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

Um diese Probleme zu lösen, werden Algorithmen benötigt,

- um Paragraphen in Zeilen zu zerlegen,
- um geeignete Trennungsstellen in Wörtern zu finden und
- für die Seitenaufteilung.

Voraussetzung für die Algorithmen sind geeignete Datenstrukturen.

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.

Einfache Schnittstelle für eine Metrik

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 fehlt noch.
- Die Unterstützung für kombinierte Zeichen fehlt ebenfalls.

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.

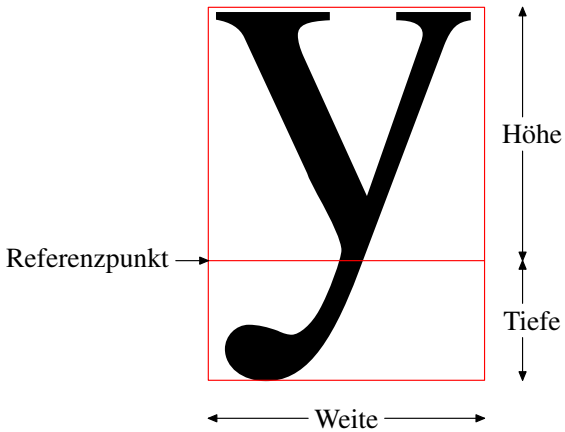
TestAV.java

```
public class TestAV {
   private final static String AFM_INPUT =
      "/usr/local/texlive/2013/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.
- Zusätzlich zu den durch die Dehnfugen repräsentierten Sollbruchstellen kann es weitere geben, die selbst keinen Platz einnehmen. Denkbar sind hier etwa Stellen unmittelbar nach einem Binde- oder Gedankenstrich. Ferner kann ein Wörtertrennungsalgorithmus Sollbruchstellen innerhalb eines Worts ermitteln. In letzterem Falle kommt bei einer Trennung noch ein Trennungszeichen hinzu.

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. (Noch 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 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.

Schachteln für einzelne Zeichen

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 können, 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*.

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 ist vielmehr eines der Resultate eines zeilenbrechenden Algorithmus.

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

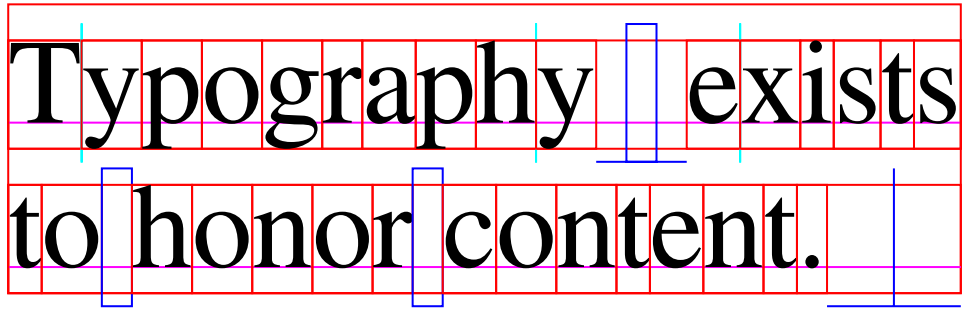
```
public VerticalBox(int baselineskip) {
   items = new LinkedList<Item>();
   width = 0; height = 0; depth = 0;
   this.baselineskip = baselineskip;
}

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.

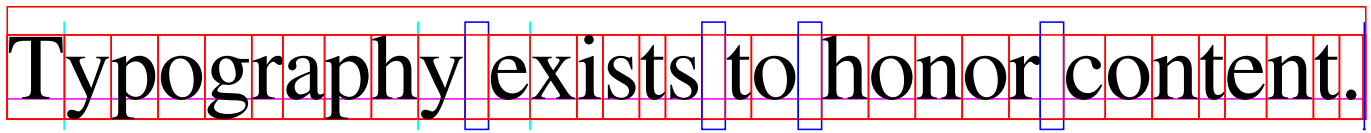
VerticalBox.java

```
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;
}
```



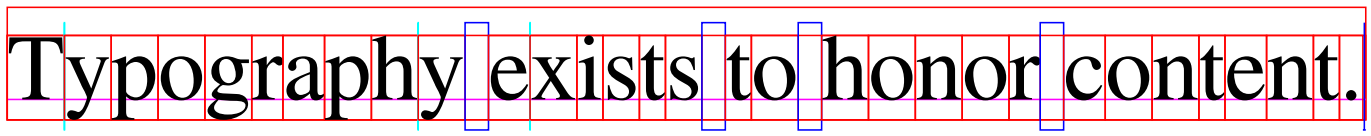
- Paragraphen werden durch vertikale Schachteln repräsentiert, bei denen horizontale Schachteln die einzelnen Zeilen einpacken, die wiederum aus den aneinandergereihten Zeichen, Kerning-Schachteln und Dehnfugen bestehen. Jede der Zeilen wurde an die gewünschte Paragraphenweite angepasst.
- Reguläre Schachteln 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 359

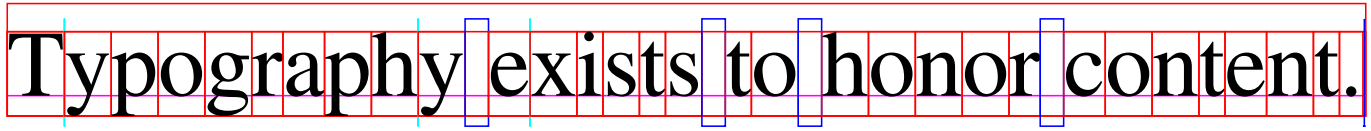


- Bevor eine Zerlegung eines Paragraphen beginnen kann, benötigen wir eine Datenstruktur, bei der alle Schachteln aufgereiht sind.
- Dies kann noch keine reguläre horizontale Schachtel sein, da horizontale Schachteln nicht mehr aufgebrochen werden können.

Ausgangssituation bei der Zerlegung eines Paragraphen 360



- 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 beliebig dehnbare Dehnfuge mit einer regulären Weite von 0 an.

Schnittstelle für Schachtel-Sequenzen 362

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.

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
```

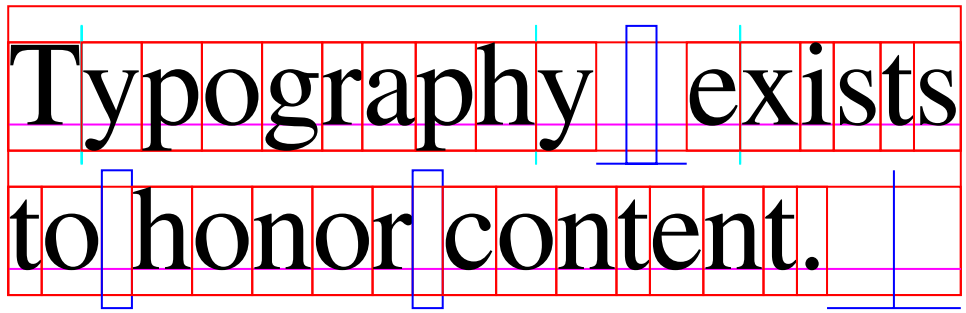
- Da diese Schnittstelle die Schnittstelle für *Iterable* beinhaltet, ist das Durchiterieren der einzelnen Schachteln möglich.
- Der Iterator *getWords* erlaubt es, über alle enthaltenen Worte zu iterieren. Wörter werden ebenfalls als Sequenzen repräsentiert. Herausgegriffene Wörter erlauben den Zugriff 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 großer 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.

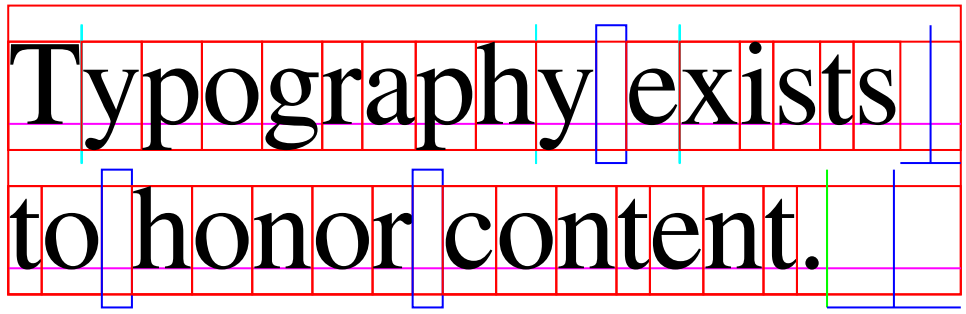


- Eine unendlich dehnbare Dehnfuge ist insbesondere am Ende eines Paragraphen sinnvoll, damit die letzte Zeile nicht bis auf die Paragraphenweite ausgedehnt wird.
- Wie an diesem Beispiel zu sehen ist, wurden die normalen Dehnfugen zwischen den Wörtern der letzten Zeile nicht ausgedehnt, sondern nur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

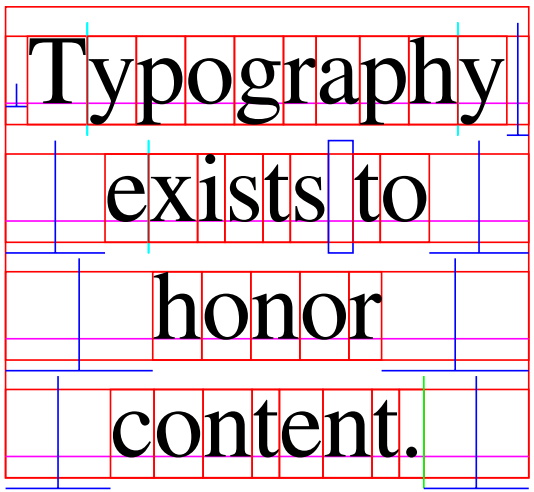
LineBreaker.java

```
public interface LineBreaker {
   public void setLineWrapper(SequenceWrapper wrapper);
   public void setHorizontalBoxWrapper(ItemWrapper hboxwrapper);
   public Item breakParagraph(HorizontalSequence hseq,
      int parwidth, int baselineskip);
} // interface LineBreaker
```

- Da es mehrere zeilenbrechende Algorithmen gibt, ist eine allgemeine Schnittstelle hierfür sinnvoll.
- Der eigentliche Algorithmus wird verpackt durch die Methode *breakParagraph()*, die eine horizontale Sequenz in Zeilen bricht, die einzelnen Zeilen an die vorgegebene Weite anpasst und diese dann mit dem gegebenen Zeilenabstand in einer vertikalen Box übereinander stapelt.
- Optional kann mit *setLineWrapper()* eine Bearbeitung einer Zeilensequenz vor der Anpassung an die vorgegebene Weite eingebaut werden. Ferner ist mit *setHorizontalBoxWrapper()* eine Nachbearbeitung der horizontalen Schachteln für die einzelnen Zeilen möglich.



- Mit *setLineWrapper()* ist es möglich, vom traditionellen Blocksatz abzuweichen.
- Wenn der Wrapper an die horizontalen Sequenzen jeweils eine Dehnfuge mit unendlicher Dehnkapazität anfügt, erhalten wir einen aufgerauhten rechten Rand.
- Der grüne Strich repräsentiert hier eine Sollbruchstelle, die an das Ende des Paragraphen angehängt wurde. Auf diese Weise wird auch das letzte Wort mit einer Trennstelle terminiert, wenn zu Beginn die sonst übliche unendlich dehnbare Dehnfuge am Ende des Paragraphen fehlt.



- Wenn durch den Wrapper für die einzelnen Zeilen jeweils am Anfang und Ende jeweils eine unendlich dehnbare Dehnfuge hinzugefügt wird, erhalten wir eine zentrierte Formatierung.

Basisklasse für zeilenbrechende Algorithmen 370

BasicLineBreaker.java

```
public abstract class BasicLineBreaker implements LineBreaker {
   protected SequenceWrapper wrapper;
   protected ItemWrapper hboxwrapper;

   public BasicLineBreaker() {
      wrapper = null; hboxwrapper = null;
   }

   public void setLineWrapper(SequenceWrapper wrapper) {
      this.wrapper = wrapper;
   }

   public void setHorizontalBoxWrapper(ItemWrapper hboxwrapper) {
      this.hboxwrapper = hboxwrapper;
   }

   // ...

   public abstract Item breakParagraph(HorizontalSequence hseq,
      int parwidth, int baselineskip);
} // class BasicLineBreaker
```

BasicLineBreaker.java

```
protected void add(VerticalBox vbox,
      HorizontalSequence line, int parwidth) {
   if (wrapper != null) {
      line = wrapper.wrap(line);
   }
   HorizontalFitter.fit(line, parwidth);
   Item hbox = new HorizontalBox(line);
   if (hboxwrapper != null) {
      hbox = hboxwrapper.wrap(hbox);
   }
   vbox.add(hbox);
}
```

- Die *add*-Methode wird von den abgeleiteten Klassen aufgerufen, um eine fertiggestellte *HorizontalSequence* an eine *VerticalBox* anzuhängen.
- An dieser Stelle werden die beiden Wrapper aufgerufen: *wrapper.wrap* passt ggf. die Formatierung an (Blocksatz, aufgerauht, zentriert etc) und der *hboxwrapper* dient nur u.U. der Visualisierung des Schachtelsystems.

- Gegeben ist eine horizontale Sequenz, die in einzelne Wörter zerlegt ist.
- Für den Algorithmus wird zusätzlich eine horizontale Sequenz verwaltet, die für die aktuelle Zeile steht. Diese ist zu Beginn leer.
- Für jedes Wort aus der horizontalen Sequenz wird überprüft, ob sich dieses noch auf die aktuelle Zeile quetschen lässt, d.h. ob der bereits vorhandene Zeileninhalt und die dazwischenliegende Trennstelle und das neue Wort bei einer maximalen Stauchung noch in eine Zeile passt.
- Wenn das neue Wort hineinpasst, dann wird das Wort in die aktuelle Zeile mitsamt der davorliegenden Trennstelle hinzugefügt. Die Schleife wird danach fortgesetzt.
- Wenn es nicht hineinpasst, wird die bislang vollendete Zeile zu den fertiggestellten Zeilen hinzugefügt (in eine vertikale Schachtel) und die aktuelle Zeile mit dem neuen Wort initialisiert. Danach geht es in der Schleife weiter.
- Am Ende der Schleife wird die aktuelle Zeile noch hinzugefügt, falls sie nicht leer ist.

FirstFitLineBreaker.java

```
public class FirstFitLineBreaker extends BasicLineBreaker {
   public Item breakParagraph(HorizontalSequence hseq,
         int parwidth, int baselineskip) {
      VerticalBox vbox = new VerticalBox(baselineskip);

      HorizontalSequence line = null; // current line
      Item bp = null; // last breakpoint
      // ... for loop over all words of hseq ...
      if (line != null) {
         line.add(bp);
         add(vbox, line, parwidth);
      }
      return vbox;
   }
} // class FirstFitLineBreaker
```

- Zu Beginn wird eine neue *VerticalBox* erzeugt, dann werden in der **for**-Schleife dieser sukzessive Zeilen hinzugefügt (siehe folgende Folie) und schließlich der Rest, sofern vorhanden, angehängt.

FirstFitLineBreaker.java

```
for (HorizontalSequence word: hseq.getWords()) {
   if (line == null) {
      line = word;
   } else {
      Width sum = new Width(line.getWidth());
      sum.add(bp); sum.add(word.getWidth());
      if (sum.getWidth() > parwidth) {
         if (sum.getWidth() - sum.getShrinkability()
               <= parwidth) {
            line.add(bp); line.add(word);
            add(vbox, line, parwidth); line = null;
         } else {
            add(vbox, line, parwidth); line = word;
         }
      } else {
         line.add(bp); line.add(word);
      }
   }
   bp = word.getFollowingBreakpoint();
}
```

- Der Best-Fit-Algorithmus bemüht sich um lokale Minima bezüglich der Häßlichkeit.
- Die Häßlichkeit wird danach bewertet, wie dramatisch der zur Verfügung stehende Spielraum ausgenutzt wird.
- Ein Wert von 0 bedeutet, dass nichts verändert werden muss. Wenn ein Wert von 1 erreicht wird, dann wird der Spielraum vollständig ausgenutzt. Wenn der Wert von 1 überschritten wird, dann überstrapazieren wir die vorhandene Flexibilität.
- Der Best-Fit-Algorithmus vergleicht alle Trennungskandidaten am nächsten Zeilenende und vergleicht sie anhand des Maßes.
- Best-Fit liefert nicht unbedingt bessere Resultate als First-Fit, da möglicherweise das globale Minimum erst dann erreicht wird, wenn zunächst eine etwas häßlichere Variante gewählt wird.

BestFitLineBreaker.java

```
public class BestFitLineBreaker extends BasicLineBreaker {
   public Item breakParagraph(HorizontalSequence hseq,
         int parwidth, int baselineskip) {
      VerticalBox vbox = new VerticalBox(baselineskip);

      HorizontalSequence line = null; // current line
      HorizontalSequence candidate = null;
      HorizontalSequence nextline = null; // beginning of next line
      double badness = 0; // of candidate
      Item bp = null; // last breakpoint
      // ... for loop over all words ...
      if (line != null) {
         line.add(bp);
         add(vbox, line, parwidth);
      }
      return vbox;
   }
} // class BestFitLineBreaker
```

BestFitLineBreaker.java

```
    for (HorizontalSequence word: hseq.getWords()) {
       if (line == null) {
          line = word;
       } else {
          line.add(bp); line.add(word);
       }
       if (candidate != null) {
          if (nextline == null) {
             nextline = word;
          } else {
             nextline.add(bp); nextline.add(word);
          }
       }
       Width width = line.getWidth();
       // ... analysis ...
       bp = word.getFollowingBreakpoint();
    }
```

- Zu dem aktuellen Kandidaten *candidate* gehört, falls definiert, auch *nextline* (was gehört in die nächste Zeile) und *badness* (gemessener Wert).

BestFitLineBreaker.java

```
if (width.getWidth() - width.getShrinkability() > parwidth) {
   if (candidate == null) {
      add(vbox, line, parwidth); line = null;
   } else {
      add(vbox, candidate, parwidth);
      line = nextline; candidate = null; nextline = null;
   }
} else if (width.getWidth() + width.getStretchability() >= parwidth) {
   double newbadness;
   if (width.getWidth() < parwidth) {
      newbadness = ((double) parwidth - width.getWidth()) /
         width.getStretchability();
   } else {
      newbadness = ((double) width.getWidth() - parwidth) /
         width.getShrinkability();
   }
   if (candidate == null || newbadness < badness) {
      candidate = line.clone(); nextline = null; badness = newbadness;
   }
} else {
   candidate = line.clone(); nextline = null;
   badness = Item.INFINITY;
}
```

Überblick der Struktur einer horizontalen Sequenz nach Knuth

- Eine horizontale Sequenz aus m Elementen kann durch die folgenden sechs Folgen repräsentiert werden:

$t_1 \ldots t_m$	Typ des Elements: *box*, *glue* oder *penalty*.
$w_1 \ldots w_m$	Die Weite des Elements.
$y_1 \ldots y_m$	Der Ausdehnungsspielraum bei Dehnfugen (*glue*), bei anderen Elementen 0.
$z_1 \ldots z_m$	Der Schrumpfungsspielraum bei Dehnfugen (*glue*), bei anderen Elementen 0.
$p_1 \ldots p_m$	Der Strafwert bei Sollbruchstellen, bei anderen Elementen 0.
$f_1 \ldots f_m$	Der Schalter bei Sollbruchstellen, bei anderen Elementen 0.

- Gegeben ist eine Folge $\{l_i\}$ von gewünschten Zeilenlängen. Normalerweise gilt $l_1 = l_2 = \ldots$, aber im Falle von Illustrationen oder anderen ungewöhnlichen Randbedingungen können diese voneinander abweichen.
- Wenn für die Zeile j die Elemente $a_j \ldots b_j$ in Betracht gezogen werden, dann ergeben sich aufsummiert folgende Werte:

$$
\begin{aligned}
L_j &= \sum_{i=a_j}^{b_{j-1}} w_i + \begin{cases} w_{b_j} & \text{falls } t_{b_j} = \mathit{penalty} \\ 0 & \text{sonst} \end{cases} \\
Y_j &= \sum_{i=a_j}^{b_{j-1}} y_i \\
Z_j &= \sum_{i=a_j}^{b_{j-1}} z_i
\end{aligned}
$$

- In Abhängigkeit von L_j und l_j lässt sich ein Justierverhältnis r_j bestimmen:
  - Falls $L_j = l_j$, dann passt die Zeile ganz genau und es sei $r_j = 0$
  - Falls $L_j < l_j$, dann ist die Zeile zu kurz und es sei $r_j = \frac{l_j - L_j}{Y_j}$, falls $Y_j > 0$, und undefiniert andernfalls.
  - Falls $L_j > l_j$, dann hat die Zeile Überlänge und es sei $r_j = \frac{l_j - L_j}{Z_j}$, falls $Z_j > 0$, und undefiniert andernfalls.
- Beispiel: Falls $r_j = \frac{1}{2}$, dann muss der Ausdehnungsspielraum zur Hälfte ausgenutzt werden.
- Die einzelnen Weiten aller Dehnfugen innerhalb der Zeile sind dann auf $w_i + r_j y_i$ zu setzen, falls $r_j \geq 0$ und $w_i + r_j z_i$, falls $r_j < 0$.

- Bereits 1963 schlug C. J. Duncan einen Algorithmus vor, der eine Zerlegung akzeptierte, sobald $|r_j| \leq 1$ galt. In diesem Falle haben wir eine Lösung, die im Rahmen der vorgesehenen Spielräume für Ausdehnungen und Schrumpfungen liegt.
- Da es jedoch häufig mehrere solcher Lösungen gibt, lohnt es sich, diese zu vergleichen. Auch falls es keine Lösung gibt, ist es wohl im Notfall sinnvoll, die am wenigsten schlechte Lösung auszusuchen.
- Knuth hat für TEX folgendes Maß als für sinnvoll befunden, das die Unschönheit einer Zeile bewertet:

$$\beta_j = \begin{cases} \infty, & \text{falls } r_j \text{ undefiniert oder } r_j < -1 \\ \lfloor 100|r_j|^3 + .5 \rfloor & \text{sonst} \end{cases}$$

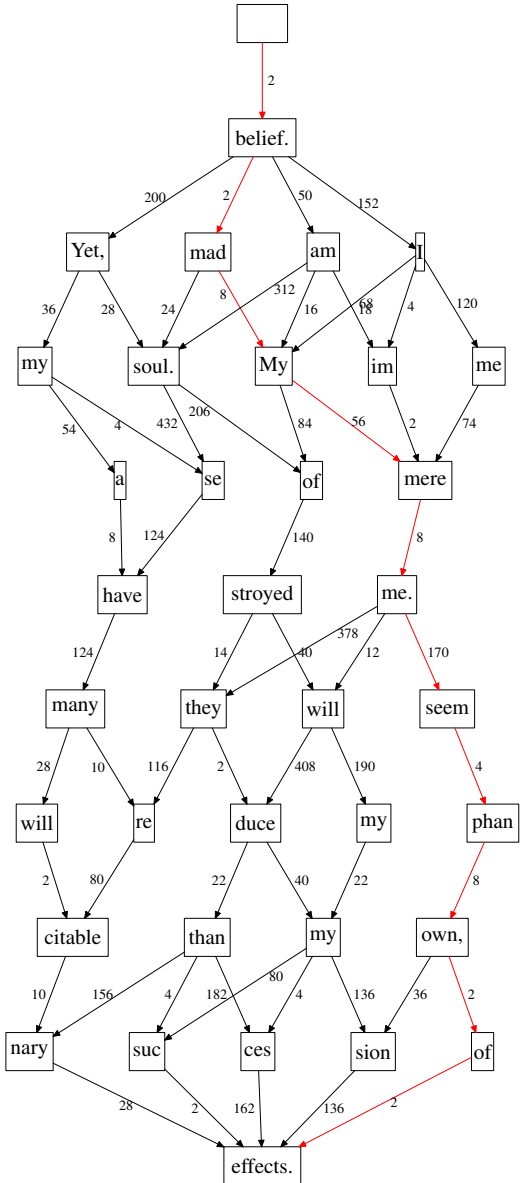
- Bei einer Aggregierung der Unschönheiten der einzelnen Zeilen geht der Strafwert der Sollbruchstelle ein: $\pi_j = p_{b_j}$
- Ferner sei α_j positiv (z.B. 3000), falls die j-te Zeile von zwei Sollbruchstellen eingefasst ist, bei der die Schalter jeweils auf 1 stehen. (Aufeinanderfolgende Trennungen von Wörtern sind zu vermeiden.)
- Darauf aufbauend definiert Knuth folgendes Maß für die Unschönheit einer Zeile:

$$\delta_j = \begin{cases} (1+\beta_j+\pi_j)^2+\alpha_j & \text{falls } \pi_j \geq 0 \\ (1+\beta_j)^2-\pi_j^2+\alpha_j & \text{falls } -\infty < \pi_j < 0 \\ (1+\beta_j)^2+\alpha_j & \text{falls } \pi_j = -\infty \end{cases}$$

- Das Maß für die Unschönheit einer Zerlegung ergibt sich dann aus der Summe der δ_j.

- Knuth versuchte mit seinem Maß folgende Ziele umzusetzen:
  - Wenn eine einzelne Zeile unvermeidlich schlecht ist, sollen die anderen Zeilen dennoch möglichst schön gehalten werden.
  - Die Unschönheit wächst mit den Strafwerten der Sollbruchstellen π_j. Bei negativen Strafwerten wird ein Bruch an der Stelle gerne gesehen. Allerdings wird bei $\pi_j = -\infty$ dies ignoriert, da hier ein Bruch unvermeidlich ist.
  - Da jeweils 1 dazu addiert wird, werden bei ansonsten annäherungsweise gleich guten Lösungen diejenigen mit weniger Zeilen bevorzugt.

Zerlegung eines Paragraphen als Optimierungsproblem 385



- Optimierungsproblem: Finde die Zerlegung mit der minimalen Summe der Unschönheitsmaße.
- Das Problem kann als gerichteter, antizyklischer Graph (DAG) dargestellt werden, bei dem die möglichen Bruchstellen als Knoten repräsentiert werden und die zugehörigen Unschönheitsmaße der jeweils vorangehenden Zeile als Kostenwert.
- Einen extra Knoten gibt es für den Anfang und ebenso gibt es nur einen Endknoten.
- Das linke Beispiel ist ein zusammengekürzter Graph, bei dem Kanten mit extremem Unschönheiten herausgefiltert wurden.

- Gegeben sei ein Paragraph mit $n+1$ Sollbruchstellen, wobei die letzte Sollbruchstelle unvermeidlich ist.
- Zulässige Zerlegungen sind dann Teilmengen aller Sollbruchstellen, die die letzte Sollbruchstellstelle enthalten.
- Dann gibt es insgesamt $\sum_{i=0}^{n} \binom{n}{i}$ mögliche Zerlegungen.
- Der Aufwand, sämtliche zulässige Zerlegungen zu untersuchen, wäre extrem.
- Wenn es als graphentheoretisches Problem betrachtet wird (Finden des kürzesten Wegs), dann lässt sich der Aufwand unter Verwendung des Algorithmus von Dijkstra auf $O(n^2)$ begrenzen. Dies ist immer noch sehr aufwendig.

- Knuth hat gezeigt, dass sich der Aufwand auf $O(n * m)$ begrenzen lässt, wobei dann m dann der Zahl der zu erwartenden Bruchstellen pro Zeile entspricht.
- Der Wert m lässt sich noch weiter reduzieren, wenn
  - Trennstellen in Wörtern erst dann betrachtet werden, wenn klar ist, dass sie benötigt werden, und
  - Zerlegungen mit extrem hohen Unschönheitsmaßen an einem Zeilenbruch von vorneherein aussortiert werden.
- In der Praxis bedeutet dies, dass der Aufwand sich auf $O(n)$ beschränkt, d.h. mit linearem Aufwand eine Zerlegung eines Paragraphen erfolgen kann. Voraussetzung dafür ist allerdings, dass Paragraphen nicht ungewöhnlich lang werden.

- Der gesamte Paragraph wird in einem Durchgang sequentiell bearbeitet.
- Es gibt eine Menge der Bruchstellen (zu Beginn nur der Anfang des Paragraphen). Eine Teilmenge davon sind die aktiven Bruchstellen (zu Beginn ebenfalls nur der Anfang des Paragraphen).
- Bei jeder Sollbruchstelle beim sequentiellen Durchgang wird die Unschönheit der Zeile gemessen ausgehend von jeder aktiven Bruchstelle bis zur neuen Sollbruchstelle.
- Wenn keine aktive Bruchstelle gefunden wird mit $|r| \leq 1$, dann wird die Sollbruchstelle nicht weiter beachtet.
- Andernfalls wird die Sollbruchstelle in die Liste der aktiven Bruchstellen aufgenommen. Dabei wird notiert, zu welcher vorherigen aktiven Bruchstelle δ minimal war.
- Aktive Bruchstellen, die zur aktuellen Sollbruchstelle einen Wert $r < -1$ haben, werden aus der Liste der aktiven Bruchstellen entfernt.
- Inaktive Bruchstellen, auf die keine aktive Bruchstelle verweist, können entfernt werden (Sackgassen).

First-Fit:

For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject their own evidence. Yet, mad am I not--and very surely do I not dream. But to-morrow I die, and to-day I would unburden my soul. My immediate purpose is to place before the world, plainly, succinctly, and without comment, a series of mere household events. In their consequences, these events have terrified--have tortured--have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but horror--to many they will seem less terrible than baroques. Hereafter, perhaps, some intellect may be found which will reduce my phantasm to the commonplace--some intellect more calm, more logical, and far less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than an ordinary succession of very natural causes and effects.

Total-Fit:

For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject their own evidence. Yet, mad am I not--and very surely do I not dream. But to-morrow I die, and to-day I would unburden my soul. My immediate purpose is to place before the world, plainly, succinctly, and without comment, a series of mere household events. In their consequences, these events have terrified--have tortured--have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but horror--to many they will seem less terrible than baroques. Hereafter, perhaps, some intellect may be found which will reduce my phantasm to the commonplace--some intellect more calm, more logical, and far less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than an ordinary succession of very natural causes and effects.

Best-Fit:

For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject their own evidence. Yet, mad am I not--and very surely do I not dream. But to-morrow I die, and to-day I would unburden my soul. My immediate purpose is to place before the world, plainly, succinctly, and without comment, a series of mere household events. In their consequences, these events have terrified--have tortured--have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but horror--to many they will seem less terrible than baroques. Hereafter, perhaps, some intellect may be found which will reduce my phantasm to the commonplace--some intellect more calm, more logical, and far less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than an ordinary succession of very natural causes and effects.

Total-Fit:

For the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject their own evidence. Yet, mad am I not--and very surely do I not dream. But to-morrow I die, and to-day I would unburden my soul. My immediate purpose is to place before the world, plainly, succinctly, and without comment, a series of mere household events. In their consequences, these events have terrified--have tortured--have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but horror--to many they will seem less terrible than baroques. Hereafter, perhaps, some intellect may be found which will reduce my phantasm to the commonplace--some intellect more calm, more logical, and far less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than an ordinary succession of very natural causes and effects.

- Für einen Blocksatz mit einer einheitlichen Farbe des Texts ist die Möglichkeit, Wörter bei Bedarf trennen zu können, essentiell.
- Einfache Trennungsregeln stehen nicht zur Verfügung oder liefern zuviele Falschtrennungen oder finden zuwenig Trennungsstellen.
- Die erste TEX-Implementierung von 1977 entfernte zunächst die Nachsilbe, die Vorsilbe und suchte dann nach Folgen von Vokal-Konsonant-Konsonant-Vokal und trennte dann zwischen den beiden Konsonanten. Damit werden jedoch nur ca. 40% der zulässigen Trennstellen erfasst und es bleiben Fehler wie etwa in „prog-ram“, die von längeren Ausnahmenlisten erfasst werden müssen.
- Einzelne Wörterbücher nennen umfassend Trennstellen (wie etwa Webster's oder der Duden). Diese Listen sind jedoch recht umfangreich und trotz ihres Umfangs nicht umfassend. Dies gilt insbesondere für Sprachen, die Zusammensetzungen erlauben und zahlreiche Beugungen (Flexionen) vorsehen wie etwa die deutsche Sprache.

- Gegeben sei eine Liste mit Trennstellen für einen umfangreichen Wortschatz. (Bei TEX wurde hier das *Merriam-Webster Pocket Dictionary* von 1974 mit ca. 50.000 Einträgen verwendet. Für die deutsche Sprache gibt es eine von Werner Lemberg gepflegte Liste mit 448.508 Einträgen, siehe `http://repo.or.cz/w/wortliste.git`.)
- Gesucht werden
  - ein sprachunabhängiges Regelsystem,
  - ein Verfahren, dass das Regelsystem für eine Sprache mit Hilfe von einer umfangreichen Trennliste automatisiert konfiguriert, so dass die Zahl der erzeugten Regeln möglichst minimal ist und
  - Datenstrukturen und Algorithmen, die mit geringem Laufzeit- und Speicheraufwand die möglichen Trennstellen eines Wortes finden, selbst wenn es nicht auf der Liste enthalten war.
- Ziel sollte es sein, dass die Mehrheit der korrekten Trennstellen gefunden wird und nur marginal wenige falsche Trennstellen geliefert werden, die notfalls mit Hilfe von Ausnahmelisten behandelt werden können.

- Das Regelsystem betrachtet nur vier aufeinanderfolgende Buchstaben $b_1 \ldots b_4$, um zu beurteilen, ob zwischen b_2 und b_3 getrennt werden kann.
- Da eine Tabelle mit $26^4 = 456976$ Einträgen zu umfangreich ist, gibt es stattdessen drei Tabellen für die Buchstabenpaare (b_1, b_2), (b_2, b_3) und (b_3, b_4), die jeweils die Wahrscheinlichkeiten angeben, dass hinter, zwischen und vor dem jeweiligen Buchstabenpaar in der gegebenen Wortliste getrennt wird.
- Die drei Werte aus den Tabellen werden miteinander multipliziert und es werden alle Trennstellen berücksichtigt, bei denen das Produkt einen vorgegebenen Mindestwert erreicht. (Hierbei wird ignoriert, dass die drei Wahrscheinlichkeiten nicht unabhängig voneinander sind.)
- Franklin Liang stellte in seiner Untersuchung jedoch fest, dass mit diesem Verfahren nur ca. 40% der Trennstellen gefunden werden bei einer Fehlerquote von 8%.

- Flexibler als die Wahrscheinlichkeitstabellen für die Buchstabenpaare sind Trennungsmuster, da sie beliebig lang sein können und auch gängige Vorsilben und Nachsilben berücksichtigen können.
- Beispiele für Trennungsmuster der englischen Sprache aus der Arbeit von Franklin Liang:

  .in-d .in-s .in-t .un-d b-s -cia con-s con-t e-ly erl-l er-m ex- -ful it-t i-ty -less l-ly -ment n-co -ness n-f n-l n-si n-v om-m -sion s-ly s-nes ti-ca x-p

  (Der Punkt repräsentiert jeweils den Anfang oder das Ende eines Worts.)
- Beispiele für die deutsche Sprache:

  .es-p .ob-l a-bl an-kl eu-e e-z ge-s g-q h-d h-h i-che i-d ll-b o-ra.

- Auch wenn Trennungsmuster viele Fälle korrekt erfassen, so bleibt doch eine signifikante Zahl von Fällen, bei denen falsch getrennt wird.
- Beispiel: Obwohl die Regel *-tion* in vielen Fällen zutrifft, darf „cation“ nicht getrennt werden.
- Wenn Ausnahmen nur über eine explizite Ausnahmeliste geregelt werden können, wird sie zu umfangreich.
- Deswegen ist es sinnvoll, auch Ausnahmeregeln in Form von Regeln zu formulieren. Als Ausnahmeregel würde sich hier *.cat* empfehlen.
- Das System lässt sich fortsetzen mit Ausnahmen der Ausnahmeregeln etc.

- Das Regelsystem besteht aus einer beliebigen Zahl von Trennungsmustern und einer Festlegung der Variablen *leftmin* und *rightmin*. (Die beiden Variablen geben den Mindestumfang eines Wortteils an, der vorne oder am Ende des Worts abgetrennt werden kann. In der englischen Sprache sind beide Werte 2, in der deutschen Sprache kann *leftmin* auf 1 gesetzt werden.)
- Jedes Trennungsmuster besteht aus Buchstaben, dem Punkt "." als Zeichen für den Wortanfang oder das Wortende und Ziffern, die Trennstellen bewerten.
- Ziffern repräsentieren den Rang einer Trennstelle in einer Regel. Ungerade Ränge (beginnend ab 1) befürworten eine Trennung, gerade Ränge (beginnend ab 2) stehen für eine Unterdrückung einer Trennstelle.
- Für ein zu trennendes Wort werden sämtliche zutreffenden Regeln berücksichtigt. Wenn sich mehrere Regeln für eine potentielle Trennregel widersprechen, dann setzt sich die Regel mit dem höheren Rang für die Trennstelle durch.

- Zu trennen ist das Wort „typography“. Folgende Trennungsmuster bei TEX treffen darauf zu (siehe Datei *hyphen.tex*):
  - *1ty*
  - *y3po*
  - *5po4g*
  - *1gr*
  - *4graphy*
  - *3raphy*
  - *1phy*
- Das Resultat ist „ty-pog-ra-phy“.
- Zu sehen ist hier, dass normalerweise durchaus vor „gr“ getrennt wird (4. Regel), dies in diesem konkreten Fall jedoch durch zwei Ausnahmeregeln unterbunden wird: *5po4g* und *4graphy* – in beiden Fällen schlägt der Rang 4 den Rang 1.
- Die vorgeschlagene Trennung entspricht genau der, die von Merriam-Webster vorgegeben wird.

- Jede Trennungsregel enthält mindestens einen Buchstaben.
- Entsprechend müssen bei einem Wort der Länge n insgesamt $\frac{n(n+1)}{2}$ Teilzeichenfolgen untersucht werden, was einem Aufwand von $O(n^2)$ entspricht.
- Dieser Aufwand multipliziert sich mit dem Aufwand, nach einer Regel für eine Zeichenfolge zu suchen.
- Das entspricht zumindest der einfachsten Vorgehensweise, die alle Regeln beispielsweise über eine *HashMap* zugänglich macht.
- Alternativ wäre es denkbar, einen endlichen Automaten für ein Regelsystem zu erzeugen. Dann reduziert sich der Aufwand auf $O(n)$. Allerdings wäre der Automat ziemlich umfangreich.
- Eine weitere Alternative sind Tries.

- Der Name Trie leitet sich ab von *re-trie-val*. Die Datenstruktur wurde zuerst von Briandais (1959) und Fredkin (1960) beschrieben. Der Begriff geht auf Fredkin zurück.
- Anders als bei assoziativen Arrays (Maps) wird davon ausgegangen, dass Schlüssel sich definieren als eine Sequenz von Zeichen aus einem endlichen Alphabet.
- Entsprechend erfolgt der Zugriff zeichenweise. An jedem erreichten Punkt können wir dabei auf eine Regel stoßen, die Suche aber noch fortsetzen. Die Suche wird abgebrochen, sobald eine Zeichenfolge als Anfang eines Schlüssels nicht vorkommen kann.
- Tries können durch Baumstrukturen repräsentiert werden. Baumknoten können auf Regeln verweisen und die Zahl der möglichen Unterbäume zu einem Baumknoten ist nur durch den Umfang des Alphabets begrenzt.

Lesende Schnittstelle eines Tries

TrieReader.java

```
package de.uniulm.mathematik.tries;

public interface TrieReader<TrieInfo> {

   public interface TriePointer<TrieInfo> {
      public TriePointer<TrieInfo> descend(int code);
      public TrieInfo getInfo();
   }

   public TriePointer<TrieInfo> getRoot();
   public int getNumberOfEntries();
   public int getNumberOfNodes();
}
```

- Der Typparameter *TrieInfo* repräsentiert hier den Typ der durch die Datenstruktur auffindbaren Objekte (hier: Trennungsregeln).
- Eine Suche beginnt mit dem Aufruf von *getRoot*, das einen Zeiger (*TriePointer*) in die Datenstruktur liefert. An jedem Punkt lässt sich mit *getInfo* abfragen, ob ein Objekt zu finden ist und ein weiterer Abstieg ist mit *descend* möglich, das *null* zurückliefert, wenn die Zeichenfolge als Anfang eines Schlüssels nicht vorkommen kann.

HashedTrie.java

```
public class TrieNode<TrieInfo> implements TrieIterator<TrieInfo> {
   protected TrieInfo info;
   protected HashMap<Integer, TrieNode<TrieInfo> > subnodes;
   protected TrieNode() {
      info = null;
      subnodes = new HashMap<Integer, TrieNode<TrieInfo> >();
   }
   public TrieNode<TrieInfo> descend(int code) {
      return subnodes.get(new Integer(code));
   }
   public TrieInfo getInfo() { return info; }
   public Iterator<Integer> iterator() {
      return subnodes.keySet().iterator();
   }
}
```

- Die einem Baumknoten untergeordneten Unterbäume werden hier mit einer *HashMap* verwaltet.
- Franklin Liang hat für die TEX-Implementierung eine komprimierte Datenstruktur (*packed tries*) entwickelt, die noch effizienter ist und insbesondere nur sehr wenig Speicherplatz benötigt. (Allerdings lässt sich das in Java nicht in gleicher Weise umsetzen.)

HashedTrie.java

```
public class HashedTrie<TrieInfo>
   implements TrieReader<TrieInfo>, TrieConstructor<TrieInfo> {

   // public class TrieNode<TrieInfo> ...

   private TrieNode<TrieInfo> root;
   private int numberOfNodes; private int numberOfEntries;

   public HashedTrie() {
      root = new TrieNode<TrieInfo>();
      numberOfNodes = 0; numberOfEntries = 0;
   }

   public TrieNode<TrieInfo> getRoot() {
      return root;
   }

   // public void insert(String word, TrieInfo info) ...

   public int getNumberOfEntries() { return numberOfEntries; }
   public int getNumberOfNodes() { return numberOfNodes; }
}
```

HashedTrie.java

```
public void insert(String word, TrieInfo info) {
   assert info != null;
   TrieNode<TrieInfo> node = root;
   for (int index = 0; index < word.length();
         index = word.offsetByCodePoints(index, 1)) {
      int ch = word.codePointAt(index);
      Integer key = new Integer(ch);
      TrieNode<TrieInfo> subnode = node.subnodes.get(key);
      if (subnode == null) {
         subnode = new TrieNode<TrieInfo>();
         node.subnodes.put(key, subnode);
         ++numberOfNodes;
      }
      node = subnode;
   }
   node.info = info;
   ++numberOfEntries;
}
```

HyphenationPoint.java

```
public interface HyphenationPoint {
   public int getPosition();
   public int getValue();
}
```

- Die Trennungsposition ist relativ zum Beginn der Trennungsregel und *getValue* liefert den zugehörigen Rang.

HyphenationRule.java

```
public interface HyphenationRule
   extends Iterable<HyphenationPoint> {
}
```

- Eine Trennungsregel besteht dann nur noch aus einer Folge von Trennungspositionen und zugehörigen Rängen.
- Die Buchstabenfolge der Regel gehört zum Schlüssel und wird in dieser Datenstruktur nicht wiederholt.

Hyphenator.java

```
public int[] hyphenate(String s) {
   // extract array of codepoints
   // where upper case letters are mapped to lower case
   String word = "." + s + ".";
   int len = word.codePointCount(0, word.length());
   int[] codepoints = new int[len];
   for (int i = 0; i < word.length(); i = word.offsetByCodePoints(i, 1)) {
      codepoints[i] = word.codePointAt(i);
      if (Character.isUpperCase(codepoints[i])) {
         codepoints[i] = Character.toLowerCase(codepoints[i]);
      }
   }
   // find matching hyphenation patterns and apply them to value[] ...
   // create array with hyphenation positions ...
}
```

- Ein zu trennendes Wort wird mit zwei Punkten ergänzt, damit auch die Trennregeln berücksichtigt werden, die sich nur auf das Anfang oder das Ende eines Worts beziehen.
- Das Eingabealphabet wird standardisiert, indem hier Groß- auf Kleinbuchstaben abgebildet werden. Es ist hier auch mehr denkbar wie etwa die Wegnahme von Akzenten.

Hyphenator.java

```
// find matching hyphenation patterns and apply them to value[]
int[] value = new int[len+1];
for (int start = 0; start+leftmin < len; ++start) {
   int pos = start;
   for (TrieReader.TriePointer<HyphenationEntry>
         ptr = patterns.getRoot();
         ptr != null;
         ptr = pos < len? ptr.descend(codepoints[pos++]): null) {
      HyphenationEntry rule = ptr.getInfo();
      if (rule != null) {
         for (HyphenationPoint point: rule) {
            int index = point.getPosition() + start - 1;
            if (index >= leftmin && index + rightmin < len-1) {
               int val = point.getValue();
               if (val > value[index]) {
                  value[index] = val;
               }
            }
         }
      }
   }
}
```

Hyphenator.java

```
// create array with hyphenation positions
int count = 0;
for (int i = 0; i <= len; ++i) {
   if (value[i] % 2 == 1) {
      ++count;
   }
}
if (count == 0) {
   return null;
}
int[] result = new int[count]; int index = 0;
for (int i = 0; i <= len; ++i) {
   if (value[i] % 2 == 1) {
      result[index++] = i;
   }
}
return result;
```

- Der Trennungsalgorithmus von Franklin Liang ist bis heute das wichtigste Trennungsverfahren, das auch außerhalb von TEX bei anderen Programmen wie etwa *troff* oder *QuarkXPress*.
- Verschiedene Verbesserungen wurden jedoch bzw. bereits umgesetzt (in Ω_2):
  - Trennungsregeln sollten Gewichtungen vorsehen, damit gute Trennstellen von weniger guten unterschieden werden können. In der deutschen Sprache sollte eher die Trennung „Trennungs-regel“ vor „Trennungsre-gel“ bevorzugt werden. (Dies könnte in den Strafwert der entsprechenden Sollbruchstelle einfließen.)
  - Bei gleich guten Trennungsstellen sollten die Trennungsstellen in der Mitte eines Worts bevorzugt werden.
  - Mehrdeutigkeiten sollten nicht getrennt werden: „Wach-stube“ vs. „Wachs-tube“.