nur temporär erstellt und nicht als XML abgelegt, bevor sie mit noch weiteren Daten gefüllt worden sind. In XML-Notation sähe ein solcher Knoten aber wie folgt aus:

```
<ElementName id="UniqueID" classname="AClassName"/>
```

Ein solcher Knoten kann damit vom XMLManager eindeutig identifiziert und – durch die Angabe des Klassen-Namens – in eine Objekt-Instanz der angegebenen Klasse gelesen werden.

Jedes Objekt, das mithilfe des TCadXMLManagers nach XML konvertiert werden soll, bekommt dazu einen minimalen XML-Node übergeben, um darin weitere Informationen abzulegen.

Die wesentlichen Methoden in TCadXMLMananger sind:

```
function makeDataNode( Obj : TCadXMLItem; NodeName : String = '' ) :
TxDataNode; virtual;

function makeObj( Node : TxDataNode; ChildName : String = '' ) :
TCadXMLItem; virtual;
```

Die Semantik von makeDataNode(...) ist, aus dem übergebenen Objekt einen TxDataNode – also einen XML-Node – zu machen, der den Bezeichner in NodeName hat. Wird kein NodeName angegeben, wird stattdessen der Bezeichner „object" verwendet. In makeDataNode(...) wird ein minimaler Knoten erzeugt und anschließend die Methode

ToXMLNode(...) des übergebenen Objektes mit diesem Knoten aufgerufen, um den Inhalt des Objektes in den Knoten zu schreiben.

Analog dazu funktioniert die Methode makeObj, die aus einem übergebenen Node oder ggf. einem seiner Kinder eine Objekt-Instanz erzeugt. Hierbei ist zu bemerken, dass dafür eine Umwandlung einer Zeichenkette – nämlich des Klassennamens, der aus dem Attribut classname gelesen wird – in eine Delphi-Klassenreferenz erforderlich ist.
Anders als z.B. Java besitzt Delphi jedoch keine System-Funktion, um diese Umwandlung durchzuführen. Das hat insofern Konsequenzen, als jede Klasse, die auf diese Weise verarbeitet werden soll, explizit vom Programm registriert werden muss.

Durch die Baumstruktur eines XML-Dokumentes sind alle XML-Elemente vom Root-Element aus erreichbar. Das bedeutet für die (De-)Serialisierung, dass nur genau ein Node als Repräsentation des gesamten XML-Baumes verwendet wird.
Zur Serialisierung wird deshalb lediglich makeDataNode mit dem Objekt aufgerufen, das auch die Wurzel des Objekt-Baumes im Speicher ist. Daraus wird dann genau ein XML-Node, der dann mit makeObj wieder in die Wurzel des Objekt-Baumes verwandelt werden kann.

Wie bereits geschildert, arbeitet die Klasse TCadXMLManager mit Objekten vom Typ TCadXMLItem. Innerhalb dieser Objekte werden dann die Aktionen ausgeführt, die spezifisch für jedes Objekt sind. Eine genauere Erläuterung mit Beispielen folgt.

1.1.4 Serialisierung

Die Serialisierung der Objekte geschieht zum großen Teil in der Methode

```
TCadXMLItem.function ToXMLNode ( Manager : TCadXMLManager; Node :
TxDataNode ) : TxDataNode; virtual;
```

Die Semantik dieser Methode ist, den Inhalt des aktuellen Objektes in den XML-Knoten Node abzulegen, und den neuen Knoten als Ergebnis zurückzugeben. In der Regel wird das Ergebnis der übergebene Node sein, dies ist aber nicht unbedingt notwendig.

Die Methode kann den übergebenen Node benutzen, um beliebige Daten darin abzulegen. Insbesondere zum einen einzelne Eigenschaftswerte, die in der Regel in Attributen des Knotens abgelegt werden, und zum anderen Wertelisten und andere dem aktuellen Objekt untergeordnete Objekte, die selber wiederum in eigenen XML-Knoten abgelegt werden.
Hierbei wird unterschieden zwischen Objekten, die von TCadXMLItem abgeleitet sind, und sonstigen Objekten. Solche, die von TCadXMLItem abgeleitet sind, können einfach rekursiv mittels der übergebenen TCadXMLManager-Instanz in einem TxDataNode abgelegt werden.
Für die Verarbeitung anderer Objekte muss das aktuelle Objekt selber sorgen, also die aktuelle Methode ToXMLNode. Dies sollte jedoch möglichst vermieden werden, indem möglichst alle verwendeten Objekte, die für die Serialisierung relevant sind, von TCadXMLItem abgeleitet werden und damit ihre eigenen ToXMLNode-Methoden definieren.

In der Klasse TCadXMLItem ist die Methode ToXMLNode leer. Lediglich der übergebene Node wird als Standardverhalten aller Spezialisierungen von TCadXMLItem zurückgegeben.

Ein Beispiel für eine einfache Implementierung der Methode ToXMLNode findet man in der Klasse TCadPoint, die einen Punkt mit zwei Koordinaten repräsentiert:

```
function TCadPoint.ToXMLNode(Manager: TCadXMLManager;
  Node: TxDataNode): TxDataNode;
begin

  result := inherited ToXmlNode( Manager, Node );

  result.Attribute['xpos'] := FloatToStr( Self.x );
  result.Attribute['ypos'] := FloatToStr( Self.y );

end;
```

In dieser Methode wird zunächst die geerbte Implementierung aufgerufen, um einen TxDataNode zu erhalten. In diesem Fall ist das die Standard-Implementierung in TCadXMLItem, die lediglich den übergebenen Node zurückgibt.

Anschließend werden die beiden Eigenschaften für die Koordinaten in den Attributen ‚xpos' und ‚ypos' des Knotens abgelegt. Dieser geänderte Node wird schließlich auch an den Aufrufer zurückgegeben.

Bei etwas komplexeren Objekten gibt es wie schon erwähnt zwei Möglichkeiten, die Struktur abzulegen. Beide Möglichkeiten werden in TCadItem genutzt, da hier sowohl eine Instanz von TCadPoint abgelegt wird, als auch Eigenschaften des Zeichenstiftes (Pen), der nicht von TCadXMLItem abgeleitet ist:

```
function TCadItem.ToXMLNode(Manager: TCadXMLManager;
  Node: TxDataNode): TxDataNode;
var pn : TxDataNode;
begin
  pn := NIL;

  result := inherited ToXmlNode( Manager, Node );
  result.Attribute['linewidth'] := floattostr( Self.LineWidth );
  result.AddChild( Manager.makeDataNode( Self.Anchor , 'anchor' ) );

  try
    pn := Manager.getDataNode('pen');
    pn.Attribute['color'] := inttostr(Self.Pen.Color);
    pn.Attribute['style'] := inttostr(integer(Self.Pen.Style));
    pn.Attribute['mode']  := inttostr(integer(Self.Pen.Mode));
    result.AddChild( pn );
  except
    pn.Free;
  end;
end;
```

Hier wird die Eigenschaft LineWidth in einem Attribut abgelegt und danach mit einem Aufruf von TCadXMLManager.makeDataNode(...) ein neuer TxDataNode erzeugt, der das Objekt Self.Anchor (Instanz von TCadPoint) in einem Knoten mit dem Namen ‚anchor' enthält.
Dieser neue Node wird dann mit AddChild als Kinds-Knoten (ChildNode) an den aktuellen Knoten angefügt.

Die Eigenschaft Pen von TCadItem ist kein von TCadXMLItem abgeleitetes Objekt, sondern vom Typ Graphics.TPen. Daher besitzt das TPen-Objekt auch keine Methode ToXMLNode und muss somit von seinem Besitzer, also dem aktuellen TCadItem, serialisiert werden.
Dazu wird zunächst vom XML-Manager ein neuer TxDataNode mit dem Namen ‚pen' angefordert (Manager.getDataNode(...)), dieser dann mit Daten befüllt und an den aktuellen Knoten als Kind angefügt.

Mit diesem Vorgehen lassen sich also verschiedene Daten-Strukturen in XML ablegen. Eine weitere Möglichkeit zur Serialisierung der Eigenschaften von TPen wäre, einen TPen in einer eigenen Klasse ‚einzupacken' (wrap), die dann wiederum von TCadXMLItem abgeleitet sein könnte und in der die (De-)Serialisierung vorgenommen würde. Dies ist jedoch mit deutlichem Mehraufwand verbunden, der kaum gerechtfertigt erscheint im Bezug auf die geringe Menge der abzulegenden Daten, deren flache Struktur und die fehlende Notwendigkeit zur Wiederverwendung innerhalb des Projekts.

Ein ganz anderer Design-Ansatz ließe sich in Delphi ähnlich wie z.B. in Java verfolgen, indem die Schnittstelle von TCadXMLItem in ein Interface umgewandelt wird. Dieses Interface muss dann von betroffenen Klassen implementiert werden, die aber ansonsten von ganz anderen Klassen abgeleitet sein können. Diese ‚simulierte' Art der Mehrfachvererbung hat aber in Delphi einige gravierende Nachteile, weshalb hier von einem solchen Vorgehen abgesehen wird.

1.1.5 Deserialisierung

Die Hauptarbeit bei der Deserialisierung wird in der Methode

```
TCadXMLItem.procedure FromXMLNode ( Manager : TCadXMLManager; Node :
TxDataNode ); virtual; abstract;
```

erledigt.

Diese Methode ist das genaue Gegenstück zu ToXMLNode und hat die Semantik, das aktuelle Objekt mit den Werten aus dem XML-Knoten Node zu befüllen.

Dies kann sie tun, indem sie Attribut- und andere Werte aus dem Node ausliest und Eigenschaftswerte entsprechend setzt.

Ein einfaches Beispiel für eine Implementierung findet sich wiederum in der Klasse TCadPoint:

```
procedure TCadPoint.FromXMLNode(Manager: TCadXMLManager; Node:
TxDataNode);
begin
  inherited;
  Self.Fx := StrToFloat(Node.Attribute['xpos']);
  Self.Fy := StrToFloat(Node.Attribute['ypos']);
end;
```

Diese Methode sorgt dafür, dass die Attribute ‚xpos' und ‚ypos' aus dem Node gelesen werden und ihre Fließkommaentsprechung in die Eigenschaften für x- und y-Koordinate des Punktes übernommen werden.

Für die Deserialisierung stehen der Methode alle Funktionen zur Verfügung, die der Node bereitstellt, und damit vor allem auch der Zugriff auf Kinds-Knoten, also XML-Elemente, die dem aktuellen untergeordnet sind. Somit können die verschachtelten Strukturen, die bei der Serialisierung erstellt worden sind, auch wieder gelesen werden.

Sollen aus Kinds-Knoten wieder Objekte erzeugt werden, so kann dafür – analog zur Serialisierung – der übergebene XML-Manager benutzt werden, sofern es sich um Objekte handelt, die von TCadXMLItem abgeleitet wurden und bei der Serialisierung mittels eines TCadXMLManagers abgelegt wurden.

Dieses Vorgehen ist u.a. in TCadItem zu finden:

```
procedure TCadItem.FromXMLNode(Manager: TCadXMLManager; Node:
TxDataNode);
var p : TCadPoint;
    n : TxDataNode;
begin
```

```
    p := NIL;

    try

      Self.LineWidth := StrToFloat( Node.Attribute['linewidth'] );

      p := Manager.makeObj( Node, 'anchor' ) as TCadPoint;

      if Assigned( p ) then
      begin
        Self.Anchor.SetTo( P );
      end;

      n := Node.ChildByName['pen'];

      if Assigned( n ) then
      begin
        Self.Pen.Color := StrToInt( n.Attribute['color'] );
        Self.Pen.Style := TPenStyle( StrToInt( n.Attribute['style'] )
);
        Self.Pen.Mode  := TPenMode( StrToInt( n.Attribute['mode'] ) );
      end;

    finally
      p.Free;
    end;

    Self.ItemChanged;

  end;
```

Durch einen Aufruf der Methode TCadXMLManager.makeObj(...) wird aus einem benannten Kinds-Knoten des übergebenen Nodes eine Instanz von TCadXMLItem erzeugt und initialisiert. Dieses Objekt muss anschließend in die ursprüngliche bzw. die benötigte Klasse zurück-gewandelt werden.

Wie bereits beim Serialisieren, bedarf es einer gesonderten Behandlung, um die Eigenschaft Pen (TPen) wiederherzustellen. Hierzu wird direkt auf einen Kinds-Knoten zugegriffen und der Knoten mit der Bezeichnung ‚pen’ermittelt, aus dessen Attributen die entsprechenden Werte ausgelesen werden.

2 Repräsentation der Zeichnungs-Objekte

Alle Objekte, die in einer CAD-Zeichnung vorkommen, sind abgeleitet von TAbstractCadItem. Abstrakt bezieht sich dabei nicht nur auf die fehlende Implementierung bestimmter Methoden, sondern auch darauf, dass ‚abstrakte' Objekte keine wirkliche Entsprechung in einer Zeichnung haben, so wie etwa Punkte oder Bounding-Boxes.

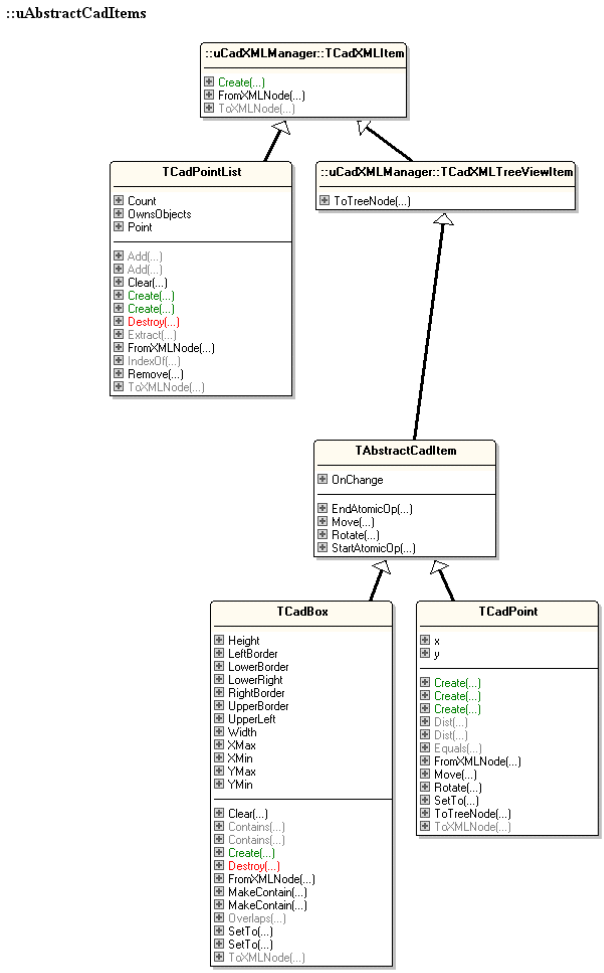

Abbildung 3: Klassenhierarchie in uAbstractCadItems

In der Unit uAbstractCadItems sind die wichtigsten Klassen zum einen TAbstractCadItem als Vorfahr aller weiteren CAD-Objekte und zu anderen die Klasse TCadPoint.

2.1 Ereignisbehandlung

Im Wesentlichen wird in TAbstractCadItem die Schnittstelle für einfache Operationen mit CAD-Objekten definiert (rotate(...) und move(...)). Außerdem stellt diese Klasse das grundlegende Verhalten zur Verfügung, das benötigt wird, um die Ereignisbehandlung der Objekte zu ermöglichen.

Die Ereignisbehandlung im Umfeld der Zeichnungs-Objekte bezieht sich darauf, Änderungen am Zustand eines Objektes so zu kommunizieren, dass andere Objekte, die im Objektbaum näher an der Wurzel liegen, falls nötig ihren Zustand den geänderten Werten anpassen können.

Das Konzept, das in TAbstractCadItem implementiert ist, verwendet dazu – wie in Delphi üblich – eine Eigenschaft (Property) OnChange, die ein anderes Objekt mit einem eigenen Methoden-Zeiger setzen kann. Die durch OnChange referenzierte Methode wird dann bei Zustands-Änderungen aufgerufen.

Des Weiteren besitzt TAbstractCadItem Methoden, um Operationen mit einem Objekt ‚atomar' zu machen – bezogen auf die Ereignisbehandlung. Das bedeutet, dass das Objekt selber oder ein anderes Objekt mehrere Zustands-Änderungen vornehmen kann, ohne dass für jede das Ereignis OnChange eintritt.
Sinnvoll ist dies vor allem dann, wenn ein Objekt ein oder mehrere andere Objekte referenziert und sich bei diesen als Listener für deren OnChange-Ereignis registriert hat. Kommt es nun zu einer Änderung, die sowohl das aktuelle Objekt als auch die Referenzierten betrifft, so kann mittels einer atomaren Operation dafür gesorgt werden, dass das OnChange-Ereignis dieser Objekt-Struktur nur einmal und zu einem genau definierten Zeitpunkt nach außen gereicht wird.

Ein Beispiel:
Eine Linie (TCadLine) enthält zwei Objekte vom Typ TCadPoint (Start- bzw. Endpunkt der Linie). Wird nun einer der beiden Punkte verschoben, so wird die Linie darüber benachrichtigt, kann sich anpassen und reicht dann das eine Änderungs-Ereignis weiter.

Wird allerdings die Linie verschoben, so muss sie ihre beiden Punkte nacheinander verändern. Dies würde ohne das Konzept der atomaren Operationen in zwei OnChange-Ereignissen resultieren, zwischen denen die Linie einen inkonsistenten Zustand hat, der nach außen nicht sichtbar gemacht werde sollte. Mithilfe der atomaren Operationen hingegen wird erst nach Abschluss aller Änderungen an der Linie genau ein OnChange-Ereignis ausgelöst, das über den neuen Gesamtzustand der Linie informiert.

Die Funktionalität der Ereignisbehandlung ist in folgenden Typen, Methoden und Eigenschaften von TAbstractCadItem implementiert:

```
Type TCadItemChangeNotify = procedure ( EventType : TCadEventType;
Sender : TObject; AffectedItems : TObjectList ) of object;
```

In TAbstractCadItem:

```
property OnChange : TCadItemChangeNotify read FOnChange write
SetOnChange;

procedure ItemChanged( EventType : TCadEventType = etUnknown; Sender
: TObject = nil; AffectedItems : TObjectList = nil ); overload;
virtual;

procedure StartAtomicOp;
procedure EndAtomicOp( EventType : TCadEventType = etUnknown );
```

Die Methode ItemChanged wird immer dann aufgerufen, wenn sich eine Zustands-Änderung am aktuellen Objekt ergeben hat. In der Basis-Implementierung in TAbstractCadItem wird hier zunächst geprüft, ob zurzeit eine atomare Operation stattfindet. Falls dies nicht der Fall ist, wird – falls gesetzt – die als OnChange registrierte Methode aufgerufen, um das Ereignis weiterzureichen. Ansonsten wird nur das Flag fWasChanged gesetzt, um zu signalisieren, dass eine Änderung stattgefunden hat.

Die Parameter für den Methodenaufruf sind der Typ des Ereignisses (EventType), das Objekt, dass das Ereignis zuletzt ausgelöst hat (Sender) und eine Liste von Objekten, die durch das Ereignis bereits betroffen sind (AffectedItems).
Die Signatur von ItemChanged ist kompatibel mit dem Typ TCadItemChangeNotify der OnChange-Eigenschaft. Dadurch ist es im einfachsten Fall möglich, ein Ereignis

unbehandelt weiterreichen zu lassen, indem X.OnChange := Self.ItemChanged gesetzt wird. Tritt dann ein OnChange-Ereignis in X auf, so wird auch in Self das OnChange-Ereignis ausgelöst.

Die Methoden StartAtomicOp und EndAtomicOp werden benutzt, um atomare Operationen zu beginnen oder zu beenden. Durch einen Aufruf-Zähler ist es auch möglich, atomare Operationen beliebig tief zu verschachteln.
Zu bemerken ist, dass EndAtomicOp nicht nur eine atomare Operation als beendet deklariert, sondern natürlich auch selber ein OnChange Ereignis auslöst, falls 1. der Aufruf die äußerste der verschachtelten atomaren Operationen betrifft und 2. eine Zustandsänderung durch ein gesetztes fWasChanged-Flag vermerkt ist. Dies ist auch der Grund, warum EndAtomicOp als Parameter einen Event-Typ erhalten kann: Dieser wird ggf. als Typ für das ausgelöste Ereignis verwendet.

2.2 Zeichenoperationen

Windows – und damit Delphi – stellt bereits eine Reihe von Zeichenoperationen zur Verfügung, die in Anwendungen u.a. genutzt werden können, um Steuerelemente innerhalb eines Programm-Fensters auf dem Bildschirm anzuzeigen. Dazu gehören Funktionen zum Zeichnen von Linien, Kreisen, Bögen und ähnlichem ebenso wie verschiedene „Stifte“ (Pens).
Dazu gehört aber auch eine einheitliche Abstraktion des Zeichnungs-Untergrundes, also der Zeichenfläche, der in Delphi in der Klasse TCanvas gekapselt ist. Dieser Canvas ist eine triviale Umschreibung von Objekten, auf denen gezeichnet werden kann, und ist quasi die Basisklasse, die alle Geräte(-Treiber) implementieren, um Zeichenoperationen zu ermöglichen. Insbesondere sind dies zum einen der Bildschirm und zum anderen der Drucker.
Zu den Anforderungen an ein Zeichenprogramm gehört fast zwangsläufig auch die Fähigkeit, Zeichnungen auf Papier, also über einen Drucker, auszugeben.

Das Konzept des Geräteunabhängigen Canvas schafft die Grundlage für die nahtlose Verwendung verschiedener Ausgabegeräte – aber nicht mehr. Denn ein Canvas-Objekt besitzt bedingt durch seine allgemeine Semantik keine Funktionen um auf spezifische Geräte-Eigenschaften einzugehen. Eine Canvas-Zeichenfläche besitzt nicht einmal eine Größe (in Pixeln o.ä.), die sich abfragen oder setzen ließe und ebenso wenig eine Auflösung bzw. Pixel-Dichte (DPI o.ä.).

All das macht es nötig, dass eine Anwendung, die auf verschiedenen Geräten Zeichnungen ausgeben soll, eigene Operationen und Eigenschaften mit den Geräten assoziiert.
Bei der vorliegenden Implementation werden deshalb die Klasse TCadDrawContext und ihre Ableitungen deklariert, die jeweils eine Referenz auf einen Canvas enthalten und darüber hinaus noch die Eigenschaft PixelPerUnit, die die Pixel-Dichte bezogen auf eine Zeichnungseinheit für den jeweiligen Kontext angibt.
Mit den Operationen PixelX/PixelY bzw. OrdX/OrdY kann dann direkt zwischen Pixeln und Ordinaten umgewandelt werden, und mit den entsprechenden Pixel-Werten auf den Canvas gezeichnet werden.

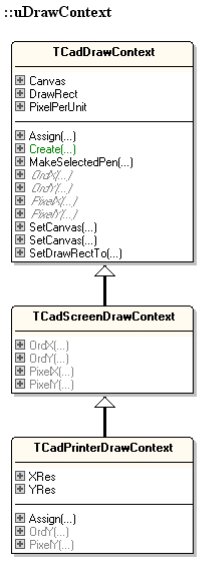

Abbildung 4: Klassen in uDrawContext

Somit wird jedes Zeichnungsobjekt mit einer Instanz eines TCadDrawContext assoziiert, das es anschließend benutzt, um sich selber auszugeben. Natürlich macht es i.d.R. keinen Sinn, innerhalb einer Zeichnung, also eines Zeichnungs-Objekt-Baumes, unterschiedliche Zeichnungs-Kontexte zu verwenden, weshalb eine Zeichnung als solche mit einem Kontext assoziiert werden kann und diesen rekursiv an ihre Bestandteile propagiert.

Während in TCadScreenContext für X- und Y-Ordinate der gleiche Faktor in Form von PixelPerUnit verwendet wird, besitzt TCadPrinterDrawContext zusätzlich die Eigenschaften XRes und YRes, die verschiedene Auflösungen für X- und Y-Richtung festlegen können. Hierbei wird bei der Umrechnung aber lediglich das Verhältnis der beiden zueinander berücksichtigt, um zusammen mit PixelPerUnit die korrekten Werte für eine Darstellung im richtigen Seitenverhältnis zu erhalten.

3 Die Zeichenobjekte

In der Unit uBasicCadItems werden letztlich die eigentlichen Bestandteile einer Zeichnung deklariert. Die Bestandteile sind dabei die geometrischen Objekte Linie, Kreis und Bogen sowie Container, die diese Objekte zusammenfassen, und die Klasse, die eine ganze Zeichnung repräsentiert.

Das Diagramm auf der folgenden Seite zeigt die Klassen in der Unit uBasicCadItems.

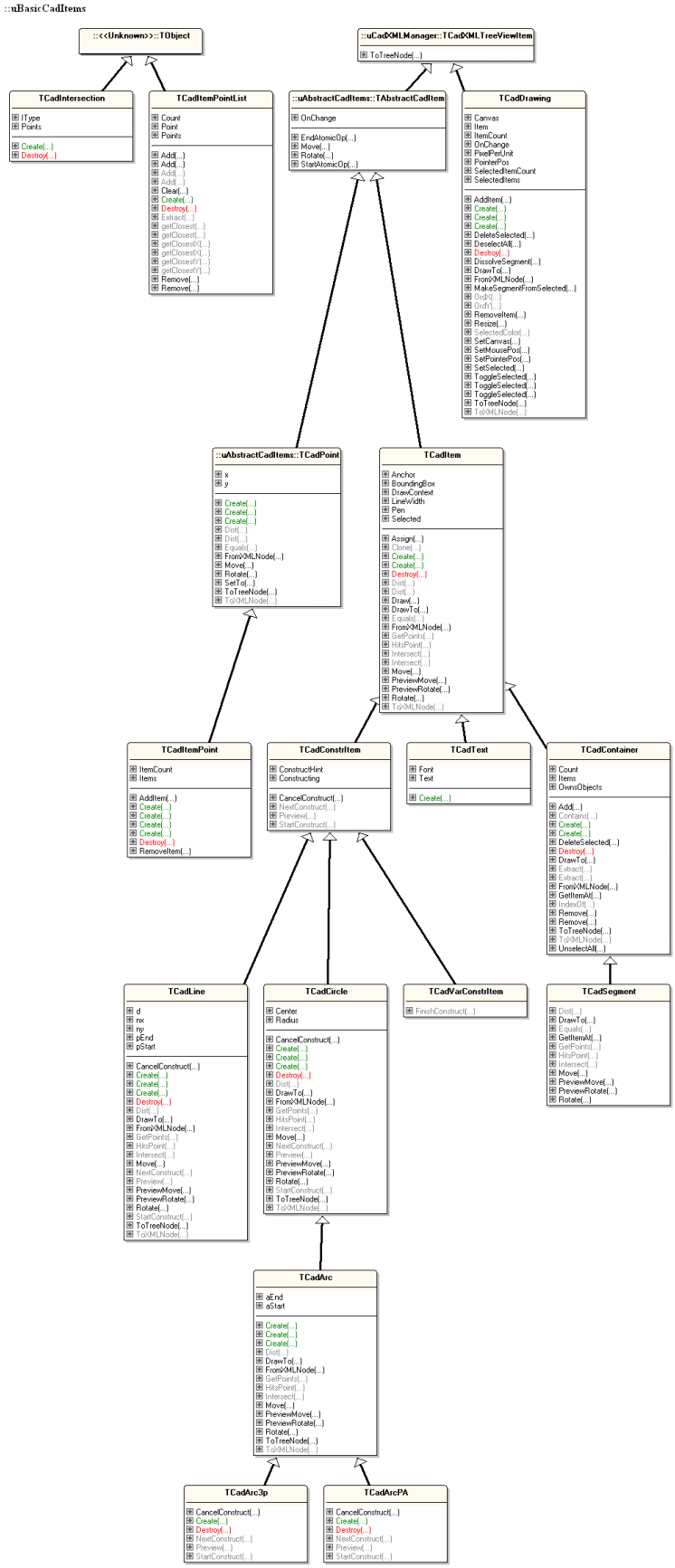

Abbildung 5: Klassen in uBasicCadItems

3.1 Das Zeichnungs-Objekt TCadDrawing

Die Klasse TCadDrawing definiert den äußersten Container einer Zeichnung. Alle Objekte der Zeichnung werden in einer Instanz von TCadDrawing verwaltet. Die Zeichnungsobjekte werden nach ihrer Instanziierung mit AddItem der Zeichnung hinzugefügt und gehen damit in den ‚Verantwortungsbereich' des Zeichnungsobjektes über.
Obwohl das Interface von TCadDrawing viele Gemeinsamkeiten mit dem in TAbstractCadItem definierten hat, ist TCadDrawing nicht von TAbstractCadItem abgeleitet. Der wichtigste Grund hierfür ist, dass TCadDrawing per Definition der äußerste Container von Zeichnungs-Objekten ist. Es soll also nicht möglich sein, eine Zeichnung in einen anderen Container einzufügen. Durch diese Deklaration von TCadDrawing wird das verhindert. TCadDrawing wird quasi markiert mit ‚dies ist kein normales Zeichnungs-Element'.

Die Operationen, die TCadDrawing bereitstellt, decken die folgenden Bereiche ab:

- Hinzufügen und Entfernen von Zeichnungs-Objekten,
- Verwaltung von markierten (selected) Elementen der Zeichnung,
- Erzeugen und Auflösen von Segmenten,

sowie außerdem die Funktionen zum Zeichnen der gesamten Zeichnung und zum (De-) Serialisieren.

3.2 Die Klasse TCadItem

Die Grundlage und der Vorfahr aller Zeichnungs-Objekte, die in einer Zeichnung verwendet werden, ist die Klasse TCadItem. Ähnlich wie ihr Vorfahr TAbstractCadItem definiert sie (fast) kein konkretes Verhalten, sondern hauptsächlich die gemeinsame Schnittstelle aller Zeichnungs-Objekte. Zu dieser Schnittstelle gehören u.a.:

- ein repräsentativer Punkt (Anchor) des Objektes,
- ein Stift (TPen), der zum Zeichnen verwendet wird,
- Operationen, um den Abstand des gesamten Objektes zu einem Punkt zu ermitteln,

- die DrawTo-Prozedur, die das Objekt auf einen gegebenen DrawContext zeichnet,
- Preview-Funktionen, die eine Vorschau des Objektes währen einer Benutzer-Interaktion erzeugen

und die Intersect-Operationen, die Schnittpunkte von Objekten ermitteln.

3.2.1 Die Funktion Dist(…)

Die Funktion, die zum Ermitteln des Abstandes des Objektes zu einem Punkt benutzt wird

```
function Dist( x, y : Float ) : Float; overload; virtual;
```

besitzt zwar bereits eine Standard-Implementierung in TCadItem, die den Abstand zum Anchor-Punkt zurückgibt, in Ableitungen von TCadItem muss sie jedoch i.d.R. überschrieben werden, um den Abstand des jeweiligen *Objektes* zum Punkt zu liefern. Wie dieser Abstand ermittelt wird, ist komplett objektspezifisch. Bei einer Linie wird z.B. zunächst versucht, einen Lotpunkt zu ermitteln und der Abstand zu diesem ggf. zurückgegeben, bei einem Kreis muss nur der Abstand zum Mittelpunkt abzüglich des Radius ermittelt werden.

3.2.2 Die Prozedur DrawTo(…)

Jedes Mal, wenn ein Objekt auf einem Canvas gezeichnet werden soll, wird die Methode

```
procedure DrawTo ( C : TCadDrawContext = nil ); overload; virtual;
```

aufgerufen. Sie wird in Nachkommen von TCadItem überschrieben, um die nötigen Zeichenoperationen auf dem Canvas im entsprechenden Kontext auszuführen. Dabei ist es Sinnvoll, die eigentlichen Zeichenoperationen in einer eigenen parametrisierten Methode zu kapseln, da diese Funktionalität i.d.R. auch von den Preview-Methoden benötigt wird.

3.2.3 Die Prozeduren PreviewMove(…) und PreviewRotate(…)

Während der Benutzer eine der Operationen ‚verschieben' oder ‚rotieren' interaktiv durchführt, sollen jeweils Vorschau-Versionen des bewegten Objektes gezeichnet werden, um dem Benutzer das Ergebnis der Operation anzuzeigen, bevor die sie abgeschlossen ist. Dafür werden die Methoden PreviewMove und PreviewRotate implementiert, so dass sie eine Vorschau des aktuellen Objektes entsprechend den übergebenen Parametern der betreffenden Operation zeichnen.

3.2.4 Die Prozedur GetPoints(…)

Zur Unterstützung des Benutzers bei Zeichenoperationen sollen markante Punkte in der Zeichnung markiert werden, mit denen auch eine gewisse Interaktion im Sinne von Skizzierhilfen möglich ist. Dazu implementiert jedes Zeichnungsobjekt die Methode

```
function GetPoints( PL : TCadPointList ) : integer; virtual;
```

Die jeweilige Implementierung entscheidet dann darüber, welche Punkte für das aktuelle Objekt relevant sind und trägt diese in die übergebene Punkt-Liste ein. Bei einer Linie sind das der Start- und Endpunkt, bei einem Bogen z.B. Start-, End- und Mittelpunkt und bei einem Kreis nur der Mittelpunkt.

3.3 Schnittmengen von Objekten

Da es in verschiedenen Situationen sinnvoll ist, dem Benutzer Schnittpunkte von Zeichenobjekten anzuzeigen und sie ggf. als Skizzierhilfe zu verwenden, wird ein Mechanismus benötigt, der die Schnittpunkte eines beliebigen Zeichnungs-Objektes mit einem anderen ermitteln kann. Es wird also folgende Funktion benötigt:

$$\text{Intersect} : \overline{TCadItem} \times \overline{TCadItem} \rightarrow TCadPoint^{n}$$

Diese Funktion so universell zu implementieren, wie sie in der Notation erscheint, ist aber praktisch nicht direkt umsetzbar: Diese Funktion müsste in der Lage sein, jedes beliebige TCadItem mit jedem anderen zu schneiden. Dafür müsste sie die Details jedes möglichen TCadItem ‚kennen' und ‚wissen', wie jedes mögliche Item mit jedem anderen geschnitten wird. Das würde auch bedeuten, dass diese Funktion erweitert werden muss, sobald neue Zeichen-Objekte implementiert werden.

Im vorliegenden Fall wird deshalb eine Lösung verwendet, die – basierend auf einer impliziten Ordnungsrelation – die Schnittmengen leicht erweiterbar ermitteln kann. Die zugrunde liegende Annahme ist hierbei, dass in jedem Fall die Kommutativität des Schnitt-Operators gilt:

$$\text{Intersect}(A,B) = \text{Intersect}(B,A)$$

Hierfür wird im Folgenden die Notation A.Intersect(B) bzw. B.Intersect(A) verwendet.

Des Weiteren wird eine Ordnungsrelation $\succ$ auf allen Klassen definiert, die von TCadItem abgeleitet sind (im Folgenden mit $\overline{TCadItem}$ bezeichnet). Diese Relation kann man bezeichnen mit ‚kennt' oder ‚knows'. Ihre Bedeutung ist, dass eine Klasse A eine andere Klasse B ‚kennt', wenn in A die Methode A.Intersect(B) implementiert ist. Das heißt umgangssprachlich also, wenn A ‚weiß', wie es selber mit B geschnitten werden kann. Dabei muss gelten:

I. $\forall A, B \in \overline{TCadItem} : A \succ B \Leftrightarrow A = B \vee \neg(B \succ A)$

Jede Klasse muss also zu jeder anderen eindeutig in Relation zu setzen sein. Ist dies gegeben, lassen sich alle Klassen für jede Klasse X in zwei Mengen einteilen:

$$Items_{known}(X) = \{i \mid i \in \overline{TCadItem} \wedge X \succ i\}$$
$$Items_{unknown}(X) = \{i \mid i \in \overline{TCadItem} \wedge i \notin Items_{known}(X)\}$$

Damit kann die Schnittfunktion definiert werden als:

$$\text{Intersect}(A,B) = \begin{cases} A.\text{Intersect}(B), \text{falls } B \in Items_{known}(A) \\ B.\text{Intersect}(A), \text{sonst } (\Leftrightarrow B \in Items_{unknown}(A)) \end{cases}$$

Mit dieser Definition kann also die Funktion Intersect so implementiert werden, dass jede Implementierung nur eine bestimmte, feste Menge von Klassen behandeln können muss. Soll die Schnittmenge mit einer ‚unbekannten' Klasse ermittelt werden, so muss die ‚unbekannte' Klasse die Operation implementieren und somit kann ihre Implementierung benutzt werden.

Es muss in jedem Fall sichergestellt sein, dass die ‚knows'-Relation gemäß der Forderung I. für alle Klassen definiert ist. Wie diese Ordnung konkret umgesetzt wird, ist vollkommen offen und der jeweiligen Implementation überlassen – die Ordnung wird durch die Implementierung der Intersect-Methoden gegeben, nicht umgekehrt. Wenn aber ein bestehendes System durch neue Klassen erweitert werden soll, so ist klar, dass die Menge $Items_{known}$ der neuen Klassen auf jeden Fall alle bereits im System bestehenden Klassen umfassen muss, da diese ja die neuen Klassen **nicht** ‚kennen'.

3.3.1 Die Intersect(…)-Methode

Im Programm wird das gerade beschriebene Verhalten in der Methode TCadItem.Intersect(…) und ihren Overrides umgesetzt.

```
function Intersect( Item : TCadItem; Intersection :
TCadIntersection; TryReverse : boolean = true) : TCadIntersection;
overload; virtual;
```

Diese Methode hat die Semantik, die Schnittmenge (genau genommen: die Menge diskreter Schnittpunkte) des aktuellen Objektes mit dem übergebenen Item zu ermitteln und in einem TCadIntersection-Objekt zurückzugeben. Wird ein Intersection-Objekt übergeben, so wird dieses benutzt, ansonsten wird ein neues Objekt instanziiert.

Mit dem Parameter TryReverse wird das Verhalten festgelegt für den Fall, dass das aktuelle Objekt das Item nicht ‚kennt'. Ist TryReverse auf True gesetzt, so wird in diesem Fall letztendlich mit einem Aufruf von Item.Intersect(Self,…) versucht, die Schnittmenge zu ermitteln. Dies geschieht in der Standard-Implementierung in TCadItem:

```
function TCadItem.Intersect(Item: TCadItem; Intersection:
TCadIntersection; TryReverse: boolean): TCadIntersection;
begin
  if ( TryReverse ) then
    result := Item.Intersect( Self, Intersection, False )
  else
    result := Intersection; // raise "Cannot intersect"
end;
```

Diese Implementierung ist ein Spezialfall. Der abstrakten Natur von TCadItem entsprechend ist die Menge $Items_{known}$ von TCadItem nämlich leer. Deshalb findet keine Überprüfung statt, ob das übergebene Item vielleicht in $Items_{known}$ enthalten ist – es ist es nicht. Damit muss, falls TryReverse auf True gesetzt ist, die Intersect-Methode des Items aufgerufen werden. Ist TryReverse False, so handelt es sich genau genommen um einen Programm-Fehler, da dies bedeuten würde, dass weder $A \succ B$ noch $B \succ A$ gilt und dass somit kein Schnitt von A und B berechnet werden kann.
TryReverse kann auch als Flag verstanden werden, das den Abbruch der Rekursion von Intersect(…)-Aufrufen bewirkt, wenn es auf False gesetzt wird.

Für überschriebene Implementierungen gibt dieser Mechanismus den Aufbau der Intersect-Methode vor:

1. Prüfen, ob das übergebene Item in $Items_{known}$ enthalten ist
2. Wenn ja, Schnittmenge berechnen und zurückgeben
3. Wenn nein, inherited Intersect(…) aufrufen.

Ein Beispiel aus TCadLine:

```
function TCadLine.Intersect(Item: TCadItem; Intersection:
TCadIntersection; TryReverse: boolean): TCadIntersection;
…
begin
```

```
  …

      if Item is TCadLine then
      begin
        // Schnittmenge berechnen und in Itersection eintragen…
        …
        result := Intersection;

      end
      else
        result := inherited Intersect( Item, Intersection,
TryReverse );

  …

end;
```

Hier ist ersichtlich, dass TCadLine quasi das erste Element in der Ordnung ist:

$$Items_{known}(TCadLine) = \{TCadLine\},$$

TCadLine ‚kennt' also nur sich selbst. (TCadLine ist quasi direkt von TCadItem abgeleitet und erbt daher keine ‚bekannten' Items.)

In TCadCircle.Intersect findet sich bereits die Unterscheidung in TCadLine und TCadCircle, damit ist

$$Items_{known}(TCadCircle) = \{TCadLine, TCadCircle\}$$

Eine Instanz von TCadCircle kann also den Schnitt von sich selber mit einer anderen TCadCircle-Instanz oder einer TCadLine-Instanz berechnen, aber keinen anderen.

Bemerkenswert ist auch die Klasse TCadArc, die von TCadCircle abgeleitet ist. In ihr werden keine eigenen Schnitt-Berechnungen implementiert, sondern die geerbte Methode aus TCadCircle verwendet und danach die Punkte verworfen, die nicht auf dem Bogen liegen. Insofern kann man sagen, dass

$$Items_{known}(TCadArc) = Items_{known}(TCadCircle)$$

Allerdings unterschieden sich die beiden Klassen in den Implementierungen der HitsPoint()-Methode, die bei der Schnittberechnung verwendet wird, wodurch sich wiederum auch die Schnittberechnung in beiden Klassen unterscheidet, so dass sich zumindest auch sagen lässt, dass

$$Items_{known}(TCadArc) = Items_{known}(TCadCircle) \cup \{TCadArc\}$$

4 Konstruktion von Zeichenobjekten

Die interaktive, schrittweise Konstruktion von Zeichnungselementen ist eine wichtige und zugleich recht komplexe Funktion in einem CAD-System. Die Komplexität liegt darin begründet, dass zum einen mehrere Arten der Benutzer-Eingabe berücksichtigt werden sollten wie z.B. Maus und Tastatur, und zum anderen verschiedenen Zeichnungs-Elemente eine große Varianz an notwendigen Schritten und Eingaben zur Konstruktion aufweisen: Eine Linie kann z.B. durch zwei Punkte definiert werden, ebenso ein Kreis. Der Kreis besitzt aber einen Freiheitsgrad weniger als die Linie und kann deshalb genauso gut durch einen Punkt und einen Abstand definiert werden.

Möglichst viele dieser Varianten sollten im Benutzerinterface abgedeckt sein, um hohe Flexibilität zu erhalten – sowohl bezogen auf die Auswahlmöglichkeiten des Benutzers bzgl. einer Konstruktions-Methode, als auch bezogen auf die Erweiterbarkeit des Systems um weitere komplexe Elemente.

Im vorliegenden System wird zur Lösung der Problemstellung der Konstruktion ein Ansatz verfolgt, der versucht, folgende zwei Hauptforderungen umzusetzen:

1. Strikte Trennung von Benutzerschnittstelle und Zeichnungsobjekten
2. Hohe Flexibilität für Erweiterungen um neue Zeichnungsobjekte

Diese Forderungen implizieren mehr oder weniger direkt die Feststellung, dass ein Mechanismus benötigt wird, der eine Art bidirektionale Kommunikation zwischen dem User-Interface und einem Zeichnungsobjekt ermöglicht. Zum einen soll nämlich der Benutzer über das User-Interface die Schritte zur Konstruktion kontrollieren, zum anderen muss ein zu konstruierendes Objekt aber auch Vorgaben machen, welche Schritte in welcher Reihenfolge nötig sind.

4.1 InputItems

Um die Unabhängigkeit von Zeichnungsobjekten und Benutzerschnittstelle zu erreichen, wird eine Abstraktionsschicht verwendet: InputItems.

InputItems, also Klassen, die von TCadInputItem abgeleitet sind, werden benutzt, um die Eingaben des Benutzers in ein allgemeines, zweckgerechtes Format zu bringen. Außerdem werden sie benutzt, um der User-Interface-Schicht mitzuteilen, welche Art von Eingabe benötigt wird.

Die Input-Items funktionieren als Schnittstelle, als eine Art Adapter, zwischen dem Benutzer-Interface und den Zeichnungsobjekten. Dies können sie tun, indem sie quasi zwei verschiedene Sichten auf die gleichen Daten über zwei verschiedene Schnittstellen bieten: Auf der einen Seite funktionieren die verschiedenen InputItems (fast) ausschließlich über die Ein- oder Ausgabe von Punkten, also Koordinaten, auf der anderen Seite stellen sie spezielle Daten bereit, die spezifisch für jedes InputItem aus den Punkten ermittelt werden. Dadurch wird eine hohe Unabhängigkeit der beiden Nutzer der InputItems (Zeichnungsobjekte und Benutzer-Interface) erreicht und damit eine flexibel erweiterbare Struktur.

Da die InputItems (i.d.R.) lediglich die Eingabe von einem oder mehreren Punkten vom Benutzer erfordern, kann sich die Benutzer-Schnittstelle wo nötig oder erwünscht auf die minimale Funktion, Punkt-Eingaben zur Verfügung zu stellen, beschränken.

Entgegen der einfachen Annahme, Daten müssten nur vom Benutzer an die Objekte übermittelt werden, ist auch ein – wenn auch geringer – Informationsfluss in die umgekehrte Richtung zumindest wünschenswert. Beide Informationsflüsse werden in den InputItems abgebildet.

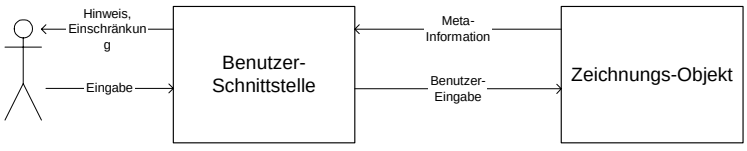

Abbildung 6: Informationsfluss

Der eigentliche Vorgang des Konstruierens eines Objektes besteht in der Regel aus mehreren Schritten. Diese Schritte müssen nicht alle genau gleich sein, sondern können auch innerhalb eines Konstruktions-Vorganges variieren. Dies gilt insbesondere für das Benutzer-Interface, das durch seine interaktive und dynamische Struktur auf verschiedene

Eingaben reagieren muss, aber auch für abstraktere Konstruktionsschritte, wie z.B. die Eingabe eines Punktes und eines Radius' für einen Kreis.

Die flexibelste Art, eine Konstruktion vorzunehmen wäre damit, alle nötigen bzw. möglichen Konstruktionsschritte zu ermitteln und dem Benutzer alle gleichzeitig zur Auswahl anzubieten. Der Benutzer kann dann flexibel die Reihenfolge auswählen, in der er die Schritte bearbeiten möchte. Ein Kreis (Punkt und Radius) könnte dann jederzeit konstruiert werden, indem entweder erst ein Abstand gewählt wird und dann ein Mittelpunkt, oder umgekehrt. Dieses Vorgehen würde es dem Benutzer in bestimmten Situationen erleichtern, Objekte in Abhängigkeit von anderen Objekten zu erzeugen und ihm jederzeit die freie Wahl lassen, ob er den einen oder anderen Weg zur Konstruktion gehen möchte.

Der Nachteil einer solchen Benutzer-Schnittstelle ist natürlich die höhere Komplexität, die darauf beruht, dass der Benutzer auswählen muss, was in der aktuellen Situation ein günstiges Vorgehen zur Konstruktion ist. Des weiteren lassen sich Konstruktionen denken, bei denen nicht von vorneherein bestimmt werden kann, welche Schritte notwendig sind, weil u.U. einzelne Schritte von anderen in der Art abhängen, dass die Möglichkeiten für den Schritt n+1 überhaupt erst bekannt sind, wenn die Schritte 1 bis n bereits ausgeführt worden sind. So eine Konstruktion würde ein komplexes Benutzer-Interface und hochpolymorphe Objekte dahinter erfordern. Dies ist praktisch aber nur mit geringem Nutzen verbunden, so dass das gesamte Konzept zu hinterfragen wäre.

In der vorliegenden Implementation wird deshalb eine strikt sequenzielle Abfolge von Konstruktionsschritten für jedes Zeichnungsobjekt vorausgesetzt. Die Abfolge an sich muss aber nicht statisch sein, sondern kann sich vor bzw. nach jedem Schritt den bereits gemachten Schritten anpassen. Es steht lediglich zu jedem Zeitpunkt fest, welcher Konstruktionsschritt jetzt stattfinden muss.

4.2 Eine einfache State-Machine

Das schrittweise Abarbeiten von Konstruktionsschritten mit dynamischen Entscheidungen zwischen den Schritten legt schon die Umsetzung als einfachen Automaten, als State-

Machine nahe. Das Verhalten der Zeichnungsobjekte zur Konstruktion implementiert daher eine einfache State-Machine, die jeweils einen Schritt weiter arbeitet, wenn der Benutzer eine Eingabe gemacht hat. Wegen der hohen Varianz der Konstruktionsschritte erfordert nämlich jedes Zeichnungsobjekt seinen eigenen Automaten, der die Art und Reihenfolge der Schritte ermittelt und abarbeitet.

Die Schnittstelle, die hierfür benötigt wird, ist in der Klasse TCadConstrItem definiert (Constr für Constructable). Im Wesentlichen sind es die Methoden:

```
function StartConstruct( StartPoint : TCadPoint = NIL ) :
TCadInputItem; virtual;

function NextConstruct : TCadInputItem; virtual;

procedure CancelConstruct; virtual;
```

Die StartConstruct und NextConstruct Methoden kann man als ‚pseudo-abstrakt' bezeichnen, weil sie zwar nicht als abstrakt deklariert sind, aber ohne überschrieben zu werden auch kein sinnvolles Verhalten nach außen hin aufweisen. Man kann also eine abgeleitete Klasse instanziieren, ohne die Methoden überschrieben zu haben, jedoch ist das entsprechende Zeichnungsobjekt nicht konstruierbar, was dem Sinn der Klasse TCadConstrItem widerspricht. Aus diesem Grund ist es auch denkbar, TCadConstrItem als Interface zu deklarieren, das von anderen Klassen bei Bedarf implementiert werden kann.

Der einfache Automat wird mit der Methode StartConstruct initialisiert. Als Parameter kann ggf. der Start-Punkt der Konstruktion bereits übergeben werden. (In der Regel wird der Start-Punkt, falls gesetzt, als erster Punkt des ersten InputItems übernommen. Dies ist jedoch nicht zwingend erforderlich.) Je nach Art der benötigten Eingabe und abhängig vom übergebenen Start-Punkt wird ein InputItem zurückgeliefert, das den nächsten Eingabe-Schritt bestimmt. Wenn die Eingabe abgeschlossen ist, wird NextConstruct verwendet, um die State-Machine einen Schritt machen zu lassen, und als Ergebnis wird wiederum ein passendes InputItem zurückgegeben oder eine leere Referenz (Null-Pointer), um das Ende der Konstruktionsschritte zu signalisieren. Dies entspricht dann dem finalen oder Halte-Zustand des Automaten und kennzeichnet damit den Zeitpunkt, an dem das konstruierte Objekt bereit ist, in die Zeichnung übernommen und gezeichnet zu werden.

Bei jedem Aufruf von NextConstruct muss das Element, das sich in Konstruktion befindet, also prüfen, in welchem Zustand es sich bereits befindet, und anhand dessen entscheiden, was für eine Eingabe als nächstes benötigt wird, und schließlich ein entsprechendes InputItem zurückgeben.

Die Methode CancelConstruct kann verwendet werden, um ein vorzeitiges Ende der Konstruktion z.B. einen Abbruch durch den Benutzer zu signalisieren.

Für Konstrukte, bei denen z.B. der Benutzer die Anzahl der Schritte vorgibt, wie z.B. bei Linienzügen, wird in der Klasse TCadVarConstrItem die Deklaration der Methode

```
function FinishConstruct : boolean; virtual; abstract;
```

zu denen aus TCadConstrItem hinzugefügt, mit der sich – anders als mit CancelConstruct – das erfolgreiche Ende der Konstruktion an das Zeichnungs-Element kommunizieren lässt. Damit können bei einem TCadVarConstrItem beide Seiten, also das Zeichnungs-Element und das Benutzer-Interface, die Konstruktion für beendet erklären, wobei beim Zeichnungs-Element immer noch die Möglichkeit eines ‚Veto' besteht, indem in FinishConstruct die Beendigung zurückgewiesen wird und dies über die Rückgabe von False mitgeteilt wird. Im letzteren Fall wäre der Aufruf von FinishConstruct gleichbedeutend mit dem Aufruf von CancelConstruct.

Der gesamte Mechanismus kommt also einem interaktiven Benutzerinterface sehr entgegen, weil lediglich zwischen zwei Eingabe-Schritten eine Methode aufzurufen ist und die eigentliche Eingabe völlig unabhängig davon beliebig komplex und zeitlich unbeschränkt stattfinden kann.

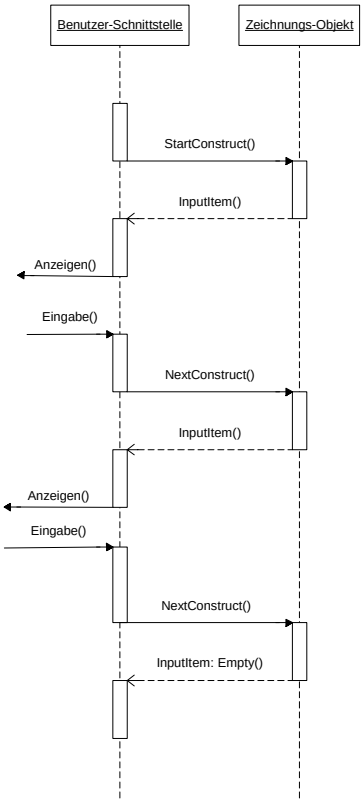

Abbildung 7: Beispielhafte Sequenz bei der Eingabe/Konstruktion

4.3 Verwendung von InputItems

Die InputItems sind in der Unit uCadInputItems deklariert:

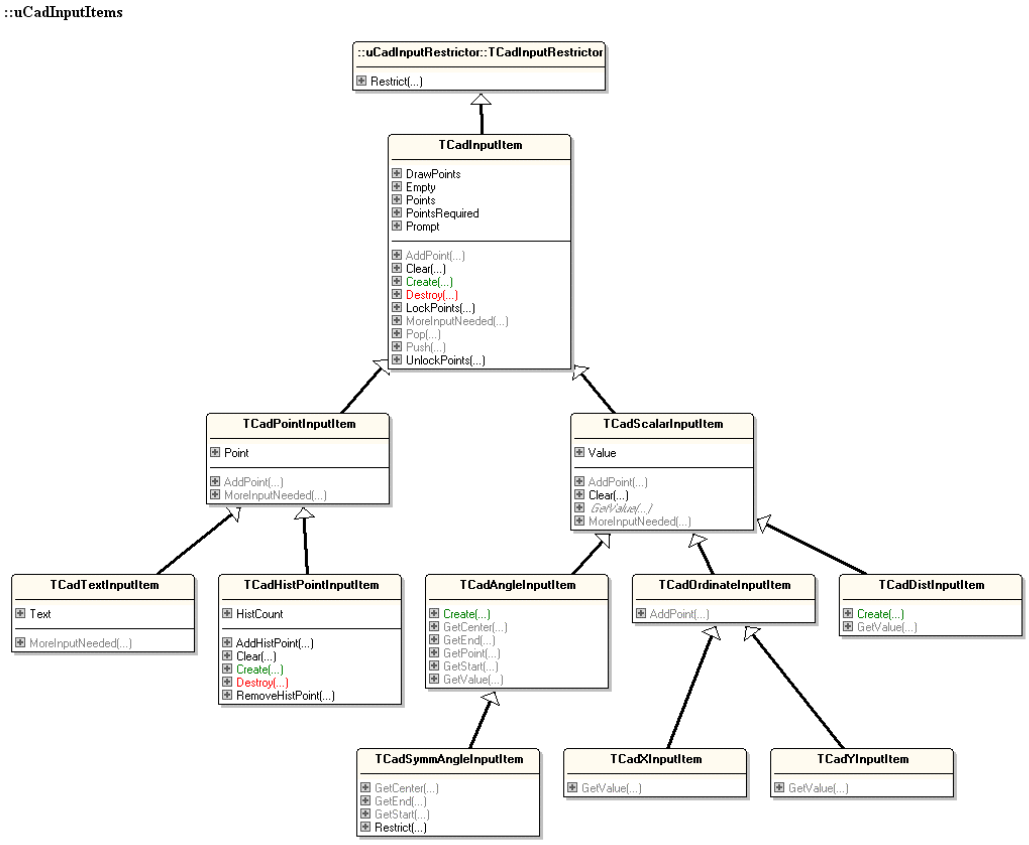

Abbildung 8: Klassen in uCadInputItems

Alle InputItems sind von der Klasse TCadInputRestrictor abgeleitet und erben von ihr die Schnittstelle, um Eingaben von Koordinaten auf bestimmte Werte zu beschränken.

In TCadInputItem werden dann die beiden Methoden hinzugefügt, die für die Benutzer-Interface-Schicht am wichtigsten sind:

```
function AddPoint ( P : TCadPoint ) : boolean; virtual;
```

und

```
function MoreInputNeeded : boolean; virtual;
```

Im einfachsten Fall nimmt die Benutzer-Schnittstelle solange die Eingabe von Punkten vom Benutzer an und fügt diese jeweils mit AddPoint dem InputItem hinzu, bis die Funktion MoreInputNeeded als Ergebnis False liefert. Wenn dieser Zustand erreicht ist, bedeutet dies, dass keine weitere Eingabe mehr nötig ist, damit das InputItem seine Aufgabe erfüllen kann – es bedeutet aber nicht unbedingt, dass keine Eingabe mehr erlaubt ist, was u.U. auch sinnvoll sein kann.

Die von TCadInputItem abgeleiteten Klassen unterscheiden sich im Wesentlichen darin, wie viele Punkte sie erwarten, bis sie keine weiteren mehr benötigen, und darin, welche Verarbeitung der eingegebenen Punkte sie für das Zeichnungsobjekt zur Verfügung stellen. So erwarten z.B. TCadXInputItem und TCadYInputItem jeweils nur einen Punkt als Eingabe, liefern aber über GetValue entweder die X- oder die Y-Ordinate zurück. TCadDistInputItem erwartet zwei, TCadAngleInputItem drei Punkte als Eingabe.

Je nach Klasse eines InputItems kann das Benutzer-Interface aber auch entscheiden, wie es die Eingabe vom Benutzer entgegennimmt. Für von TCadScalarInputItem abgeleitete Klassen kann z.B. einfach ein skalarer Wert erfragt und direkt im InputItem gesetzt werden. Dies ist im Eingabe-Form in der Unit uInputDlg beispielhaft umgesetzt:

```
procedure TInputDlg.SetInputItem(const Value: TCadInputItem);
begin
…
  if Value is TCadYInputItem then
  begin

    Self.inputY( Value );

  end
  else
  if Value is TCadPointInputItem then
  begin

    Self.inputPoint( Value );

  end
  else
  if Value is TCadDistInputItem then
  begin
```

```
      Self.inputDistance( Value );

    end
    else …
…
end;
```

Damit übernehmen die InputItems also nicht nur die Aufgabe eines einfachen Adapters sondern können sehr flexibel und feingranular verwendet werden, um die Benutzereingaben zu koordinieren, entgegenzunehmen und auszuwerten.

4.4 Das Benutzerinterface

Neben der Kommunikation zwischen der Benutzerschnittstelle und den Zeichnungsobjekten gibt es natürlich auch die Kommunikation zwischen Benutzer und Anwendung (s Abbildung 6: Informationsfluss). Um diese Kommunikation effizient zu gestalten, sollte die Anwendung spezielle Funktionen zur Verfügung stellen, die man mit ‚Skizziertechniken' umschreiben kann.

4.4.1 Skizziertechniken

In der Regel gibt es in einer Konstruktion minimale Strukturen, die in ihrer Art häufig erstellt werden müssen. Solche minimalen Strukturen sind z.B. Linien, die genau waagerecht oder senkrecht verlaufen, ebenso wie Linien, die exakt an andere Objekte anschließen etc. Um solche häufig wiederkehrenden Elemente einfach eingeben zu können, muss das User-Interface genau hierfür Unterstützung anbieten.
Dies kann u.a. dadurch erfolgen, dass während der Eingabe laufend bestimmte Bedingungen überprüft werden, und bei Erreichen bestimmter Grenzwerte die Eingabe durch die Software automatisch ‚korrigiert' wird. Diese Korrektur lässt sich als ein Vorschlag der Software für den Benutzer auffassen.

Die wohl einfachste Unterstützung in dieser Art ist die Konstruktion entlang eines Gitter-Rasters: Für jeden Punkt, der eingegeben wird, wird der nächstliegende Punkt des Rasters vorgeschlagen.

Andere gebräuchliche Methoden, möglichst sinnvolle Vorschläge zu generieren, sind z.B.:

- Winkel einer Linie zur Horizontalen oder Vertikalen
  Wenn der Winkel unter z.B. 5° liegt, wird entsprechend eine exakte horizontale bzw. vertikale Linie vorgeschlagen.
- Abstand des aktuellen Punktes zu einem anderen Punkt oder Zeichnungsobjekt
  Wenn der Abstand unter z.B. 10 Punkten liegt, wird für den aktuellen Punkt der bestehende Punkt oder das Zeichnungsobjekt vorgeschlagen.

Etwas komplexere Bedingungen wären

- Horizontaler bzw. vertikaler Abstand zu allen anderen Punkten in der Zeichnung
  Wenn der horizontale bzw. vertikale Abstand zu *irgendeinem* Punkt in der Zeichnung unter z.B. 5 Punkten liegt, so wird eine exakte vertikale bzw. horizontale Ausrichtung mit dem Punkt vorgeschlagen, dessen entsprechender Abstand am kleinsten ist
- Winkel im Bereich von jeweils 360° geteilt durch z.B. 1, 2, 4, 8, die bei Annäherung vorgeschlagen werden.

Bei dieser beispielhaften Auflistung wird bewusst nicht definiert, was ein ‚Punkt' in einer Zeichnung ist, oder was der ‚Abstand' zu einem Objekt konkret bedeutet. Tatsächlich hängt nämlich allein von dieser Definition bereits ab, wie effektiv die Unterstützung ausfällt.
Ein einfacher Ansatz ist, als ‚Punkt' nur die Punkte heranzuziehen, die zuvor explizit bei der Konstruktion eingegeben wurden, wie z.B. der Start- oder Endpunkt einer Linie etc. Bricht man diesen Ansatz aber auf und erlaubt verschiedene andere Punkte, die aus der Konstruktion berechnet werden können, vervielfacht sich damit der Nutzen. Für einen Kreis(-Bogen) muss nämlich u.U. bei der Konstruktion gar kein Mittelpunkt explizit eingegeben werden, trotzdem ist es ggf. sehr nützlich, sich auf den Mittelpunkt beziehen zu können.
Unabhängig davon, wie die Skizzierhilfe letztlich arbeitet, ist das, was sie bietet, stets als Vorschlag für den Benutzer zu verstehen, den er akzeptieren kann oder eben auch nicht. Wie bereits erwähnt, soll die Hilfe häufige Konstruktionsschritte erleichtern, sie soll

jedoch nicht unübliche Konstruktionsschritte unmöglich machen. Bei der strikten Konstruktion entlang eines (quadratischen) Gitter-Rasters lassen sich z.B. keine gleichseitigen Dreiecke erstellen. Der Benutzer muss also jederzeit die Möglichkeit haben, zumindest zeitweise auf die automatische Hilfe zu verzichten.

Im Vorliegenden System wird eine einfache Skizzierhilfe umgesetzt, die letztlich auf vorberechneten Punkten basiert, und zwar zum einen auf Punkten, die für das jeweilige Zeichnungsobjekt relevant sind und zum anderen auf den Schnittpunkten der Zeichnungsobjekte untereinander.
Die eigentliche Hilfe ist dabei sehr einfach: Wenn ein Punkt eingegeben werden soll, der nahe an einem Punkt aus der Menge der vorberechneten Punkte liegt, wird der vorberechnete Punkt stattdessen verwendet.
Diese Menge von vorberechneten Punkten wird immer dann aktualisiert, wenn sich ein Objekt in der Zeichnung ändert (z.B. verschoben wird) oder wenn ein neues Objekt hinzugefügt wird.
Die Punkte, die für jedes Zeichnungsobjekt als ‚relevant' eingestuft werden, werden über die Methode

```
function TCadItem.GetPoints( PL : TCadPointList ) : integer;
virtual;
```

ermittelt. Diese Methode wird überschrieben, um jeweils den Besonderheiten verschiedener Objekte zu entsprechen. Für eine TCadLine werden hier z.B. Start- und Endpunkt zurückgegeben, für einen TCadCircle nur der Mittelpunkt, etc.

Mittels der Intersect-Methode (s.o.) werden zusätzlich noch alle Schnittpunkte jedes Zeichnungsobjektes mit jedem anderen ermittelt und auch für die Skizzierhilfe zur Verfügung gestellt, so dass im Endeffekt der Benutzer dabei unterstützt wird, Punkte einzugeben, die solchen entsprechen, die entweder für ein bestehendes Zeichnungsobjekt relevant sind, oder die Schnittpunkte zwischen bestehenden Objekten bilden.

Teil 2: Parametrische Modellierung

Frühe CAD-Systeme halfen dem Benutzer dabei, geometrische Objekte zu erstellen. Der Benutzer konnte dazu Daten wie Punktkoordinaten oder Winkel eingeben und nach diesen Vorgaben konnten dann die geometrischen Objekte erzeugt und gezeichnet werden. Diese Art der Modellierung beschränkt sich aber eben auf die geometrischen Objekte. Der Benutzer gibt nichts weiter ein als die einfachen Objekt-Daten und auch nur diese werden im System gespeichert und verwaltet.

Ein solches System geht nur mit geometrischen Objekten um, die aber einige wünschenswerte Eigenschaften nicht besitzen, die bei realen Objekten ganz selbstverständlich vorhanden sind. Beispielsweise können zwei reale (solide) Quader nur so weit aufeinander zu bewegt werden, bis sie sich berühren. Sie können nicht einfach teilweise ‚ineinander stecken'. Mit einem einfachen CAD-System sind solche Operationen aber leicht durchführbar (und unter Umständen auch erwünscht).

Darüber hinaus ist ein System, das konkrete Werte für jeden möglichen Parameter einer geometrischen Konstruktion vom Benutzer fordert, relativ schlecht geeignet, bereits erstellte Objekte in veränderter Form aber bei gleichen Grundeigenschaften direkt an anderer Stelle wieder zu verwenden. Möchte der Benutzer z.B. einen Würfel, dessen Oberfläche aus Quadraten zusammengestellt ist, verschieben, so muss er bei einem einfachen CAD-System jedes Quadrat an die korrekte Position verschieben.

An dieser Stelle setzt parametrische Modellierung (auch: Modellierung mit Varianten) mit den verschiedenen Algorithmen an: Ein System, das parametrische Modellierung (im Sinne dieses Textes) unterstützt, erlaubt es dem Benutzer, zusätzlich zu den geometrischen Objekten Beziehungen zwischen ihnen festzulegen. Dies geschieht in der Form von Regeln oder Einschränkungen, den Constraints. Sind diese Constraints vorhanden, so sorgt das CAD-System dafür, dass keine Operationen durchgeführt werden können, die eines oder mehrere der Constraints verletzen.

Es lassen sich verschiedene Typen von Constraints nennen, beispielsweise der Abstand zweier Punkte zueinander. Mit einem solchen Constraint wird zwischen zwei Punkten eine Abstands-Beziehung festgelegt, die jederzeit gelten muss, z.B.

Abstand(P1, P2) := 100.

Das CAD-System hat nun dafür zu sorgen, dass diese Abstands-Beziehung jederzeit gilt, unabhängig davon, was für Operationen mit den Objekten durchgeführt werden. Als Constraint-Typen, also Arten von Beziehungen, kommen dabei unter anderem folgende in Frage:

- Abstand zweier Objekte
  In erster Linie ist hier der Abstand zweier Punkte von Bedeutung

- Tangential-Beziehung zwischen zwei Objekten
  Ein Spezialfall hiervon ist das Tangieren eines Punktes, was effektiv bedeutet, dass das Objekt durch den Punkt gehen muss.

- Winkel zwischen drei Punkten bzw. zwei Geraden oder Strecken
  Hierzu kann man u.U. auch Bedingungen wie „horizontal“ und „vertikal“ zählen

- Gleichheit von X- oder/und Y-Ordinate zweier Punkte

Die genannten Beispiele für Constraint-Typen beziehen sich jeweils nur auf zwei bzw. drei Objekte. Dies ist jedoch keine Einschränkung, sondern lediglich eine Vereinfachung der Beschreibung: Durch die Verbindung eines Objektes mit mehreren Constraints lassen sich transitiv beliebig lange Ketten von Abhängigkeiten herstellen.
Beispiel: Drei Punkte P1, P2 und P3 sollen alle die gleiche X-Ordinate haben. Dann lässt sich das formulieren als zwei Constraints, z.B. P1.x = P2.x und P2.x = P3.x.
Zu den genannten kommen ggf. noch weitere, spezielle Typen hinzu, die durch die Zeichnungs-Objekte bestimmt werden. Da verschiedene Zeichnungs-Elemente sehr unterschiedliche Strukturen haben können, sollte dabei der Bezug zum Constraint festgehalten werden, z.B. im Sinne von „Mittelpunkt“, „Endpunkt“ usw.

Es bleibt also festzuhalten, dass in einem System, das parametrische Modellierung unterstützt, zwei Grundlegende Typen von Elementen verwaltet werden: Zum einen die Elemente der Zeichnung bzw. des eigentlichen Modells und darüber hinaus zusätzlich Objekte, die die geometrischen Beziehungen der Objekte untereinander abbilden.

5 Techniken

Die allgemeine Zielsetzung, Constraints so zu verarbeiten, dass das System damit wie gefordert ihre Einhaltung garantieren kann, lässt sich mithilfe verschiedener Techniken erreichen. Die im Folgenden vorgestellten Möglichkeiten unterscheiden sich vor allem in der Komplexität ihrer Umsetzung und in der Flexibilität, welche sie bei der Konstruktion und Wiederverwendung lassen.

Obwohl Constraints im Prinzip beliebig komplexe Bedingungen beinhalten können, werden hier im Wesentlichen solche Constraint-Typen betrachtet, die zum einen in der Praxis relevant sind und zum anderen von einem CAD-System relativ leicht zu handhaben sind.

5.1 Freiheitsgrade

Im Zusammenhang mit den Constraints erhalten immer wieder ‚Freiheitsgrade' eine wichtige Bedeutung. Ein Freiheitsgrad kann verstanden werden als eine elementare, eindimensionale, direkt bestimmende Größe. Verschiedene geometrische Objekte haben u.U. verschieden viele Freiheitsgrade, durch deren konkrete Ausprägung oder Belegung sie erst eindeutig definiert werden. Als Anzahl der Freiheitsgrade eines geometrischen Objektes wird dabei die minimale Zahl von eindimensionalen Größen bezeichnet, die seine Lage im Raum eindeutig bestimmen.
Offensichtlich kann ein Punkt im zweidimensionalen Raum durch zwei eindimensionale Größen eindeutig bestimmt werden, z.B. durch die X- und die Y-Ordinate. Ebenso gut kann ein Punkt aber auch durch einen Abstand (z.B. zum Koordinaten-Ursprung) und einen Winkel festegelegt werden (Polarkoordinaten) oder durch eine X-Ordinate und einen Winkel etc.. Eine solche eindimensionale Größe kann prinzipiell beliebige Ausprägungen haben. So kann eine Gerade (z.B. $y = (0 \bullet x) + \mathbf{c}$) ebenso wie ein Kreis oder ein beliebiges anderes eindimensionales Objekt einen Freiheitsgrad belegen. Man spricht dabei auch von Ortskurven.

Ist für einen Punkt nur ein Freiheitsgrad belegt, also von außen vorgegeben, so gibt es für die Position des Punktes entlang einer eindimensionalen (es bleibt ja noch ein Freiheitsgrad) Kurve unbegrenzt viele Möglichkeiten, der Punkt kann also nicht eindeutig bestimmt werden.
Für komplexere Objekte gilt dies im Prinzip genauso, allerdings haben mehrdimensionale geometrische Objekte auch mehr Freiheitsgrade. Eine Gerade (1-dimensional) hat (im 2D!) z.B. 3 Freiheitsgrade (z.B. einen Punkt (2 Freiheitsgrade) und einen Winkel), ebenso wie ein Kreis (z.B. Mittelpunkt und Radius) oder ein Dreieck, dessen Form bereits festgelegt ist (z.B. ein Punkt und ein Lage-Winkel). Ein Geradensegment (Strecke) hat noch einen Freiheitsgrad mehr, da ja auch die Länge bestimmt werden muss.

Für jeden Freiheitsgrad eines Objektes muss also ein Wert vorgegeben oder berechenbar sein, damit das Objekt eindeutig im Raum positioniert werden kann. Ein Objekt oder eine Konstruktion heißt „überbestimmt“, wenn für das Objekt mehr Werte vorgegeben oder berechenbar sind, als Freiheitsgrade zu belegen waren. Analog heißt ein Objekt „unterbestimmt“, wenn noch Freiheitsgrade offen sind.

6 Ein einfacher graphen-basierter Ansatz

Im Folgenden soll zunächst eine mögliche Umsetzung eines geometrischen Constraint-Systems mittels Repräsentation in einem Graphen beschrieben werden. Der Grundgedanke ist dabei, die Abhängigkeit zweier Elemente im Constraint-System durch eine verbindende Kante festzuhalten.
Wie oben bereits erwähnt, gibt es zwei Klassen von Objekten im Constraint-System, nämlich Zeichnungs-Elemente und die eigentlichen Constraints. Die Zeichnungs-Elemente haben Einfluss auf die Constraints und Constraints haben Einfluss auf die Zeichnungs-Elemente.
Somit bietet sich also eine Repräsentation mittels eines bi-partiten Graphen an, in dem es zwei Klassen von Knoten gibt:

1. Zeichnungs-Objekte
2. Constraints

Ein solcher Graph könnte z.B. so aussehen wie im ersten Beispiel:

6.1 Ein einfaches Beispiel

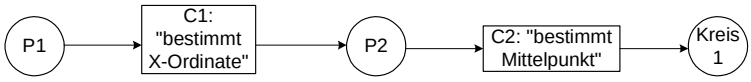

Abbildung 9

Die kreisförmigen Knoten repräsentieren hier die Zeichnungs-Elemente, die rechteckigen Knoten stehen für Constraints.

Die Bedeutung dieses Graphen ist also:

1. Punkt P1 bestimmt durch das Constraint C1 die X-Ordinate von Punkt P2 und
2. Punkt P2 bestimmt durch C2 den Mittelpunkt des Kreises Kreis1

(3. folgt daraus transitiv, dass P1 auch die X-Ordinate von Kreis1 bestimmt.)

Die entsprechende Zeichnung enthielte also einen Punkt (P1) und einen Kreis, dessen Mittelpunkt (gleichzeitig P2) genau vertikal über oder unter dem Punkt liegt.

Bei diesem einfachen Beispiel fallen bereits einige Dinge unmittelbar auf: Zum einen wurde ein *gerichteter* bi-partiter Graph benutzt, des Weiteren fehlen in dem Graphen noch Informationen, die nötig wären, um die Objekte auf der Ebene zu positionieren. Im Folgenden werden wir zunächst auch nur zyklenfreie Graphen, also DAGs, betrachten und erst später auf Graphen mit Zyklen eingehen.

Um herauszufinden, welche Informationen noch benötigt werden, um die Lage der Objekte eindeutig zu bestimmen, kann man eine Betrachtung der Freiheitsgrade der Objekte heranziehen. Dabei ist es wichtig zu bemerken, dass ein Constraint einen oder mehrere Freiheitsgrade belegt, „konsumiert“, wenn es ein Objekt einschränkt. Die X-Ordinate von P2 wird z.B. durch das Constraint C1 bereits festgelegt, so dass hier nur noch ein Freiheitsgrad zu belegen ist, der die Y-Ordinate (ggf. auch nur indirekt) festlegt. Bei P1 ist noch kein Freiheitsgrad konsumiert, es werden also noch zwei Dimensionen benötigt, und dem Kreis1 fehlt noch eine Angabe, aus der der Radius bestimmt werden kann. Alle diese

Werte können mit den bestehenden Constraints nicht ermittelt werden. In diesem Fall ist die einzige Informationsquelle, die das Problem lösen kann, der Benutzer des Systems, der die nötigen Werte vorgeben muss:

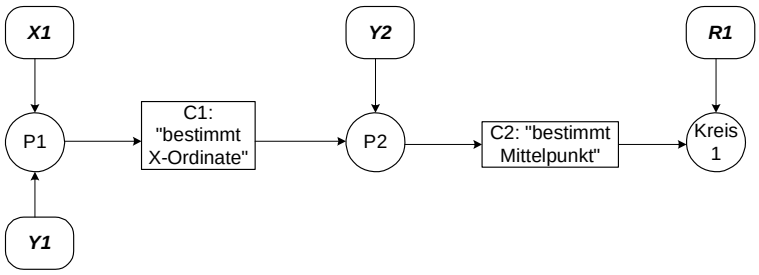

Abbildung 10

6.2 Elemente des Graphen

Die Elemente in den abgerundeten Rechtecken entsprechen hierbei skalaren Werten, die in diesem Fall vom Benutzer eingegeben werden müssen. Genau genommen handelt es sich bei dieser Repräsentation also um einen tri-partiten Graphen mit den Knotenmengen

K_G = der Menge der geometrischen Objekte in der Konstruktion

K_C = der Menge der Constraints

K_I = der Menge der von außen eingegebenen Werte (Input)

Für den Graphen in Abbildung 10 sind dies:

K_G = { P1, P2, Kreis1 }

K_C = { C1, C2 }

K_I = { X1, Y1, Y2, R1 }

Je nach Konstruktion ergeben sich verschiedene Typen von Graphen, die auch verschieden komplexe Algorithmen erfordern. Diese Typen sind: Baum, gerichteter azyklischer Graph (DAG) und gerichteter Graph.

6.3 Konstruktion des Graphen

Ein solcher Graph lässt sich aus den Knotenmengen konstruieren:

1. Jeder Knoten *k* aus K_I ist im Graph direkt über (eine oder mehrere) auslaufende Kanten verbunden mit allen Knoten aus $K_G \cup K_C$, für deren Elemente der Skalar in *k* direkt bestimmend ist.
2. Jeder Knoten *k* aus K_G ist im Graph direkt über (eine oder mehrere) auslaufende Kanten verbunden mit allen Knoten aus $K_C \cup K_I$, für die das Objekt in *k* direkt bestimmend ist.
3. Jeder Knoten *k* aus K_C ist im Graph direkt über (eine oder mehrere) auslaufende Kanten verbunden mit allen Knoten aus $K_I \cup K_G$, für die das Constraint in *k* direkt bestimmend ist.

Diese Bedingungen beschreiben, dass die Knoten *k*, für die in der topologischen Sortierung gilt $k > x$, von den Knoten *x* abhängig sind. In Spezialfällen, in denen jedes Element für maximal ein anderes Element bestimmend ist, kann der Graph auch als Baum dargestellt werden, wobei auslaufenden Kanten von den Kinds- zu den Elternknoten verlaufen. Dies ist aber natürlich bereits dann nicht mehr möglich, wenn ein Objekt für zwei andere Objekte bestimmend ist.

Der Graph aus Abbildung 10 wird also zu folgendem Baum:

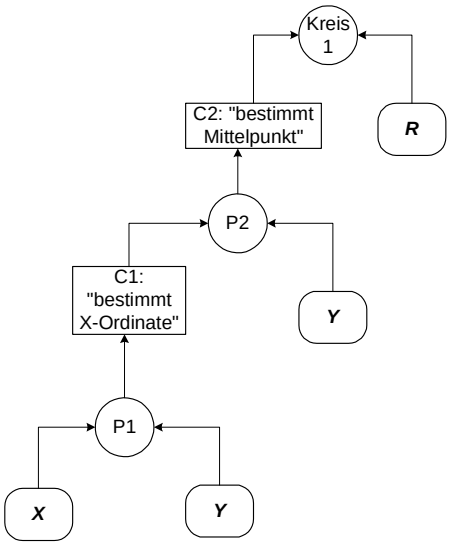

Abbildung 11: Constraint Graph in Baumdarstellung

Die skalaren Elemente, die durch den Graphen nicht festgelegt werden, sind die Parameter der Zeichnung. Sie können einfach – auch nachträglich – manipuliert werden und bewirken damit, dass sich die ganze Zeichnung entsprechend verändert. Bei allen möglichen Parameter-Kombinationen gibt es aber immer bestimmte Invariante in der Zeichnung, und dies sind genau die, die durch den Graphen selber repräsentiert werden. Durch den Graphen werden als Invariante i.A. keine festen Werte vorgegeben, sondern vielmehr die „Struktur“ einer Zeichnung.

Somit ergibt sich, dass eine solche Zeichnung mit einem Constraint-Graphen klar in zwei Teile zerfällt:

1. Die Parameter
2. Die „Struktur“

6.3.1 Der Graph im Detail

Der letztgenannte Graph ist in der Semantik nicht ganz eindeutig, weil z.B. nicht näher bestimmt wird, wie P1 die X-Ordinate von P2 bestimmt: Wird dafür die X- oder die Y-Ordinate von P1 herangezogen? Oder etwas ganz anderes?

Will man diese Eindeutigkeit schaffen, so kann man den Graphen wiederum erweitern. Diesmal kommen aber keine neuen Knotenklassen hinzu, es werden lediglich Knoten aufgespalten:

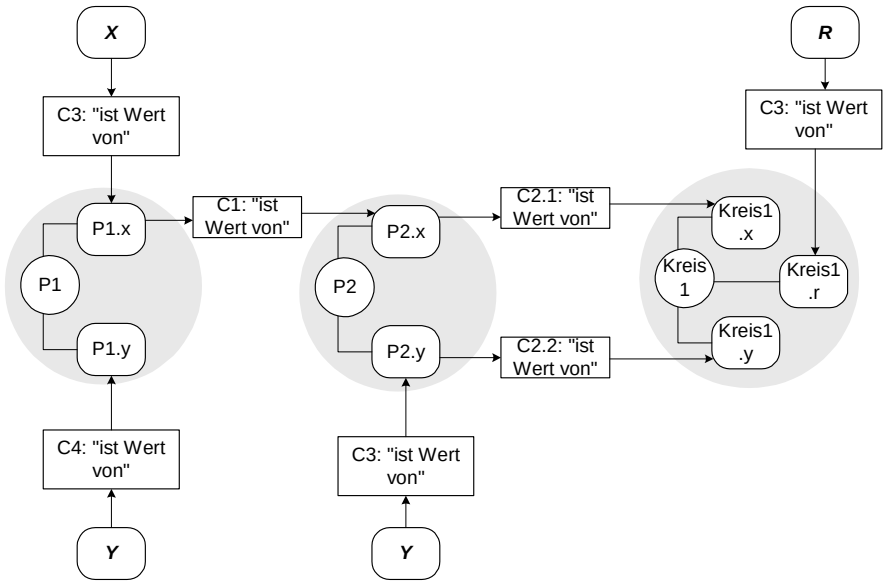

Abbildung 12: Der Graph aus Abbildung 11 ‚hineingezoomt'

Der Graph ist also so erweitert worden, dass jedes Constraint eindeutig einen skalaren Wert einem anderen zuordnet. Hiermit kann dann auch ein Constraint wie „X-Wert von A ist Y-Wert von B" abgebildet werden. Allerdings gibt es auch Fälle, in denen es überhaupt nicht sinnvoll oder sogar unmöglich ist, Skalare auf Skalare abzubilden. Beispielsweise im Falle eines Tangential-Constraints zwischen einem Kreis und einer Geraden. Aus keinem der vorhandenen skalaren Werte kann direkt z.B. der Kreis-Radius ermittelt werden. Daher ist es notwendig, die Constraints zwischen geometrischen Objekten zu definieren und nicht (nur) zwischen ihren atomaren Bestandteilen. Aus diesem Grund werden hier weiterhin Graphen verwendet, die so aufgebaut sind.

6.4 Ein komplexeres Beispiel

Zur Veranschaulichung soll hier noch ein etwas komplexeres Beispiel für Constraint-Graphen gegeben werden. Die zugehörige Zeichnung soll etwa so aussehen:

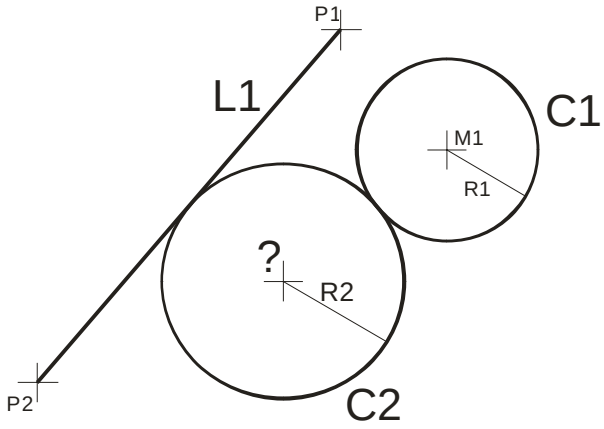

Abbildung 13: Beispiel-Zeichnung

Hier handelt es sich um eine Konstruktion aus zwei Kreisen und einer Linie, bei der für C2 kein Mittelpunkt explizit angegeben ist. Stattdessen sollen folgende Bedingungen gelten:

1. C2 tangiert die Linie L1
2. C2 tangiert den Kreis C1

Dass diese Konstruktion eindeutig bestimmt ist, lässt sich anhand der Freiheitsgrade verdeutlichen: C2 hat insgesamt drei Freiheitsgrade, von denen einer durch den Radius und zwei weitere durch zwei Tangential-Constraints konsumiert werden.

Der Constraint-Graph ist der folgende:

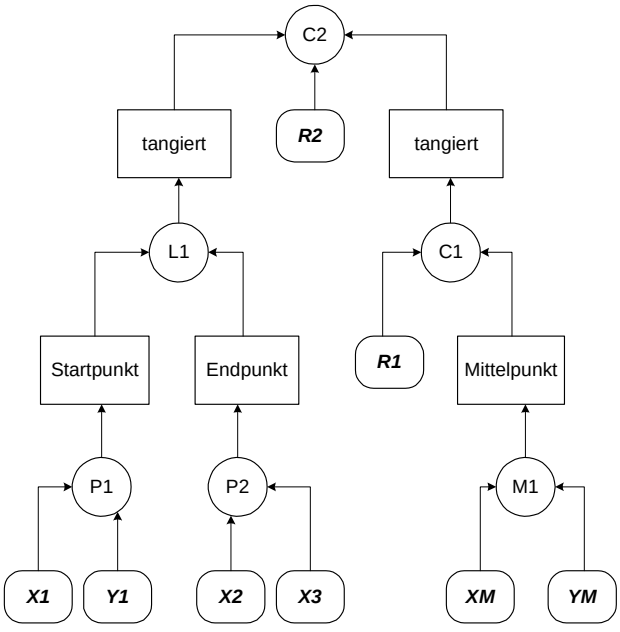

Abbildung 14: Constraint-Graph zu Abbildung 13

6.5 Algorithmus zur Berechnung des Graphen

Jedes Mal, wenn (bei gegebenem Graphen) einer der Parameter geändert wird, muss die konkrete Zeichnung, die ja vom Graphen und den Parametern abhängt, neu ermittelt werden. Da prinzipiell jeder Wert im Graphen von jedem Parameter indirekt abhängig sein kann, muss also jedes Mal die gesamte Zeichnung überprüft werden. Ähnliches gilt natürlich auch, wenn sich ein Constraint ändert, oder ein neues hinzugefügt wird.

Wie oben bereits erwähnt, wird hier ein gerichteter Constraint-Graph verwendet. Dadurch wird im Graphen bereits definiert, welches Objekt von welchem abhängig ist. In den einfachen Beispielen ist der Graph jeweils auch als Baum darstellbar. In diesem Constraint-Baum befinden sich alle eingegebenen Parameter in den Blättern. Somit kann auch nur in den Blättern einen Änderung stattfinden, indem dort ein Wert geändert wird. Um nun die konkrete Zeichnung zu ermitteln, müssen die Werte von den Blättern über die Constraints hin zur Wurzel propagiert werden. Man sagt deshalb auch, dass die Constraint-Propagation von den Blättern zur Wurzel hin stattfindet. Praktisch wird dies erreicht, indem von der Wurzel aus der Baum in einer Depth-First –Weise traversiert wird und jeweils die Elemente berechnet werden, für die bereits alle benötigten Knoten berechnet – also traversiert – worden sind. Daraus resultierend kann man sagen, dass die Berechnung – „demand driven“ – von der Wurzel zu den Blättern propagiert wird („computation propagation“). Dies entspricht im allgemeinen Fall eines DAG dem Propagieren der Berechnung entlang der topologischen Sortierung, also der Richtung der gerichteten Kanten.

Die eigentliche Berechnung eines jeden Knotens ist jeweils abhängig von den beteiligten geometrischen Objekten und Constraints. Es handelt sich hier um normale geometrische Berechnungen, wie im Beispiel u.a. bei der Berechnung des Mittelpunktes von C2 aus gegebenem Radius und zwei (zu dem Zeitpunkt bereits fertig berechneten) tangentialen Elementen.

6.6 Beschränkungen

Der einfache Ansatz, der hier erläutert worden ist, ermöglicht es, mit geringem Aufwand jedes Element der Zeichnung zu berechnen. Diese Einfachheit der Berechnung wird aber mit einigen Einschränkungen erkauft. Es wird hier vorausgesetzt, dass eine eindeutige Reihenfolge der Constraints und Objekte fest vorgegeben ist. Daraus folgt, dass zwar aus gegebenen Parametern alle Objekte berechnet werde können, aber umgekehrt nicht ausgehend von einem Objekt beliebige Parameter. Ohne diese Möglichkeit ist jedoch die gesamte Konstruktion streng sequenziell – sowohl in der Berechnung als auch in der Erstellung durch den Benutzer. Es ist hier also nicht möglich, ein beliebiges Objekt der Zeichnung zu manipulieren (z.B. zu verschieben) und dann den Rest der Zeichnung entsprechend den Constraints automatisch anpassen zu lassen.

Darüber hinaus gibt es Konstruktionen, die sich nicht ohne zyklische Abhängigkeiten darstellen lassen. Um diese abzubilden, muss man Zyklen im Constraint-Graphen zulassen, was dann die Komplexität der Berechnung deutlich erhöht.

Implizit wurde hier auch vorausgesetzt, dass jede Konstruktion eindeutig bestimmt werden kann. Dies ist genau dann der Fall, wenn jeder Freiheitsgrad jedes geometrischen Objektes durch ein Constraint oder einen Skalar konsumiert wird. Diese Voraussetzung erschwert das Erstellen einer umfangreichen Konstruktion, weshalb es im Sinne des Benutzers wäre, nur „relevante“ Constraints vorgeben zu müssen und alle anderen implizit mit „sinnvollem“ Verhalten zu erzeugen. „Sinnvoll“ bedeutet hier, für nicht eingegebene Constraints automatisch Standardconstraints anzunehmen, die vermutlich in vielen Fällen ein gutes Ergebnis liefern. Im einfachsten Fall könnten so z.B. alle Punkte als fix angenommen werden, die nicht anderweitig explizit über Constraints bestimmt werden, oder diese Punkte könnten z.B. in konstanter Lage zum nächsten explizit bestimmten Punkt angenommen werden.

Beispiel:

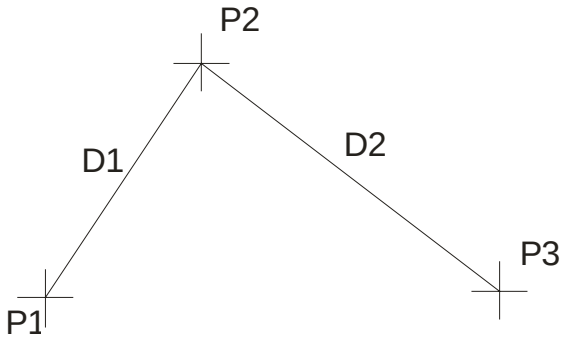

Abbildung 15

Hier seien drei Punkte gegeben, die über Abstands-Constraints (D1, D2) miteinander in Beziehung stehen. Sind keine weiteren Constraints vorgegeben, kann die Konstruktion in Abbildung 15 zunächst nicht einmal dargestellt werden. Hier wird also i.d.R. bereits angenommen, dass die einzelnen Punkte absolut so positioniert sind, wie sie ursprünglich eingegeben wurden.

Wenn jetzt aber Punkt P3 verschoben werden soll, müssen wiederum andere Annahmen gemacht werden. Die Möglichkeiten hier sind:

- P1 und P2 werden als fix angenommen, P3 kann also nur auf einer Kreisbahn um P2 bewegt werden.
- P1 wird als fix angenommen. Hierbei ist es noch recht einfach möglich, aus einer Verschiebung von P3 ggf. eine neue Position von P2 zu bestimmen (z.B. von den maximal zwei Möglichkeiten diejenige, die näher an der letzten Position liegt).
- Keiner der Punkte wird als fix angenommen. In diesem Fall kann die Konstruktion bei Verschiebung eines Punktes sowohl ihre Lage/Position als auch ihre Form ändern. Die Frage ist hier, ob die Lage/Position, die Form oder beides verändert wird, um die Constraints weiterhin zu erfüllen.

Komplexere (z.B. heuristische) Algorithmen können in einer solchen Situation dem Benutzer die Arbeit abnehmen, jedes Constraint explizit angeben zu müssen, indem sie für solche Fälle möglichst „sinnvolles“ Verhalten an den Tag legen.

7 Erweiterung des graphen-basierten Ansatzes

Nun sollen einige der zuvor beschriebenen Einschränkungen des einfachen DAG-Ansatzes aufgehoben werden, indem das Modell aus Constraints und Objekten, das der Graph abbildet und das damit durch den Graphen auch eingeschränkt wird, sowie der Graph selbst um weitere Eigenschaften erweitert wird.

7.1 Zyklische Graphen

Zunächst wieder ein einfaches Beispiel:

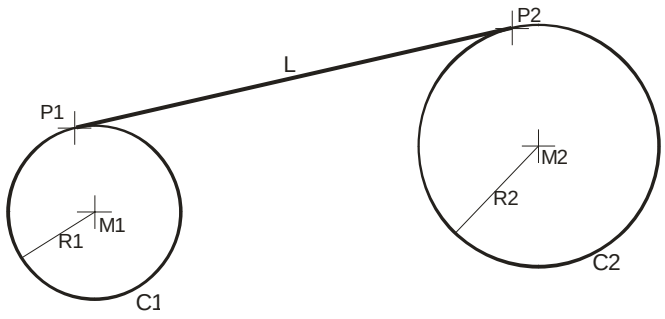

Abbildung 16

In dieser Zeichnung soll die Linie L von ihrem Tangentialpunkt mit C1 P1 zum Tangentialpunk mit C2 P2 verlaufen. Von der Reihenfolge, in der Constraints ausgewertet werden sollen, hängt nun der Constraint-Graph ab. Allerdings ist es hier nicht möglich, eine eindeutige Reihenfolge festzustellen: Die Lage von P1 hängt nämlich von C1 und P2 ab, P2 aber hängt von C2 und P1 ab. Es besteht also eine zyklische Abhängigkeit.
Beispielhaft soll hier angenommen werden, die Linie L soll lediglich durch ihre Tangentialbeziehungen und ihre Länge festgelegt sein. Dabei kann die Länge (vom Benutzer) verändert werden, und die Elemente der Zeichnung müssen sich daran anpassen. Hierfür gibt es wieder mehrere Möglichkeiten, hier soll angenommen werden, dass sich nur die X-Ordinate von M1 verändern kann, um die Constraints zu erfüllen.
Dann sieht der Constraint-Graph so aus:

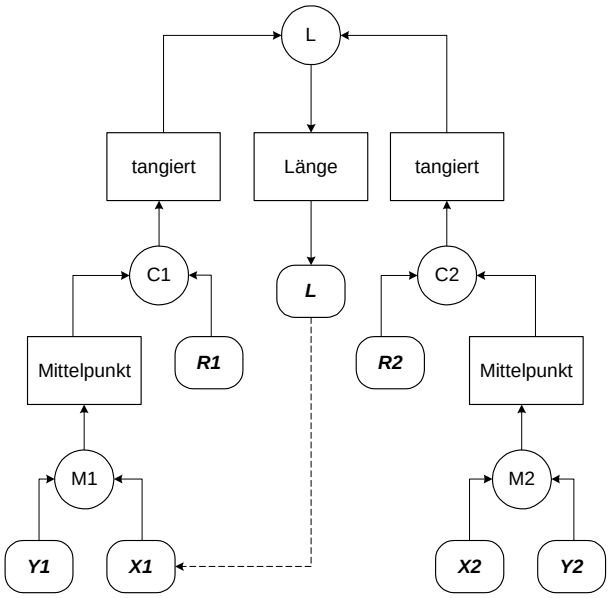

Abbildung 17

Dieser Graph ist also nicht mehr zyklenfrei und kann deshalb auch nicht sequenziell in Einzelschritten evaluiert werden. Die Bedingungen im Graphen können jedoch als Gleichungssystem formuliert werden, die Berechnung der Konstruktion entspricht dann dem Lösen dieses Gleichungssystems. Eine Möglichkeit, die Berechnung durchzuführen, ist, ein Intervall-Schachtelungs-Verfahren zu verwenden. Hierbei werden für die Variablen zunächst Werte aus einem bestimmten Intervall benutzt und dann ein Wert für die gesuchte(n) Variable(n) ermittelt und das Intervall für die nächste Iteration so verkleinert, dass der dann errechnete Wert die geforderte Bedingung (wahrscheinlich) „besser“ erfüllt, dass also die Abweichung vom gewünschten Zustand (z.B. Punkt liegt genau auf Kreis, Abweichung hier: Abstand Punkt-Kreis) kleiner wird.

Um möglichst flexibel bei der Konstruktion vorgehen zu können, ist es notwendig, verschiedene Elemente in beliebiger Reihenfolge verändern zu können. Daraus folgt, dass ein Ansatz mit einem Graphen, der von vornherein gerichtet ist, die Flexibilität einschränkt. Um dies zu umgehen, kann man darauf verzichten, den Graphen per se gerichtet zu halten und ihn erst dann – vorübergehend – in einen gerichteten umwandeln, wenn es explizit notwendig ist, nämlich dann, wenn der Graph evaluiert werden soll.

7.2 Ungerichtete Constraint-Graphen

Die Umwandlung eines gerichteten Constraint-Graphen in einen ungerichteten ist eine triviale Aufgabe. Die dynamische Rückführung in einen gerichteten Graphen ist nicht ganz so einfach, aber in der Regel möglich, da die notwendige Information implizit noch im Graphen vorhanden ist. Hierbei kommt den Freiheitsgraden große Bedeutung zu. Zunächst sollen aber wieder einfache Beispiele zur Veranschaulichung angeführt werden.

7.2.1 Beispiel

Eine sehr einfache Konstruktion soll lediglich aus zwei Punkten bestehen. Diese beiden Punkte sollen

- waagerecht ausgerichtet sein (gleiche Y-Ordinate)
- einen bestimmten Abstand *d* haben.

Der zugehörige ungerichtete Constraint-Graph ist folgender:

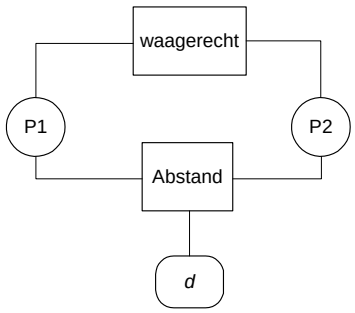

Abbildung 18: Ein ungerichteter Constraint-Graph

Durch den Graphen wird hier keine Reihenfolge der Constraints oder anderer Knoten vorgegeben. Im Unterschied zu den oben besprochenen Graphen wird hier und im Folgenden auch nicht vorausgesetzt, dass die skalaren Variablen von außen in die Konstruktion eingebracht werden. Die Skalare können auch durch andere Größen innerhalb der Konstruktion bestimmt sein. Es soll aber weiterhin die Möglichkeit bestehen, einzelne Werte oder Punkte als unveränderbar zu kennzeichnen, sie zu „fixieren“, um die Konstruktion zu bestimmen.

Auf der Ebene der Interaktion mit der Zeichnung durch den Benutzer gibt es nun verschiedene Möglichkeiten, Operationen durchzuführen. Eine Möglichkeit ist z.B., dass der Abstand *d* als fest angenommen wird. Verändert der Benutzer nun die Position von P1, so muss sich die Position von P2 entsprechend ändern, damit der Abstand (und die waagerechte Ausrichtung) zu P1 erhalten bleibt.
Ist der Abstand nicht fest, so kann bei dieser Operation mit P1 der Punkt P2 ggf. seine Position beibehalten und der Abstand d von P1 zu P2 wird verändert.
Sind zwei der drei Elemente festgelegt, so ist auch das dritte eindeutig in seiner Position bestimmbar. In dieser Konstruktion sind jedoch keine Größen als „fest“ gekennzeichnet. Dies ist auch nicht erwünscht, weil der Graph dann wieder äquivalent zu einem gerichteten wäre.
Eine der Herausforderungen bei einem System, das mit dieser Art von Graphen arbeitet, ist, bei jeder Operation aufs neue möglichst „sinnvoll“ zu entscheiden, welche Elemente der Konstruktion von welchen anderen wie beeinflusst werden und welche als „fest“ angenommen werden, wenn der Benutzer an anderen Änderungen vornimmt. Beispielsweise könnte man Elemente als unterschiedlich „fest“ annehmen, abhängig davon, wann der Benutzer sie zuletzt geändert hat.

7.2.2 Ein weiteres Beispiel

Drei Punkte sollen auf einem Kreis liegen:

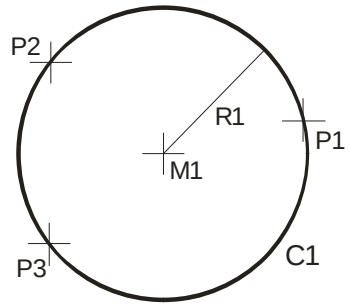

Abbildung 19

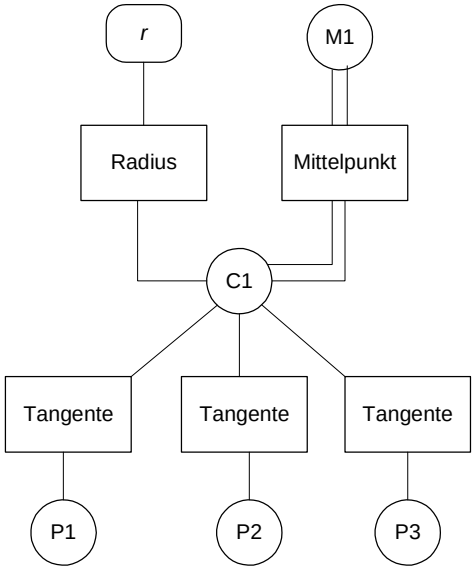

Abbildung 20: Constraint-Graph zu Abbildung 19[1]

In diesem Fall gibt es deutlich mehr mögliche Ausrichtungen für den Constraint-Graphen als im letzten Beispiel. Aus Gründen der Einfachheit wurde hier auch darauf verzichtet, die Punkte P1 bis P3 mit einem weiteren Constraint eindeutig festzulegen. So gibt es in diesem Graphen im Prinzip für jeden der drei Punkte eine Ortskurve, die den gesamten Kreis umfasst. Die zwei Kanten zwischen M1 und C1 verdeutlichen, dass der Mittelpunkt-Constraint genau wie der Kreismittelpunkt selber zwei Freiheitsgrade hat.

Mögliche Ausrichtungen des Graphen könnten u.a. sein:

- Mittelpunkt und Radius des Kreises bestimmen die drei anderen Punkte
- P1 bis P3 bestimmen den Mittelpunkt und Radius des Kreises
- P1, P2 und M1 bestimmen den Kreisradius und darüber P3
- Etc.

[1] Es sei hier angemerkt, dass die Constraints „Tangente“ an Kreisen (insbesondere für Punkte) auch als Abstands-Constraints zum Mittelpunkt des Kreises mit dem Kreisradius als Wert formuliert werden können.

7.3 Orientierung des Constraint-Graphen

Wie bereits erwähnt, muss der ungerichtete Constraint-Graph in einen gerichteten umgewandelt – orientiert – werden, damit die Reihenfolge der Constraint-Evaluation bestimmt werden kann. Hierbei gibt es mehrere Aspekte, die berücksichtigt werden müssen, u.a. besteht natürlich die Anforderung, dass möglichst jeder Constraint-Graph „sinnvoll“ evaluiert werden soll. Dies umfasst zusätzlich zu eindeutigen Graphen auch unterbestimmte und überbestimmte. Um festzustellen, ob ein ungerichteter Constraint-Graph über- oder unterbestimmt ist, müssen die Freiheitsgrade in der Konstruktion untersucht werden.
Daraus folgt also, dass nicht irgendeine beliebige Orientierung des Graphen gefunden werden soll, sondern möglichst diejenige, die die Konstruktion eindeutig bestimmt – falls eine solche überhaupt existiert.

7.4 Untersuchung der Freiheitsgrade

Jedes Element einer Konstruktion hat eine bestimmte, feste Anzahl an Freiheitsgraden, die belegt werden müssen, um das Objekt eindeutig zu bestimmen. Werden mehr Freiheitsgrade konsumiert, so ist es überbestimmt, und es ist unterbestimmt, wenn nicht alle nötigen Freiheitsgrade belegt werden. Daher muss bei der Orientierung des Graphen versucht werden, für jedes Element der Konstruktion (und auch für jede Teil-Konstruktion) gleichzeitig genau die nötigen Freiheitsgrade mit Constraints zu belegen. Ebenso ist für die Constraints bekannt, wie viele bestimmende Größen notwendig sind, damit das Constraint ausgewertet werden kann. Ein einzelnes Constraint kann jedoch selber beliebig viele andere Elemente bestimmen, ebenso wie ein Zeichnungs-Objekt für beliebig viele Constraints bestimmend sein kann. Für den orientierten Graphen bedeutet dies:

- jedes Element der Zeichnung hat (wenn möglich) genau so viele eingehende Kanten, wie es Freiheitsgrade hat und

- jedes Constraint hat (wenn möglich) genau so viele eingehende Kanten, wie es bestimmende Größen benötigt.

Insbesondere bedeutet dies, dass kein Element oder Constraint mehr eingehende Kanten haben kann, als zur eindeutigen Bestimmung nötig sind. Jedes Element/Constraint hat also maximal so viele einlaufende Kanten, wie es Freiheitsgrade hat. Alle weiteren Kanten müssen dann ausgehend orientiert sein.
Natürlich können nicht alle möglichen Kanten diese Bedingungen so erfüllen, dass die Konstruktion eindeutig bestimmt ist: Wenn z.B. für einen Punkt zwei X- aber keine Y-Ordinate vorgegeben werden, ist der Punkt natürlich nicht eindeutig bestimmt. Das gleiche gilt natürlich auch für alle anderen Arten von Ortskurven: Wenn zwei Ortskurven 0 oder unendlich viele gemeinsame Punkte haben, können sie keinen Punkt bestimmen[2]. Man kann daraus folgen Matrix erstellen, die eine Übersicht gibt über Kombinationen von durch Ortskurven bestimmten Größen und ob diese sinnvoll bzw. erlaubt sind unter der Bedingung, dass für einen Punkt immer X- und Y-Ordinate bekannt sein müssen:

	X	Y	X und Y
X	-	+	+
Y	+	-	+
X und Y	+	+	+

Diese Tabelle kann man auch so lesen, dass zu einer gegebenen Größe eine (mögliche) noch benötigte Größe ermittelt werden kann.

7.5 Algorithmus

Die Orientierung beginnt entweder mit einem Knoten im Graphen, der explizit als „fixiert“ markiert ist, also nicht von anderen Größen bestimmt wird, oder mit dem bzw. den Knoten, die durch die vom Benutzer ausgeführte Operation direkt geändert wurden[3].

[2] Ein Sonderfall ergibt sich, wenn zwei Ortskurven weder 0 noch unendlich viele, also *n* gemeinsame Punkte haben. Dieser Fall soll hier nicht weiter behandelt werden. Es wird in einem solchen Fall einfach genau einer der Punkte ausgewählt und wie der einzige mögliche behandelt.

[3] Gibt es keinen solchen Knoten, so muss einer ausgewählt werden, der im Folgenden wie ein fixierter behandelt wird.

Zur Veranschaulichung soll im Folgenden der Graph aus Abbildung 18 herangezogen werden mit der Erweiterung, dass der Abstand *d* als fixiert gegeben ist und P2 durch den Benutzer verschoben worden ist:

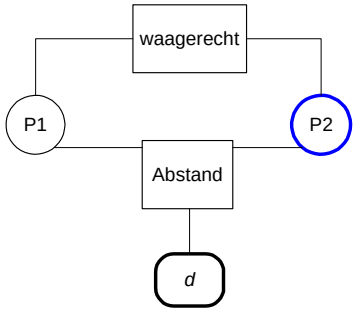

Abbildung 21: Ungerichteter Graph mit Fixierung

Zur Untersuchung der Freiheitsgrade seien auf den Knoten des Graphen folgende Mengen definiert:

Der ursprüngliche, ungerichtete Constraint-Graph kann als Menge von Knoten-Tupeln dargestellt werden. Hierzu seien noch einmal die verschiedenen Knoten-Mengen im Graphen erwähnt:

K_G = der Menge der geometrischen Objekte in der Konstruktion

K_C = der Menge der Constraints

Anstelle von K_G verwenden wir hier die Menge der Elemente der Zeichnung

$$K_E = K_G \cup K_S ,$$

wobei K_S die Menge der Knoten ist, die skalare Werte enthalten. K_E umfasst also alle Knoten des Graphen, bis auf die Constraint-Knoten.

Im Beispiel-Graphen sind dies also:

$K_E = \{P1, P2, d\}$ und

$K_C = \{[waagerecht], [Abstand]\}.$

Damit ist dann der Constraint-Graph

$$CG \subseteq K_E \times K_C,$$

wobei jedoch die Richtung der Kanten unberücksichtigt bleibt.

Um ermitteln zu können, welche Freiheitsgrade jeweils noch frei sind, wird außerdem noch eine Zuordnung von Constraints und Elementen zu Dimensionen definiert.

Die möglichen Freiheitsgrade bzw. Dimensionen sind:

$$Dim \in \overline{\{Xdim, Ydim\}},$$

wobei $\overline{\{Xdim, Ydim\}}$ die Potenzmenge über *Xdim* und *Ydim* ist, also
$\{\{\ \}, \{Xdim\}, \{Ydim\}, \{Xdim, Ydim\}\}$.

Für die Constraints wird dann die Funktion

$$Det : K_C \rightarrow Dim$$

definiert und für die Elemente und Constraints

$$Dim_E : K_E \rightarrow Dim \text{ und}$$
$$Dim_C : K_C \rightarrow Dim$$

Det legt für jedes Constraint fest, ob es eine X-, Y- oder beide Ordinaten bestimmen kann. Diese Zuordnung ist konstant und hängt lediglich von der Art des Constraints ab.
Dagegen geben Dim_E und Dim_C an, welche Freiheitsgrade für ein Element bzw. ein Constraint noch zu belegen sind, bevor es eindeutig bestimmt ist. Diese Zuordnung ist variabel und verändert sich während der Orientierung des Graphen. Ein Element *e* ist genau dann eindeutig festgelegt, wenn $Dim_E(e) = \{\ \}$. Ebenso ist ein Constraint *c* festgelegt, wenn $Dim_C(c) = \{\ \}$.

Wiederum bezogen auf den Beispiel-Graphen:

$Det([waagerecht]) = \{Ydim\}$ und

$Det([Abstand]) = \{Xdim, Ydim\}$, sowie

$Dim_C([waagerecht]) = \{Ydim\}$ und

$Dim_C([Abstand]) = \{Ydim, Ydim\}$

In der Relation

$$Fix_E : K_E \rightarrow Dim$$

wird abgebildet, für welches Element bereits welche Größe fest vorgegeben – fixiert – ist. Ist das Element *e* nicht fixiert, so ist $Fix_E(e) = \{\ \}$.

Im Beispiel:

$Fix_E(P2) = \{Xdim, Ydim\}$,

$Fix_E(P1) = \{\}$ und

$Fix_E(d) = \{Xdim, Ydim\}$

Mit diesen Definitionen lässt sich der Algorithmus zur Orientierung des ungerichteten Graphen wie folgt formulieren:

Schritt 0:
Da zunächst der gesamte Graph noch ungerichtet ist, also noch keine Information darüber enthält, welche Freiheitsgrade bereits belegt sind, wird unter Berücksichtigung der fixierten Größen

$$\forall e \in K_E : Dim_E(e) := \{Xdim, Ydim\} \setminus Fix_E(e)$$

gesetzt. Damit wird zunächst jedes Element als noch völlig unbestimmt markiert, ausgenommen diejenigen Elemente, die bereits durch Fixierung (teilweise) bestimmt sind.

Schritt 1:

Ausgehend vom ersten komplett fixierten Element *e*, für das nun gilt $Dim_E(e) = \{\ \}$, werden dann alle Kanten auslaufend orientiert[4]. Dieser Schritt wird für alle fixierten Elemente wiederholt.

Für das Beispiel ergibt sich dann:

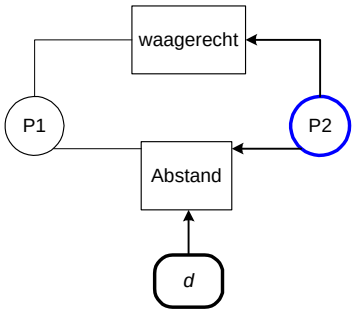

Abbildung 22

Die gerichtete Verbindung eines fixierten Elementes mit einem Constraint konsumiert bei diesem Constraint die entsprechenden Freiheitsgrade, und zwar genau diejenigen, die für das Element fixiert sind.

Im Falle des Beispiels ist P2 komplett fixiert ($Fix_E(P2) = \{Xdim, Ydim\}$). Das bedeutet also, dass sowohl das waagerecht- als auch das Abstand-Constraint auf jeden Fall mindestens einen der verbleibenden Freiheitsgrade verlieren (*d* ist auch bereits festgelegt!). Somit sind nun $Dim_C([waagerecht]) = \{\}$ und $Dim_C([Abstand]) = \{\}$.

Schritt 2:

Analog zum Vorgehen in Schritt 1 wird im zweiten Schritt das gleiche Verfahren auf die Constraints angewandt:

Für die Constraints *c*, die bereits komplett bestimmt sind ($Dim_C(c) = \{\}$), werden alle verbleibenden, ungerichteten Kanten im Graphen auslaufend orientiert. Das bedeutet für alle adjazenten Elemente, die dadurch eine einlaufende Kante erhalten, dass

$Dim_E(e) := Dim_E(e) \setminus Det(c)$ gesetzt wird.

---

[4] Da das Element bereits komplett fixiert ist, kann es keine einlaufende Kante geben, die noch einen weiteren Freiheitsgrad konsumieren würde.

Hierdurch werden also bei den Elementen ggf. weitere Freiheitsgrade konsumiert, und zwar genau diejenigen, die durch die Constraints nunmehr abgedeckt werden.

Ist nach diesem Schritt der Graph noch nicht vollständig orientiert, so wird erneut bei Schritt 1 angesetzt und solange Schritt 1 und 2 abwechselnd vollzogen, bis die Orientierung komplett ist. Hierbei ist zu bemerken, dass Schritt 1 Elemente beeinflusst, die im Schritt 2 als Eingabe-Menge dienen und umgekehrt, so dass bei korrektem Graphen auch in jedem Fall eine Lösung gefunden wird.

Im Beispiel ist die Verarbeitung allerdings bereits nach Schritt 2 zu Ende und das Ergebnis ist das folgende:

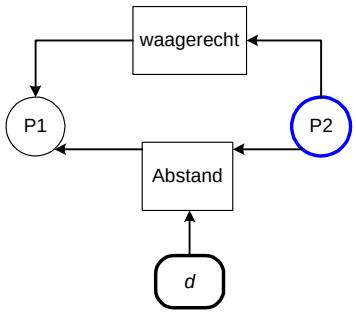

Abbildung 23

Hier ergibt sich also offensichtlich die Position von P1 durch das Propagieren fixierter Größen durch die Constraints.

8 Weitere Algorithmen

Im Bereich der graphenbasierten Constraint-Bearbeitung gibt es noch weitere, komplexere Algorithmen und Vorgehensweisen, die insgesamt zum Ziel haben, möglichst in jedem gegebenen Graphen die optimale Lösung bzw. Orientierung zu finden. Das gleiche Ziel kann man auch anders formulieren: Zum einen soll tatsächlich möglichst jeder Graph als Eingabe für einen Orientierungs-Algorithmus dienen können. Dies bedeutet im Endeffekt, dass der Benutzer bei der Erstellung der Konstruktion keinen Einschränkungen unterliegen soll, die durch den Orientierungs-Algorithmus bestimmt werden. Zum anderen soll

möglichst immer dann, wenn es eine Möglichkeit gibt, den Graphen konstruktiv zu orientieren, diese auch gefunden werden.

Diese beiden Vorgaben zusammen zu erfüllen ist nicht ganz einfach, und es bedarf u.U. einiger Optimierungen, um mit praktikablem Laufzeitverhalten zum Ziel zu kommen. Beispielsweise lässt sich jede mögliche Lösung finden, indem ein Brute-Force-Ansatz umgesetzt wird, der alle möglichen Kombinationen von Orientierungen nacheinander untersucht. Dies ist aber natürlich mit hohem (exponentiellem) Aufwand verbunden, also kaum praktikabel, wenn nicht weitere Optimierungsmöglichkeiten ausgeschöpft werden. Im Folgenden sollen weitere Möglichkeiten beleuchtet werden, mittels Constraints – aber ohne Graphen – Konstruktionen zu bestimmen.

8.1 Weitere Algorithmen zur Parametrischen Modellierung

Neben den bereits erläuterten graphenbasierten Ansätzen zur parametrischen Programmierung gibt es noch verschiedene andere Ansätze, die alle das gleiche zu erreichen versuchen, aber auf grundlegend unterschiedliche Weise. Einige werden nun kurz beschrieben.

8.2 Ein regelbasiertes System

Eine vom graphenbasierten Ansatz gänzlich verschiedene Methode soll nun vorgestellt werden, die mithilfe von Deduktionsregeln die Lösung eines Constraint-Systems durchführt. Dieses Vorgehen ist dem innerhalb von Expertensystemen ähnlich und zwar insofern, als dass aus gegebenen Fakten (hier: Constraints) zusammen mit festen Regeln neue Fakten erzeugt werden.
Damit auf beliebige Constraints möglichst allgemeine Regeln angewendet werden können, müssen die Constraints jedoch zunächst auf einfache und grundlegende Constraints zurückgeführt werden, auf denen dann mit Regeln operiert wird. Diese Basis-Constraints können z.B. sein Winkel- und Abstands-Restriktionen, die jeweils nur auf Punkten definiert werden.
Diese Restriktionen kann man wie folgt definieren:

- Eine Abstands-Bedingung soll auf zwei Punkten definiert sein und den Abstand der Punkte zueinander angeben:
  $D(P_1, P_2) = d$, mit d konstant

- Eine Winkel-Bedingung soll auf vier Punkten (zwei Punkt-Paare) definiert sein und den eingeschlossenen Winkel zwischen den zwei Geraden angeben, die durch die vier Punkte festgelegt sind:
  $A((P_1, P_2), (P_3, P_4)) = a$, mit a konstant; zwei der Punkte können auch identisch sein.

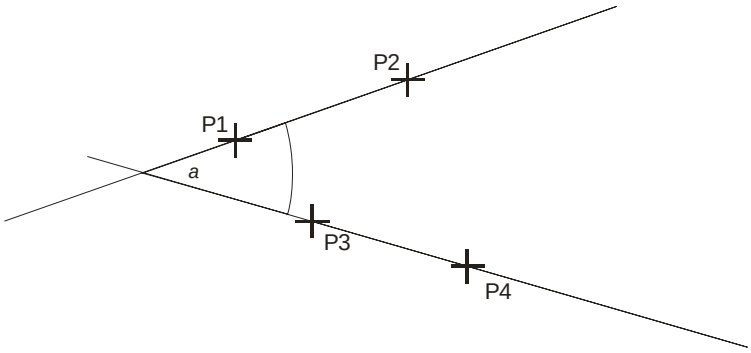

Abbildung 24

Dies sind also die Basis-Constraints, in die andere Constraints zunächst zerlegt werden müssen, damit sie mit diesem System bearbeitbar werden. In der Regel ist das ohne Schwierigkeiten möglich: Ein Tangential-Constraint zweier Kreise lässt sich z.B. direkt als Abstands-Bedingung ihrer Mittelpunkte umschreiben; eine Tangente an einen Kreis lässt sich über eine Abstands-Bedingung zum Mittelpunkt und einen rechten Winkel definieren, etc.

Die beiden Basis-Constraints werden im Folgenden mit Mengen von Punkten identifiziert, die die eine oder die andere Bedingung erfüllen, und zwar so:

- Ein *CA-Set* ist eine Menge von Punktepaaren, deren Winkel zueinander festgelegt ist
  und
- Ein *CD-Set* ist eine Menge von Punkten, deren Lage zueinander festgelegt ist.

Auf einer Menge von initialen CA- und CD-Sets werden dann deduktiv Regeln angewandt, die beschreiben, wie sich daraus weitere, größere CA- und/oder CD-Sets ergeben. Das Ziel hierbei ist klar: Wenn sich ausgehend von fixierten Punkten über eine Kette von Winkel- und Abstands-Restriktionen alle Punkte der Konstruktion definieren lassen, so ist die ganze Konstruktion eindeutig bestimmt und berechenbar. In diesem Fall bedeutet das zugleich, dass dann alle Punkte in einem CD-Set liegen.

8.2.1 Die Regeln

Die Regeln, nach denen die Verarbeitung stattfindet, sind einfache Deduktionsregeln, d.h., jede Regel hat die Form *Prämisse* $\Rightarrow$ *Konklusion* - natürlichsprachlich also: Wenn die

Prämisse gilt, dann folgt daraus, dass auch die Konklusion gilt. Im Fall der CA- und CD-Sets kann man die Regeln aber auch als Handlungsanweisungen verstehen: Wenn die Prämisse gilt, dann gilt auch die Konklusion, also sorge man dafür, dass die Konklusion anstelle der Prämisse verwendet werde, was in dem Fall ein neues CA- und/oder CD-Set ist.

Die grundlegenden Regeln sind die folgenden:

Erzeugung der CA-/CD-Sets aus Constraints:

- Wenn ein Abstand-Constraint zwischen zwei Punkten besteht, dann gibt es ein CD-Set mit diesen Punkten.
- Wenn ein Winkel-Constraint zwischen zwei Punkt-Paaren besteht, dann gibt es ein CA-Set mit diesen Punkt-Paaren.
- Wenn zwei Punkte fixiert sind, dann gibt es ein CD-Set mit diesen Punkten und ein CA-Set mit diesen Punkten und zwei Punkten auf der X-Achse.

CA-Sets lassen sich zusammenfassen:

- Wenn ein Punktepaar zu mehreren verschiedenen CA-Sets gehört, dann werden diese CA-Sets zusammengefasst zu einem größeren CA-Set.

Hier ist zu bemerken, dass ein CA-Set ja eine Menge von Punkt-Paaren (und Winkeln) ist (s.o.). Somit kann man, wenn ein Punktepaar in verschiedenen CA-Sets vorkommt, diese Mengen vereinigen und die fehlenden Winkel per Addition einfach berechnen.

Diese grundlegenden Regeln können zunächst erweitert werden durch einfache Regeln, die Dreiecksstrukturen (genauer: Strukturen aus drei Punkten) behandeln. Sie ergeben sich aus den bekannten Bedingungen für Dreiecke, dass aus verschiedenen Kombinationen von Winkeln und Abständen ein Dreieck bestimmt werden kann.

- Wenn drei CD-Sets jeweils paarweise einen gemeinsamen Punkt haben, werden sie vereinigt.

  Dies entspricht der Konstruktion eines Dreiecks aus den drei Seitenlängen.

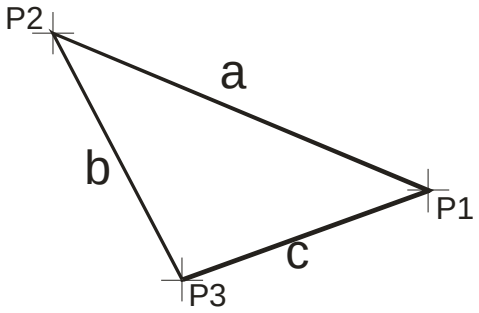

Abbildung 25: Dreieck konstruiert aus drei CD-Sets (a,b,c)

- Wenn zwei CD-Sets einen gemeinsamen Punkt haben und der gemeinsame Punkt und aus beiden CD-Sets jeweils ein Punkt in einem CA-Set liegt, so werden die beiden CD-Sets vereinigt.

  Ein Dreieck wird also durch zwei Längen und den eingeschlossenen Winkel definiert.

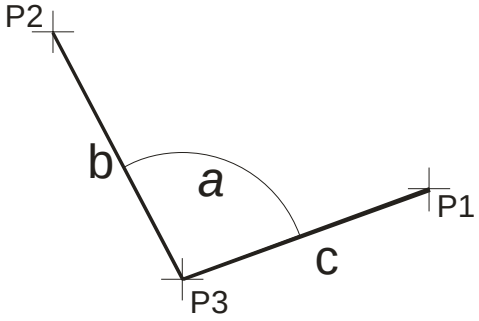

Abbildung 26: Dreieck aus zwei CD- (b,c) und einem CA-Set (a)

- Wenn zwei CA-Sets zwei Punkte gemeinsam haben, die in einem CD-Set liegen, dann bilden die drei Sets ein CD-Set.

  Aus einer Strecke und zwei Winkeln wird damit auch ein Dreieck konstruiert.

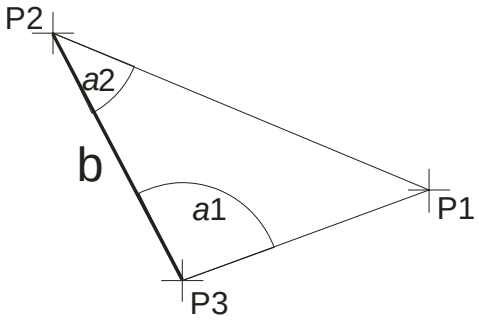

Abbildung 27: Zwei CA-Sets (a1,a2) und ein CD-Set (b) bilden ein Dreieck

- Wenn zwei CD-Sets einen Punkt gemeinsam haben und in beiden CD-Sets jeweils ein Punktepaar aus einem CA-Set vorkommt, dann werden die Sets zu einem CD-Set vereinigt.

  Hier wird das Dreieck durch zwei Strecken und einen nicht-eingeschlossenen Winkel festgelegt.

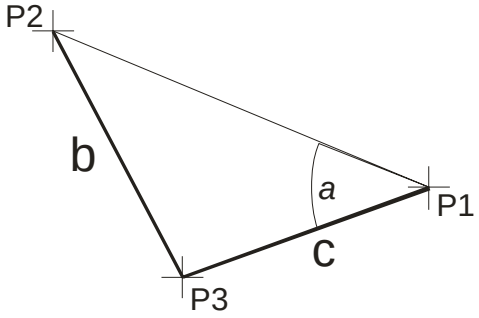

Abbildung 28: Ein CA-Set (a) und zwei CD-Sets (b,c) reichen auch für ein Dreieck

Ebenso wie für Strukturen mit drei Punkten lassen sich auch Regeln für solche mit vier Punkten herleiten und darüber hinaus solche für allgemeine Strukturen aus n Punkten mit n-1 Winkel- und n-2 Abstands-Constraints.

Diese Art der Constraint-Behandlung unterscheidet sich also offenbar grundlegend vom graphenbasierten Ansatz – es gibt schlicht keinen Graphen – und vermeidet damit auch Probleme wie das der zyklenarmen Graphen-Orientierung.
Der Algorithmus, der bei diesem Verfahren verwendet wird, prüft lediglich iterativ, welche der bekannten, invarianten Regeln er in jeder Iteration zur Anwendung bringen kann, sucht

also gleichsam in der Konstruktion nach Prämissen, wobei Reihenfolge und ähnliche Faktoren keine Rolle spielen.

8.3 Ausblick

Neben den hier kurz vorgestellten Algorithmen gibt es natürlich noch einige mehr. Allen gemein ist letztlich nur das Ziel, einmal erstellte Konstruktionen flexibel wieder verwenden zu können. Diesem Ziel werden verschiedene Algorithmen verschieden gut gerecht. Auch die Anforderungen an den Benutzer sind bei verschiedenen Lösungen unterschiedlich. So gibt es Systeme, die es erfordern, Konstruktionen in einer Art Programmiersprache (Makro-Sprache) zu formulieren, also quasi ein kleines Programm zu schreiben, dass dann immer wieder mit unterschiedlicher Eingabe aufgerufen werden kann. Eine andere Möglichkeit bietet sich in der assoziativen Bemaßung. Hier werden über einfache mathematische Gleichungen verschiedene Maße in Bezug zueinander gebracht (wie Höhe = 2* Breite), so dass bestimme Maße eingegeben und andere daraus direkt berechnet werden können. Wenn ein solches System nicht das Umformen und Lösen von Gleichungen mit Unbekannten beherrscht, muss also bereits bei der Eingabe festgelegt werden, welche Maße von welchen anderen abhängen.

Dies führt dann auch zu einem sehr komplexen Ansatz, bei dem das gesamte System von Bedingungen durch eine Anzahl von Gleichungen ausgedrückt wird, die alle zusammen gelten sollen, also ein Gleichungssystem. Das CAD-System muss dann solche u.U. umfangreichen Gleichungssysteme lösen können und dabei noch möglichst sinnvolles Verhalten zeigen für den Fall, dass das Gleichungssystem unterbestimmt ist.

9 Zusammenfassung

Wie bereits erwähnt, nehmen CAD-Systeme heute einen festen Platz in einer Reihe von Systemen ein, die an der Herstellung verschiedenster Teile, Geräte und Bauelementen beteiligt sind. Erst sie ermöglichen den direkten Datenaustausch zwischen Entwicklung einer Konstruktion und weiteren Fertigungsschritten und tragen so schon zu einer Effizienzsteigerung bei.

Damit ein CAD-System die Möglichkeiten der digitalen Informationsverarbeitung optimal nutzen kann, muss es jedoch über einige Komplexität verfügen, die nicht unbedingt vom Benutzer als solche zu erkennen ist. Zum einen muss natürlich bereits die Benutzerschnittstelle so gestaltet sein, dass sie leicht zu beherrschen ist und zugleich möglichst viel der Konstruktionsarbeit bereits automatisch erledigen kann (durch Skizziertechnik etc.) , und zum anderen muss die Wiederverwendbarkeit von Konstruktionselementen möglichst flexibel in Form von beliebigen Varianten möglich sein. Hierbei sollen dann alle Parameter eines Konstruktionselementes veränderbar sein, und dennoch bereits gemachte Konstruktionsschritte – also Eingaben – nicht erneut notwendig werden. Bestenfalls gibt ein CAD-System dem Benutzer auch die Möglichkeit, Varianten mit dem ‚Original' zu verknüpfen, so dass bei Änderungen an einer Stelle diese sich auch auf die Varianten übertragen.

Allein das Feld der Möglichkeiten zur Variantenprogrammierung ist recht umfangreich. So gibt es verschiedene graphenbasierte, regelbasierte und gleichungsbasierte Verfahren sowie u.a. Programmierung in Makro-Sprachen und weiteren. Wurde hier auch nur CAD im zweidimensionalen Raum betrachtet, lässt sich das Gesagte aber ebenso auf Konstruktionen in drei Dimensionen übertragen, die im Alltag – vor allem bei der Fertigung – eine größere Rolle spielen. Im 3D-Raum kommt aber natürlich bei allen Problemstellungen noch eine Dimension hinzu, bei der Eingabe ebenso wie bei der Lösung von Schnitt-Problemen und anderen Gleichungen in der Konstruktion. Alleine die Eingabe einer Konstruktion wird um ein vielfaches komplexer (und damit auch die Anforderung an das CAD-System, dies zu vereinfachen), wenn in drei Dimensionen gearbeitet wird, ganz abgesehen von der Schwierigkeit, die auftritt, wenn schon die beteiligte Hardware (Benutzerein-/-ausgabe) nur mit zwei Dimensionen arbeitet, wie bei normalen Maus/Bildschirm-Arbeitsplätzen üblich.

Insgesamt sind CAD-Systeme heute aus der Planung und Fertigung also nicht mehr wegzudenken. Um die möglichen Vorteile der Digitaltechnik auszuschöpfen, müssen sie zu verschiedenen, komplexen Problemstellungen noch komplexere Lösungen anbieten.

10 Literaturverzeichnis

- Berling R., Eine constraint-basierte Modellierung für Geometrische Objekte, Dissertation.
- Du C., Rosendahl M., Berling R., Variation of Geometry and Parametric Design, Proc. 3rd. international conference on CAD and computer graphics, Beijing, Aug. 23-26, 1993, pp 400-405, international academic publishers
- Rosendahl M., Vorlesung „CAD-Systeme“ WS2004/05, Universität Koblenz
- Rosendahl M., Objektorientierte Implementierung einer Constraint basierten geometrischen Modellierung, http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-12-2004.pdf, 2004
- Verroust A., Schonek F., Roller D., „Rule-oriented method for parameterized Computer-aided design“, Computer Aided Design, 24 (10): 531-540, October 1992.

11 Anhang

Anlage: 1 CD (s. Einband)

11.1 Inhalt der CD

Folgende Dateien befinden sich auf der CD:

`\CAD-Systeme.doc`
`\CAD-Systeme.pdf`

Die vorliegende Arbeit als Microsoft Word bzw. Adobe PDF Dokument

Die Verzeichnisse haben folgenden Inhalt:

`\Implementation`

Die Implementierung des entworfenen CAD-Programms, Delphi 7 Projektdatei (`BasicCAD.dpr`)

`\Sources`

Die Quelldateien der Units, die die Hauptfunktionalität und die Formulare des Programms enthalten

`\lib`

Externe Hilfs-Units

`\bin`

Ausgabeverzeichnis für kompilierte Units, enthält die .DCU-Dateien.

`\release`

Die fertig kompilierte Windows-Anwendung (`BasicCAD.exe`)