CodeZine
デベロッパーの成長と課題解決に貢献するメディア

.NET向けExcel・PDF操作ライブラリ「DioDocs」を使ったアプリケーション構築～究極に軽量な帳票生成環境を手に入れる～

開発ツール

中村 充志（リコージャパン株式会社）[著] 2019/02/01 11:00

グレープシティ社からリリースされた.NET向けのExcelファイルおよびPDFを操作するライブラリ、「DioDocs（ディオドック） for Excel」および「DioDocs for PDF」。前回の記事ではその魅力と簡単な利用方法についてお伝えしました。今回はDioDocsを使って、汎用的な帳票生成ライブラリを設計・実装するアイディアを紹介します。そしてその後、ライブラリを実践的なアプリケーション アーキテクチャの中にどう組み込むのか解説します。

- DioDocs（公式Webサイト）
- 前回記事：.NET向けExcel・PDF操作ライブラリ「DioDocs」の衝撃！ その魅力に迫る

はじめに

今回実装するライブラリはすでにOSSとしてGitHubとNuGetに公開しています。そのライブラリを利用した帳票生成は今すぐに試すことができます。DioDocs自体がNuGetに公開されており、評価版の申し込みなしにある程度利用できるためです。これはうれしいですね。

さあ、それでは改めてDioDocsの世界へ足を踏み入れてみましょう！

本稿の構成

主に次の3つの内容を取り扱います。

1. GitHubとNuGetに公開しているライブラリの紹介と設計の解説
2. エンタープライズ領域において、DioDocsが.NET Standardをサポートする価値とは
3. 汎用帳票ライブラリを組み込んだアプリケーション アーキテクチャ例の紹介

まずはNuGetで公開しているパッケージを利用して帳票生成するコードを紹介し、DioDocsを利用するとどれだけ簡単に帳票が生成できるか見ていただきます。また、ライブラリの実装を解説します。

続いて、前回の記事でも記載しましたが、いまひとつ正しく伝わりきっていなかったようなので、改めてDioDocsが.NET Frameworkではなく.NET Standardをサポートしている価値の大きさを、もう一歩踏み込んで説明したいと思います。

そして最後に、帳票生成を扱うアプリケーション全体のアーキテクチャ例について紹介します。1．の内容では帳票生成にフォーカスした狭い視点での解説になりますが、それを利用して実際のアプリケーションレベルでどのように取り込むべきか、その一例を示します。

そして、最初に種明かしをしましょう。「1．は.NET Frameworkで」「3．は.NET Coreで」そして「汎用帳票生成ライブラリは .NET Standardで」書かれています。そして、1．と3．から同じバイナリを利用しています。

ほら、興味をひかれませんか？

なお本稿のソースコードは、すべてこちらのリポジトリに公開しています。併せてご覧ください。

筆者について

私は普段、主に金融業界向けの受託開発のアーキテクトを務めています。このため本稿はエンタープライズ領域でのアプリケーション開発者の視点で記載しています。

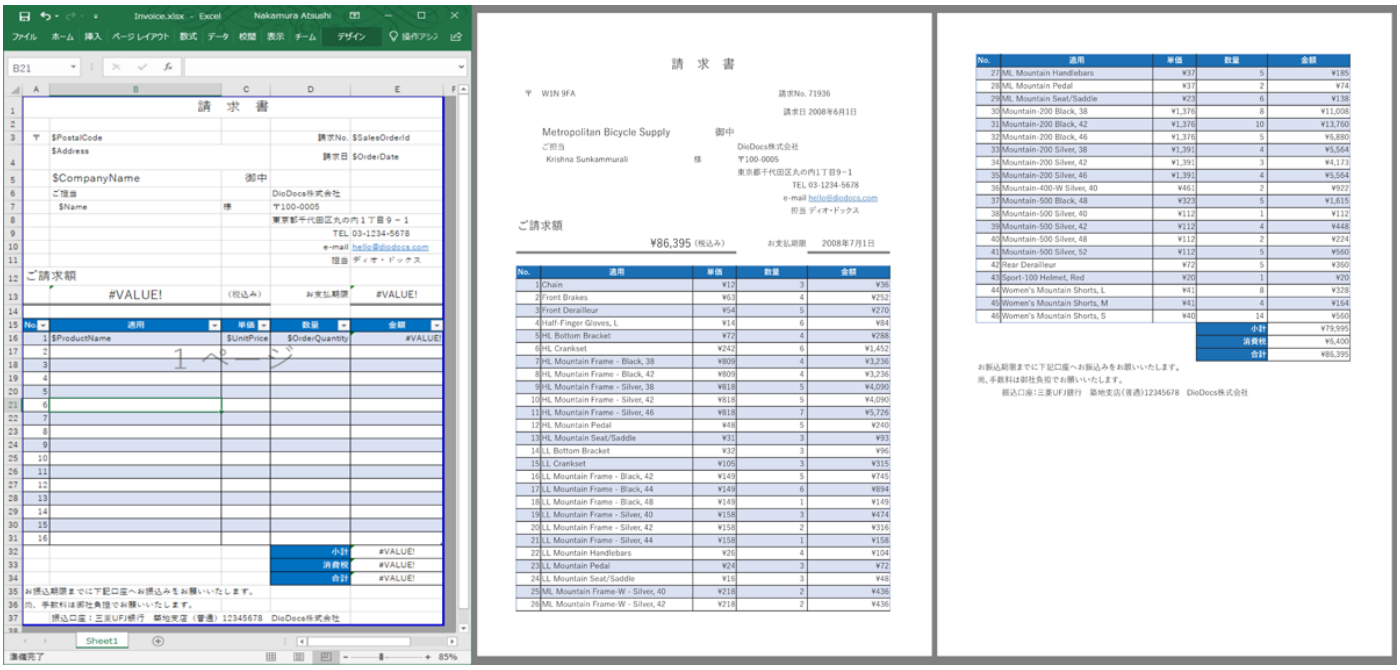
本稿では.NET Standardや.NET Core、.NET Frameworkに対する今後の取り組み方の指針について触れますが、実務の領域が異なれば大きな考え方の相違が生じるかもしれません。仮にそうであっても、特定の分野から見るとそういう考えもあるものと捉えていただければ幸いです。

汎用帳票生成ライブラリの仕様

早速、DioDocsを利用した帳票生成ライブラリを見ていただきましょう。

完成イメージ

まずは下の図をご覧ください。



完成イメージ

上図の左のExcelのテンプレートを用意し、下のコードを実行すると右の帳票が生成されます。

```
var reportBuilder =
    new ReportBuilder<InvoiceDetail>(template)
        // 単一項目のSetterを設定
        .AddSetter("$SalesOrderId", cell => cell.Value = invoice.SalesOrderId)
        .AddSetter("$OrderDate", cell => cell.Value = invoice.OrderDate)
        .AddSetter("$CompanyName", cell => cell.Value = invoice.CompanyName)
        .AddSetter("$Name", cell => cell.Value = invoice.Name)
        .AddSetter("$Address", cell => cell.Value = invoice.Address)
        .AddSetter("$PostalCode", cell => cell.Value = invoice.PostalCode)
        // テーブルのセルに対するSetterを設定
        .AddTableSetter("$ProductName", (cell, detail) => cell.Value = detail.ProductName)
        .AddTableSetter("$UnitPrice", (cell, detail) => cell.Value = detail.UnitPrice)
        .AddTableSetter("$OrderQuantity", (cell, detail) => cell.Value = detail.OrderQuantity);

reportBuilder.Build(invoice.InvoiceDetails, outputStream, SaveFileFormat.Pdf);
```

ライブラリの利用者は、Excelのテンプレートと上のコード（とデータを生成するビジネスロジック）以外、何も用意する必要はありません。驚異的な生産性なのが見て取れるでしょう。

では、もう少し手順を追って解説していきます。

利用方法

まずは完成した帳票生成ライブラリを使って、どのように帳票を出力するのか見ていきましょう。

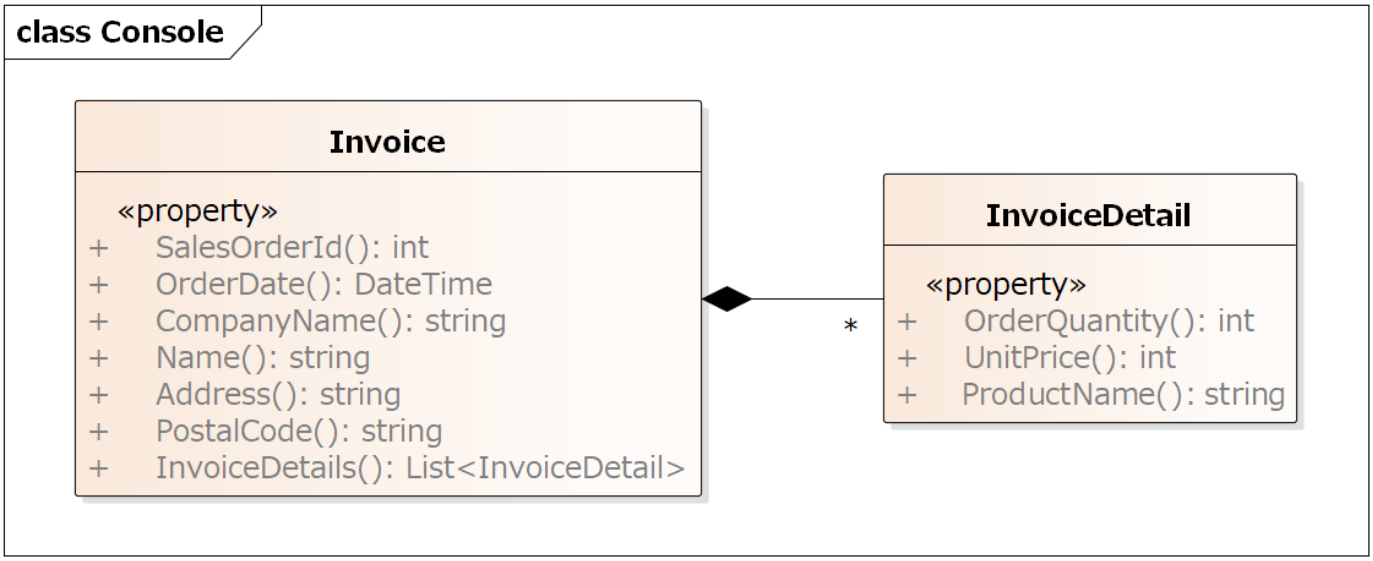
GitHubのこちらから完成したすべてのコードが確認できます。

まず .NET Frameworkのプロジェクトを作成し、NuGetから今回の対象となる帳票生成ライブラリ「**DioDocs.FastReportBuilder**」をインストールします。

次のようにPackage Managerからコマンドを入力してもいいですし、GUIからインストールしてもいいでしょう。

```
PM> Install-Package DioDocs.FastReportBuilder -Version 0.1.0
```

その上で帳票に表示する情報を保持した、次のような請求書クラスと請求明細クラスを作ります。



請求書クラスと請求明細クラス

請求書クラスは複数の請求明細をListとして保持しています。実際のコードは次の通りです。

```
public class Invoice
{
    public int SalesOrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public string CompanyName { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string PostalCode { get; set; }
    public List<InvoiceDetail> InvoiceDetails { get; } = new List<InvoiceDetail>();
}

public class InvoiceDetail
{
    public int OrderQuantity { get; set; }
    public int UnitPrice { get; set; }
    public string ProductName { get; set; }
}
```

続いて次のような帳票のテンプレートをExcelを用意しましょう。**こちらから**ダウンロードできます。

帳票のテンプレート

先のクラスのプロパティ名がExcelの中に「$プロパティ名」で定義されているのが見て取れるかと思います。

では実際に帳票生成ライブラリを使って帳票を生成します。まずは、次のように実行して帳票生成ライブラリをセットアップします。

```
var reportBuilder =
    new ReportBuilder<InvoiceDetail>(stream)
        // 単一項目のSetterを設定
        .AddSetter("$SalesOrderId", cell => cell.Value = invoice.SalesOrderId)
        .AddSetter("$OrderDate", cell => cell.Value = invoice.OrderDate)
        .AddSetter("$CompanyName", cell => cell.Value = invoice.CompanyName)
        .AddSetter("$Name", cell => cell.Value = invoice.Name)
        .AddSetter("$Address", cell => cell.Value = invoice.Address)
        .AddSetter("$PostalCode", cell => cell.Value = invoice.PostalCode)
        // テーブルのセルに対するSetterを設定
        .AddTableSetter("$ProductName", (cell, detail) => cell.Value = detail.ProductName)
        .AddTableSetter("$UnitPrice", (cell, detail) => cell.Value = detail.UnitPrice)
        .AddTableSetter("$OrderQuantity", (cell, detail) => cell.Value = detail.OrderQuantity);
```

Excelのテンプレートに定義した「$プロパティ名」に対して、値を設定させるためのセッターをラムダ式で指定します。

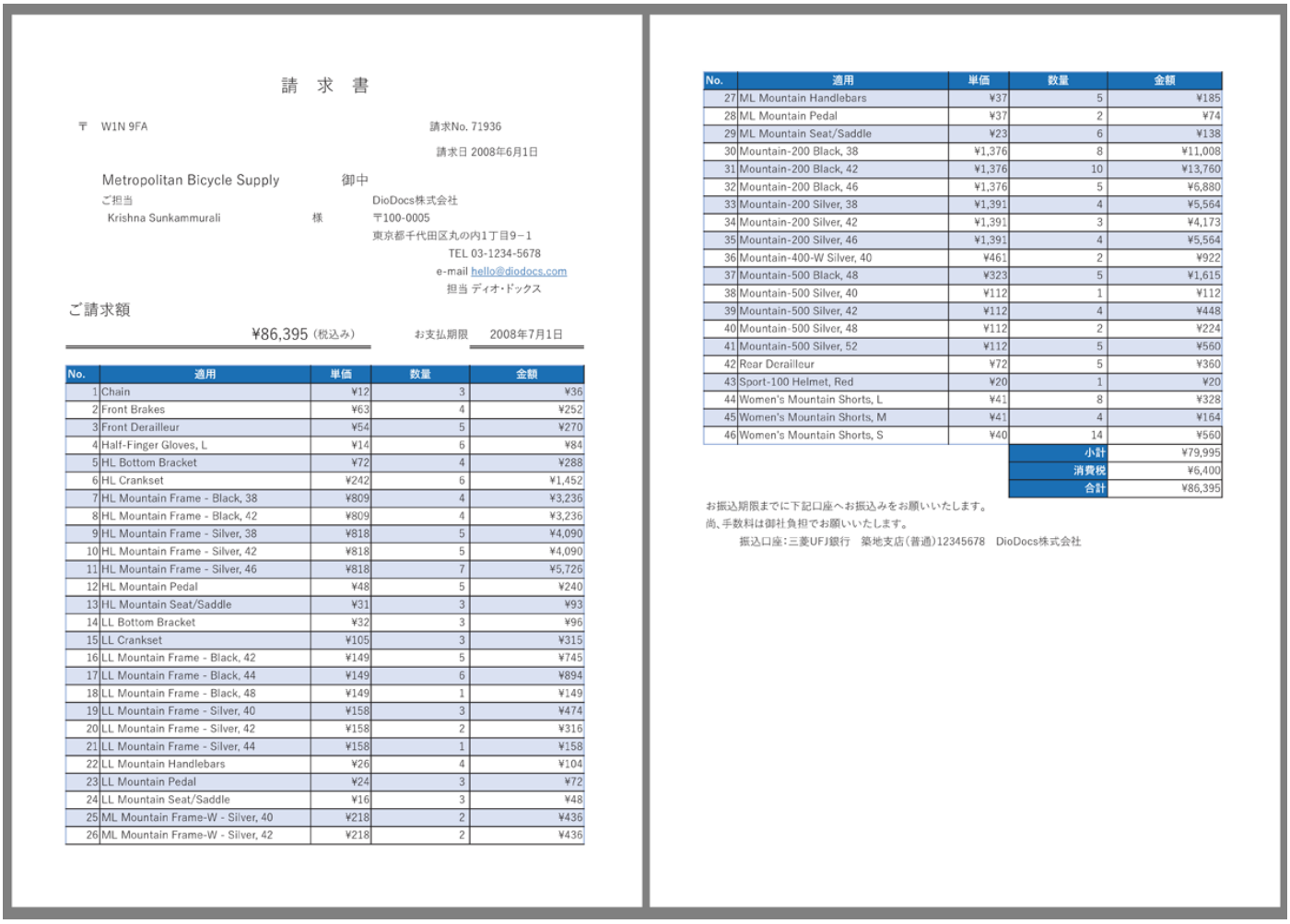
ラムダ式をコールバックさせることで型安全性を損なわず、リフレクションも利用せず、文字列ではなくオブジェクトをセルに設定できます。書式設定を利用するためには文字列ではなく、オブジェクトを設定する方が柔軟に対応できます。

セッターを追加するには2種類の方法があります。AddSetterでは単項目に対するコールバック関数を登録し、AddTableSetterでは表の列に対して値を設定するコールバック関数を登録します。AddTableSetterではコールバックの引数に1行を表すオブジェクトを受け取ります。この時、ReportBuilderクラスのインスタンス生成時に指定した型パラメーターのオブジェクトを受け取ります。

これで準備は完了です。おもむろに次の通り実行します。

```
reportBuilder.Build(
        invoice.InvoiceDetails,
        outputStream,
        SaveFileFormat.Pdf);
```

表の行を表すオブジェクトのリストと、生成した帳票を出力するためのストリーム、そして出力形式を渡します。そしてでき上がったのが次のPDFです。



請　求　書

〒　W1N 9FA

請求No. 71936

請求日 2008年6月1日

Metropolitan Bicycle Supply　御中

ご担当
Krishna Sunkammurali　様

DioDocs株式会社
〒100-0005
東京都千代田区丸の内1丁目9－1
TEL 03-1234-5678
e-mail hello@diodocs.com
担当 ディオ・ドックス

ご請求額

¥86,395 (税込み)　お支払期限　2008年7月1日

No.	適用	単価	数量	金額
1	Chain	¥12	3	¥36
2	Front Brakes	¥63	4	¥252
3	Front Derailleur	¥54	5	¥270
4	Half-Finger Gloves, L	¥14	6	¥84
5	HL Bottom Bracket	¥72	4	¥288
6	HL Crankset	¥242	6	¥1,452
7	HL Mountain Frame - Black, 38	¥809	4	¥3,236
8	HL Mountain Frame - Black, 42	¥809	4	¥3,236
9	HL Mountain Frame - Silver, 38	¥818	5	¥4,090
10	HL Mountain Frame - Silver, 42	¥818	5	¥4,090
11	HL Mountain Frame - Silver, 46	¥818	7	¥5,726
12	HL Mountain Pedal	¥48	5	¥240
13	HL Mountain Seat/Saddle	¥31	3	¥93
14	LL Bottom Bracket	¥32	3	¥96
15	LL Crankset	¥105	3	¥315
16	LL Mountain Frame - Black, 42	¥149	5	¥745
17	LL Mountain Frame - Black, 44	¥149	6	¥894
18	LL Mountain Frame - Black, 48	¥149	1	¥149
19	LL Mountain Frame - Silver, 40	¥158	3	¥474
20	LL Mountain Frame - Silver, 42	¥158	2	¥316
21	LL Mountain Frame - Silver, 44	¥158	1	¥158
22	LL Mountain Handlebars	¥26	4	¥104
23	LL Mountain Pedal	¥24	3	¥72
24	LL Mountain Seat/Saddle	¥16	3	¥48
25	ML Mountain Frame-W - Silver, 40	¥218	2	¥436
26	ML Mountain Frame-W - Silver, 42	¥218	2	¥436

No.	適用	単価	数量	金額
27	ML Mountain Handlebars	¥37	5	¥185
28	ML Mountain Pedal	¥37	2	¥74
29	ML Mountain Seat/Saddle	¥23	6	¥138
30	Mountain-200 Black, 38	¥1,376	8	¥11,008
31	Mountain-200 Black, 42	¥1,376	10	¥13,760
32	Mountain-200 Black, 46	¥1,376	5	¥6,880
33	Mountain-200 Silver, 38	¥1,391	4	¥5,564
34	Mountain-200 Silver, 42	¥1,391	3	¥4,173
35	Mountain-200 Silver, 46	¥1,391	4	¥5,564
36	Mountain-400-W Silver, 40	¥461	2	¥922
37	Mountain-500 Black, 48	¥323	5	¥1,615
38	Mountain-500 Silver, 40	¥112	1	¥112
39	Mountain-500 Silver, 42	¥112	4	¥448
40	Mountain-500 Silver, 48	¥112	2	¥224
41	Mountain-500 Silver, 52	¥112	5	¥560
42	Rear Derailleur	¥72	5	¥360
43	Sport-100 Helmet, Red	¥20	1	¥20
44	Women's Mountain Shorts, L	¥41	8	¥328
45	Women's Mountain Shorts, M	¥41	4	¥164
46	Women's Mountain Shorts, S	¥40	14	¥560
			小計	¥79,995
			消費税	¥6,400
			合計	¥86,395

お振込期限までに下記口座へお振込みをお願いいたします。
尚、手数料は御社負担でお願いいたします。
振込口座：三菱UFJ銀行　築地支店（普通）12345678　DioDocs株式会社

でき上がったPDFファイル

よく見ていただきたい点がたくさんあります。

- テンプレートは1ページだったがPDFはページングされている
- 2ページ目のテーブルにも列ヘッダーが表示されている
- 日付や金額が適切にフォーマットされている
- C#のコードから設定したのは明細の品目・単価・数量だけだが、次が正しく表示されている
  - 単価×数量から求めた「金額」
  - 小計
  - 消費税
  - 合計
  - ご請求額
- 同様に請求書情報から請求年月日を設定しただけだが、「お支払期限」が表示されている

これほどに簡単に、ページング可能な帳票を作れた記憶が私にはありません。DioDocsの使い勝手が良い事もありますが、私自身が日頃からExcelに慣れ親しんでいたことも大きな要素だと思います。とは言え、特別にExcelをよく理解しているという程でもなく、ありふれたレベルでしかありません。自身が持っているExcelスキルを100%活用できるのがDioDocsの魅力です。

では、帳票生成ライブラリの実装を見ていきましょう。

汎用帳票生成ライブラリの実装

帳票生成のソフトウェア アーキテクチャ

さて、具体的な実装に触れる前に帳票生成の仕組みをまず整理しましょう。

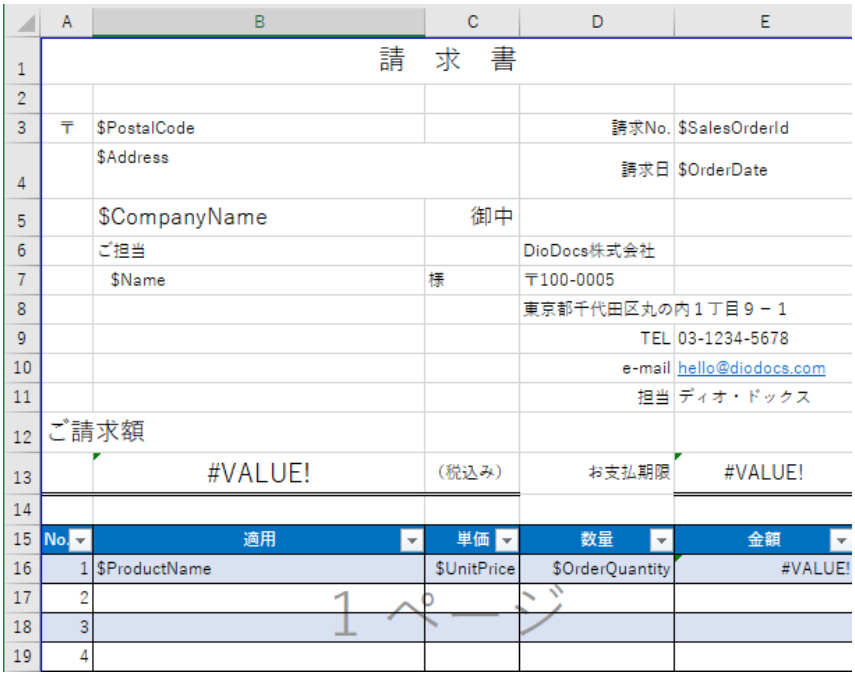
今回の帳票生成ライブラリでは、帳票のテンプレートにはExcelを利用することを大前提とします。このためライブラリの設計としては、Excelの存在を隠ぺいするような抽象化は行わないこととします。

また、帳票の出力項目が変わらない限りは自由にExcelのテンプレートを編集することを可能とし、プログラムの改修を伴わずに帳票のレイアウトを変更できるようにします。このため項目値の埋め込みに具体的なセル（A1やB2など）を指定することはできません。

帳票の形式は1つの表を含む帳票を前提とし、表以外に任意の単項目を設定できるものとします。表にはExcelのテーブルを利用します。テーブルを利用する理由については後述します。テーブルを利用するため、表でセルの結合を利用することができません。このため帳票内で複数の表を扱うことは現実的ではないでしょう。テーブルを使わなければ複数の表に埋め込むことも不可能ではないでしょうが、実装の難易度やユーザビリティが大幅に悪化することが想定できます。多くの帳票が単一の表でカバーできるであろうことから、今回は対象としません。

Excelへ値の適用ルール

下図はExcelのテンプレートの上半分を表示したものです。



	A	B	C	D	E
1		請　求　書			
2					
3	〒	$PostalCode		請求No.	$SalesOrderId
4		$Address		請求日	$OrderDate
5		$CompanyName	御中		
6		ご担当		DioDocs株式会社	
7		$Name	様	〒100-0005	
8				東京都千代田区丸の内１丁目９－１	
9				TEL	03-1234-5678
10				e-mail	hello@diodocs.com
11				担当	ディオ・ドックス
12	ご請求額				
13		#VALUE!	（税込み）	お支払期限	#VALUE!
14					
15	No.	適用	単価	数量	金額
16	1	$ProductName	$UnitPrice	$OrderQuantity	#VALUE!
17	2				
18	3				
19	4				

1ページ

テンプレートの上半分

大切な点がいくつかあります。今回は次のルールでテンプレートへ値の適用を行います。

- 単項目は、項目名を判別可能な形でテンプレートへ記載する（この例では「$～」のセルを動的な値で置き換える）
- 明細はExcelのテーブルを利用する
- テーブルは、1行目に項目名を定義する

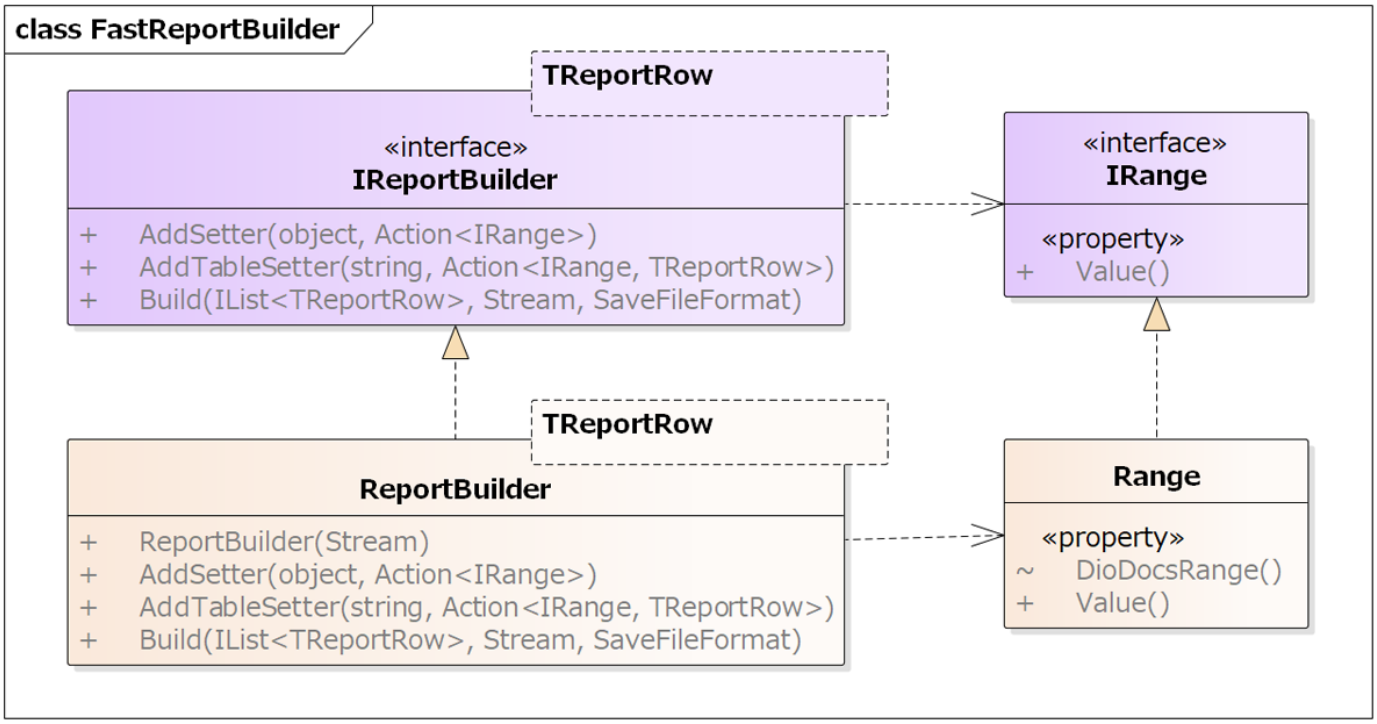
ヘッダーは業務要求で変更が発生する可能性が高いので、マッピングへの利用を避けます。

クラス設計と実装

さて先にも見ていただいたように、帳票生成ライブラリを使うにあたり次のコードでライブラリを初期化していました。

```
var reportBuilder =
    new ReportBuilder<InvoiceDetail>(stream)
        // 単一項目のSetterを設定
        .AddSetter("$SalesOrderId", cell => cell.Value = invoice.SalesOrderId)
        ...
        // テーブルのセルに対するSetterを設定
        .AddTableSetter("$ProductName", (cell, detail) => cell.Value = detail.ProductName)
        ...
```

帳票生成はReportBuilderクラスを利用して行います。このクラスの設計は次のモデルのようになっています。



ReportBuilderクラスのモデル

IReportBuilderとIRange、2つのインターフェースと、それぞれの実装クラスが存在します。IReportBuilderは帳票生成の処理を抽象化したインターフェースで、次の3つのメソッドが定義されています。

```
public interface IReportBuilder<TReportRow>
{
    /// <summary>
    /// 単項目をアプリケーション側に設定させるため、設定対象のセル（IRange）を引数に
    /// コールバックさせるためのActionを登録する。
    /// </summary>
    IReportBuilder<TReportRow> AddSetter(object key, Action<IRange> setter);

    /// <summary>
    /// 表項目をアプリケーション側に設定させるため、設定対象のセル（IRange）を引数に
    /// コールバックさせるためのActionを登録する。
    /// </summary>
    IReportBuilder<TReportRow> AddTableSetter(string key, Action<IRange, TReportRow> setter);

    /// <summary>
    /// 表の行オブジェクトを引数に指定して帳票を生成する
    /// </summary>
    void Build(IList<TReportRow> rows, Stream stream, SaveFileFormat saveFileFormat);
}
```

IReportBuilderは型パラメーターとして、表の行を表すクラスを指定します。

ポイントは次の通りです。ライブラリを生成する側は、設定する先をセルの座標で指定しません。論理的な名称（つまり文字列）を指定し、ライブラリ側がその位置を特定してセルオブジェクト（Excelのオブジェクト設計の慣例に従いIRangeとしています）を引数にアプリケーション側をコールバックし、アプリケーション側で必要な値をラムダ式内で設定しています。

こうすることでテンプレートとなるExcelを編集し、項目を表示すべきセルの座標が変わってもプログラム側を編集することなく適用可能になります。

Add～の戻り値がIReportBuilder<TReportRow>になっていますが、下のコードサンプルのように設定用コールバックをメソッドチェーンで設定できるようにするためのものです。メソッドチェーンには好みの問題がありますが、利用者が選択できるようにしています。

```
var reportBuilder =
    new ReportBuilder<InvoiceDetail>(stream)
        // 単一項目のSetterを設定
        .AddSetter("$SalesOrderId", cell => cell.Value = invoice.SalesOrderId)
        .AddSetter("$OrderDate", cell => cell.Value = invoice.OrderDate)
        .AddSetter("$CompanyName", cell => cell.Value = invoice.CompanyName)
        ...
```

IRangeインターフェースはExcelのセルを表すオブジェクトです。実際には単独のセルだけではなく、結合されたセルを扱うこともあるため、Excelのオブジェクト設計の慣例にならってIRangeとしています。以下のようにsetだけがあるValueプロパティを持っており、コールバックしてアプリケーション側からExcelのセルに値を設定できるようにしています。

```
public interface IRange
{
    object Value { set; }
}
```

IRangeの実装クラスは次のようになっています。

```
public class Range : IRange
{
    internal GrapeCity.Documents.Excel.IRange DioDocsRange { private get; set; }

    public object Value
    {
        set => DioDocsRange.Value = value;
    }
}
```

内部的にDioDocsのIRangeオブジェクトを所持していて、そのラッパーの役割を担っています。帳票生成ライブラリはExcelベースのライブラリであることを隠ぺいしませんが、DioDocs自体は隠ぺいする設計にしています。これには主に2つの理由があります。

- 帳票生成ライブラリの使用者のテストコードを書く際、サードパーティクラスに依存しているとテストの自由度が下がるため
- DioDocs以外のライブラリに差し替えられる余地を残すため

それではいよいよIReportBuilderの実装クラスを見ていきましょう。全体をここに載せると見通しが悪くなるので、少しずつ切り取ったコードで解説していきます。コードの全体像は**こちら**をご覧ください。

まずフィールドを見ていきましょう。

```
private readonly byte[] _template;
private readonly string _tableName;

private readonly Dictionary<object, Action<IRange>> _setters =
    new Dictionary<object, Action<IRange>>();
private readonly Dictionary<object, Action<IRange, TReportRow>> _tableSetters =
    new Dictionary<object, Action<IRange, TReportRow>>();
```

テンプレートとなるExcelファイルを読み込んだバイト列と、帳票内の表として使うExcelのテーブルの名称（Excelのデザインタブでテーブルに名前を付けられます）、それから単項目と表の項目へ値を設定するためのコールバックActionを保持するDictionaryを持っています。

Excelの読込先について、バイト列ではなくIWorkbookにすることも考えられます。しかし、ReportBuilderのインスタンスを再利用するケースを考えた場合、2度目の生成時に1度目では値を設定したセルが2度目に設定しないとなると、前回の値を引き継いでしまう不具合が発生する可能性があります。こうした不具合の温床となる可能性があるため、Workbookは毎回開き直す実装にしています。

続いてコンストラクタとコールバックアクションの登録メソッドを見てみましょう。

```
public ReportBuilder(Stream template)
{
    _template = new byte[template.Length];
    template.Read(_template, 0, (int)template.Length);
    _tableName = typeof(TReportRow).Name;
}

public IReportBuilder<TReportRow> AddSetter(object key, Action<IRange> setter)
{
    _setters[key] = setter;
    return this;
}

public IReportBuilder<TReportRow> AddTableSetter(string key, Action<IRange, TReportRow> setter)
{
    _tableSetters[key] = setter;
    return this;
}
```

今回、コンストラクタはStreamを受けるようにしていますが、byte[]やFileへのパスを受け取ってもいいでしょう。

また、テーブルの名称はデフォルトではテーブルの行に該当する情報を保持しているクラス（今回はInvoiceDetail）の名称を設定しています。これは任意に指定するためのコンストラクタがあってもいいと思います。

それでは、実際に帳票を生成しているBuildメソッドを見ていきます。ここには大きく分けて次の4つの処理が含まれています。

1. 単項目への値の設定
2. テーブルへ設定する行数がテンプレートのExcelのテーブルの行数より多かった場合の調整
3. テーブルの列設定を解析し、どの項目を何列目に設定すべきか解析
4. テーブルへ値を設定

順番に見ていきましょう。まずは単項目の値を設定します。

```
public byte[] Build(IList<TReportRow> rows)
{
    IWorkbook workbook;
    using (var inputStream = new MemoryStream(_template))
    {
        workbook = new Workbook();
        workbook.Open(inputStream);
    }
    var worksheet = workbook.Worksheets[0];

    // コールバックに渡すためのIRangeオブジェクト
    // 都度生成すると、大きな帳票ではインスタンス生成コストが無視できない
    // 可能性があるため、インスタンスを使いまわす
    var range = new Range();
    // 利用している領域を走査して、単一項目を設定する
    var usedRange = worksheet.UsedRange;
    for (var i = 0; i < usedRange.Rows.Count; i++)
    {
        for (var j = 0; j < usedRange.Columns.Count; j++)
        {
            var cell = usedRange[i, j];
            if (cell.Value != null && _setters.ContainsKey(cell.Value))
            {
                range.DioDocsRange = cell;
                _setters[cell.Value](range);
            }
        }
    }
```

帳票生成は先頭のワークシートを利用します。

値の設定はコールバックにて行いますが、その際にDioDocsのRangeをラップして渡します。値を設定する回数が多いとラップオブジェクトの生成コストが気にかかりますので（実際は影響ないと思いますが）、Rangeクラスのインスタンスは使いまわしています。

単項目を設定する範囲の走査はWorksheetのUsedRangeプロパティによって、テンプレートで利用している領域にのみ行います。

使用領域を走査していき、セルに値設定用のコールバックを登録したときのkeyと同じ値が設定されている箇所を探します。該当するセルを見つけたら、コールバックを呼び出して値を設定させるという形で処理しています。

続いてテーブルの行数の調整です。

```
var templateTable = worksheet.Tables[_tableName];

// テーブルの行数を確認し、不足分を追加する
if (templateTable.Rows.Count < rows.Count)
{
    var addCount = rows.Count - templateTable.Rows.Count;
    for (var i = 0; i < addCount; i++)
    {
        templateTable.Rows.Add(templateTable.Rows.Count - 1);
    }
}
```

ワークシートから対象のテーブルを名称を指定して取得します。取得したテーブルの行数が、Buildメソッドを呼び出されたときに渡された行オブジェクトよりも少ない場合はテーブルに行を追加します。

ここで素晴らしいのは、行を追加するだけで前後の行とおなじExcel式が新しい行にも設定される点にあります。この挙動がExcelと同じであることは、ご存知の方も多いでしょう。そのため可変の表を取り扱いやすく、ページングなども特に細かな配慮の要なく実現できます。

続いてテーブルの1行目を解析して、値を設定すべき列の番号を調べます。

```
// テーブルの1行目から項目の列番号を探索する
var rowSetters = new List<(int index, Action<IRange, TReportRow> setter)>();
var firstRow = templateTable.Rows[0];
for (var i = 0; i < firstRow.Range.Columns.Count; i++)
{
    var value = firstRow.Range[0, i].Value;
    if (value != null && _tableSetters.ContainsKey(value))
    {
        rowSetters.Add((i, _tableSetters[value]));
    }
}
```

表項目のコールバックを登録した際に指定されたkeyに該当する列を探索し、その列のインデックスとコールバックをrowSetter変数に保持します。そしてテーブルに値を設定していきます。

```
// テーブルに値を設定する
for (var i = 0; i < rows.Count; i++)
{
    var row = templateTable.Rows[i];
    foreach (var rowSetter in rowSetters)
    {
        range.DioDocsRange = row.Range[rowSetter.index];
        rowSetter.setter(range, rows[i]);
    }
}
```

あとは帳票を生成するだけです。

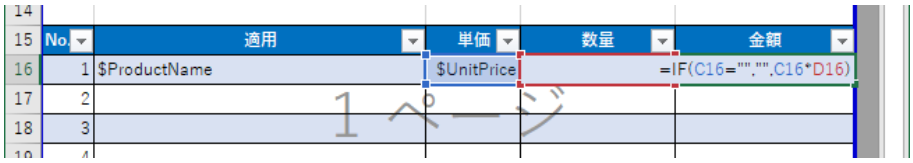
```
workbook.Save(stream, (GrapeCity.Documents.Excel.SaveFileFormat)saveFileFormat);
}
```

ちょっとしたアイディアだけで、簡単に汎用的な帳票生成ライブラリを作ることができました。

さて、ライブラリの実装は以上ですが、今回の帳票を生成するために利用している、Excelのテンプレート側の設定についても解説しておきましょう。

Excelの式の活用

.NETのプログラムから明示的に値を設定する箇所は最小限にし、Excelの式で計算できる箇所は式を利用します。今回はNo. ・金額・小計・消費税・合計・お支払期限（請求日から）はすべてExcelの式で計算します。

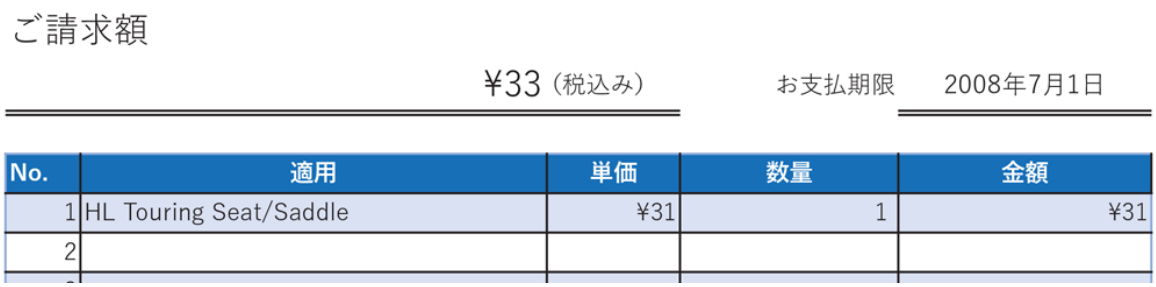


Excelの式で計算する

なお基本的にDioDocsが式を評価して計算結果を出力してくれますが、一点注意が必要です。

明細を表示する「表」の部分は、明細の件数によって増加します。その際、単純に行を追加して式が失われることや、小計を計算する範囲が含まれないなんてことが起きてはいけません。今回はテーブルを利用して、行の増減の操作を楽にできるようにしています。

もう一点見ていただきたいのは、金額列が空欄だった場合です。


ご請求額

¥33（税込み） お支払期限 2008年7月1日

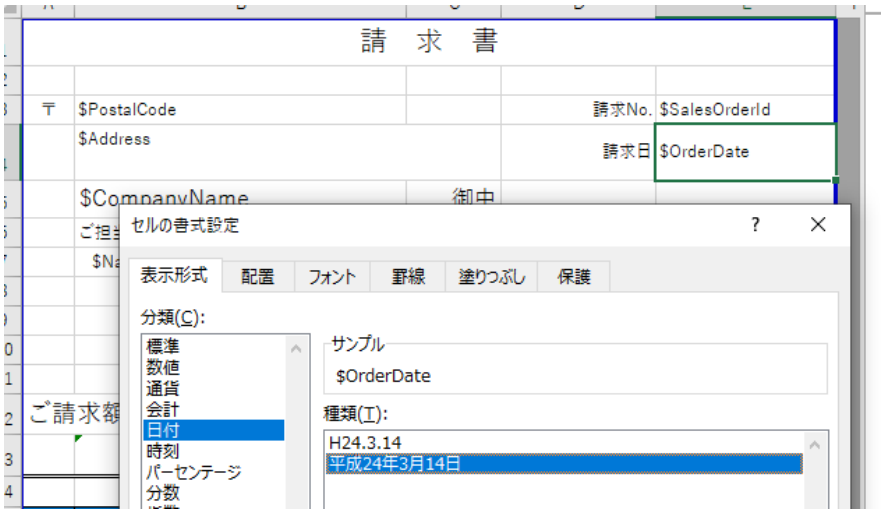
No.	適用	単価	数量	金額
1	HL Touring Seat/Saddle	¥31	1	¥31
2				

金額列が空欄の場合

金額列の式でIF文を利用し、単価列が未入力であった場合に「0」と表示されないように制御しています。ここにExcelの強みが見て取れるのではないでしょうか？

Excelの書式の活用

数値のカンマフォーマット、日付の書式、そういった部分の実装は簡単ではあっても手間がかかるものですし、テストも面倒です。そこで活躍するのがExcelの書式です。恐らく和暦の新元号対応も、Excelが正しく動作すれば楽に対応できるのではないでしょうか。



Excelの書式を活用

改ページ時のヘッダーの表示

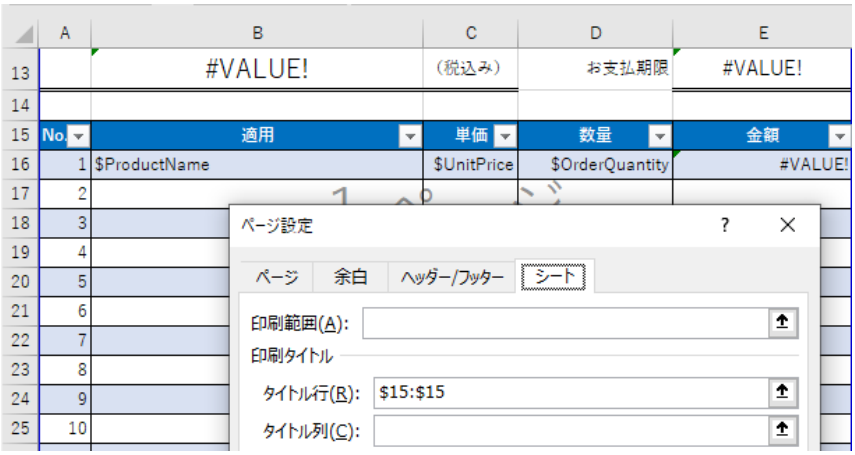
さて、表の行数が可変の場合、悩ましいのが改ページ時のヘッダーの取り扱いです。下図のように2ページ目にまたがった場合、表のヘッダーは毎ページ表示したいというのは一般的な要求だと思います。私も作ってもらうならそう要求するでしょうが、作る側からすると面倒なことこの上ありません。

No.	適用	単価	数量	金額
27	Road-550-W Yellow, 38	¥672	5	¥3,360
28	Road-550-W Yellow, 40	¥672	5	¥3,360
29	Road-550-W Yellow, 42	¥672	4	¥2,688
30	Road-550-W Yellow, 44	¥672	3	¥2,016
31	Road-550-W Yellow, 48	¥672	5	¥3,360
32	Road-750 Black, 44	¥323	3	¥969
33	Road-750 Black, 48	¥323	5	¥1,615
34	Road-750 Black, 52	¥323	4	¥1,292
35	Road-750 Black, 58	¥323	4	¥1,292
36	Short-Sleeve Classic Jersey, L	¥32	3	¥96
37	Short-Sleeve Classic Jersey, S	¥32	1	¥32
38	Short-Sleeve Classic Jersey, XL	¥32	5	¥160
39	Sport-100 Helmet, Black	¥20	5	¥100
40	Sport-100 Helmet, Blue	¥20	3	¥60
41	Sport-100 Helmet, Red	¥20	3	¥60
42	Water Bottle - 30 oz.	¥2	8	¥16
			小計	¥74,074
			消費税	¥5,926
			合計	¥80,000

お振込期限までに下記口座へお振込みをお願いいたします。
尚、手数料は御社負担でお願いいたします。
振込口座：三菱UFJ銀行　築地支店（普通）12345678　DioDocs株式会社

改ページするケース

これもExcelのページ設定機能を利用して実現します。



Excelのページ設定機能を利用

これでExcel＆DioDocsの利便性と生産性の高さを、さらに感じていただけたのではないでしょうか？ 私の場合、これまでActiveReportsを利用してきたシーンの半分以上はDioDocsを使うことになると思います。

.NET Standardである意味

それではライブラリを組み込んだアプリケーションのアーキテクチャについて説明します。しかしその前に、今一度DioDocsが.NET Standardで実装されているメリットについてお話ししたいと思います。

前回、DioDocsに対して私が特に魅力を感じる点を次の通り挙げました。

- ExcelファイルからPDFファイルの生成
- .NET Standard 2.0準拠
- ランタイムフリー
- Excel準拠のオブジェクトモデル
- 開発元は日本が本社の企業なので直接サポートが受けられる

2番目に .NET Standardに準拠していることを挙げています。これは否が応でも徐々に.NET Frameworkから.NET Coreへ移行していく必要があることに端を発しています。

.NET Coreというとクロスプラットフォームな.NETのランタイムで、ASP.NET Coreといったサーバーサイド用のイメージがある方も多いかもしれません。しかし実際にはすでにその範疇を超えつつあります。

現在の.NETの大まかな方針として、すでに新規機能の追加やパフォーマンスの改善は.NET Coreが開発の中心に移行されており、新しく開発された機能などの.NET Frameworkへの適用は、後方互換性を担保しやすい範囲に限られて提供されています。

実際にC# 8.0で追加される機能のいくつかは、次のバージョンである.NET Framework 4.8に適用されないことが判明しています。.NET Frameworkのメジャーバージョンは4.8が最後？ という予測もあります。

このあたりの詳細は併せて以下をご覧ください。

- **C# 8.0 の予告 | ++C++; // 未確認飛行 C ブログ**
- **Update on .NET Core 3.0 and .NET Framework 4.8 | .NET Blog**

ではWindows FormsやWPFがどうなるかというと、.NET Core 3.0から.NET Core上でも動作するようになります。現時点ではプレビューの段階で、実プロダクトへの投入はしばらく先になるかと思いますが、非常に魅力的な話です。現状の.NET CoreにおけるWindows FormsやWPFの対応状況は、以下のブログを参考すると空気感が伝わるのではないでしょうか？

- **.NET Core 3.0 でデスクトップアプリを作る - rksoftware**

.NET Frameworkが終息に向かい、.NET Coreへの移行が求められる状況について、個人的には決して悲観的なことではないと考えています。どちらかというとワクワクしています。実際、私は自部署の技術戦略として積極的に.NET Coreへ移行・投資していこうと考えており、金融機関向けの業務システムにおいてすでに採用を進めています。

これはひとえに .NET Coreが非常に魅力的であるからです。特に次の魅力を感じます。

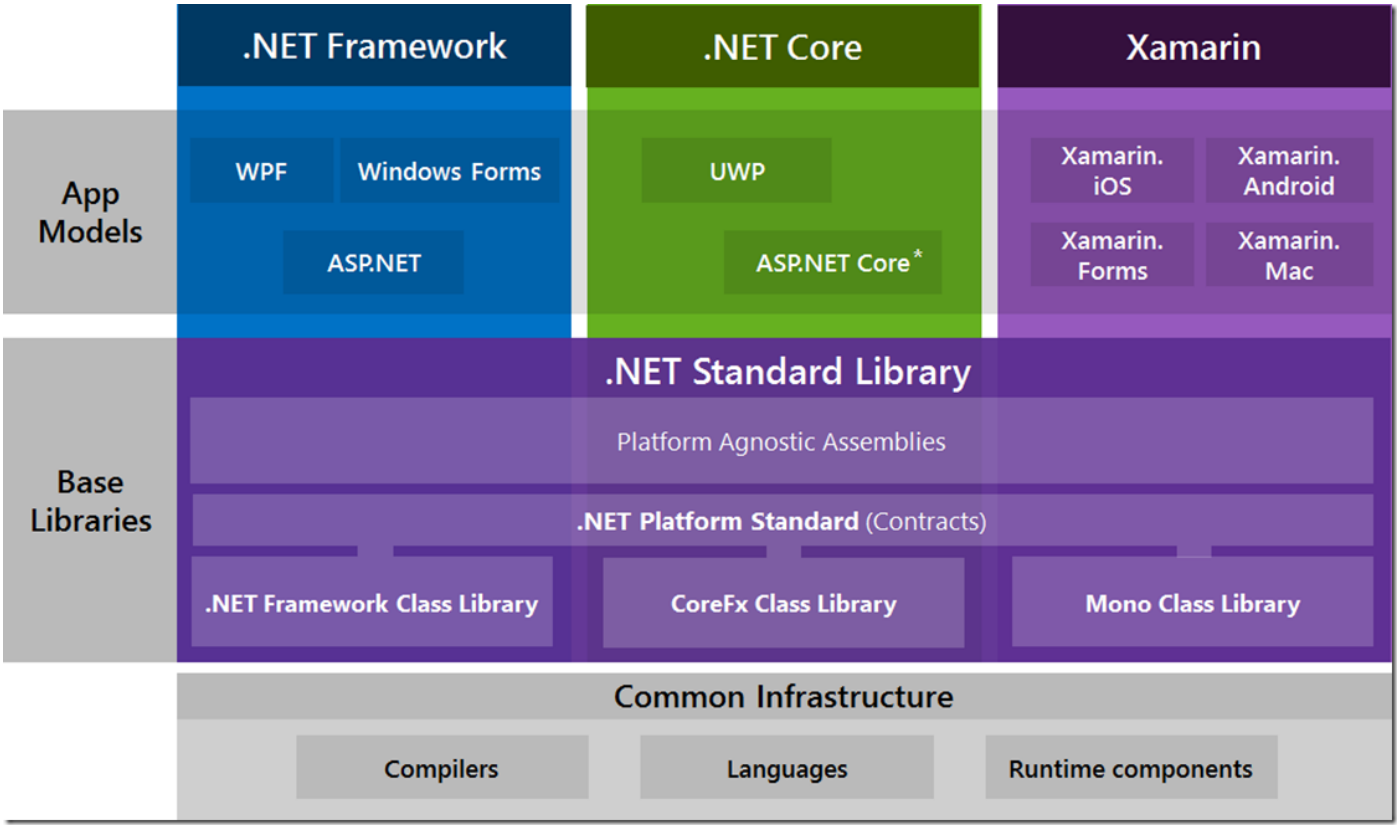
1. Side-by-sideをサポートしており、複数バージョンの.NET Coreが共存できる
2. ランタイムを同梱できるため、インストールされているバージョンに縛られない
3. .NET Coreの先進的な機能や性能を享受できる
4. クラウドを見据えたときに基盤技術を共有しやすい

特に1. と2. はエンタープライズな現場でも、非常に魅力的です。これで顧客環境の古い.NET Frameworkのバージョンに悩まされることがなくなります。さようなら、.NET Framework 3.5！

エンタープライズ領域の業務アプリケーションでも、クラウドファーストな顧客が増えており、大手金融機関でも全体的にその方向に向かっていると実感しています。さらに配布などを考慮すると、オンプレミスであってもDockerコンテナの利用などを進めていきたいと考えています。そうなった場合、OSとしてはLinuxを視野に入れていく必要があり、.NET Coreのクロスプラットフォーム対応に魅力を感じます。

総括すると、現在は.NET Frameworkから.NET Coreへ移行する過渡期にあると言えるでしょう。だからこそ、サードパーティのライブラリは.NET Standardであることに大きなメリットがあります。

ここで改めて、.NET Frameworkと.NET Core、.NET Stanadrdの関係を簡単に整理します。次の図をご覧ください。



.NETファミリーのフレームワークの関係性

- 出典：Cesar de la Torre [MSFT]、**.NET Core, .NET Framework, Xamarin – The “WHAT and WHEN to use it” – Cesar de la Torre [Microsoft] – BLOG**

この図にはUnityやGTK#など、.NET関連のすべてのプロダクトが含まれているわけではありませんし、一番上の段にXamarinと記載されている箇所はMonoといった方が適切な気がしますが、大枠を理解するには便利なモデルだと思います。

.NET Frameworkや.NET CoreはMono（Xamarin）と並んで、.NETのランタイムです。そして当然それらのランタイムの上では基本となるクラス群（stringやDateTimeなどなど）が共通して利用できます。それがBase Librariesです。

Base Librariesの実装はランタイム環境によって異なりますが、自由に実装しているわけではもちろんなく、共通の規格に則って実装されています。その共通規格が .NET Standardです。

.NET Standardにはバージョンがあり、各ランタイムのバージョンごとにサポートする.NET Standardのバージョンが異なります。具体的には次のような対応関係にあります。

.NET Standard	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**
.NET Core	1.0	1	1	1	1	1	1	2.0
.NET Framework 1	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10	10	10	10	10	10	10	10.14
Xamarin.Mac	3	3	3	3	3	3	3	3.8
Xamarin.Android	7	7	7	7	7	7	7	8
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1

- 出典：**.NET Standard - Microsoft Doc**

例えば、.NET Coreの1.0であれば.NET Standard 1.6までサポートしており、.NET Standard 2.0を利用する場合は.NET Core 2.0以上が必要になるといった形で読み解きます。

つまり.NET Frameworkと.NET Coreの過渡期である現在、採用するライブラリが.NET Standardに対応していれば、いずれの環境でも利用することが可能であり、その点に大きな魅力があると私は考えています。

前回の記事でも触れた通り、ExcelのCOM操作は非常にリスクの高い技術であるため、どうしてもExcelでないと不可能な機能の実装を除き、利用するべきではないと思います。それを除外しても、.NET Standard対応のライブラリを利用することは、中長期的な視野に立つとメリットが大きいと考えられます。

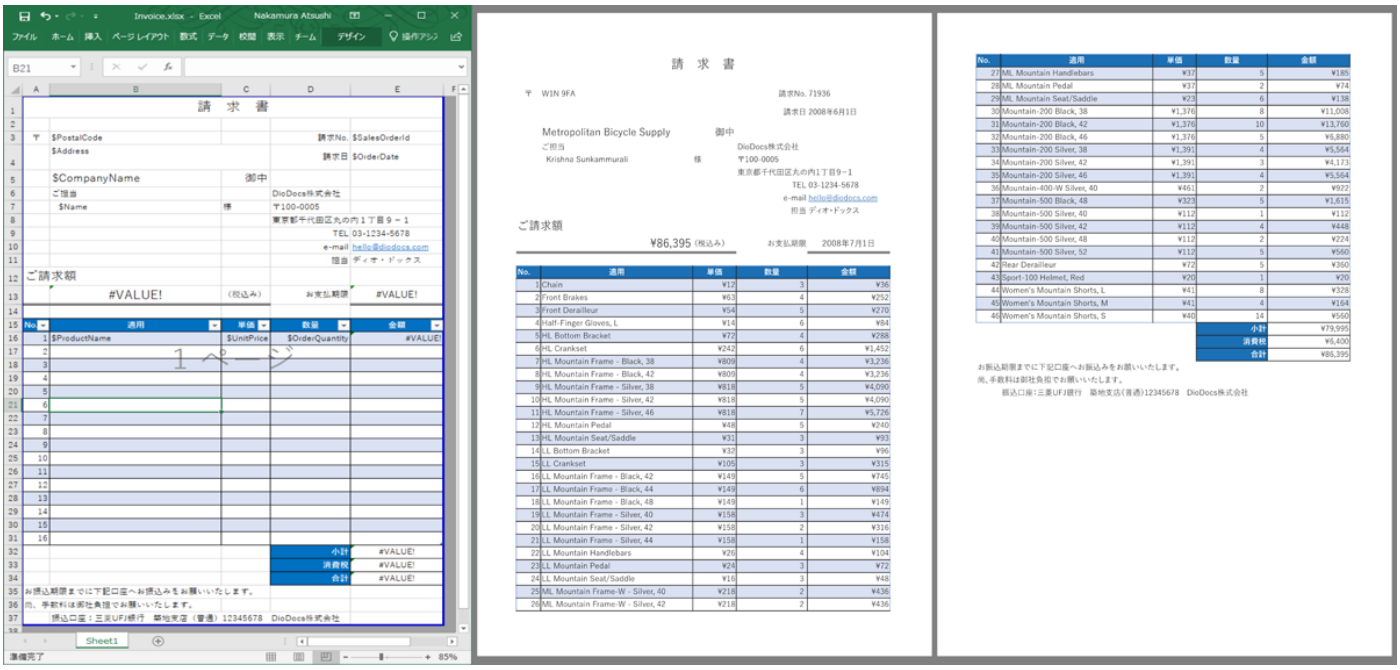
ASP.NET Coreで帳票生成サービスを作る

ここからは先に説明した帳票生成ライブラリを利用して、アプリケーションを構築する際のアーキテクチャの一例を紹介します。

まずは対象のアプリケーションの前提を確認しておきます。本稿ではExcelで設計したテンプレートから、請求書PDFを生成するサービスを構築します。せっかくExcelをテンプレートとして利用できるので、次の要件を満たすものを作ります。

- 表示する項目が変更なければレイアウトは自由に変更可能
- 対象とする帳票は固定の項目と、明細を持つ請求書（下図参照）
- 明細テーブルの項目数が多くなればページングする
- ページングした場合、テーブルの項目名は新しいページにも表示する
- Excelの式や書式を最大限に利用する

今回は先のコンソールの例と同じように、下図の一番左のExcelファイルをテンプレートとして利用し、中央と右側のようなPDF帳票を生成します。



作成するPDF帳票

また今回のサービスは、ASP.NET Coreで構築し、Azure上で動作させることとします。ASP.NET Coreを選定した主な理由は次の通りです。

- Windows FormsやWPFが動作する.NET Core 3.0はプレビュー段階なため
- .NET Standardの恩恵を、現時点でもっとも受けられるため
- Excelアプリケーションが動作しないサーバーサイド アプリケーションのため

採用技術

本稿のサンプルアプリケーションは以下の環境で構築しています。

- Visual Studio 2017 version 15.9.5
- Azure SQL Database
- .NET Core 2.1
- .NET Standard 2.0

- DioDocs for Excel 1.5.0.8

サンプルコードはGitHubで公開しています。併せてご覧ください。

- **nuitsjp/DioDocsStudy**

アプリケーションの概要

アプリケーションの題材としては、Azure SQL Database用のサンプルデータベースであるAdventureWorksLTを利用します。データは英語になってしまいますが、手頃な日本語のサンプルデータベースもないので、そこは妥協します。

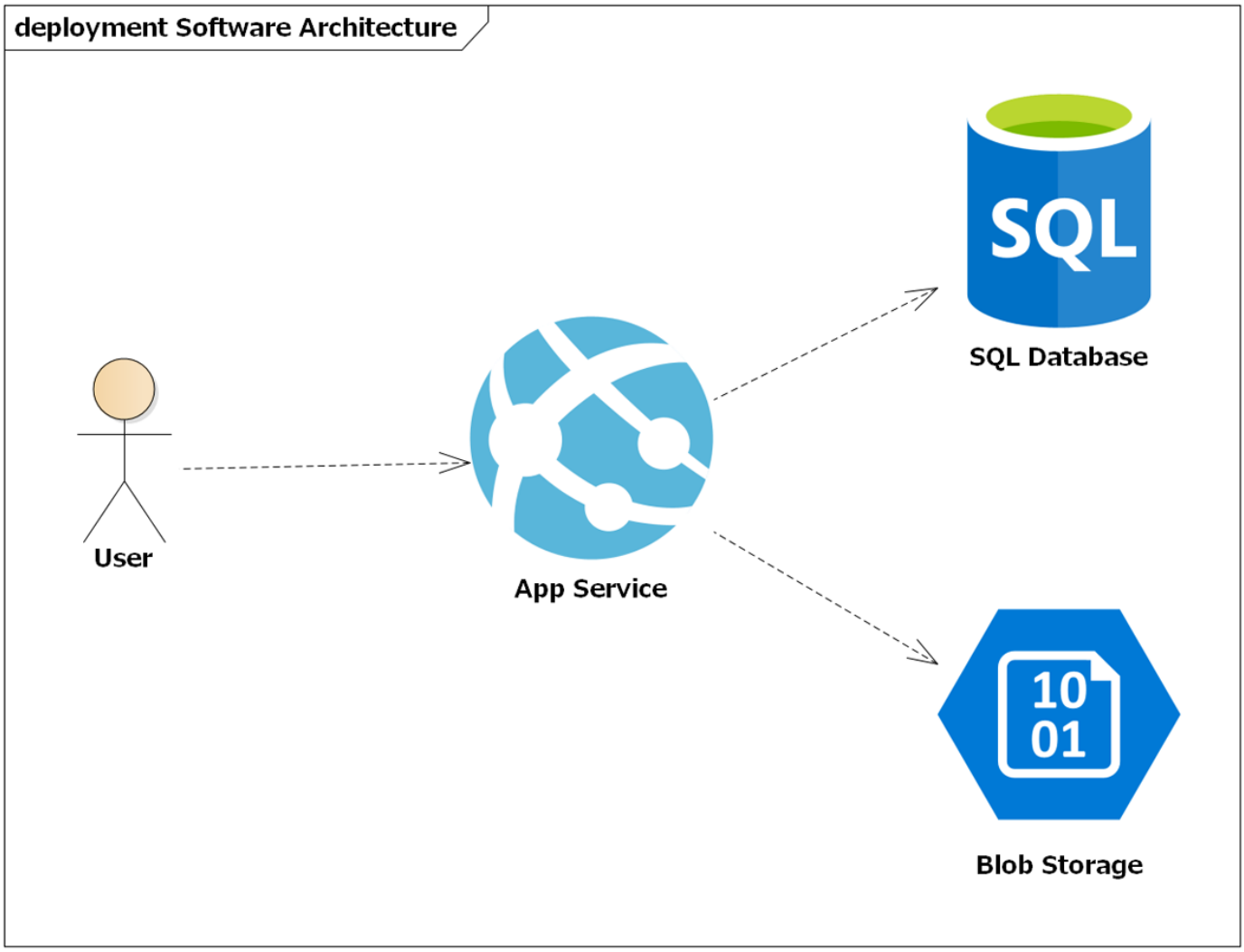
以下は、AdventureWorksサンプルデータベースのビジネスシナリオとして公開されている文章の抜粋です。

> AdventureWorksサンプルデータベースは、Adventure Works Cyclesという架空の大規模多国籍製造企業をベースにしています。この企業は、北米、ヨーロッパ、およびアジアのマーケットを対象に、金属製自転車や複合材製自転車の製造および販売を行っています。

- 出典：**Adventure Works Cycles のビジネス シナリオ**

今回はAdventure Works Cycles社における販売情報をもとに、販売情報の一覧を表示し、その中から指定の顧客の販売情報に対する請求書をPDFで生成してダウンロードするWebアプリケーションを構築していきます。

先にも記載したように、今回はAzure上でASP.NET Coreでアプリケーションを構築します。Azure上の全体の大まかな配置モデルは次の通りです。



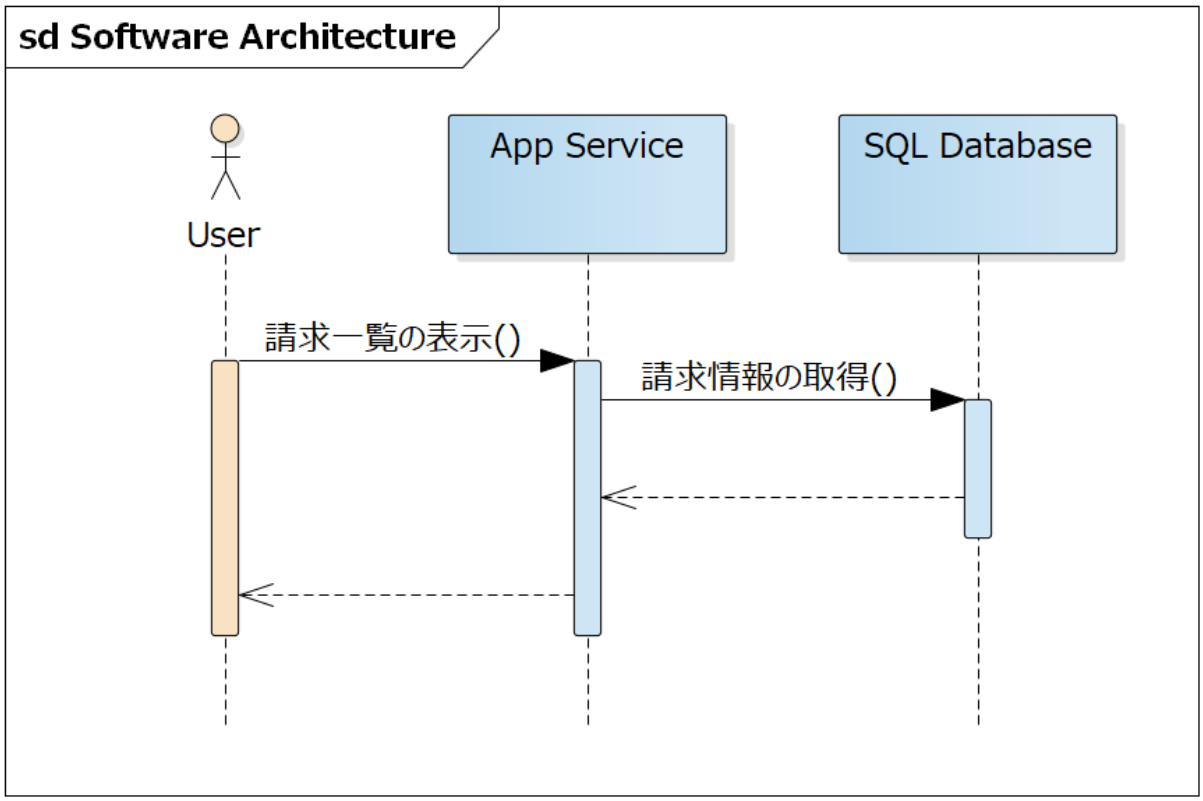
Azure上の配置モデル

Azureの利用するサービスは以下の通りです。

- Azure App Service
- Azure SQL Database
- Azure Blob Storage

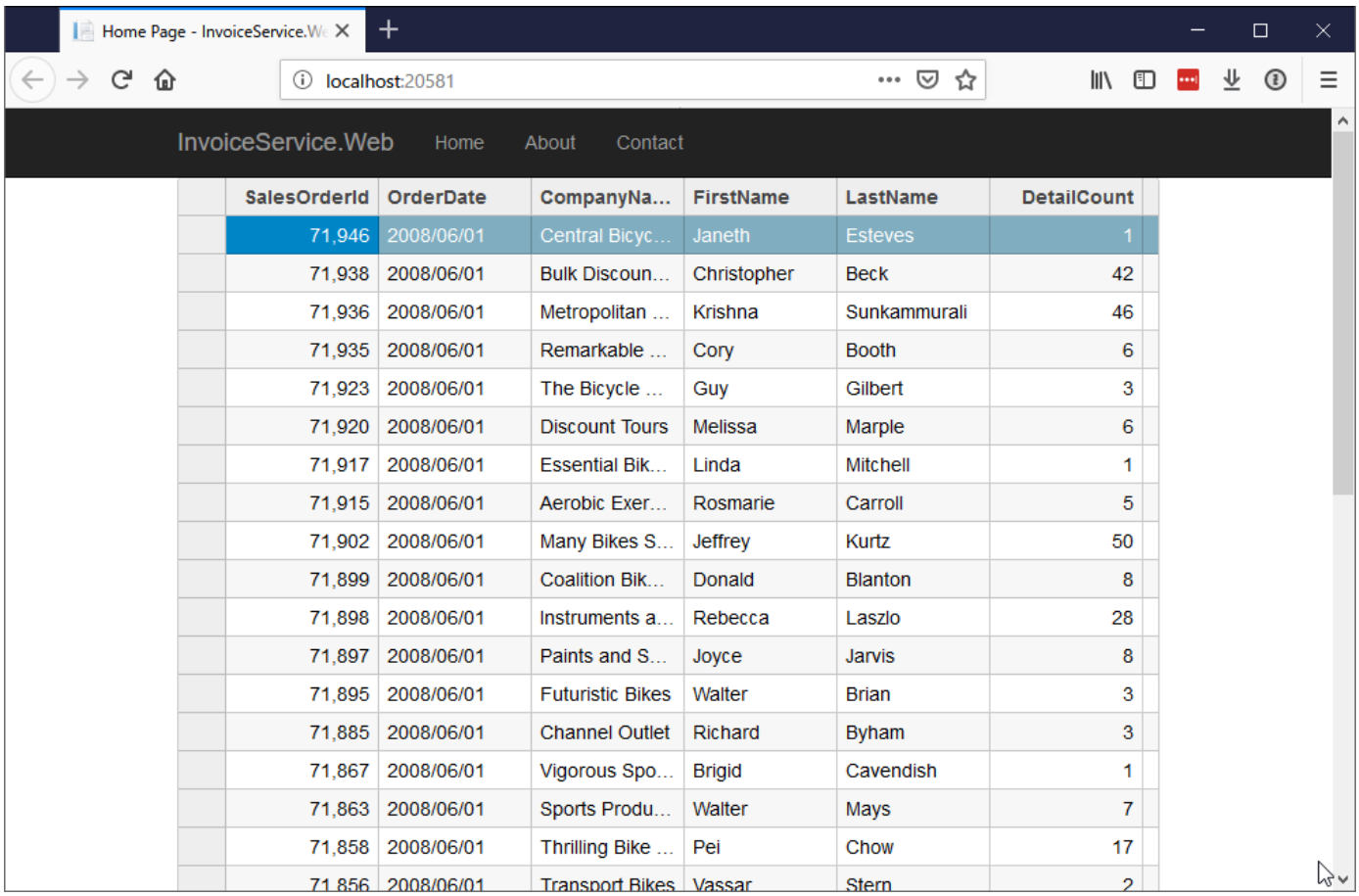
Blob StorageにExcelのテンプレートファイルを登録、そこから都度ダウンロードして帳票を生成します。テンプレートの管理は今回の趣旨からそれるため、今回はAzure Portalから直接Blob Storageにテンプレートファイルをアップロードして利用します。

これらが次のようにコラボレーションして、まずは請求の一覧を表示します。



請求の一覧を表示するシーケンス図

ユーザーがASP.NET Coreで構築したWebアプリケーションを開くと、App Serviceに配備されたWebアプリケーションはSQL Databaseへ接続し請求書の一覧情報を取得、ユーザーのブラウザに次のように表示します。

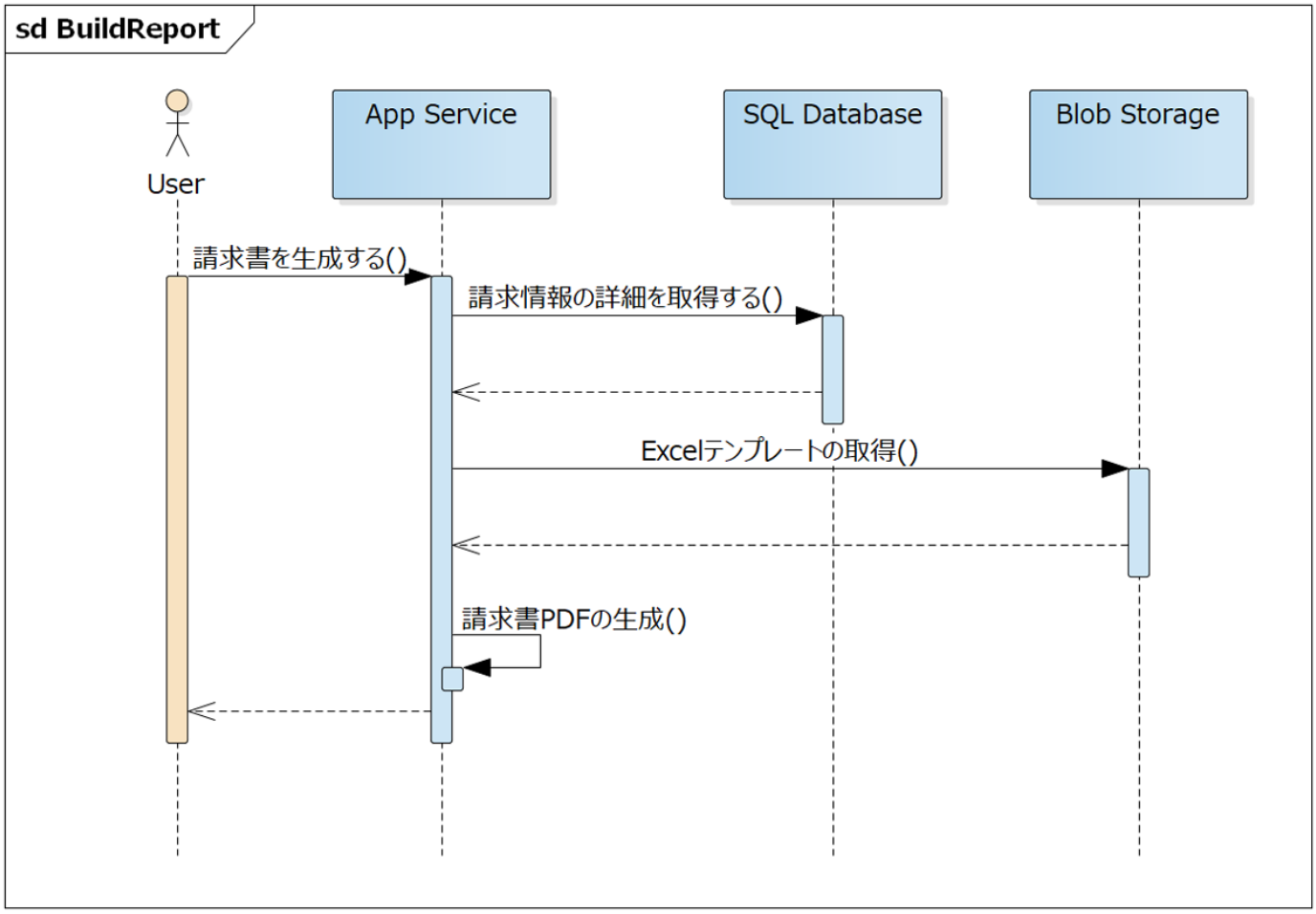


SalesOrderId	OrderDate	CompanyNa…	FirstName	LastName	DetailCount
71,946	2008/06/01	Central Bicyc…	Janeth	Esteves	1
71,938	2008/06/01	Bulk Discoun…	Christopher	Beck	42
71,936	2008/06/01	Metropolitan …	Krishna	Sunkammurali	46
71,935	2008/06/01	Remarkable …	Cory	Booth	6
71,923	2008/06/01	The Bicycle …	Guy	Gilbert	3
71,920	2008/06/01	Discount Tours	Melissa	Marple	6
71,917	2008/06/01	Essential Bik…	Linda	Mitchell	1
71,915	2008/06/01	Aerobic Exer…	Rosmarie	Carroll	5
71,902	2008/06/01	Many Bikes S…	Jeffrey	Kurtz	50
71,899	2008/06/01	Coalition Bik…	Donald	Blanton	8
71,898	2008/06/01	Instruments a…	Rebecca	Laszlo	28
71,897	2008/06/01	Paints and S…	Joyce	Jarvis	8
71,895	2008/06/01	Futuristic Bikes	Walter	Brian	3
71,885	2008/06/01	Channel Outlet	Richard	Byham	3
71,867	2008/06/01	Vigorous Spo…	Brigid	Cavendish	1
71,863	2008/06/01	Sports Produ…	Walter	Mays	7
71,858	2008/06/01	Thrilling Bike …	Pei	Chow	17
71,856	2008/06/01	Transport Bikes	Vassar	Stern	2

一覧情報をブラウザに表示

ASP.NET Core MVCのプロジェクトテンプレートから生成した画面に、ComponentOneのFlexGridを表示しています。

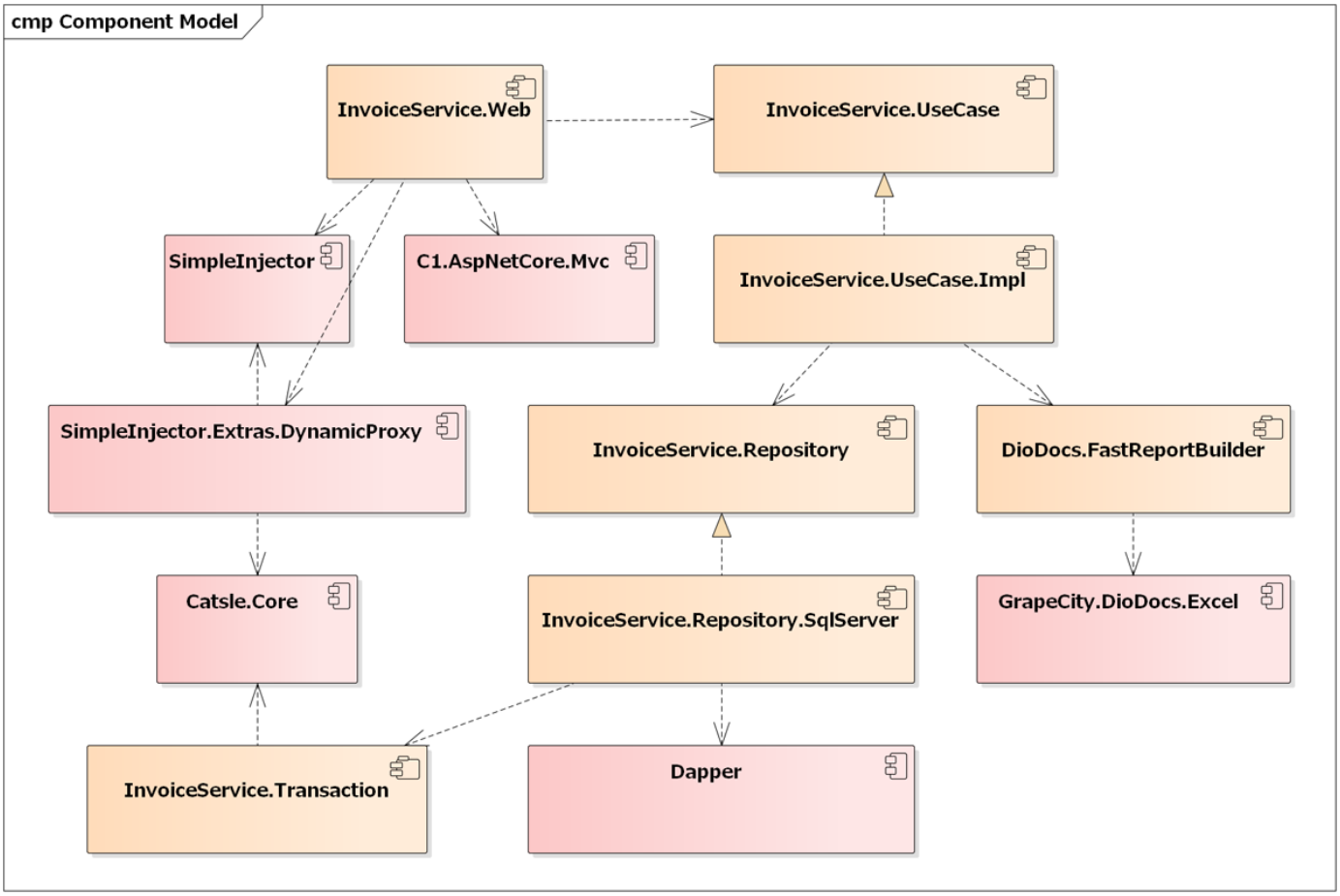
ユーザーがグリッドのいずれかの請求を選択すると、Webアプリケーションは選択された請求に対する請求書を作成するのに必要な情報をSQL Databaseから取得します。そして、請求書のテンプレートとなるExcelファイルをBlob Storageから取得後、DioDocsを利用してPDFを生成しダウンロードされます。



請求書PDFを生成するシーケンス図

ソフトウェア アーキテクチャ概要

下図がAzure App Serviceに展開する、ASP.NET Core MVCアプリケーションのコンポーネント図です。



ASP.NET Core MVCアプリケーションのコンポーネント図

薄だいだい色のコンポーネントが今回作成したものです。ピンク色のコンポーネントは既存のコンポーネントで、DioDocs含めてNuGetからインストールして利用します。

自作のコンポーネントは「InvoiceService.Web」と「DioDocs.FastReportBuilder」を除き、インターフェースパッケージと実装パッケージに分かれています。他のコンポーネントへの依存はすべてインターフェースに対して発生するように設計し、実体はDependency Injection（以降DI）パターンを利用して解決します。

DIパターンを実現するにあたり、DI ContainerとしてSimpleInjectorを利用しています。DI Containerへモジュールを登録するのは「InvoiceService.Web」で、ASP.NET Coreを起動するStarupクラスの中で実施しています。このため「InvoiceService.Web」だけは例外的に実装コンポーネントを含むすべてのコンポーネントに対して依存しています。図には線が重なり合いすぎるため記載していません。

SimpleInjectorの利用方法については以下のコードもしくはブログの記事をご覧ください。

- **Starup.cs - GitHub**
- **ASP.NET CoreでSimpleInjectorを利用する - nuits.jp blog**

個別のコンポーネントについて大まかに説明していきましょう。

DioDocs.FastReportBuilder

今回もっとも重要になる、帳票生成サービスを提供するためのコンポーネントです。構成要素はインターフェースのその実装クラスになりますが、インターフェースはDioDocsに一切依存しない形で定義しています。

ExcelからPDFを生成できる代替手段を、DioDocsを利用して作成した帳票生成ライブラリと同じインターフェースで用意できる保証はありませんが、UseCase.Implのユニットテストをする際に、DioDocsに依存していない方がテストの自由度が上がるためこのようにしています。

さて「InvoiceService.UseCase」と「InvoiceService.Repository」も同様ですが、利用者へ対して提供するサービスの「ルール」つまり「契約（Contract）」をインターフェースとして提供します。この「契約」が変更にならない限り、実装の詳細が変更になっても利用者側は影響を受けないように実装します。

インターフェースに対する実体の解決はDIコンテナで行うため、利用者は実装に対して一切の依存なく利用することができます。

DIパターンとDI Containerの詳細については、以下も併せてご覧ください。

- **Service LocatorとDependency InjectionパターンとDI Container**

本サンプルでのDIパターンの扱いについては「InvoiceService.UseCase.Impl」コンポーネントの項でクラス図とコードを見ながら解説します。

InvoiceService.Web

「InvoiceService.Web」は大きく2つの役割を実装しています。

- Presentation（View＋Controller）層の実装
- DI含む依存関係と、設定項目の管理

ビジネスロジックの呼び出し結果をもとに、ユーザーとの対話を実装します。ビジネスロジックは「InvoiceService.UseCase」コンポーネントを呼び出すことによって利用します。

InvoiceService.UseCase

ユースケースつまりビジネスロジックのインターフェースと、そのインターフェースに登場するクラスを定義しています。

InvoiceService.UseCase.Impl

「InvoiceService.UseCase」の実装コンポーネント。ユースケースを実現するためのビジネスロジックを実装しています。

Implという名前空間に違和感がある人もいるかもしれませんが、UseCaseをポリモーフィズムで拡張することは現実的に考えられず、テスト用のMockを除くと正規の実装モジュールは他に発生しえないため、このような名称にしています。

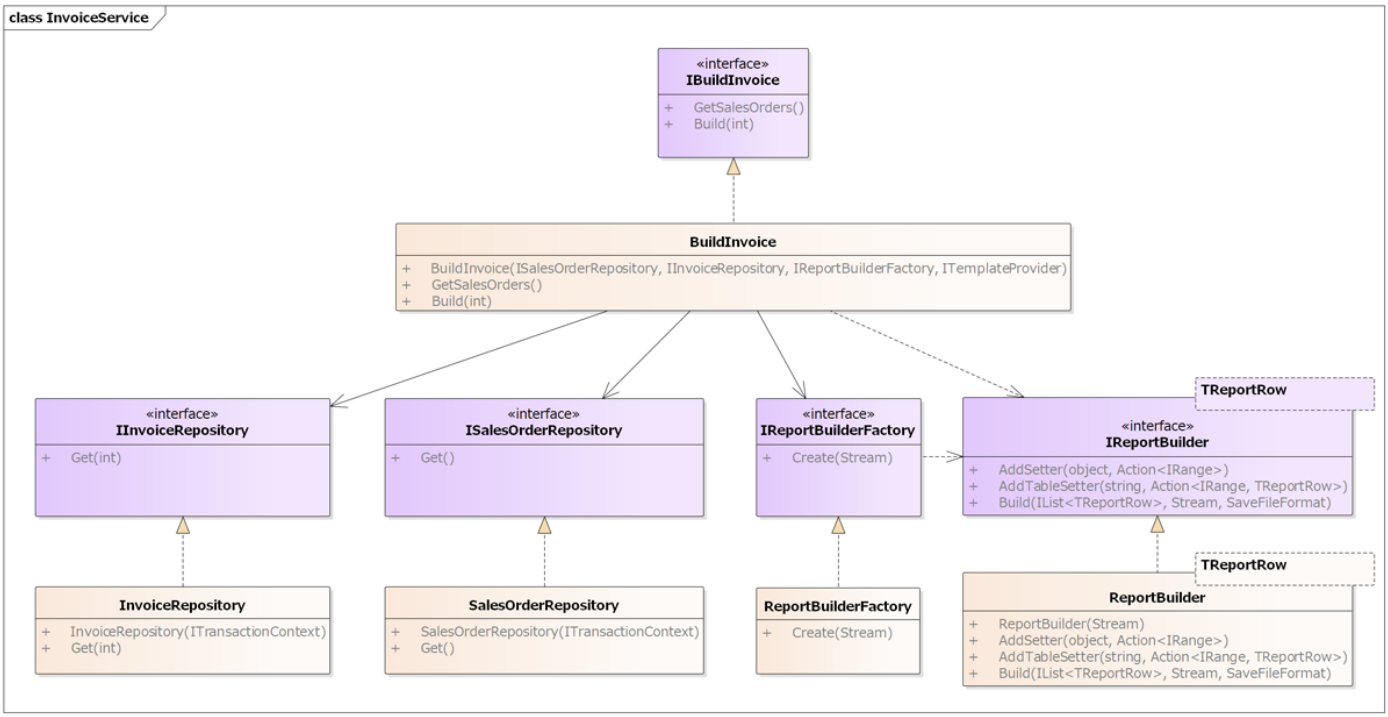
ビジネスロジックを実現するにあたり、次の2つの実装については他のコンポーネントに分離しています。

- 永続化層（今回はSQL Database）から必要な情報の取得
- 請求書オブジェクトからPDF帳票の生成

これらは「InvoiceService.Repository」と「DioDocs.FastReportBuilder」を利用することで実現しています。

「InvoiceService.UseCase.Impl」コンポーネントを詳しく見ることで、コンポーネント間の結合をどのように実現しているか理解することができます。ここはもう少し詳しく掘り下げて見ていきましょう。

下図はコンポーネントに含まれる代表的なインターフェースと実装、その依存関係を表記したモデルです。



代表的なインターフェースと実装、その依存関係

UseCaseの実装クラスが他のクラスの実装に全く依存していないことが見て取れます。私の経験上これは非常に重要なポイントです。なぜならテスト戦略に幅を持たせることができるからです。

テストを自動化していくにあたって、「単体テスト」「結合テスト」「システムテスト」のどこでどれだけテストをするのかというのは永遠の課題であると思います。すべてのフェーズで完ぺきなテストを実施することは、コストや開発期間だけでなく、保守容易性を下げる結果にもつながります。そのため最低限のコストで最高の評価を求められます。

システムの価値はユーザーにどれだけ価値を提供できるかで決定されます。したがって、価値が正しく提供できていることをより確かに確認できるテストほど価値の高いテストになります。つまりシステムテストやユーザーの受け入れテストが自動化されると、テストの価値は最大化されます。

しかしそれは何らかの課題の発見が先送りされることも意味するため、プロジェクトのリスクも増加します。

個人的にはUIのテストは統合した上で行いたいケースが多いと思っていますが、ビジネスロジックは早い段階でテストしておきたいと考えています。ここでビジネスロジック（今回はUseCase）とRepositoryやReportBuilderが疎結合になっていることの価値が発生します。

テストの価値はそれらを統合した形で実施することで最大化されます。しかし同時にRepositoryで扱うデータが大きなものであるなどして、PDFを扱う箇所をビジネスロジックも含めて一体でテストすることは困難を伴うことがあります。

PDFは内部に生成日時を持っており、生成されるバイナリが毎回異なります。そのため、自動テストの実施は困難となります。ReportBuilderの品質評価ではビジネスロジックと分離したいという要求が必ず発生するので、UseCaseとReportBuilderは疎結合になっており、テスト時にMockと差し替えが可能な設計が必要です。

Repositoryはもう少し状況が面倒です。Repositoryで取り扱うデータが小さい場合はビジネスロジックと統合してテストした方がいいですし、逆に広範なデータを扱う（多数のDBのテーブルを扱うなどの）場合は、分離してテストをした方が効率的です。

面倒なのはシステム全体でルールを統一すると、いずれかの箇所で必ず非効率なテストが発生してしまうということです。そのため統合した形でテストをするか、分離してテストをするか、選択できる余地を残しておく必要があります。

UseCaseとReporitoryの疎結合を保つ理由がここにあります。こうした話は私のブログの以下の記事でも取り扱っています。カンファレンスで登壇した内容のまとめなので、やや読みにくいかと思いますがよろしければご覧ください。

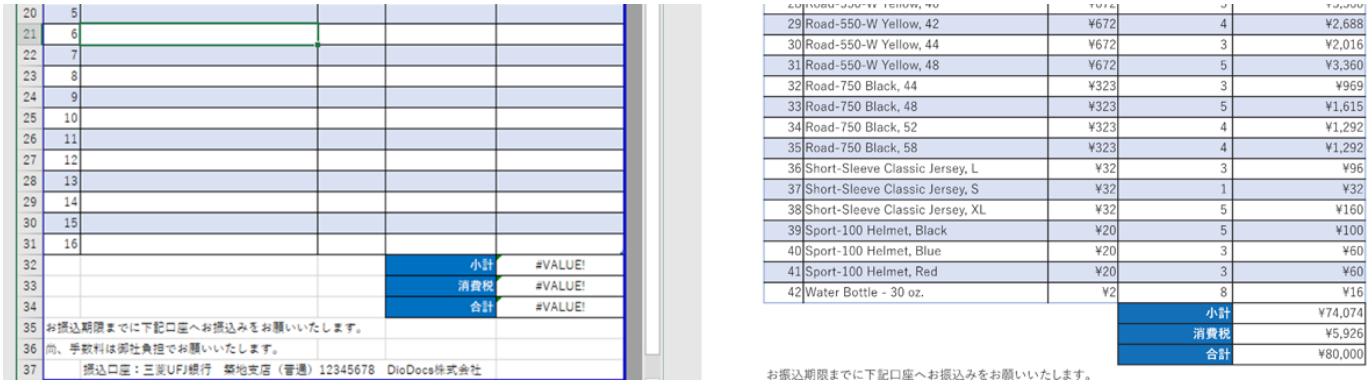
- **継続的にテスト可能な設計を考える .NET Conf 2018 Tokyo, Japan - nuits.jp blog**

このように、特に帳票生成を他のロジックから切り離して、インターフェースと実体を分離して差し替えられるようにしておくことで、開発生産性やテスト容易性を確保することができます。繰り返しになりますが、PDFは生成時に期待結果のPDFとバイナリ比較できないため、ビジネスロジックから帳票生成のロジックは分離しておかないとテストが困難になってしまうので注意してください。

さて今回はアプリケーションのアーキテクチャが本論ではないため、このくらいにしておきましょう。詳細が気になる方は、**こちら**にすべてのコードを公開しているのでご覧ください。

本稿執筆時に発見した不具合について

ところで本稿を執筆している間に、期待した動作と異なる振る舞いに遭遇しました。下図の表の、外枠の罫線をよく見てください。



29	Road-550-W Yellow, 42	¥672	4	¥2,688
30	Road-550-W Yellow, 44	¥672	3	¥2,016
31	Road-550-W Yellow, 48	¥672	5	¥3,360
32	Road-750 Black, 44	¥323	3	¥969
33	Road-750 Black, 48	¥323	5	¥1,615
34	Road-750 Black, 52	¥323	4	¥1,292
35	Road-750 Black, 58	¥323	4	¥1,292
36	Short-Sleeve Classic Jersey, L	¥32	3	¥96
37	Short-Sleeve Classic Jersey, S	¥32	1	¥32
38	Short-Sleeve Classic Jersey, XL	¥32	5	¥160
39	Sport-100 Helmet, Black	¥20	5	¥100
40	Sport-100 Helmet, Blue	¥20	3	¥60
41	Sport-100 Helmet, Red	¥20	3	¥60
42	Water Bottle - 30 oz.	¥2	8	¥16
			小計	¥74,074
			消費税	¥5,926
			合計	¥80,000

お振込期限までに下記口座へお振込みをお願いいたします。

外枠の罫線が一部水色になっている

Excelのテンプレートは黒ですが、生成されたPDFでは一部水色になってしまっています。グレープシティ社に問い合わせたところ、DioDocs for Excelの不具合が原因で発生する問題のようです。今後、修正する項目として確定したとのことでした。

そもそもExcelやWordでは、Microsoftの製品を利用していても画面の表示と印刷に差異が発生することが起こりがちです。ミリ単位でのズレや細かな表現の差異が許容できない場合は、DioDocsではなくActiveReportsといった帳票専用製品を利用したほうがいいでしょう。

ただし、必ずしもそこまでシビアに捉えるべき問題かどうかは一考の余地があります。今回のケースでも表のデザインを修正すれば枠線を黒くすることも可能でした。細かなズレも帳票に余裕を持って設計しておけば済むことがほとんどでしょう。いずれにせよ、製品選定の際にそのあたりの評価は必要になると思います。

ただ私は、このことをPR記事である本稿にぜひ載せてほしいとおっしゃったグレープシティ社に改めて信頼感を覚えました。

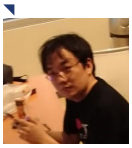
不具合がない製品はありませんが、それを隠ぺいせず、向き合って対処する姿勢がある開発元とは、継続的なパートナーとして協力していきたいと考えています。

さいごに

ここまでお付き合いいただきありがとうございました。DioDocsの魅力が十分に伝わったのではないでしょうか。DioDocsは掛け値なしに素晴らしい可能性を秘めた製品だと思います。実績が伴ってくれば、単純構造の帳票の生成領域に関して、その勢力図を大幅に塗り替えるかもしれません。

非常に面白い製品ですし、NuGetに公開されているDioDocsをライセンスなし版として試すことができます。また、この記事で作成した帳票生成サービスを改良してより多くのDioDocsの機能を試したい場合は、評価ライセンスを申請すると有効期限内に限りすべての機能が利用可能になります。

著者プロフィール



中村 充志（リコージャパン株式会社）（ナカムラ アツシ）

Microsoft MVP for Visual Studio and Development Technologies　リコージャパン株式会社 金融事業部 金融ソリューション開発部所属。　エンタープライズ領域での業務システム開発におけるアプリケーション アーキテクト・プログラマおよび中間...

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です

ページトップへ

CodeZineについて

各種RSSを配信中

プログラミングに役立つソースコードと解説記事が満載な開発者のための実装系Webマガジンです。
掲載記事、写真、イラストの無断転載を禁じます。
記載されているロゴ、システム名、製品名は各社及び商標権者の登録商標あるいは商標です。

ヘルプ
広告掲載のご案内
著作権・リンク
免責事項
会社概要
サービス利用規約
プライバシーポリシー

スタッフ募集！
メンバー情報管理
メールバックナンバー
マーケティング
エンタープライズ

人事
教育ICT
マネー・投資
ネット通販
イノベーション
セールス
クリエイティブ
ホワイトペーパー

プロジェクトマネジメント
書籍・ソフトを買う
電験3種対策講座
電験3種ネット
第二種電気工事士