- Gegeben seien
  - zu formatierender Text,
  - die Metriken und weiteren Tabellen der benötigten Schriftschnitte,
  - diverse Rahmenparameter (z.B. der Satzspiegel) und
  - Trennungstabellen.
- Zu erzeugen ist eine druckbare Ausgabe.

Um diese Probleme zu lösen, werden Algorithmen und Datenstrukturen benötigt,

- um den Text in Wörter zu gliedern (zusammenhängende Zeichenfolge ohne Leerraum, die einheitlich gesetzt ist),
- geeignete Trennungsstellen in Wörtern zu finden,
- um unmittelbar aufeinanderfolgende Eingabezeichen (also Wörter) in eine Folge von Ausgabezeichen abzubilden (z.B. Berücksichtigung von Ligaturen) und diese relativ zueinander zu positionieren (Kerning),
- um Paragraphen in Zeilen zu zerlegen und
- um eine Folge von Paragraphen in Seiten aufzuteilen.

Voraussetzung für die Algorithmen sind geeignete Datenstrukturen und Schnittstellen, die für einen ausgewählten Schriftschnitt die benötigten Metriken und Tabellen bereitstellen.

- FreeType bietet eine Schnittstelle zu Schriftschnitten und für die Rasterisierung. Eine Unterstützung des Kerning gehört nicht dazu. Das liegt daran, dass die Bibliothek sich auf die Rasterisierung konzentriert, wobei Kerning keine Rolle mehr spielt.
- HarfBuzz ist eine Bibliothek, die deutlich über das einfache Kerning hinausgeht und entsprechend der modernen Schrifttechnologien in der Lage ist, Zeichen einer Zeichenfolge zu ersetzen, zu kombinieren und zu platzieren. Solche Werkzeuge werden *shaping engines* genannt.

Für Java gibt es an mehreren Stellen Bibliotheken, die mit Metriken arbeiten:

- Die Klasse *java.awt.FontMetrics* bietet eine Methode namens *charWidth* an, unterstützt jedoch kein Kerning. Zwar lässt sich seit Java 6 die Unterstützung von Kerning und Ligaturen einschalten, aber die Bibliothek gewährt keinen Zugang zu den entsprechenden Datenstrukturen. Ferner betrifft dies nur von Java unmittelbar unterstützte Schriften, die auch dargestellt werden können.
- Das FOP-Projekt (Apache Formatting Objects Processor), das sich zum Ziel setzt, XML-Repräsentierungen in PDF abzubilden, unterstützt das Laden diverser Schriftrepräsentationen, darunter auch Adobe Type 1.
- Das sfntly-Projekt von Google bietet Schnittstellen für OpenType und TrueType. Jedoch sehe ich da noch keine unmittelbare Unterstützung für das Kerning.

FontMetrics.java

```
package de.uniulm.mathematik.typo.afm;

public interface FontMetrics {
   public String getName();
   public String getName(int code);
   public int minCode();
   public int maxCode();
   public int length();
   public boolean defined(int code);
   public int[] getBoundingBox();
   public int[] getBoundingBox(int code);
   public int getVersalHeight();
   public int getXHeight();
   public int getDescender();
   public int getAscender();
   public int getWidth(int code);
   public int getKerning(int code1, int code2);
}
```

- Ein Schriftschnitt kommt mit einer Kodierung. Kodiert wird mit ganzen Zahlen aus dem Bereich *minCode()* bis *maxCode()*, wobei naiverweise angenommen wird, dass die Ordinalwerte druckbarer ASCII-Zeichen direkt verwendet werden können.
- Ein Unterstützung alternativer Kodierungen (manche Schriftschnitte haben Zeichen, die nur über Umkodierungen erreichbar sind) fehlt.
- Für jedes einzelne Zeichen kann die Weite (*getWidth()*), die Bounding-Box und der Name abgefragt werden. (Namen sind die Voraussetzung für Umkodierungen.)
- Die Unterstützung für Ligaturen und die Unterstützung für kombinierte Zeichen fehlen. (Hierfür wäre eine fortgeschrittene *shaping engine* notwendig.)

Beispielhaft wird diese Schnittstelle implementiert durch eine Unterstützung der AFM-Dateien (Adobe Font Metrics), die folgende Vorteile haben:

- Das Format der AFM-Dateien ist öffentlich spezifiziert, siehe die *Adobe Font Metrics File Format Specification*.
- Da die AFM-Dateien nicht-binär als ASCII-Text repräsentiert sind, die einer gewissen Syntax genügen, können sie sowohl bequem direkt gelesen werden als auch mit überschaubaren Aufwand syntaktisch analysiert und in passende Datenstrukturen überführt werden.

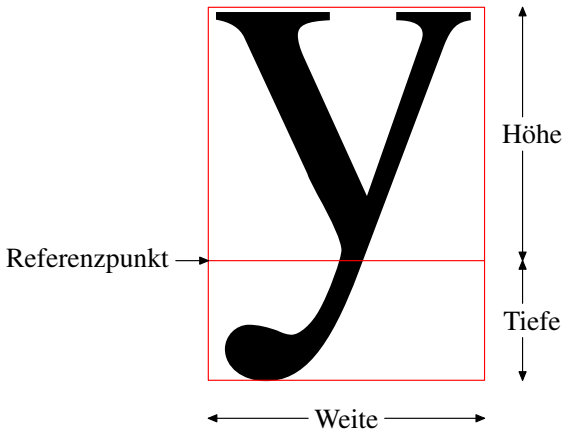
TestAV.java

```
public class TestAV {
   private final static String AFM_INPUT =
      "/usr/local/texlive/2015/texmf-dist/fonts/afm/adobe/times/ptmr8a.afm";
   public static void main(String[] args) {
      try {
         FontMetrics fm = new AdobeFontMetrics(AFM_INPUT);
         System.out.println("width of A = " + fm.getWidth('A'));
         System.out.println("width of V = " + fm.getWidth('V'));
         System.out.println("kerning = " + fm.getKerning('A', 'V'));
      } catch (Exception e) {
         e.printStackTrace();
      }
   } // main
} // class TestFontMetrics
```

- Die Klasse *AdobeFontMetrics* implementiert die Schnittstelle *FontMetrics* und bietet dabei zwei Konstruktoren an, die entweder die Angabe eines Dateinamens (wie hier) oder eines *BufferedReader* unterstützen.

```
clonard$ TestAV
width of A = 722
width of V = 722
kerning = -135
clonard$
```

- Es wurde hier die Konvention von Adobe übernommen, mit Einheiten zu arbeiten, die einem 1/1000 Punkt entsprechen.
- Das erlaubt es, weitgehend mit ganzzahligen Werten zu arbeiten.
- TEX macht das ähnlich, arbeitet aber mit Einheiten, die $\frac{1}{65536}$ pt entsprechen, wobei in TEX die Einheit pt dem nordamerikanischen Pica-Punkt entspricht.



- Die fundamentale Datenstruktur für die Verarbeitung von Texten sind Schachteln.
- Jedes Zeichen wird nur in Form einer Schachtel repräsentiert, die eine gewisse Weite, Höhe und Tiefe hat.
- Auf der Höhe des Referenzpunkts verläuft die Basislinie.
- Die Angaben werden der Metrik entnommen, die zu der Schriftform gehört.
- Eine Schachtel ist jedoch keine strenge Bounding-Box, d.h. es ist sehr wohl möglich, dass die Darstellung eines Zeichens die Grenzen der Schachtel überschreitet.
- Die Gestaltung des Zeichens ist uns hier ohnehin nicht bekannt.

- Im horizontalen Modus werden die Schachteln horizontal auf der Höhe der Basislinien aneinandergereiht.
- Aneinandergereihte Schachteln werden wiederum als Schachteln betrachtet.
- Längere horizontale Schachtelketten, die zu einem Paragraphen gehören, müssen an geeigneten Stellen auseinandergebrochen werden.
- Ein gebrochener Paragraph ist dann eine vertikale Aneinanderreihung horizontaler Schachtelketten.
- Eine Folge von vertikalen Schachtelketten muss an geeigneten Stellen in Seiten gebrochen werden.
- Ziel typografischer Algorithmen ist es, eine Texteingabe in ein geeignetes Schachtelsystem zu überführen.

Neben den regulären Schachteln werden für die typografischen Algorithmen noch spezielle Schachteln benötigt, die der Analyse der zur Verfügung stehenden Spielräume und Trennungsmöglichkeiten dienen:

- Statt Leerzeichen mit fester Länge werden **Dehnfugen** (*glue*) verwendet, die im Normalfall einen Leerraum passend zum gewählten Schriftschnitt einnehmen, bei Bedarf aber auch etwas gestaucht oder gestreckt werden können.
- **Sollbruchstellen** markieren Stellen, bei denen eine horizontale Aneinanderreihung von Schachteln gebrochen werden kann. Im Falle von Dehnfugen ist es offensichtlich. Es gibt aber auch potentielle Trennungen, bei denen ein Trennungszeichen eingefügt wird und die einen Strafwert besitzen, weswegen Knuth sie *penalties* nannte.

Item.java

```
public interface Item {
   public final int INFINITY = Integer.MAX_VALUE / 2;
   public boolean isPenalty(); public boolean isGlue();
   public boolean isBox();
   public abstract int getWidth();
   public int getStretchability(); public int getShrinkability();
   public void shrink(int units); public void stretch(int units);
   public int getHeight(); public int getDepth();
   public int getPenalty(); public boolean getPenaltyFlag();
   public StringBuffer genPostScript(PostScriptContext context);
}
```

- Dies ist die abstrakte Schnittstelle für Schachteln einschließlich spezieller Varianten, die im folgenden verwendet wird. (Es fehlt die Unterstützung vertikaler Dehnfugen.)

isPenalty()	handelt es sich um eine Sollbruchstelle?
isGlue()	handelt es sich um eine Dehnfuge?
isBox()	handelt es sich um eine normale Schachtel?
getWidth()	horizontale Weite dieser Schachtel
getStretchability()	um wieviel Einheiten darf gedehnt werden?
getShrinkability()	um wieviel Einheiten darf gestaucht werden?
stretch()	Auseinanderziehen der Schachtel
shrink()	Zusammenstauchen der Schachtel
getHeight()	Höhe der Schachtel oberhalb der Basislinie
getDepth()	Tiefe der Schachtel unterhalb der Basislinie
getPenalty()	Je größer der Wert ist, umso unschöner ist eine Brechung an dieser Stelle
getPenaltyFlag()	konsekutive Trennungen an Stellen mit gesetztem Flag sind besonders unschön
genPostScript()	Generierung von PostScript für die Schachtel, wobei die Anfangsposition am Basispunkt liegt und der Endpunkt um die Weite nach rechts versetzt liegt.

- Diese Repräsentierung orientiert sich weitgehend an die des Artikels von Donald E. Knuth: *Breaking Paragraphs Into Lines*, der ursprünglich in 1981 in *Software–Practice and Experience* erschienen ist und innerhalb des Buchs *Digital Typography* 1999 nachgedruckt wurde.
- Hinzugekommen ist jetzt nur die Generierung von PostScript, das für unsere Beispiele die bequemste Ausgabe-Möglichkeit darstellt.
- Ein Knuth folgender Ansatz findet sich auch in der FOP-Klassenbibliothek im Paket *org.apache.fop.layoutmgr* mit der abstrakten Basisklasse *KnuthElement* und den davon abgeleiteten Klassen *KnuthBox*, *KnuthGlue* und *KnuthPenalty*.

Box.java

```
public abstract class Box implements Item {
   final public boolean isPenalty() { return false; }
   final public boolean isGlue() { return false; }
   final public boolean isBox() { return true; }
   public abstract int getWidth();
   public int getStretchability() { return 0; }
   public int getShrinkability() { return 0; }
   public void shrink(int units) {}
   public void stretch(int units) {}
   public abstract int getHeight();
   public abstract int getDepth();
   final public int getPenalty() { return Item.INFINITY; }
   public boolean getPenaltyFlag() { return false; }
   public abstract StringBuffer genPostScript(PostScriptContext context);
}
```

- *Box* ist eine abstrakte Klasse auf Basis der Schnittstelle *Item* für reguläre Schachteln ohne spezielle Trennungs- oder Dehnungseigenschaften.

GlyphBox.java

```
public class GlyphBox extends Box {
   private FontMetrics fm;
   private int code; private int size;
   private int[] bbox;

   public GlyphBox(FontMetrics fm, int size, int code) {
      this.fm = fm; this.size = size; this.code = code;
      bbox = fm.getBoundingBox(code);
   }
   public int getWidth() { return fm.getWidth(code) * size; }
   public int getHeight() { return bbox[3] * size; }
   public int getDepth() { return bbox[1] * size; }
   public StringBuffer genPostScript(PostScriptContext context) {
      return context.addTo(fm, size, code);
   }
}
```

- Für ein einzelnes Zeichen genügen uns die Infos aus der zugehörigen Metrik.

PostScriptContext.java

```
public class PostScriptContext {
   public PostScriptContext();
   public FontMetrics getCurrentFont(); public int getCurrentSize();
   public boolean insideString(); public StringBuffer closeString();
   public StringBuffer gsave(); public StringBuffer grestore();
   public StringBuffer switchTo(FontMetrics fm, int size);
   public StringBuffer addTo(FontMetrics fm, int size, int code) {
} // class PostScriptContext
```

- Um nicht vor jedem Zeichen den aktuellen Schriftschnitt auszuwählen und mehrere hintereinander kommende Einzelzeichen ohne Kerning zusammengefasst ausgeben zu müssen, steht mit *PostScriptContext* eine Klasse zur Verfügung, die diese Koordinierung übernimmt.

PostScriptContext.java

```
public StringBuffer switchTo(FontMetrics fm, int size) {
   StringBuffer result = new StringBuffer("");
   if (newcontext || fm != currentFont || size != currentSize) {
      result.append(closeString()); result.append("/");
      result.append(fm.getName()); result.append(" findfont ");
      result.append(size); result.append(" scalefont setfont\n");
      currentFont = fm; currentSize = size; newcontext = false;
   }
   return result;
}
```

- *switchTo()* generiert PostScript-Text, der den gewünschten Schriftschnitt auswählt, falls dieser sich verändert haben sollte.

PostScriptContext.java

```
public StringBuffer addTo(FontMetrics fm, int size, int code) {
   StringBuffer result = new StringBuffer("");
   result.append(switchTo(fm, size));
   if (!inString) {
      inString = true; result.append("(");
   }
   if (code == '(' || code == ')' || code == '\\') {
      result.append('\\');
      result.append(Integer.toOctalString(code));
   } else {
      result.append((char) code);
   }
   return result;
}
```

- *addTo()* fügt ein einzelnes Zeichen hinzu. Falls wir uns bereits in einer PostScript-Zeichenkette befinden, muss dabei nur das Zeichen selbst ausgegeben werden.

Schachteln für das Kerning

KerningBox.java

```
public class KerningBox extends Box {
   private int kerning;

   public KerningBox(int kerning) { this.kerning = kerning; }
   public int getWidth() { return kerning; }
   public int getHeight() { return 0; }
   public int getDepth() { return 0; }

   public StringBuffer genPostScript(PostScriptContext context) {
      StringBuffer result = context.closeString();
      result.append((float) kerning / 1000);
      result.append(" 0 rmoveto\n");
      return result;
   }
}
```

- Das Kerning wird durch Schachteln repräsentiert, die nur eine Weite haben, die im Falle des Zusammenrückens negativ ist.
- Umgesetzt in PostScript wird das Kerning durch eine relative Bewegung mit *rmoveto*.

Spezial-Schachtel für Dehnfugen

Glue.java

```
public class Glue implements Item {
   private int originalWidth; private int width;
   private int stretchability; private int shrinkability;

   private int intoRange(int value) {
      if (value > Item.INFINITY) {
         return Item.INFINITY;
      } else if (value < - Item.INFINITY) {
         return -Item.INFINITY;
      } else {
         return value;
      }
   }
   public Glue(int width, int stretchability, int shrinkability) {
      this.width = intoRange(width);
      originalWidth = this.width;
      assert(stretchability >= 0);
      this.stretchability = intoRange(stretchability);
      assert(shrinkability >= 0);
      this.shrinkability = intoRange(shrinkability);
   }
   // ...
}
```

Glue.java

```
final public boolean isGlue() { return true; }
final public boolean isPenalty() { return false; }
final public boolean isBox() { return false; }
public int getWidth() { return width; }
public int getStretchability() { return stretchability; }
public int getShrinkability() { return shrinkability; }
public void stretch(int units) { /* ... */ }
public void shrink(int units) { /* ... */ }
public int getHeight() { return 0; }
public int getDepth() { return 0; }
public int getPenalty() { return 0; }
final public boolean getPenaltyFlag() { return false; }

public StringBuffer genPostScript(PostScriptContext context) {
   StringBuffer result = context.closeString();
   if (width != 0) {
      result.append((double) width / 1000);
      result.append(" 0 rmoveto\n");
   }
   return result;
}
```

Penalty.java

```
public class Penalty implements Item {
   private int penalty;
   private boolean flag;

   private int intoRange(int value) {
      if (value > Item.INFINITY) {
         return Item.INFINITY;
      } else if (value < - Item.INFINITY) {
         return -Item.INFINITY;
      } else {
         return value;
      }
   }

   public Penalty(int penalty, boolean flag) {
      this.penalty = intoRange(penalty); this.flag = flag;
   }
   // ...
}
```

Penalty.java

```
final public boolean isPenalty() { return true; }
final public boolean isGlue() { return false; }
final public boolean isBox() { return false; }
public int getWidth() { return 0; }
public int getStretchability() { return 0; }
public int getShrinkability() { return 0; }
public void shrink(int units) {}
public void stretch(int units) {}
public int getHeight() { return 0; }
public int getDepth() { return 0; }
public int getPenalty() { return penalty; }
public boolean getPenaltyFlag() { return flag; }

public StringBuffer genPostScript(PostScriptContext context) {
   return new StringBuffer("");
}
```

HorizontalBox.java

```
public class HorizontalBox extends Box {
   private LinkedList<Item> items;
   private int width; private int height; private int depth;

   public HorizontalBox() {
      items = new LinkedList<Item>();
      width = 0; height = 0; depth = 0;
   }
   public void add(Item item) {
      items.addLast(item);
      width += item.getWidth();
      if (item.getHeight() > height) {
         height = item.getHeight();
      }
      if (item.getDepth() > depth) {
         depth = item.getDepth();
      }
   }
   // ...
}
```

HorizontalBox.java

```
public class HorizontalBox extends Box {
   // ...

   public int getWidth() { return width; }
   public int getHeight() { return height; }
   public int getDepth() { return depth; }

   public StringBuffer genPostScript(PostScriptContext context) {
      StringBuffer result = new StringBuffer("");
      for (Item item:items) {
         result.append(item.genPostScript(context));
      }
      return result;
   }
}
```

- Horizontale Schachteln akkumulieren einzelne Schachteln hintereinander.
- Eine horizontale Schachtel wird später nicht mehr aufgebrochen. Sie entsteht vielmehr als Resultat eines zeilenbrechenden Algorithmus.

Vertikal zusammengesetzte Schachteln

VerticalBox.java

```
public class VerticalBox extends Box {
   private LinkedList<Item> items;
   private int width; private int height; private int depth;
   private int baselineskip;

   public VerticalBox(int baselineskip) {
      items = new LinkedList<Item>();
      width = 0; height = 0; depth = 0;
      this.baselineskip = baselineskip;
   }

   public void add(Item item) {
      items.addLast(item);
      if (item.getWidth() > width) {
         width = item.getWidth();
      }
      height += baselineskip; depth = item.getDepth();
   }
   public int getWidth() { return width; }
   public int getHeight() { return height; }
   public int getDepth() { return depth; }
   // ...
} // class VerticalBox
```

VerticalBox.java

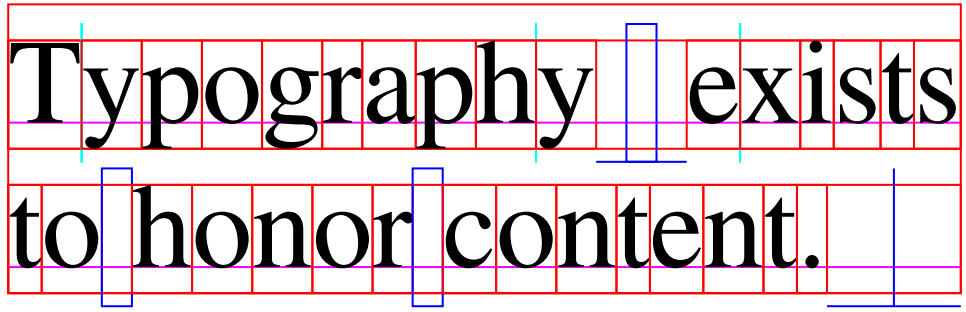
```
public VerticalBox(int baselineskip) {
   items = new LinkedList<Item>();
   width = 0; height = 0; depth = 0;
   this.baselineskip = baselineskip;
}

public void add(Item item) {
   items.addLast(item);
   if (item.getWidth() > width) {
      width = item.getWidth();
   }
   height += baselineskip; depth = item.getDepth();
}
```

- Der *baselineskip* legt den vertikalen Abstand zwischen den Basislinien der einzelnen Zeilen fest.
- Das ist genau das, was bei einem Paragraphen gewünscht wird. Für andere Zwecke werden andere vertikale Schachtelsysteme benötigt, die sich an den vertikalen Höhen der einzelnen Schachteln orientieren.

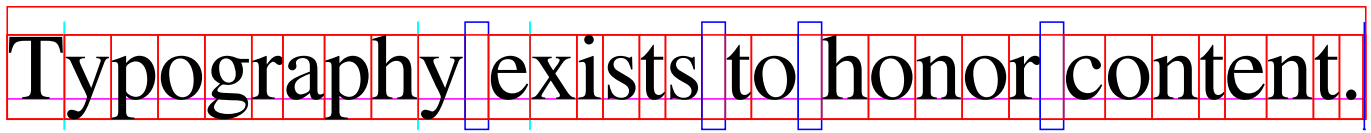
VerticalBox.java

```
public StringBuffer genPostScript(PostScriptContext context) {
   double bskip = (double) baselineskip / 1000;
   StringBuffer result = new StringBuffer("");
   if (items.size() > 0) {
      int i = 0;
      for (Item item:items) {
         result.append(context.gsave());
         if (i < items.size()-1) {
            result.append("0 ");
            result.append(bskip * (items.size()-1-i));
            result.append(" rmoveto\n");
         }
         result.append(item.genPostScript(context));
         result.append(context.closeString());
         result.append(context.grestore());
         ++i;
      }
      result.append((double) width / 1000);
      result.append(" 0 rmoveto\n");
   }
   return result;
}
```



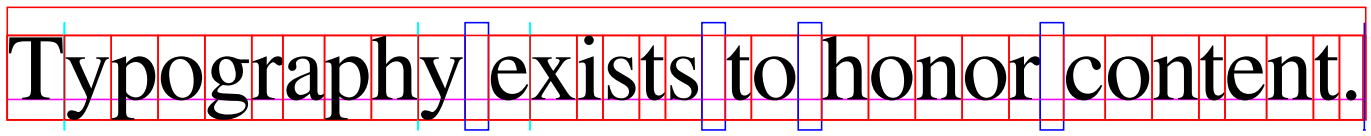
- Paragraphen werden durch vertikale Schachteln repräsentiert, bei denen horizontale Schachteln die einzelnen Zeilen einpacken, die wiederum aus den aneinandergereihten Zeichen, Kerning-Schachteln und Dehnfugen bestehen. Jede der Zeilen wurde an die gewünschte Paragraphenweite angepasst.
- Reguläre Schachteln sind in dieser Darstellung rot, Dehnfugen blau und Kerning-Schachteln hellblau. Die Schachteln der Zeichen wurden hier normalisiert (d.h. mit einheitlicher Tiefe und Höhe versehen), damit der Abstand zwischen der untersten Basislinie und der unteren Kante des Paragraphenblocks unabhängig davon ist, ob die unterste Zeile nach unten ragende Zeichen wie etwa ein »g« enthält oder nicht.

Ausgangssituation bei der Zerlegung eines Paragraphen 338

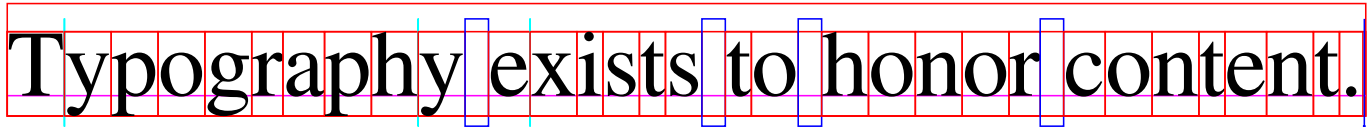


- Bevor eine Zerlegung eines Paragraphen beginnen kann, benötigen wir eine Datenstruktur, bei der alle Schachteln aufgereiht sind.
- Dies kann noch keine reguläre horizontale Schachtel sein, da horizontale Schachteln nicht mehr aufgebrochen werden können.

Ausgangssituation bei der Zerlegung eines Paragraphen 339



- Zu den einzelnen Elementen gehören
  - reguläre Schachteln, die beispielsweise ein Zeichen darstellen oder einen individuellen Abstand zwischen zwei aufeinanderfolgenden Zeichen regeln (Kerning),
  - Dehnfugen, die einen dehn- oder stauchbaren Leerraum repräsentieren ohne weitere sichtbare Darstellung und
  - Sollbruchstellen mit einem Strafwert als Häßlichkeitsmaß einer möglichen Trennung an dieser Stelle. Sollbruchstellen gibt es nach einer Folge von Binde- oder Gedankenstriche und bei potentiellen Trennstellen innerhalb von Wörtern.



- Nach der Definition von Donald E. Knuth gibt es genau zwei Arten zulässiger Trennungsstellen:
  - Sollbruchstellen mit einem Strafwert $< \infty$ und
  - Dehnungsfugen, die unmittelbar einer regulären Schachtel folgen.
- Die Eingabe des Zerlegungsalgorithmus kann auch als Folge von nicht trennbaren Schachteln betrachtet werden, denen jeweils eine zulässige Trennungsstelle folgt.
- Eine Sequenz nicht trennbarer Schachteln wird im folgenden Wort genannt.
- Damit Wörter immer durch eine zulässige Trennungsstelle terminiert werden, muss am Ende der Sequenz noch eine Trennungsstelle hinzugefügt werden. Hierfür bietet sich eine unendlich dehnbare Dehnfuge mit einer regulären Weite von 0 an.

HorizontalSequence.java

```
public interface HorizontalSequence extends Iterable<Item> {
   public void add(Item item);
   public void add(HorizontalSequence hseq);
   public Width getWidth();
   public IterableIterator<HorizontalSequence> getWords();
   public Item getFollowingBreakpoint();
   public IterableIterator<Item> getGlueItems();
   public HorizontalSequence clone();
} // class HorizontalSequence
```

- Sequenzen kann nur etwas hinzugefügt werden (entweder einzelne Schachteln oder andere Sequenzen).
- Der Datentyp *Width* vereinigt die reguläre Weite mit Hinweisen auf die Dehn- und Stauchbarkeit. Die Methode *getWidth()* liefert diese aufsummiert zurück für alle enthaltenen Schachteln.

Schnittstelle für Schachtel-Sequenzen

HorizontalSequence.java

```
public interface HorizontalSequence extends Iterable<Item> {
   public void add(Item item);
   public void add(HorizontalSequence hseq);
   public Width getWidth();
   public IterableIterator<HorizontalSequence> getWords();
   public Item getFollowingBreakpoint();
   public IterableIterator<Item> getGlueItems();
   public HorizontalSequence clone();
} // class HorizontalSequence
```

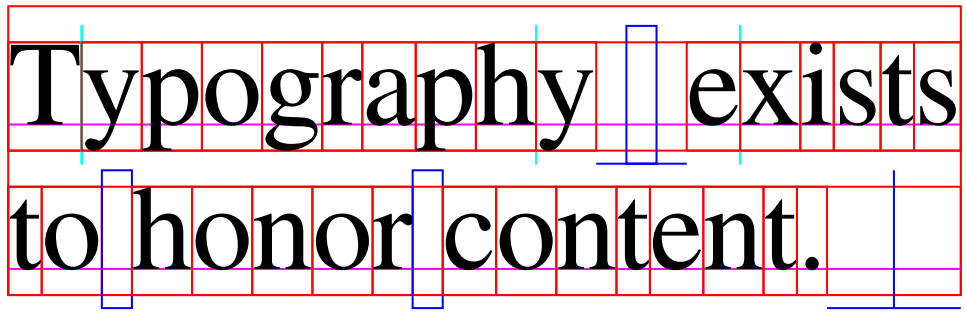
- Da diese Schnittstelle die Schnittstelle für *Iterable* beinhaltet, ist das Durchiterieren der einzelnen Schachteln möglich.
- Der Iterator *getWords* erlaubt es, über alle enthaltenen Worte zu iterieren. Wörter werden ebenfalls als Sequenzen repräsentiert, jedoch ohne die Trennungsstelle. Herausgegriffene Wörter erlauben den Zugriff auf die folgende Trennungsstelle mit der Methode *getFollowingBreakpoint()*.
- Wenn eine Sequenz an eine gegebene Weite durch Stauchen oder Dehnen anzupassen ist, dann liefert die Methode *getGlueItems()* die Dehnfugen.

- Für die Schnittstelle *HorizontalSequence* gibt es zwei Implementierungen: *SimpleHorizontalSequence* und die normalerweise nicht sichtbare *SimpleHorizontalSequence.SubSequence*.
- Die explizite Unterstützung von Subsequenzen erlaubt die Vermeidung der Duplikation von Datenstrukturen. Das hat zur Konsequenz, dass die Iteration mit *getWords* einschliesslich der Erzeugung der Subsequenzen für die einzelnen Worte nur einen Aufwand hat, der linear von der Zahl der Wörter abhängt und nicht deren Länge.
- Wenn Subsequenzen durch die *add*-Methoden verlängert werden, verändern sie nicht die zugrundeliegende Sequenz.

HorizontalFitter.java

```
public class HorizontalFitter {
   public static void fit(HorizontalSequence hseq, int width);
} // class HorizontalFitter
```

- Die Klasse *HorizontalFitter* offeriert nur eine statische Methode *fit()*, die versucht, die gegebene horizontale Sequenz an die vorgegebene Weite anzupassen, indem die zur Verfügung stehenden Dehnfugen angepasst werden.
- Wenn die Sequenz zu dehnen ist und die Sequenz Dehnfugen mit unendlich großen Dehnkapazitäten besitzt, dann wird die notwendige Dehnung gleichmäßig über alle unendlich dehnbaren Dehnfugen verteilt.
- Ansonsten wird die notwendige Dehnung oder Stauchung entsprechend der jeweiligen Dehn- oder Stauchkapazitäten gleichmäßig bei allen Dehnfugen durchgeführt.



- Eine unendlich dehnbare Dehnfuge ist insbesondere am Ende eines Paragraphen sinnvoll, damit die letzte Zeile nicht bis auf die Paragraphenweite ausgedehnt wird.
- Wie an diesem Beispiel zu sehen ist, wurden die normalen Dehnfugen zwischen den Wörtern der letzten Zeile nicht ausgedehnt, sondern nur die unendlich dehnbare Dehnfuge am Ende des Paragraphen.