関数から始めるC#クラス設計

パーシャルクラスを活用した設計

AtScopeV8.11を使って画面べったりコードから脱却

ビースラッシュ株式会社

内容

- 本書の目的

  Windows Formで自動生成したコードからクラス設計をする手順。

  ※C言語からC#に移行する場合
  いきなりオブジェクト指向のクラス設計は難しいので、
  まずはC言語風にパーシャルクラスで動くコードを作成する。
  パーシャルクラスの責務が安定した段階で、
  オブジェクト指向のクラス設計を行う手順を推奨します。

  また、Windows Formが自動生成する画面にベッタリなコードは避けましょう。
  大きい1つのかたまりになってしまい、修正箇所がわからなくなってしまいます。

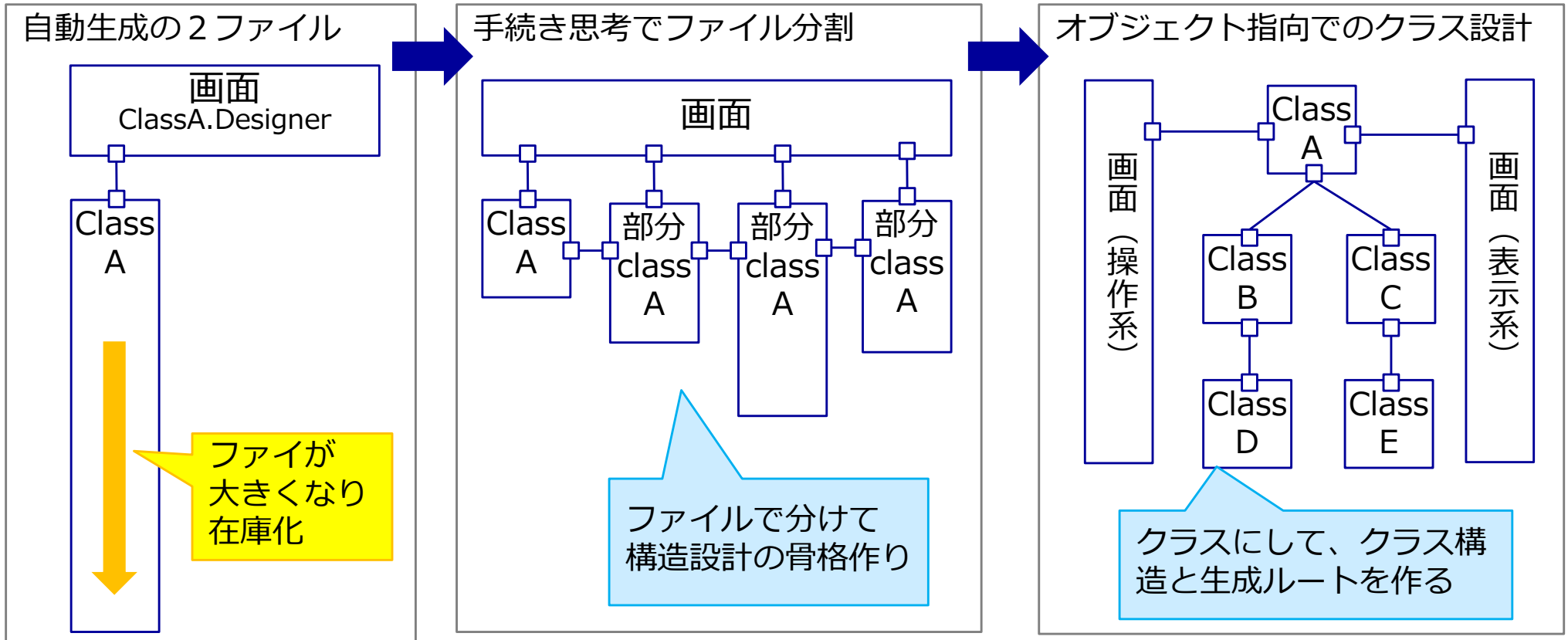
1．手続き型の構造設計
  - パーシャルクラス(部分クラス)の構造設計
  - インナークラス(内部クラス)の構造設計

2．オブジェクト指向の構造設計
  - クラス化して構造設計

関数起点のクラス設計アプローチ

コードを作りながら設計構造を形成していくアプローチ

- WindowsFormで画面を作ると、2つのファイルが自動生成されます
- そのファイル内部に処理を追加していくと巨大なファイルになってしまいます
- その解決策として、複数の「部分クラス」に分ける方法があります
  - 手続き型設計の館挙げ方で構造を作る
- 骨格となる構造が見えてきた時点でファイル（部分クラス）をクラスに置き換えます



1．AtScopeV8.11

- 図面: ファイル構造図、ファイル見取り図
- 対応言語: C#

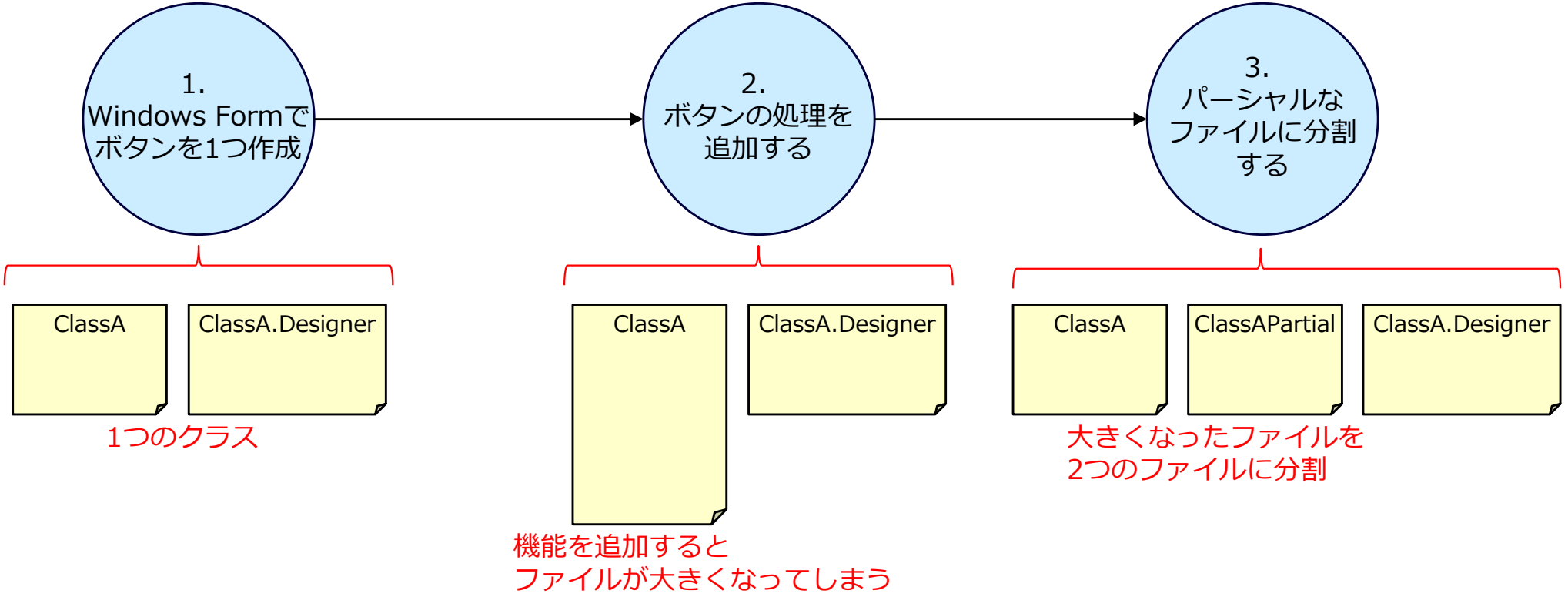
<対応項目>

#	対応項目	内容
1	パーシャルクラス対応	パーシャルクラス(部分クラス)をファイル単位で図面化するように変更。
2	インナークラス対応	インナークラス(内部クラス)をインナークラスを含んでいるクラスの一部として図面化するように変更。

1. 手続き型の構造設計

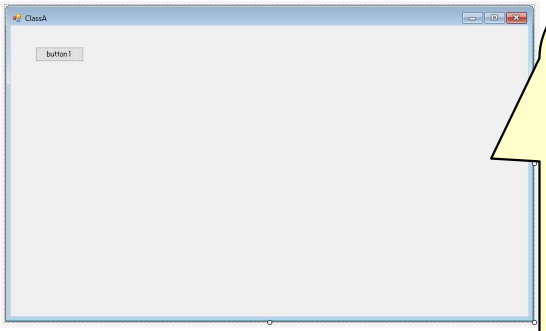
推奨する開発の流れ

- 手続き型の構造設計
  - パーシャルクラスをファイル分割して凝集度を高める
  - ファイル間の関係を構造図で理解する



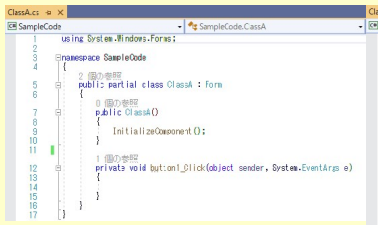
パーシャルクラスの構造設計

1. Windows Formでボタンを1つ作成

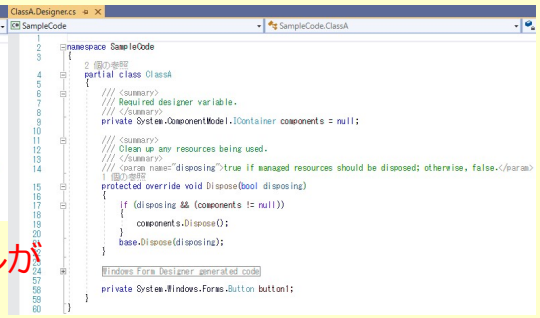


ソースコード

ClassA.cs

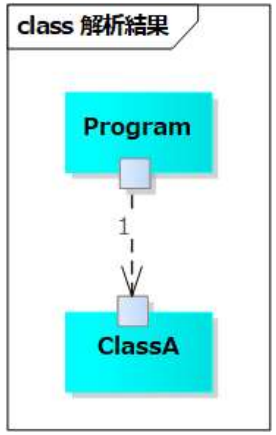


ClassA.Designer.cs



パーシャルなファイルが
自動生成される

AtScope
で図面化

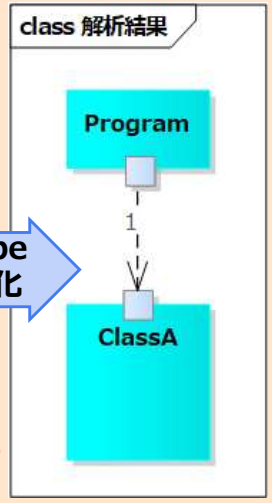


2. ボタンの処理を追加する

ClassA.cs

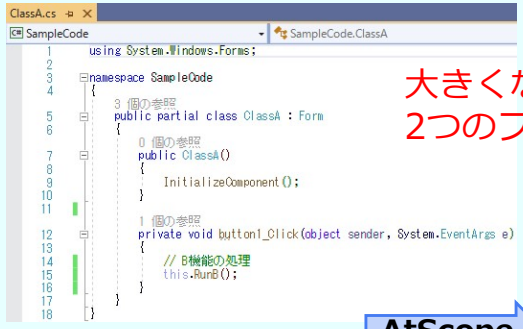
機能を追加すると
ファイルが大きくなってしまう

AtScope
で図面化



3. パーシャルなファイルに分割する

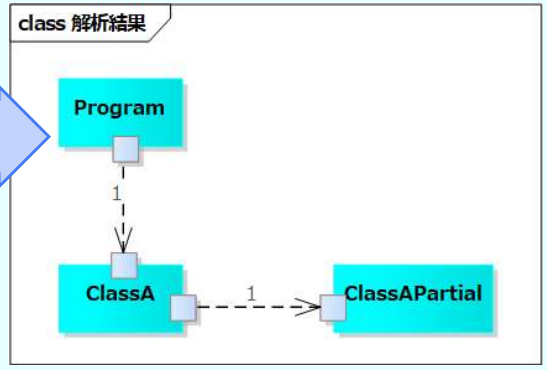
ClassA.cs



大きくなったファイルを
2つのファイルに分割

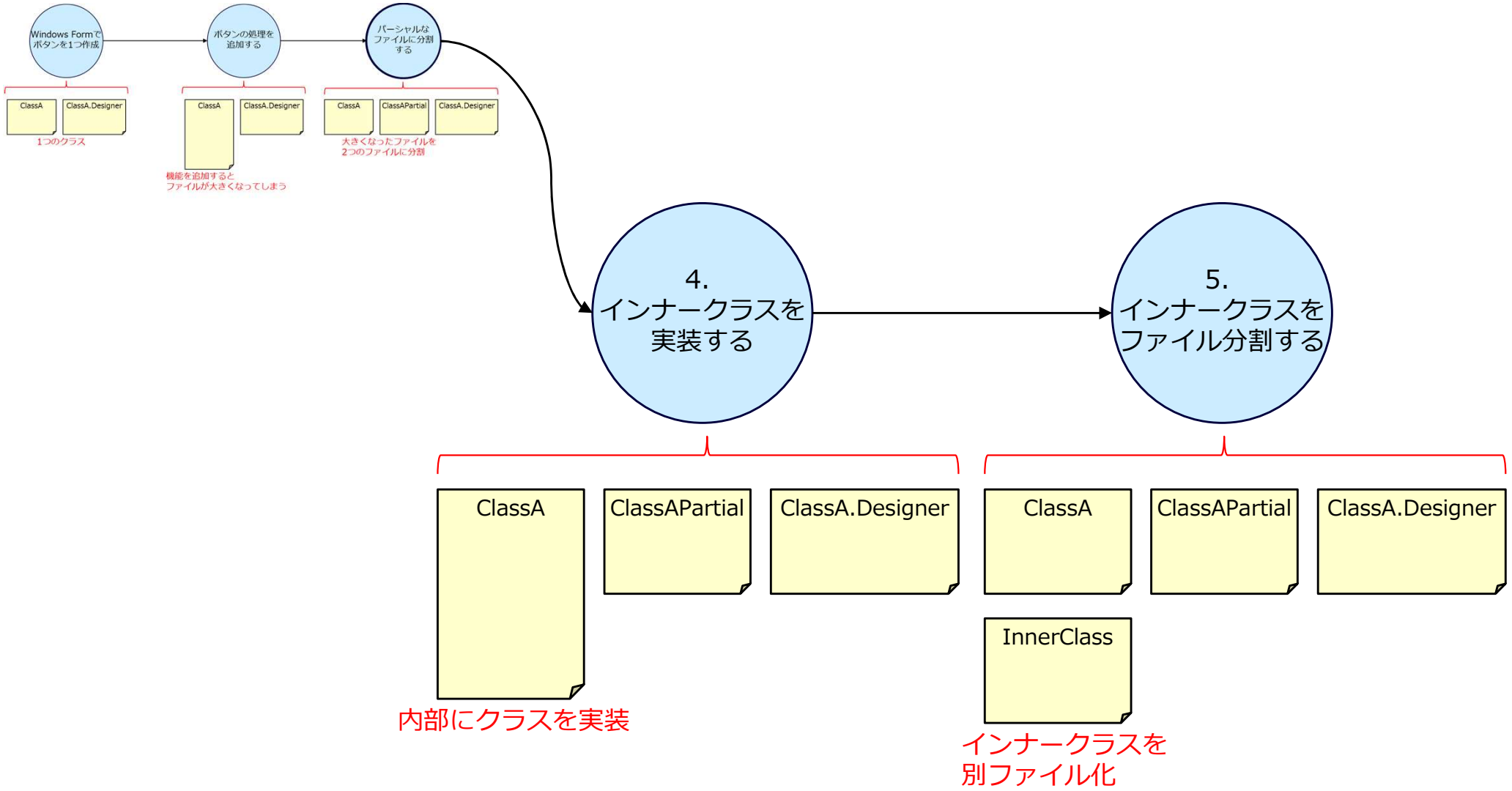
ClassAPartial.cs

AtScope
で図面化



推奨する開発の流れ

- 手続き型の構造設計
  - インナークラスをファイル分割して凝集度を高める



インナークラスの構造設計

- クラス内の構造体やクラスはインナークラスと呼ばれる
  - インナークラスは別ファイル化して凝集度を高める

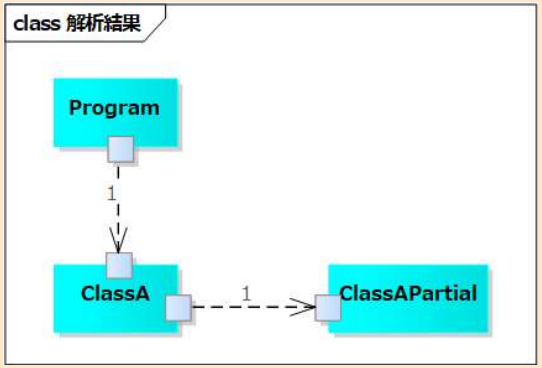
1. インナークラスを実装する

ClassA.cs

```
using System.Windows.Forms;

namespace SampleCode
{
    3 個の参照
    public partial class ClassA : Form
    {
        0 個の参照
        public ClassA()
        {
            InitializeComponent();
        }

        1 個の参照
        private void button1_Click(object sender, System.EventArgs e)
        {
            // B機能の処理
            this.RunB();

            // 内部クラスの処理
            InnerClass innerClass = new InnerClass();
            innerClass.Run();
        }

        2 個の参照
        class InnerClass
        {
            1 個の参照
            public void Run()
            {
                System.Console.WriteLine("内部クラスの処理");
            }
        }
    }
}
```

インナークラス

AtScope で図面化



2. インナークラスをファイル分割する

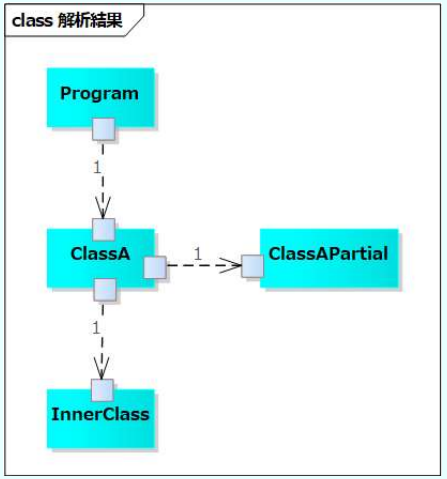
ClassA.cs

```
using System.Windows.Forms;

namespace SampleCode
{
    3 個の参照
    public partial class ClassA : Form
    {
        0 個の参照
        public ClassA()
        {
            InitializeComponent();
        }

        1 個の参照
        private void button1_Click(object sender, System.EventArgs e)
        {
            // B機能の処理
            this.RunB();

            // 内部クラスの処理
            InnerClass innerClass = new InnerClass();
            innerClass.Run();
        }
    }
}
```

InnerClass.cs

```
namespace SampleCode
{
    2 個の参照
    public class InnerClass
    {
        1 個の参照
        public void Run()
        {
            System.Console.WriteLine("内部クラスの処理");
        }
    }
}
```

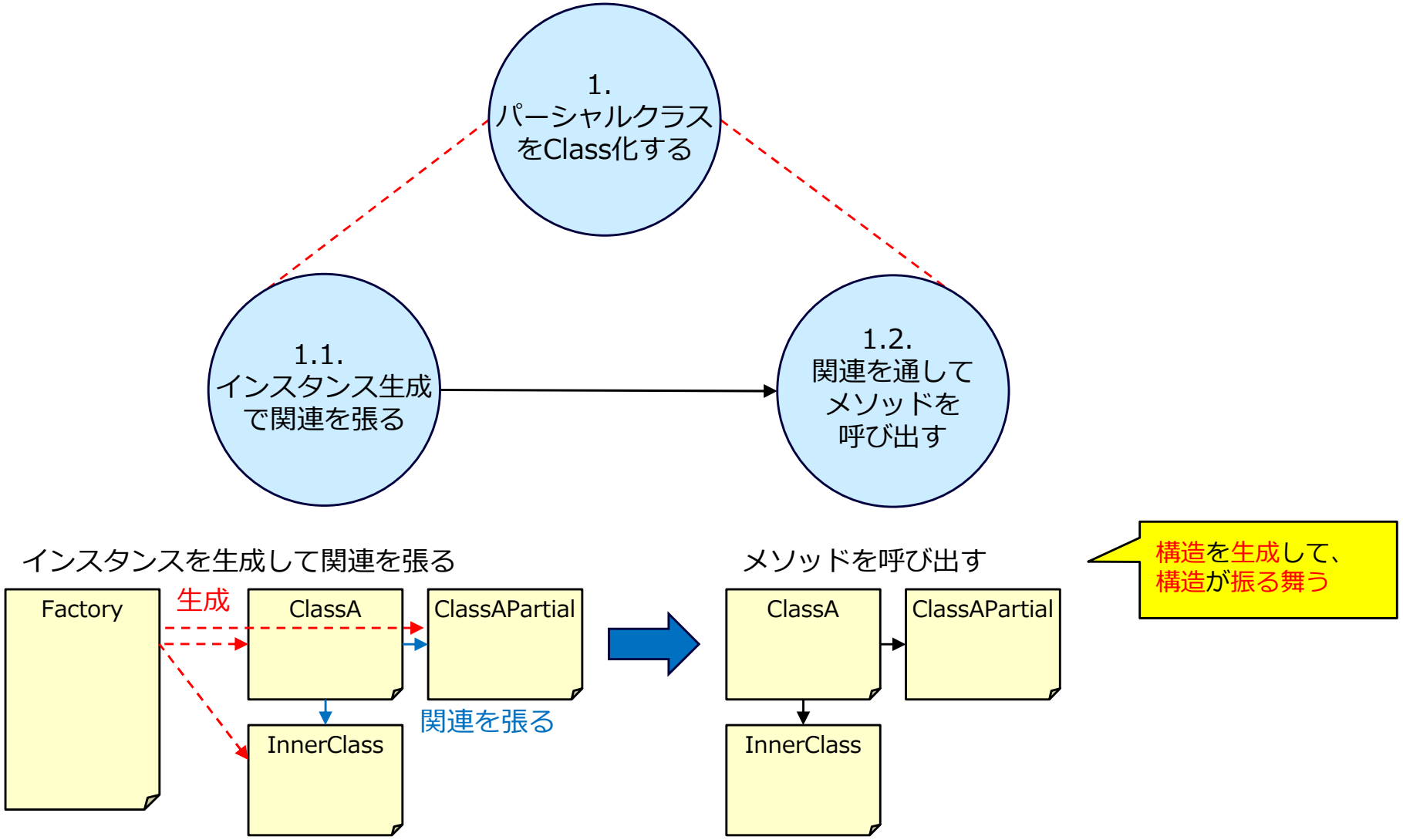
AtScope で図面化



2. オブジェクト指向の構造設計

推奨する開発の流れ

- オブジェクト指向の構造設計
  - クラス分割して凝集度を高める



クラス化して凝集度を上げる

1. インスタンス生成で関連を張る

Factory.cs

```
namespace SampleCode
{
    2 個の参照
    public class Factory
    {
        1 個の参照
        public ClassA Create()
        {
            // ClassAに関連を張るクラス
            ClassB classB = new ClassB();
            InnerClass innerClass = new InnerClass();

            // 関連を張ってClassAを生成
            ClassA classA = new ClassA(classB, innerClass);

            return classA;
        }
    }
}
```

ClassB.cs

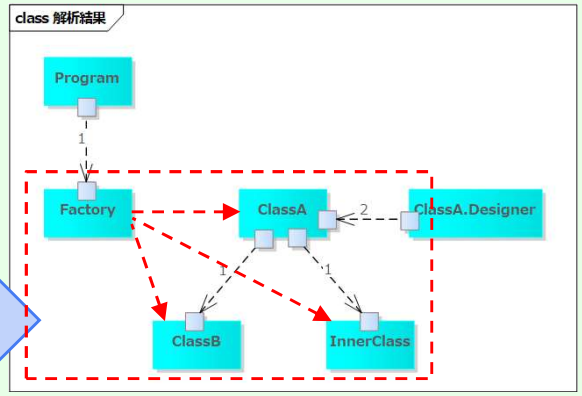
```
namespace SampleCode
{
    4 個の参照
    public class ClassB
    {
        1 個の参照
        public void RunB()
        {
            System.Console.WriteLine("B機能の処理");
        }
    }
}
```

InnerClass.cs

```
namespace SampleCode
{
    4 個の参照
    public class InnerClass
    {
        1 個の参照
        public void Run()
        {
            System.Console.WriteLine("内部クラスの処理");
        }
    }
}
```

ClassAに必要なクラス(ClassB、InnerClass)
を渡して生成

AtScope
で図面化



2. 関連を通してメソッドを呼び出す

Program.cs

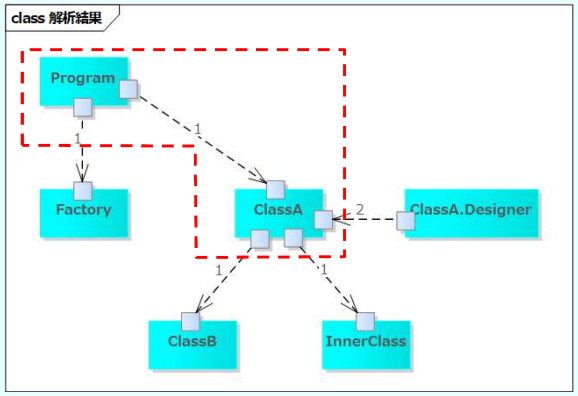
```
using System;
using System.Windows.Forms;

namespace SampleCode
{
    0 個の参照
    static class Program
    {
        /// <summary>
        /// アプリケーションのメイン エントリ ポイントです。
        /// </summary>
        [STAThread]
        0 個の参照
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            //Application.Run(new ClassA());

            // ClassAを生成
            Factory factory = new Factory();
            ClassA classA = factory.Create();

            // ClassAを実行
            classA.ShowDialog();
        }
    }
}
```

ClassA.cs

```
using System.Windows.Forms;

namespace SampleCode
{
    public partial class ClassA : Form
    {
        private ClassB classB;

        private InnerClass innerClass;


        public ClassA()
        {
            InitializeComponent();
        }

        1 個の参照
        public ClassA(ClassB classB, InnerClass innerClass)
            : this()
        {
            this.classB = classB;
            this.innerClass = innerClass;
        }

        1 個の参照
        private void button1_Click(object sender, System.EventArgs e)
        {
            // B機能の処理
            this.classB.RunB();

            // 内部クラスの処理
            this.innerClass.Run();
        }
    }
}
```

ClassAを
メソッド呼び出しする

AtScope
で図面化



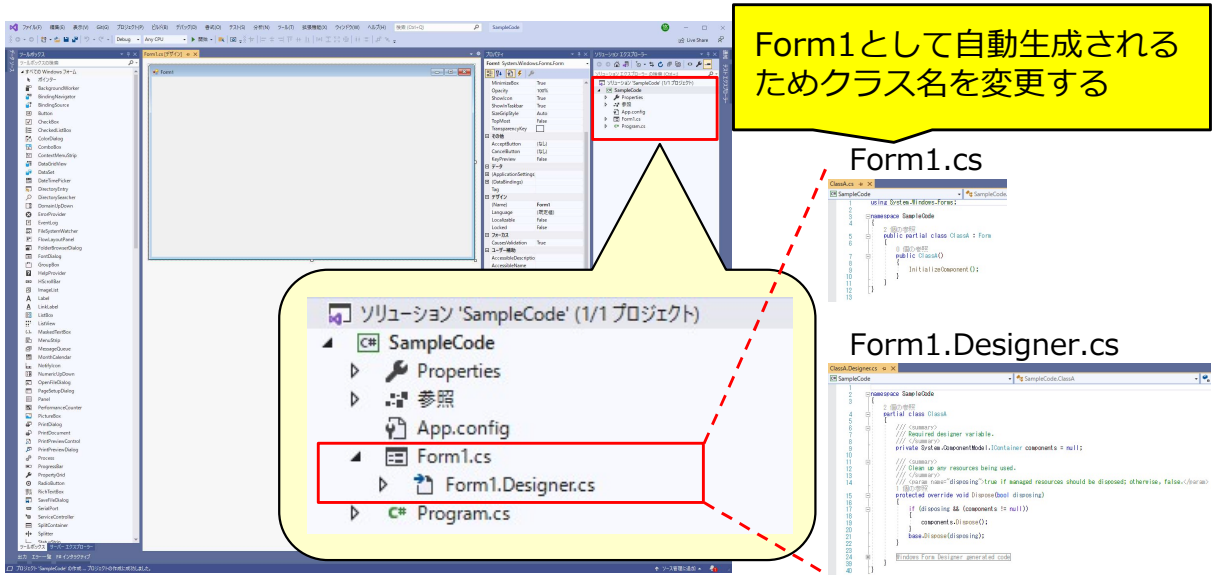
3. 補足

補足：Windows Formの操作

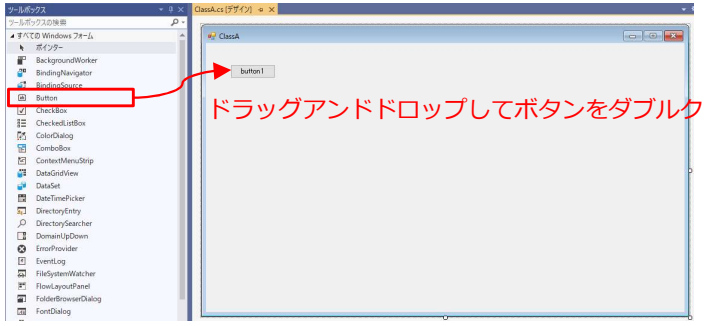
- Windows Formプロジェクトを作ると2つのファイルが作成される

1. Windows Formプロジェクトを作成

Windows Form画面と2つのファイルが生成される



2. ボタンとボタン押下時の処理を実装



ClassAのパーシャルな2ファイルが作成

ClassA.cs

```
using System.Windows.Forms;

namespace SampleCode
{
    2 個の参照
    public partial class ClassA : Form
    {
        0 個の参照
        public ClassA()
        {
            InitializeComponent();
        }

        1 個の参照
        private void button1_Click(object sender, System.EventArgs e)
        {

        }
    }
}
```

ボタン押下時の処理を記述

ClassA.Designer.cs

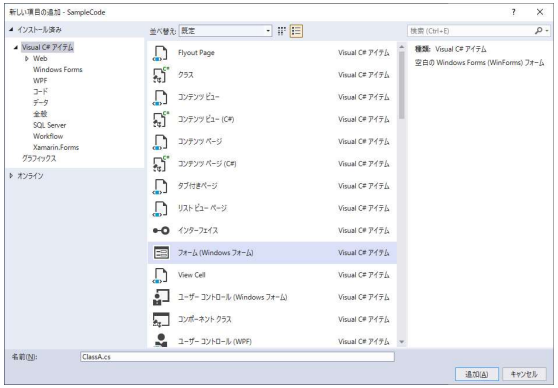
```
namespace SampleCode
{
    2 個の参照
    partial class ClassA
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        1 個の参照
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code

        private System.Windows.Forms.Button button1;
    }
}
```

このメソッドに機能追加が集中してしまい、ファイルが大きくなってしまうことが多い。

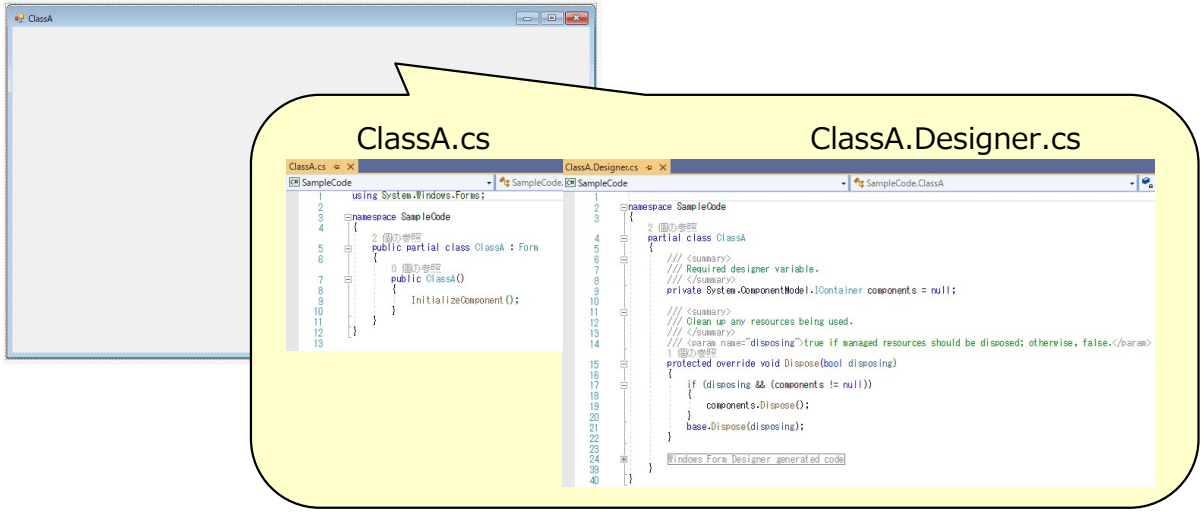
補足: Windows Formの操作

- Windows FormでClassAを作ると2つのファイルが作成される

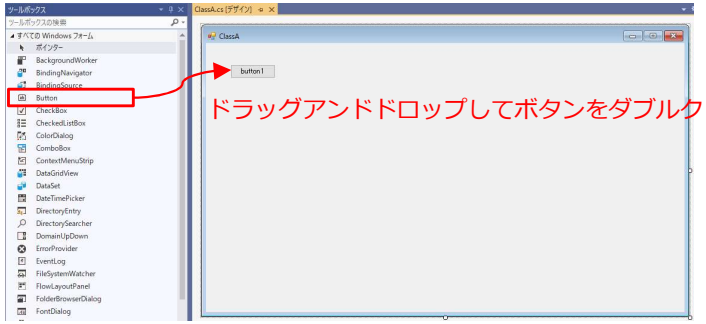
1. Windows Formクラスを作成



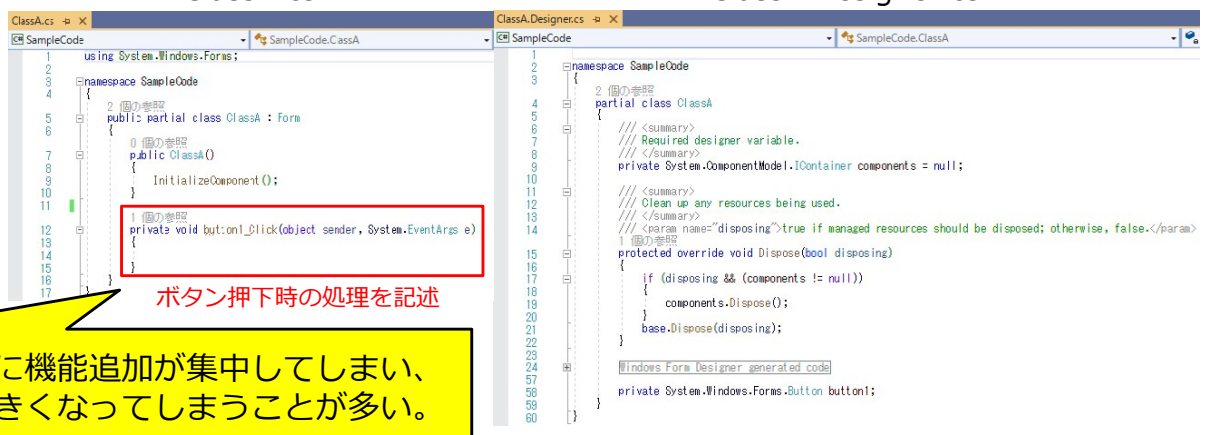
Windows Form画面と2つのファイルが生成される



2. ボタンとボタン押下時の処理を実装



ClassAのパーシャルな2ファイルが作成



このメソッドに機能追加が集中してしまい、
ファイルが大きくなってしまうことが多い。

用語解説

#	対応項目	内容
1	パーシャルクラス (部分クラス)	クラスの定義を複数のファイルに跨いで記述できる言語仕様。
2	インナークラス (内部クラス)	クラスの内部に構造体またはクラスを記述できる言語仕様。
3	凝集度	モジュールの単一責務であるかを示す指標。 1つの責務のみを持つモジュールは凝集度が高い。
4	クラス	名称と属性とメソッドを持つモジュール。
5	インスタンス	クラスが実体化されて属性の値を持ちメソッド呼び出しができるようになった状態を指す。
6	ファクトリ	インスタンスを生成するクラス。 生成と処理の流れの混在を防止するための設計。