```fsharp
(* .net core
System.Drawing.Common 5.0.2
Provides access to GDI+ graphics functionality.
https://www.nuget.org/packages/System.Drawing.Common/*)
namespace ExpandablePrinter

module FsharpPrinting =
    open System
    open System.Xml
    open System.Drawing.Printing
    open System.Drawing
    open System.Globalization
    let private Document = new PrintDocument()
    let private printerSettings = new PrinterSettings()

    /// Converts a Rectangle to a RectangleF
    let RectangleToRectangleF(r: Rectangle) = RectangleF(float32(r.X), float32
      (r.Y), float32(r.Width), float32(r.Height))

    /// Converts a RectangleF to a Rectangle
    let RectangleFtoRectangle(r: RectangleF) = Rectangle(int(r.X), int(r.Y),
      int(r.Width), int(r.Height))

    /// Size of the four margins
    let Margins = Document.PrinterSettings.DefaultPageSettings.Margins

    /// RectangleF giving the size of the paper in most cases equal to
      PageBounds
    let Bounds = RectangleToRectangleF
      (Document.PrinterSettings.DefaultPageSettings.Bounds)

    /// RectangleF giving the bounds of the page including margins.
    let PageBounds = RectangleF(Bounds.Left, Bounds.Top, Bounds.Right -
      Bounds.Left, Bounds.Bottom - Bounds.Top)

    /// RectangleF giving the bounds of the page excluding margins equals the
      size of the page container.
    let PageContainer = RectangleF(float32(Margins.Left), float32
      (Margins.Top), Bounds.Right - float32(Margins.Left + Margins.Right),
      Bounds.Bottom  - float32(Margins.Top + Margins.Bottom))

    /// <summary>Read the string value from n's attribute with the name
      "name".
    /// If "name" is not defined Some(Value) is returns or if defaultValue is
      None an exception is thrown</summary>
    /// <exception cref="Attribure is missing">If "name" is not defined and
      defaultValue is None </exception>
    /// <param name="n">XmlNode</param>
    /// <param name="name">Attribute name</param>
    /// <param name="defaultValue">Option type. None is used when a value has
      to be specified.</param>
    let readString(n: XmlNode, name, defaultValue) =
        let value = (n  :?> XmlElement).GetAttribute(name)
        match defaultValue with
        | None when value = ""      -> failwith($"Attribute {name} in tag <
          {(n  :?> XmlElement).Name}> missing")
```

```
44          | None                      -> value
45          | Some(v) when value = ""   -> v
46          | Some(_)                   -> value
47
48      /// <summary> Visual Basic and Csharp version of readString
49      /// Read the string value from n's attribute with the name "name".
50      /// If "name" is not defined defaultValue is returns or if defaultValue is
           (null or Nothing) an exception is thrown</summary>
51      /// <exception cref="Attribure is missing">If "name" is not defined and
          defaultValue is (null or Nothing) </exception>
52      /// <param name="n">XmlNode</param>
53      /// <param name="name">Attribute name</param>
54      /// <param name="defaultValue">Single. (null or Nothing) is used when a
          value has to be specified.</param>
55      let readStringVisualBasicCsharp(n: XmlNode, name, defaultValue) =
56          let value = (n  :?> XmlElement).GetAttribute(name)
57          match defaultValue with
58          | null when value = ""      -> failwith($"Attribute {name} in tag <
              {(n  :?> XmlElement).Name}> missing")
59          | null                      -> value
60          | v when value = ""         -> v
61          | _                         -> value
62
63      /// Read the string value from string element n
64      let readText(n: XmlNode) = (n  :?> XmlElement).InnerText
65
66      /// <summary>Read the float32 (Single) value from n's attribute with the
          name "name".
67      /// If "name" is not defined Some(Value) is returns or if defaultValue is
          None an exception is thrown</summary>
68      /// <exception cref="Attribure is missing">If "name" is not defined and
          defaultValue is None </exception>
69      /// <exception cref="Not a float value">If value is not a float number</
          exception>
70      /// <param name="n">XmlNode</param>
71      /// <param name="name">Attribute name</param>
72      /// <param name="defaultValue">Option type. None is used when a value has
          to be specified.</param>
73      let readFloat(n: XmlNode, name, defaultValue) =
74          let value = (n  :?> XmlElement).GetAttribute(name)
75          let i = ref 0.0f
76          match (defaultValue, Single.TryParse(value,
              System.Globalization.NumberStyles.Float,
              System.Globalization.CultureInfo.InvariantCulture, i)) with
77          | (_, true)                 -> !i
78          | (_, false) when value <> ""   -> failwith($"Attribute {name} in tag
              <{(n  :?> XmlElement).Name}> not a float number")
79          | (None, false)             -> failwith($"Attribute {name} in tag
              <{(n  :?> XmlElement).Name}> not a float number")
80          | (Some(v), false)          -> v
81
82      /// <summary> Visual Basic and Csharp version of readFloat
83      /// Read the float32 (Single) value from n's attribute with the name
          "name".
84      /// If "name" is not defined defaultValue is returns or if defaultValue is
           (null or Nothing) an exception is thrown</summary>
```

```
85      /// <exception cref="Attribure is missing">If "name" is not defined and
          defaultValue is (null or Nothing) </exception>
86      /// <exception cref="Not a float value">If value is not a float number</
          exception>
87      /// <param name="n">XmlNode</param>
88      /// <param name="name">Attribute name</param>
89      /// <param name="defaultValue">(null or Nothing) is used when a value has
          to be specified.</param>
90      let readFloatVisualBasicCsharp(n: XmlNode, name, defaultValue:
          Nullable<float32>) =
91          let value = (n  :?> XmlElement).GetAttribute(name)
92          let i = ref 0.0f
93          match (defaultValue, Single.TryParse(value,
              System.Globalization.NumberStyles.Float,
              System.Globalization.CultureInfo.InvariantCulture, i)) with
94          | (_, true)                     -> !i
95          | (_, false) when value <> ""   -> failwith($"Attribute {name} in tag
              <{(n  :?> XmlElement).Name}> not a float number")
96          | (v, false) when v = System.Nullable() -> failwith($"Attribute {name}
               in tag <{(n  :?> XmlElement).Name}> not a float number")
97          | (v, false)                    -> v.Value
98
99      ///  Read the Font from n's attribute with the name "Font", "Size" and
          "Style".
100     let readFont(n: XmlNode, f: Font) = new Font(readString(n, "Font", Some
          (f.Name)), readFloat(n, "Size", Some(f.Size)), Enum.Parse
          (typeof<FontStyle>, readString(n, "Style", Some("Regular"))) :?>
          FontStyle)
101
102     ///  Read the PointF from n's attribute with the names "Tab" and
          "VerticalTab".
103     let readTab(n: XmlNode, p: PointF) = PointF(readFloat(n, "Tab", Some
          (p.X)), readFloat(n, "VerticalTab", Some(p.Y)))
104
105     ///  Read the Color from n's attribute with the name "Colour".
106     let readColour(n: XmlNode) = Color.FromName(readString(n, "Colour", Some
          ("Black")))
107
108     ///  Read the PointF from n's attribute with the names "X" and "Y".
109     let read1PointF(n: XmlNode, xOffset, yOffset) = PointF(readFloat(n, "X",
          None) + xOffset, readFloat(n, "Y", None) + yOffset)
110
111     ///  Read a second PointF from n's attribute with the names "X2" and "Y2".
112     let read2PointF(n: XmlNode, xOffset, yOffset) = PointF(readFloat(n, "X2",
          None) + xOffset, readFloat(n, "Y2", None) + yOffset)
113
114     ///  Read a readRRectangleF from n's attribute with the names "X", "Y",
          "Width" and "Height".
115     let readRRectangleF(n: XmlNode, xOffset, yOffset) =
116         let p = read1PointF(n, 0.0f, 0.0f)
117         RectangleF(p.X + xOffset, p.Y + yOffset, readFloat(n, "Width", None),
              readFloat(n, "Height", None) )
118
119     ///  Read a font size from n's attribute with the name "Size" and return
          Font f with this new size.
120     let setFontSize(n : XmlNode, f : Font) = new Font(f.Name, readFloat(n,
```

```
          "Size", Some(10.0f)))
121
122       let offsetRectangleF(r: RectangleF, x, y, w, h) = RectangleF(r.X + x, r.Y
            + y, r.Width + w, r.Height + h)
123
124       let offsetRectangle(r: RectangleF, x, y, w, h) = Rectangle(int(r.X + x),
            int(r.Y + y), int(r.Width + w), int(r.Height + h))
125
126       let private PahragraphPrint(g: Graphics, container : RectangleF, p :
            PointF, f: Font, flag: StringFormatFlags, n: XmlNode) =
127           if n.Name <> "Format" then failwith($"Wrong tag <{n.Name}> after
              <Line>, <FreeLine> and <Paragraphs> has to be <Format>")
128           let fStyle = Enum.Parse(typeof<FontStyle>, readString(n, "Style", Some
              ("Regular"))) :?> FontStyle
129           use font = readFont(n, f)
130           let tabs = readTab(n, p)
131           let drawRect = offsetRectangleF(container, tabs.X, tabs.Y, -tabs.X, -
              tabs.Y)
132           let remainingSpace = SizeF(container.Width - tabs.X, container.Height
              - tabs.Y)
133           let paragraphs = readText(n)
134           let sizeParagraphs = g.MeasureString(paragraphs, font, remainingSpace,
                new StringFormat(flag))
135           if remainingSpace.Width < sizeParagraphs.Width ||
              remainingSpace.Height < sizeParagraphs.Height then failwith($"Text
              in <Format> out of container")
136           g.DrawString(paragraphs, font, new SolidBrush(readColour(n)),
              drawRect, new StringFormat(flag))
137           sizeParagraphs
138
139       let rec private runContainers(g: Graphics, printFont: Font, n:XmlNode,
            functions: Object, container: RectangleF) =
140           let mutable yCurrent = 0.0f
141           let graphicalElements = n.ChildNodes
142           for e in graphicalElements do
143               match e.Name with
144               | "Container"   ->  let r = readRRectangleF(e, container.X,
                  container.Y)
145                                   let eWidth = readFloat(e, "Draw", Some(0.0f))
146                                   if eWidth  > 0.0f then g.DrawRectangle( new
                        Pen(readColour(e), eWidth),
147                                                                           offsetRectangle(r,
                         eWidth / 2.0f, eWidth / 2.0f, -eWidth, -eWidth))
148                                   let r2 = if eWidth  > 0.0f then
                        offsetRectangleF(r, eWidth, eWidth, -eWidth * 2.0f, -
                        eWidth * 2.0f) else r
149                                   runContainers(g, printFont, e, functions, r2)
150               | "Line"        ->  let mutable xCurrent = 0.0f
151                                   let mutable usedHeight = 0.0f
152                                   for item in e.ChildNodes do
153                                       let usedSize = PahragraphPrint(g,
                        container, PointF(xCurrent, yCurrent), printFont,
                        StringFormatFlags.NoWrap, item)
154                                       xCurrent <- xCurrent + usedSize.Width
155                                       usedHeight <- float32(Math.Max(usedHeight,
                         usedSize.Height))
```

```
156                                 yCurrent <- yCurrent + usedHeight
157             | "FreeLine"    ->  let mutable xCurrent = 0.0f
158                                 for item in e.ChildNodes do
159                                     xCurrent <- xCurrent + PahragraphPrint(g,
                        Bounds, PointF(xCurrent, 0.0f), printFont,
                        StringFormatFlags.NoWrap, item).Width
160             | "Paragraphs"  ->  yCurrent <- yCurrent + PahragraphPrint(g,
                  container, PointF(0.0f, yCurrent), setFontSize(e, printFont),
161                                 StringFormatFlags.NoClip,
                        e.FirstChild).Height
162             | "Point"       ->  let width = readFloat(e, "Width", Some(1.0f))
163                                 let p = read1PointF(e, container.X - width /
                        2.0f, container.Y - width / 2.0f)
164                                 g.FillEllipse(new SolidBrush(readColour(e)),
                        RectangleF(p.X , p.Y, width, width))
165             | "SolidLine"   ->  g.DrawLine(new Pen(readColour(e), readFloat(e,
                   "Width", Some(2.0f))), read1PointF(e, container.X,
                  container.Y), read2PointF(e, container.X,  container.Y))
166             | "Function"    ->  if  isNull  functions then failwith($"Tag name
                   <Functions> detected, but functions is (null or Nothing)>")
167                                 let qqq = functions.GetType().GetMethods()
168                                 let MetodInf = functions.GetType().GetMethod
                        (readString(e, "Name", None))
169                                 MetodInf.Invoke(functions, [|g; container;
                        e.Attributes|]) |> ignore
170             | "FunctionXML" ->  if  isNull  functions then failwith($"Tag name
                   <Functions> detected, but functions is(null or Nothing)>")
171                                 let MetodInf = functions.GetType().GetMethod
                        (readString(e, "Name", None))
172                                 MetodInf.Invoke(functions, [|g; container;
                        e.InnerXml|]) |> ignore
173             | "#comment"     -> ()
174             | _              -> failwith($"Ilegal tag name in <container> <
                  {e.Name}>")
175
176     let mutable paragraphCount = 0
177     let mutable paragraphs = Array.empty
178     let printPages(g: Graphics, font: Font, container: RectangleF, text:
          string)=
179         if paragraphCount = 0 then paragraphs <- text.Replace("\n", "").Split
              ([|'\r'|], StringSplitOptions.None)
180         let textFit: string = ""
181         let rec addLine (s: string, l : string, index : int) =
182             let textArea = SizeF(container.Width, Single.MaxValue)
183             let sPlus = s + l
184             let z = g.MeasureString(sPlus, font, textArea,  new StringFormat
                  (StringFormatFlags.NoClip))
185             match (z.Height > container.Height, index < paragraphs.Length - 1)
                   with
186             | (false, true)     -> addLine(sPlus + "\r\n", paragraphs.[index +
                   1], index + 1)
187             | (false, false)    -> (index, sPlus)
188             | (true, _)         -> (index - 1, s)
189         let index, s = addLine(textFit, paragraphs.[paragraphCount],
              paragraphCount)
190         paragraphCount <- index + 1
```

```
191         g.DrawString(s, font, new SolidBrush(Color.Black), container, new
              StringFormat(StringFormatFlags.NoClip))
192         paragraphCount < paragraphs.Length
193
194     let mutable private pageCount = 0
195
196     let private documentPrintPage2 (xmlDoc: XmlDocument, functions: Object)
          (sender: Object)  (ev: PrintPageEventArgs) =
197         let leftMargin = ev.MarginBounds.Left |> float32
198         let rightMargin = ev.MarginBounds.Right |> float32
199         let totalWidth = rightMargin - leftMargin |> float32
200         let topMargin = ev.MarginBounds.Top |> float32
201         let bottomMargin = ev.MarginBounds.Bottom |> float32
202         let totalHight = bottomMargin - topMargin |> float32
203
204         let print = xmlDoc.FirstChild.NextSibling
205         if print.Name <> "Print" then failwith("Root tag has to be <Print>")
206         let printFont = readFont(print, new Font("Areal", 10.0f))
207         let pages = print.ChildNodes
208         match pages.[pageCount].Name with
209         | "Page"            -> runContainers(ev.Graphics, printFont, pages.
              [pageCount], functions, RectangleF(leftMargin, topMargin,
              totalWidth, totalHight))
210                                pageCount <- pageCount + 1
211                                if pageCount < pages.Count then ev.HasMorePages
                       <- true
212                                else ev.HasMorePages <- false
213         | "MultiplePages"   -> let f = readFont(pages.[pageCount].FirstChild,
              printFont)
214                                let text = readText(pages.
                         [pageCount].FirstChild)
215                                let ended = not(printPages(ev.Graphics, f,
                        RectangleF(leftMargin, topMargin, totalWidth, totalHight),
                         text))
216                                if ended then paragraphCount <- 0; pageCount <-
                         pageCount + 1
217                                ev.HasMorePages <- not ended || pageCount <
                        pages.Count
218         | _                 -> failwith("After root tag <Print> the children
              tags has to be <Page> og <MultiplePages>")
219
220     let private printing2(source: string, functions: Object) =
221         Document.PrinterSettings <- printerSettings
222         let XMLdoc = new XmlDocument()
223         XMLdoc.LoadXml(source)
224         let documentPrintPage = documentPrintPage2(XMLdoc, functions)
225         let printPageEventHandler = new PrintPageEventHandler
              (documentPrintPage)
226         Document.PrintPage.AddHandler(printPageEventHandler)
227         Document.Print()
228         Document.PrintPage.RemoveHandler(printPageEventHandler)
229         pageCount <- 0; paragraphCount <- 0
230
231     /// <summary>Start printing the XML document source to the file
232     /// functions can be (null or Nothing) if no special printing functions is
          used
```

```
233     /// functions is a referance to an object O with the special printing
          functions called by either XML tag
234     /// &lt;Functions Name = "O method name" attr1 = "Value1" attr2 =
          "Value2" ... attrN = "ValueN"/&gt; or
235     /// &lt;Function2 Name = "O method name" /&gt;
236     /// inner XML tags
237     /// &lt;/Function2&gt;
238     /// </summary>
239     /// <exception cref="Wrong tag after &lt;Line&gt; or &lt;Paragraphs&gt;
          has to be &lt;Format&gt;">Tag after &lt;Paragraphs&gt; has to be
          &lt;Format&gt;&lt;Format&gt;</exception>
240     /// <exception cref="Text in &lt;Format&gt; out of container">Text starts
          outside the container</exception>
241     /// <exception cref="Tag name &lt;Functions&gt; detected, but functions is
           (null or Nothing)>">Function is not existing</exception>
242     /// <exception cref="Ilegal tag name in &lt;container&gt;">Unknown tag
          name in container</exception>
243     /// <exception cref="Root tag has to be &lt;Print&gt;">Wrong root tag</
          exception>
244     /// <exception cref="After root tag &lt;Print&gt; the children tags has to
           be &lt;Page&gt;">description</exception>
245     /// <param name = "source">XML document defining the print</param>
246     /// <param name = "functions">object O with the special printing
          functions</param>
247     /// <param name = "file">full path to *.pdf output file</param>
248     let printingPDF(source: string, functions: Object, file) =
249         printerSettings.PrinterName <- "Microsoft Print to PDF"
250         printerSettings.PrintToFile <- true
251         Document.PrinterSettings <- printerSettings
252         printerSettings.PrintFileName <- file
253         printing2(source, functions)
254
255     /// <summary>Start printing the XML document source
256     /// functions can be (null or Nothing) if no special printing functions is
           used
257     /// functions is a referance to an object O with the special printing
          functions called by either XML tag
258     /// &lt;Functions Name = "O method name" attr1 = "Value1" attr2 =
          "Value2" ... attrN = "ValueN"/&gt; or
259     /// &lt;Function2 Name = "O method name" /&gt;
260     /// inner XML tags
261     /// &lt;/Function2&gt;
262     /// </summary>
263     /// <exception cref="Wrong tag after &lt;Line&gt; or &lt;Paragraphs&gt;
          has to be &lt;Format&gt;">Tag after &lt;Paragraphs&gt; has to be
          &lt;Format&gt;&lt;Format&gt;</exception>
264     /// <exception cref="Text in &lt;Format&gt; out of container">Text starts
          outside the container</exception>
265     /// <exception cref="Tag name &lt;Functions&gt; detected, but functions is
           (null or Nothing)>">Function is not existing</exception>
266     /// <exception cref="Ilegal tag name in &lt;container&gt;">Unknown tag
          name in container</exception>
267     /// <exception cref="Root tag has to be &lt;Print&gt;">Wrong root tag</
          exception>
268     /// <exception cref="After root tag &lt;Print&gt; the children tags has to
           be &lt;Page&gt;">description</exception>
```

```
269     /// <param name = "source">XML document defining the print</param>
270     /// <param name = "functions">object O with the special printing
          functions</param>
271     /// <param name = "printerName">Selected printers name sting</param>
272     let printingPaper(source: string, functions: Object, printerName) =
273         printerSettings.PrinterName <- printerName
274         printing2(source, functions)
275
276     let split(text: string) =
277         let mutable textFitContainers: string[] = Array.empty
278         let containers: RectangleF[] = Array.create 19 (RectangleF(0.0f, 0.0f,
              PageContainer.Width, PageContainer.Height))
279         let f = new Font("Arial", 8.0f)
280         let splitStringToFit (g: Graphics, font: Font, containers: RectangleF
            [], text: string) =
281             let paragraphs = text.Split([|'\r'|], StringSplitOptions.None)
282                                 |> Array.map (fun (item: string) ->
                        item.Replace("\n", ""))
283             let textFit: string[] = Array.zeroCreate containers.Length
284             let rec addLine (s: string, l : string, index : int, contanier :
                int) =
285                 let textArea = SizeF(containers.[contanier].Width,
                      Single.MaxValue)
286                 let sPlus = s + l
287                 let z = g.MeasureString(sPlus, font, textArea,  new
                      StringFormat(StringFormatFlags.NoClip))
288                 match (z.Height > containers.[contanier].Height, index <
                      paragraphs.Length - 1) with
289                 | (false, true)     -> addLine(sPlus + "\r\n", paragraphs.
                      [index + 1], index + 1, contanier)
290                 | (false, false)    -> (index, sPlus)
291                 | (true, _)         -> (index - 1, s)
292             let mutable j = 0
293             for i in 0 .. containers.Length - 1 do
294                 if j  < paragraphs.Length - 1 then
295                     let index, s = addLine(textFit.[i], paragraphs.[j], j, i)
296                     textFit.[i] <- s
297                     j <- index + 1
298             textFit
299         let documentPrintPage (sender: Object)  (ev: PrintPageEventArgs) =
300             textFitContainers <- splitStringToFit(ev.Graphics, f, containers,
                text)
301             ev.HasMorePages <- false
302         // Find ev.Graphics to calculate text size
303         let Document = new PrintDocument()
304         let printerSettings = new PrinterSettings()
305         printerSettings.PrinterName <- "Microsoft Print to PDF"
306         printerSettings.PrintToFile <- true
307         printerSettings.PrintFileName <- __SOURCE_DIRECTORY__ + @"\test.pdf"
308         Document.PrinterSettings <- printerSettings
309         let printPageEventHandler = new PrintPageEventHandler
              (documentPrintPage)
310         Document.PrintPage.AddHandler(printPageEventHandler)
311         Document.Print()
312         Document.PrintPage.RemoveHandler(printPageEventHandler)
313         textFitContainers
```