```
(*.Net Framework
Assemblies System.Drawing*)
namespace ExpandablePrinter

module FsharpPrinting =
    open System
    open System.Xml
    open System.Drawing.Printing
    open System.Drawing
    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")
        | None                      -> value
        | Some(v) when value = ""   -> v
        | Some(_)                   -> value
```

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

```
        exception>
84      /// <param name="n">XmlNode</param>
85      /// <param name="name">Attribute name</param>
86      /// <param name="defaultValue">(null or Nothing) is used when a value has
          to be specified.</param>
87      let readFloatVisualBasicCsharp(n: XmlNode, name, defaultValue:
          Nullable<float32>) =
88          let value = (n  :?> XmlElement).GetAttribute(name)
89          let i = ref 0.0f
90          match (defaultValue, Single.TryParse(value,
              System.Globalization.NumberStyles.Float,
              System.Globalization.CultureInfo.InvariantCulture, i)) with
91          | (_, true)                     -> !i
92          | (_, false) when value <> ""   -> failwith($"Attribute {name} in tag
              <{(n  :?> XmlElement).Name}> not a float number")
93          | (v, false) when v = System.Nullable() -> failwith($"Attribute {name}
               in tag <{(n  :?> XmlElement).Name}> not a float number")
94          | (v, false)                    -> v.Value
95
96      ///  Read the Font from n's attribute with the name "Font", "Size" and
          "Style".
97      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)
98
99      ///  Read the PointF from n's attribute with the names "Tab" and
          "VerticalTab".
100     let readTab(n: XmlNode, p: PointF) = PointF(readFloat(n, "Tab", Some
          (p.X)), readFloat(n, "VerticalTab", Some(p.Y)))
101
102     ///  Read the Color from n's attribute with the name "Colour".
103     let readColour(n: XmlNode) = Color.FromName(readString(n, "Colour", Some
          ("Black")))
104
105     ///  Read the PointF from n's attribute with the names "X" and "Y".
106     let read1PointF(n: XmlNode, xOffset, yOffset) = PointF(readFloat(n, "X",
          None) + xOffset, readFloat(n, "Y", None) + yOffset)
107
108     ///  Read a second PointF from n's attribute with the names "X2" and "Y2".
109     let read2PointF(n: XmlNode, xOffset, yOffset) = PointF(readFloat(n, "X2",
          None) + xOffset, readFloat(n, "Y2", None) + yOffset)
110
111     ///  Read a readRRectangleF from n's attribute with the names "X", "Y",
          "Width" and "Height".
112     let readRRectangleF(n: XmlNode, xOffset, yOffset) =
113         let p = read1PointF(n, 0.0f, 0.0f)
114         RectangleF(p.X + xOffset, p.Y + yOffset, readFloat(n, "Width", None),
              readFloat(n, "Height", None) )
115
116     ///  Read a font size from n's attribute with the name "Size" and return
          Font f with this new size.
117     let setFontSize(n : XmlNode, f : Font) = new Font(f.Name, readFloat(n,
          "Size", Some(10.0f)))
118
119     let offsetRectangleF(r: RectangleF, x, y, w, h) = RectangleF(r.X + x, r.Y
```

```
        + y, r.Width + w, r.Height + h)
120
121     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))
122
123     let private PahragraphPrint(g: Graphics, container : RectangleF, p :
          PointF, f: Font, flag: StringFormatFlags, n: XmlNode) =
124         if n.Name <> "Format" then failwith($"Wrong tag <{n.Name}> after
            <Line>, <FreeLine> and <Paragraphs> has to be <Format>")
125         let fStyle = Enum.Parse(typeof<FontStyle>, readString(n, "Style", Some
            ("Regular"))) :?> FontStyle
126         use font = readFont(n, f)
127         let tabs = readTab(n, p)
128         let drawRect = offsetRectangleF(container, tabs.X, tabs.Y, -tabs.X, -
            tabs.Y)
129         let remainingSpace = SizeF(container.Width - tabs.X, container.Height
            - tabs.Y)
130         let paragraphs = readText(n)
131         let sizeParagraphs = g.MeasureString(paragraphs, font, remainingSpace,
             new StringFormat(flag))
132         if remainingSpace.Width < sizeParagraphs.Width ||
            remainingSpace.Height < sizeParagraphs.Height then failwith($"Text
            in <Format> out of container")
133         g.DrawString(paragraphs, font, new SolidBrush(readColour(n)),
            drawRect, new StringFormat(flag))
134         sizeParagraphs
135
136     let rec private runContainers(g: Graphics, printFont: Font, n:XmlNode,
          functions: Object, container: RectangleF) =
137         let mutable yCurrent = 0.0f
138         let graphicalElements = n.ChildNodes
139         for e in graphicalElements do
140             match e.Name with
141             | "Container"   ->  let r = readRRectangleF(e, container.X,
                container.Y)
142                                 let eWidth = readFloat(e, "Draw", Some(0.0f))
143                                 if eWidth  > 0.0f then g.DrawRectangle( new
                        Pen(readColour(e), eWidth),
144                                                                         offsetRectangle(r,
                         eWidth / 2.0f, eWidth / 2.0f, -eWidth, -eWidth))
145                                 let r2 = if eWidth  > 0.0f then
                        offsetRectangleF(r, eWidth, eWidth, -eWidth * 2.0f, -
                        eWidth * 2.0f) else r
146                                 runContainers(g, printFont, e, functions, r2)
147             | "Line"        ->  let mutable xCurrent = 0.0f
148                                 let mutable usedHeight = 0.0f
149                                 for item in e.ChildNodes do
150                                     let usedSize = PahragraphPrint(g,
                        container, PointF(xCurrent, yCurrent), printFont,
                        StringFormatFlags.NoWrap, item)
151                                     xCurrent <- xCurrent + usedSize.Width
152                                     usedHeight <- float32(Math.Max(usedHeight,
                         usedSize.Height))
153                                 yCurrent <- yCurrent + usedHeight
154             | "FreeLine"    ->  let mutable xCurrent = 0.0f
155                                 for item in e.ChildNodes do
```

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

```
190
191     let mutable private pageCount = 0
192
193     let private documentPrintPage2 (xmlDoc: XmlDocument, functions: Object)
          (sender: Object)  (ev: PrintPageEventArgs) =
194         let leftMargin = ev.MarginBounds.Left |> float32
195         let rightMargin = ev.MarginBounds.Right |> float32
196         let totalWidth = rightMargin - leftMargin |> float32
197         let topMargin = ev.MarginBounds.Top |> float32
198         let bottomMargin = ev.MarginBounds.Bottom |> float32
199         let totalHight = bottomMargin - topMargin |> float32
200
201         let print = xmlDoc.FirstChild.NextSibling
202         if print.Name <> "Print" then failwith("Root tag has to be <Print>")
203         let printFont = readFont(print, new Font("Areal", 10.0f))
204         let pages = print.ChildNodes
205         match pages.[pageCount].Name with
206         | "Page"            -> runContainers(ev.Graphics, printFont, pages.
            [pageCount], functions, RectangleF(leftMargin, topMargin,
            totalWidth, totalHight))
207                                pageCount <- pageCount + 1
208                                if pageCount < pages.Count then ev.HasMorePages
                     <- true
209                                else ev.HasMorePages <- false
210         | "MultiplePages"   -> let f = readFont(pages.[pageCount].FirstChild,
            printFont)
211                                let text = readText(pages.
                           [pageCount].FirstChild)
212                                let ended = not(printPages(ev.Graphics, f,
                           RectangleF(leftMargin, topMargin, totalWidth, totalHight),
                            text))
213                                if ended then paragraphCount <- 0; pageCount <-
                            pageCount + 1
214                                ev.HasMorePages <- not ended || pageCount <
                           pages.Count
215         | _                 -> failwith("After root tag <Print> the children
            tags has to be <Page> og <MultiplePages>")
216
217     let private printing2(source: string, functions: Object) =
218         Document.PrinterSettings <- printerSettings
219         let XMLdoc = new XmlDocument()
220         XMLdoc.LoadXml(source)
221         let documentPrintPage = documentPrintPage2(XMLdoc, functions)
222         let printPageEventHandler = new PrintPageEventHandler
              (documentPrintPage)
223         Document.PrintPage.AddHandler(printPageEventHandler)
224         Document.Print()
225         Document.PrintPage.RemoveHandler(printPageEventHandler)
226         pageCount <- 0; paragraphCount <- 0
227
228     /// <summary>Start printing the XML document source to the file
229     /// functions can be (null or Nothing) if no special printing functions is
            used
230     /// functions is a referance to an object O with the special printing
          functions called by either XML tag
231     /// &lt;Functions Name = "O method name" attr1 = "Value1" attr2 =
```

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

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