CM-02 インスタンス管理機能

■ 概要

本機能は、TERASOLUNA フレームワークの共通基盤となる機能であり、Unity Application Block[1](以下 Unity AB)を利用して、TERASOLUNA フレームワーク上で動作するインスタンスを管理するための DI/AOP コンテナを提供する。

- DI (Dependency Injection) のサポート
  Unity AB の DI/AOP コンテナである UnityContainer により、ソースコードまたは構成ファイルで依存関係を設定することで、オブジェクト間の依存関係をオブジェクトの外部から注入することができる。本機能では UnityManager クラスが、フレームワークが利用する全ての UnityContainer を統合管理している。
  DI の主な利用箇所として、アーキテクトによる TERASOLUNA フレームワーク の拡張を想定している。TERASOLUNA フレームワークは、各コンポーネントをインタフェースで定義し DI により依存関係を注入しているため、フレームワークの各機能を拡張し容易に差し替え可能な作りになっている。
  なお、業務開発者が DI を多用するような開発スタイルは DI 設定の記述量も膨大になり生産性や品質を落とす恐れもあるため推奨しない。プロジェクト独自の利用方法を検討する場合はアーキテクトがプロジェクトの状況を考慮して適用範囲を決定すること。

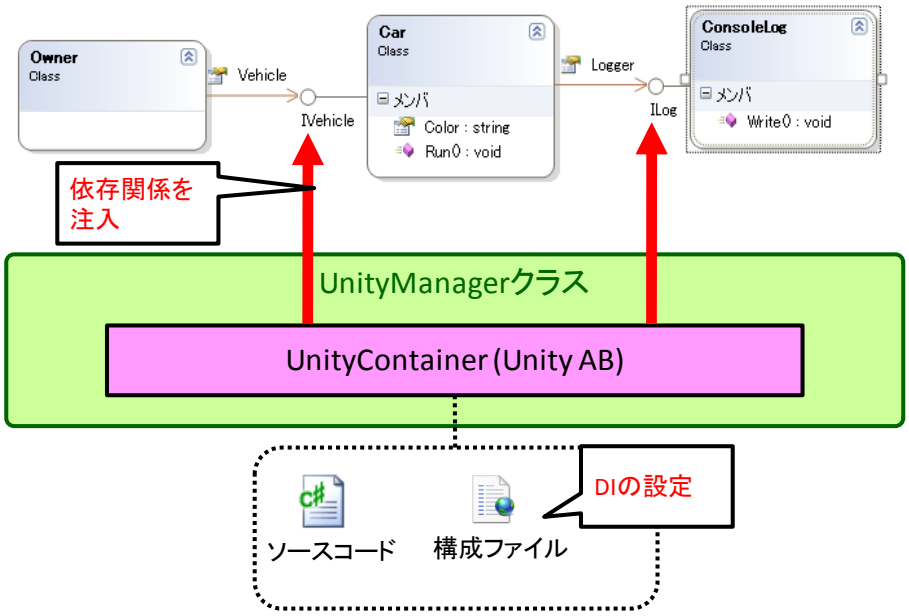

図 1 DI の動作概念図

[1] Microsoft patterns & practices が提供する DI/AOP を実現するための Application Block。
http://msdn.microsoft.com/en-us/library/dd203104.aspx

● AOP(Aspect Oriented Programming) のサポート

Unity AB の標準機能が持っているインターセプタを利用し AOP を実現する。

Unity AB の標準機能では、ソースコードまたは構成ファイルにより AOP の対象クラスを指定可能であるが、本機能は Unity AB を拡張し、SetDefaultInterceptor(または SetInterceptor)カスタム属性を付与することで指定可能である。

SetDefaultInterceptor(SetInterceptor)属性には、UnityAB が提供するインターセプタの中から適切なものを選択して指定する。

また、AOP を使って織り込む処理は、ICallHandler インタフェースを実装する。AOP 対象クラスに対して、対象メソッド等の適用範囲を指定するには以下のどちらかの方法で設定する。

- ICallHandler 実装クラスに対応するカスタム属性を、HandlerAttribute を継承して実装し各対象メソッドに付与する
- 対象を決定するマッチングルールを実装した IMatchingRule 実装クラスと、ICallHandler 実装クラスを指定して、コンテナ全体に適用するポリシーを設定する。

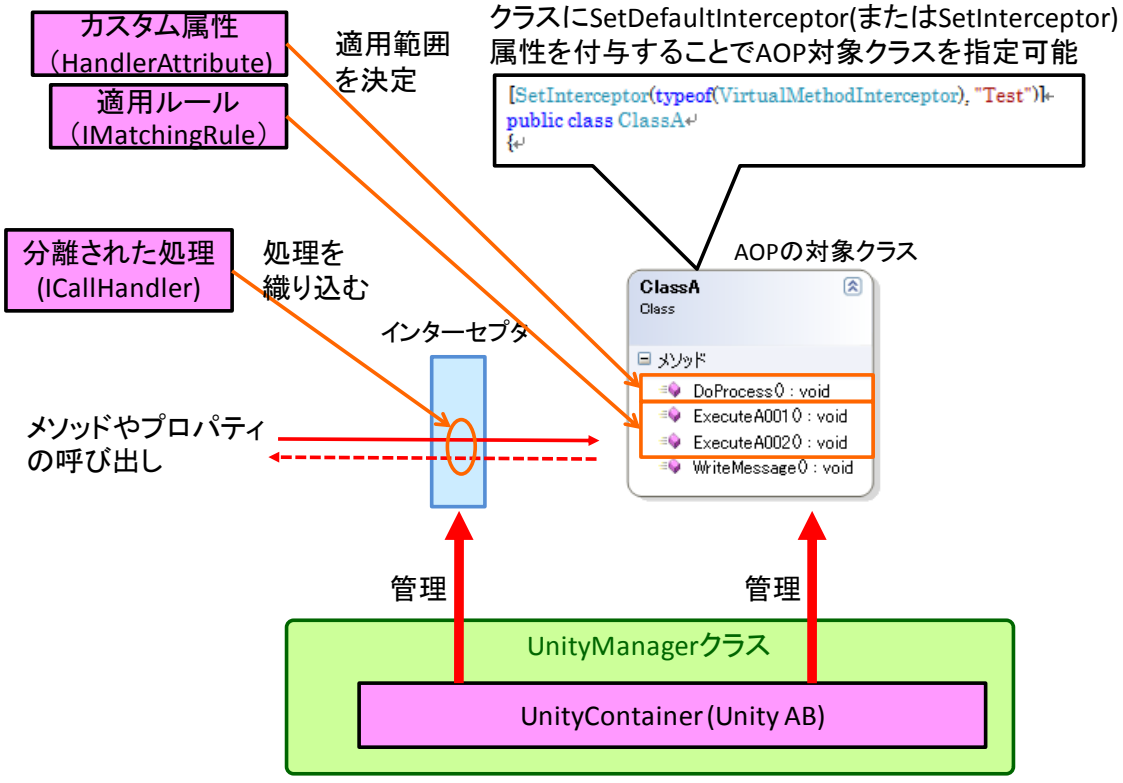

図 2 AOP 機能の動作概念図

- 構成ファイルの分割による分散開発への対応
  通常、Unity AB を利用する場合、アプリケーション構成ファイル(App.config)や Web 構成ファイル(Web.config)に記述した Unity の設定をロードして UnityContainer を利用する。
  しかし、開発プロジェクトの規模によっては、Visual Studio のプロジェクト(アセンブリ)が複数に分かれるケースも想定される。
  そこで、本機能は、フレームワーク、業務共通、業務個別といったアプリケーション単位(システムの規模に応じて、Visual Studio のソリューションやプロジェクトを分割し開発グループ間や業者間で作業分担する単位)ごとに、構成ファイルを分割可能にする。Unity に関する設定は、App.config や Web.config には記述せず、表 1 の分割された構成ファイルに記述する。これらの構成ファイルは別々のコンテナによってロードされ、UnityManager クラスで統合管理する。

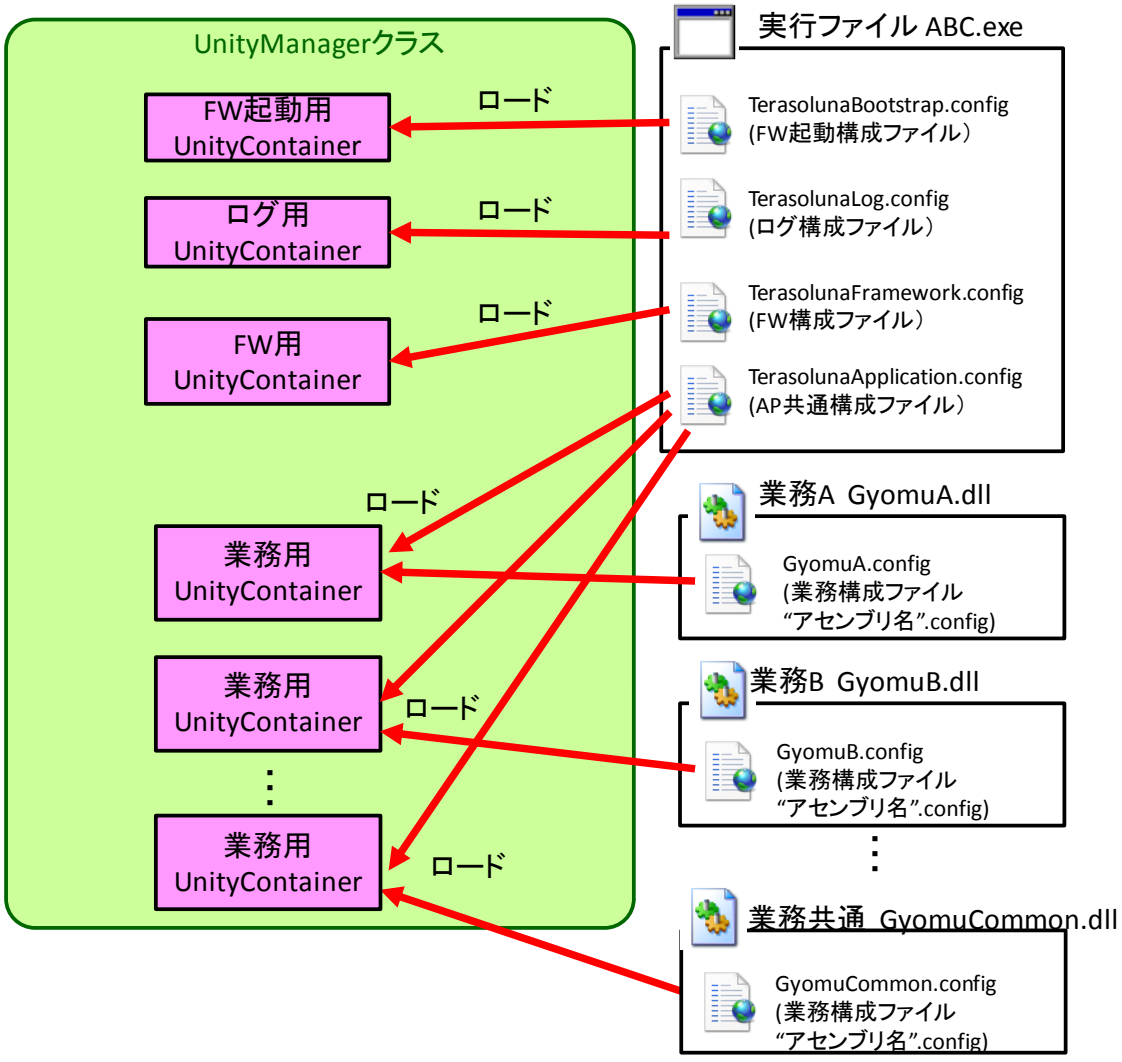

図 3 構成ファイル分割のイメージ

表 1 本機能が使用する構成ファイルとコンテナ

項番	構成ファイル名	ファイルの位置づけ	利用者	コンテナ
1	FW 構成ファイル (TerasolunaFramework .config)	業務に依存しないフレームワーク機能に関する設定ファイル。	アーキテクト	フレームワーク用 UnityContainer
2	AP 共通構成ファイル (TerasolunaApplication .config)	各業務プロジェクトに共通する機能に関する設定ファイル。	業務共通開発者 アーキテクト	業務用 UnityContainer
3	業務構成ファイル ("アセンブリ名".config)	各業務プロジェクト(Visual Studio のプロジェクト=アセンブリ)固有の設定ファイル。	業務個別開発者 業務共通開発者	
4	FW 起動構成ファイル (TerasolunaBootstrap.config)	「CM-01 アプリケーション起動・終了機能」や「CL-05 クライアントエラーハンドリング機能」、「 SV-03 サーバエラーハンドリング機能」が利用する FW 起動に関する設定ファイル。	アーキテクト	FW 起動用 UnityContainer
5	ログ構成ファイル (TerasolunaLog.config)	「CM-06 ログ出力機能」が利用する設定ファイル。	アーキテクト	ログ用 UnityContainer

- 「デザインモード」と「実行時モード」の構成ファイルロードクラスの切り替え
  TERASOLUNA フレームワーク を使ったアプリケーションを実行した場合(「実行時モード」と呼ぶ)、UnityContainer は構成ファイルを取得しロードする。この時 System.Configuration.ConfigurationManager クラスを使って、実行可能ファイル(exe や DLL)が存在するフォルダにある構成ファイルを取得している。
  一方、TERASOLUNA フレームワーク は、Visual Studio 上での開発時にも動作し(「デザインモード」と呼ぶ)、構成ファイルの設定内容をもとにプロパティエディタで候補一覧をプルダウン表示するなどの Visual Studio のデザイナ拡張機能を提供しており、開発者の生産性向上を図っている。
  しかし、「デザインモード」でも UnityAB が「実行時モード」と同じような動作をすると、その時点で実行中のアプリケーションは「Visual Studio」であるため、Visual Studio の実行可能ファイル(devenv.exe)があるフォルダの構成ファイルを取得しようとし、正しく動作しない。
  そこで、本機能は、アプリケーションの開発時(デザインモード)と実行時(実行時モード)で UnityContainer のロードクラス(UnityContainerLoader クラス)を切り替えて、デザインモード時には、開発中のプロジェクトが存在するフォルダの構成ファイルを取得するようにしている。
  TERASOLUNA フレームワーク が「デザインモード」、「実行時モード」のどちらで実行されるかは、「CM-01 アプリケーション起動・終了機能」が管理しており、起動モードによってロードクラス(IUnityContainerLoader 実装クラス)を切り替えることで、開発者はモード切

り替えなどの特別な操作をすることなく、対象アプリケーションの構成ファイルを常にロードすることができる。

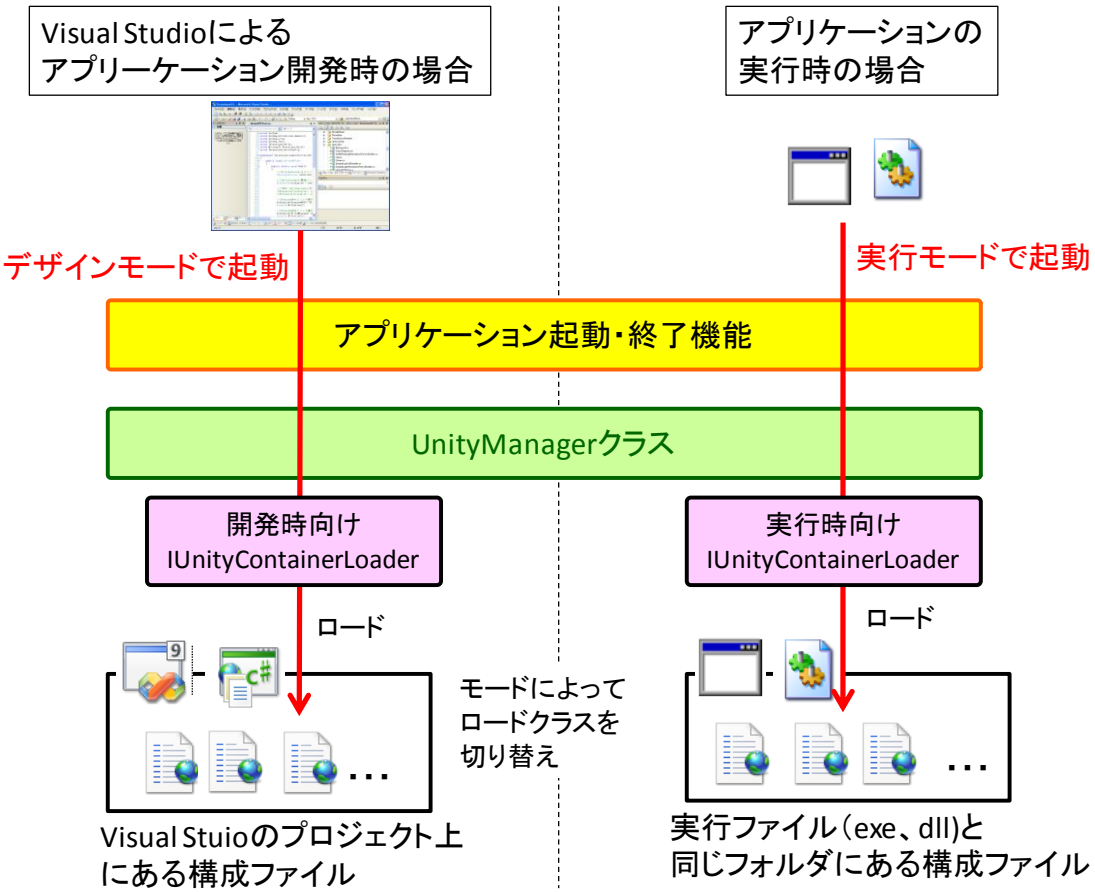

図 4 実行時モードとデザインモードの動作概念図

■ 使用方法

◆ 構成ファイルの作成(Windows Forms アプリケーションの場合)

TERASOLUNA フレームワーク を使った Windows Forms アプリケーション(クライアントAP)の場合、以下の構成ファイルが必要となる。
表 2 の通り、TerasolunaFramework.config 、TerasolunaApplication.config 、TerasolunaBootstrap.config、TerasolunaLog.config は、システム全体の設定となるため、VisualStudio 上ではエントリポイントのプロジェクトでマスタを管理する。
業務構成ファイル("アセンブリ名".config)は、各業務プロジェクトに閉じた設定となるため、対象の業務プロジェクトでマスタ管理する。

表 2 構成ファイルの配置方法(Windows Forms アプリケーションの場合)

項番	構成ファイル	Visual Studio 上でのファイルの配置		
		マスタファイルの配置場所	各業務プロジェクトへの配置(デザインモードで使用)	エントリポイントのプロジェクトへの配置(実行時モードで使用)
1	FW 構成ファイル (TerasolunaFramework.config)	エントリポイントのプロジェクト	○ (マスタをコピー)	○
2	AP 共通構成ファイル (TerasolunaApplication.config)	エントリポイントのプロジェクト	○ (マスタをコピー)	○
3	業務構成ファイル ("アセンブリ名".config)	業務プロジェクト	○ (対象業務のみ)	○ (マスタをコピー)
4	FW 起動構成ファイル (TerasolunaBootstrap.config)	エントリポイントのプロジェクト	-	○
5	ログ構成ファイル (TerasolunaLog.config)	エントリポイントのプロジェクト	-	△[2]

構成ファイルの作成方法としては、TERASOLUNA フレームワーク が提供するカスタムプロジェクトテンプレートを利用することで、プロジェクト作成時に構成ファイルのひな形も一緒に生成する。

Windows Forms アプリケーションの場合、エントリポイントとなるプロジェクトについては、「起動アプリケーション」プロジェクトテンプレートを利用し作成する。
プロジェクト生成時に TerasolunaFramework.config 、TerasolunaApplication.config 、TerasolunaBootstrap.config のひな形が生成される。

[2] TerasolunaLog.config は、「CM-06 ログ出力機能」の拡張を実施した場合に必要となる。標準提供のログ出力機能を利用する場合は不要である。

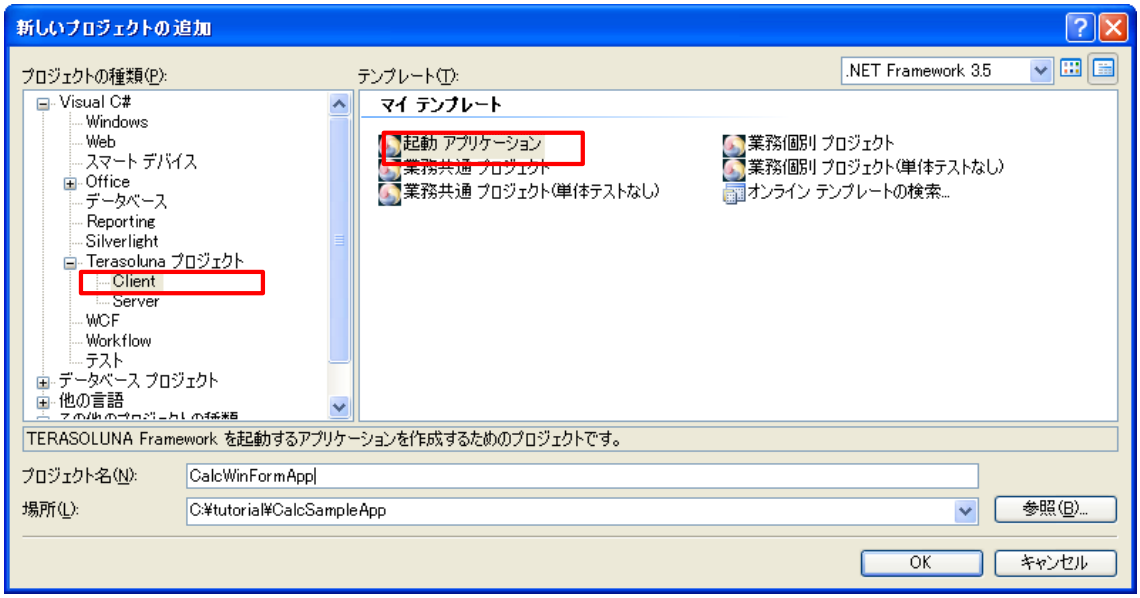

図 5 「起動アプリケーション」プロジェクトテンプレート

また、各業務プロジェクトについては、業務個別プロジェクトテンプレート[3](または業務共通プロジェクトテンプレート)を利用し作成する。
プロジェクト生成時に、ファイル名が「"アセンブリ名" + .config」の構成ファイルのひな形が生成される。

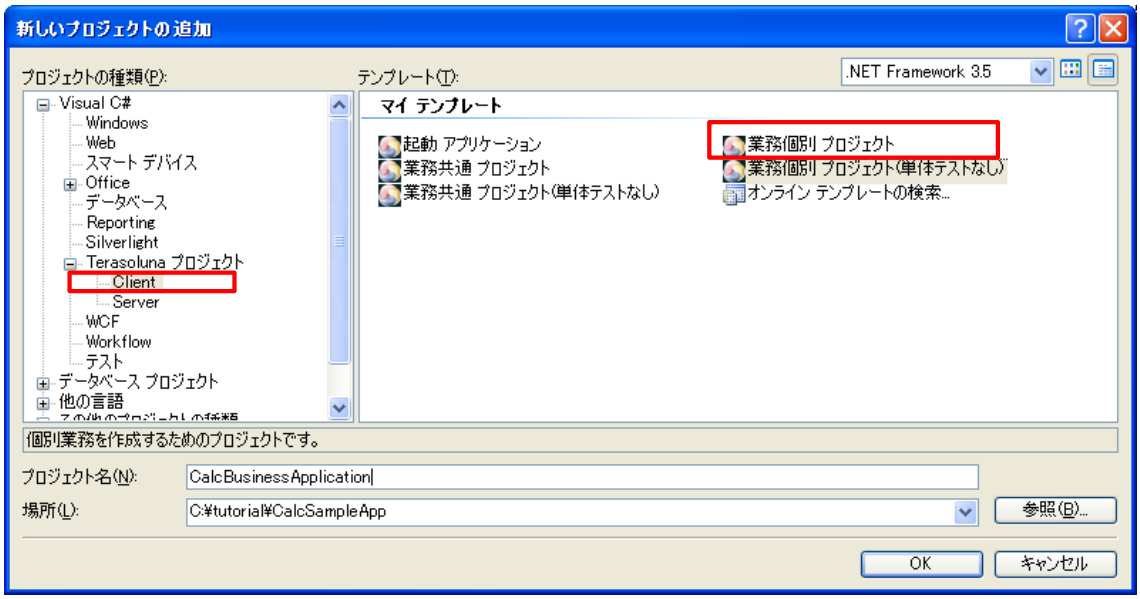

図 6 「業務個別」プロジェクトテンプレート

[3] デフォルトのテンプレートでは、単体テストプロジェクトも同時に生成される。
単体テストプロジェクトなしのテンプレートも提供している。

また、作成済みのプロジェクトに構成ファイルを追加したい場合は、TERASOLUNA フレームワーク が提供するカスタムアイテムテンプレートにより作成することもできる。

TerasolunaFramework.config は、FW 構成ファイルテンプレートにより作成する。

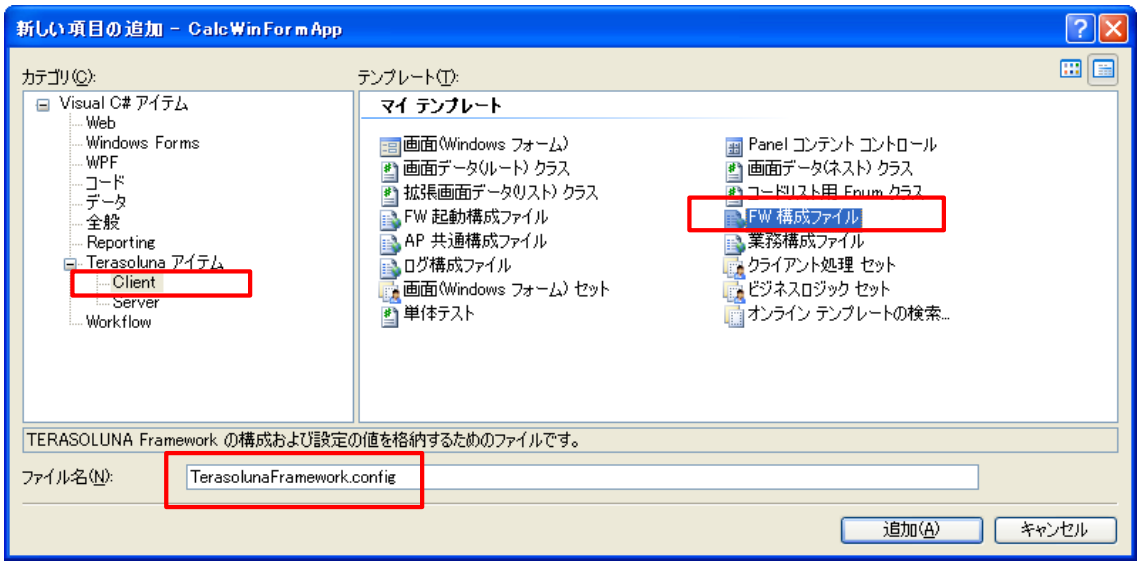

図 7「FW 構成ファイル」テンプレート

TerasolunaApplication.config は、AP 共通構成ファイルテンプレートにより作成する。

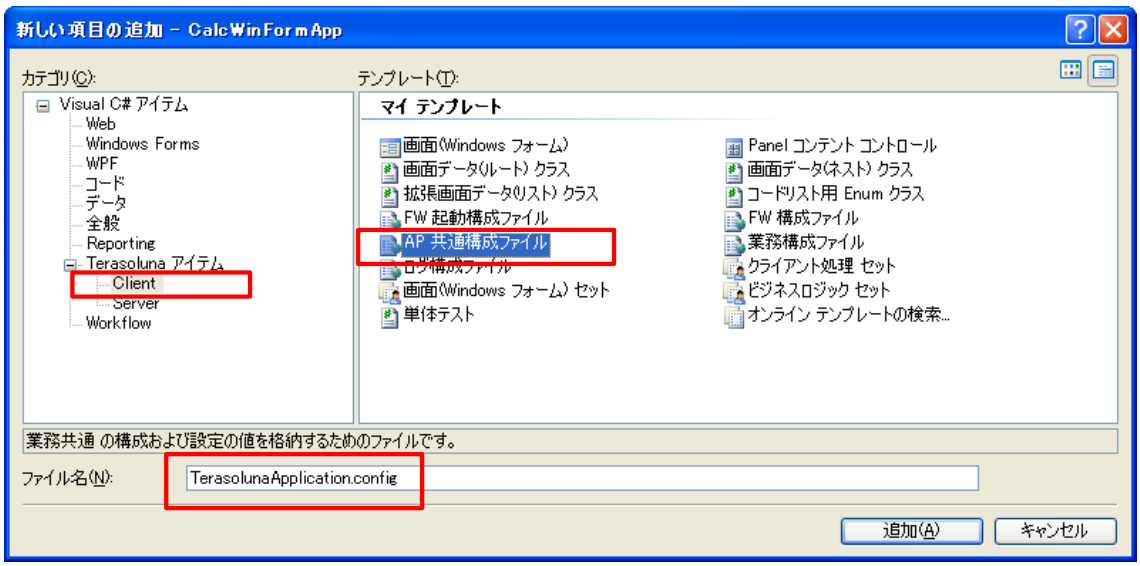

図 8「AP 共通構成ファイル」テンプレート

TerasolunaBootstrap.config は、FW 起動構成ファイルテンプレートにより作成する。

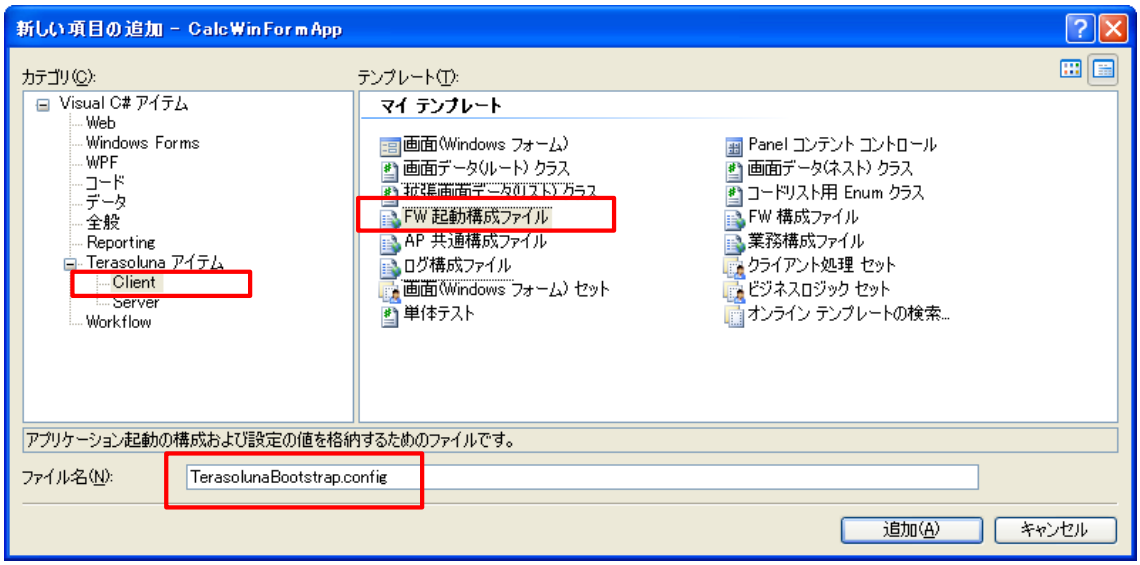

図 9 「FW 起動構成ファイル」テンプレート

図 10 に Visual Studio 上での構成ファイルの配置イメージを示す。

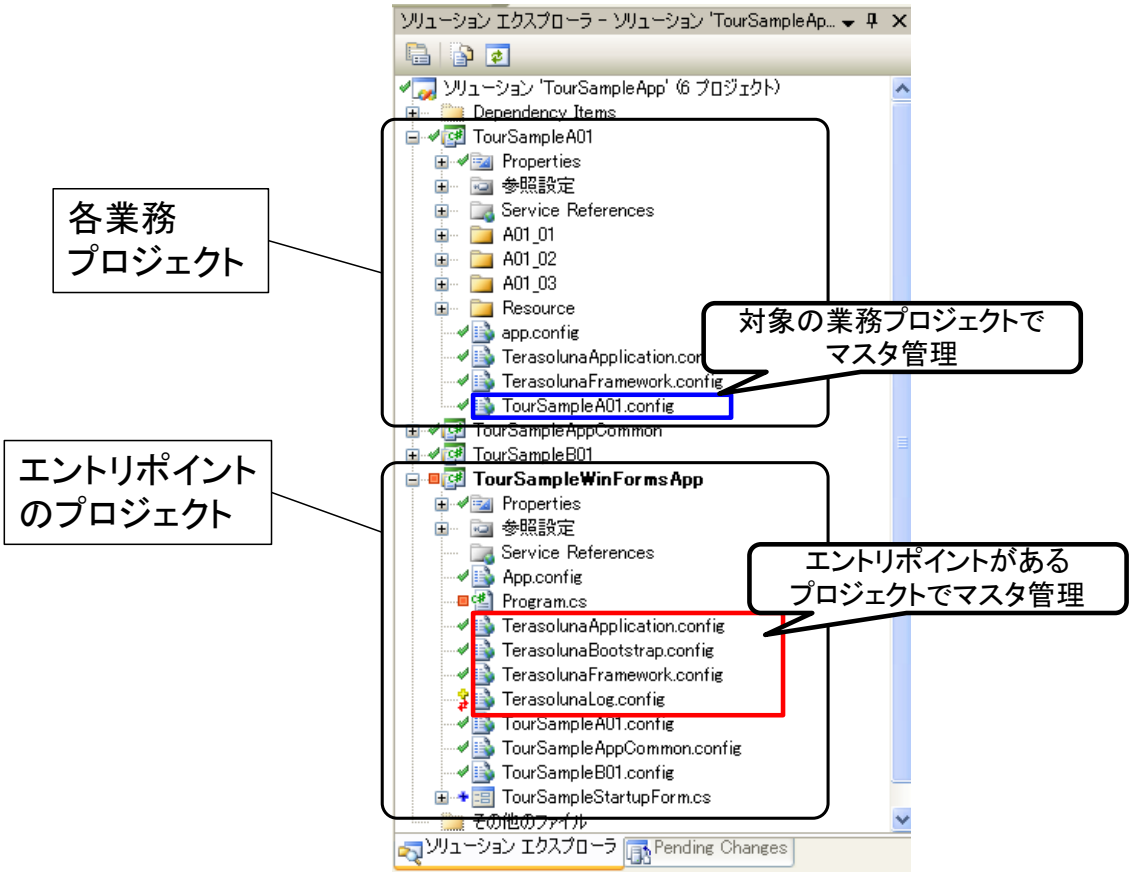

図 10 構成ファイルの配置イメージ(Windows Forms アプリケーション)

なお、上述のとおり、Visual Studio での開発時(「デザインモード」)において、TERASOLUNA フレームワーク は各業務プロジェクトに配置された構成ファイルを参照する。このため、表 2 に示したとおり、TerasolunaFramework.config や TerasolunaApplication.config は、あらかじめエントリポイントのプロジェクトにあるマスタを業務プロジェクトへコピーする必要がある。また、TerasolunaFramework.config や TerasolunaApplication.config に変更があった場合には、業務プロジェクトにコピーしたファイルに直ちに更新を反映する必要がある。
一方、アプリケーション実行時(「実行時モード」)には、すべての構成ファイルが実行可能ファイルと同じ場所に存在する必要があるため、各業務プロジェクトでマスタ管理されている構成ファイルを表 2 に示したルールで、エントリポイントとなるプロジェクトに事前にコピーしておき、ビルド時に出力されるようにしておく必要がある。

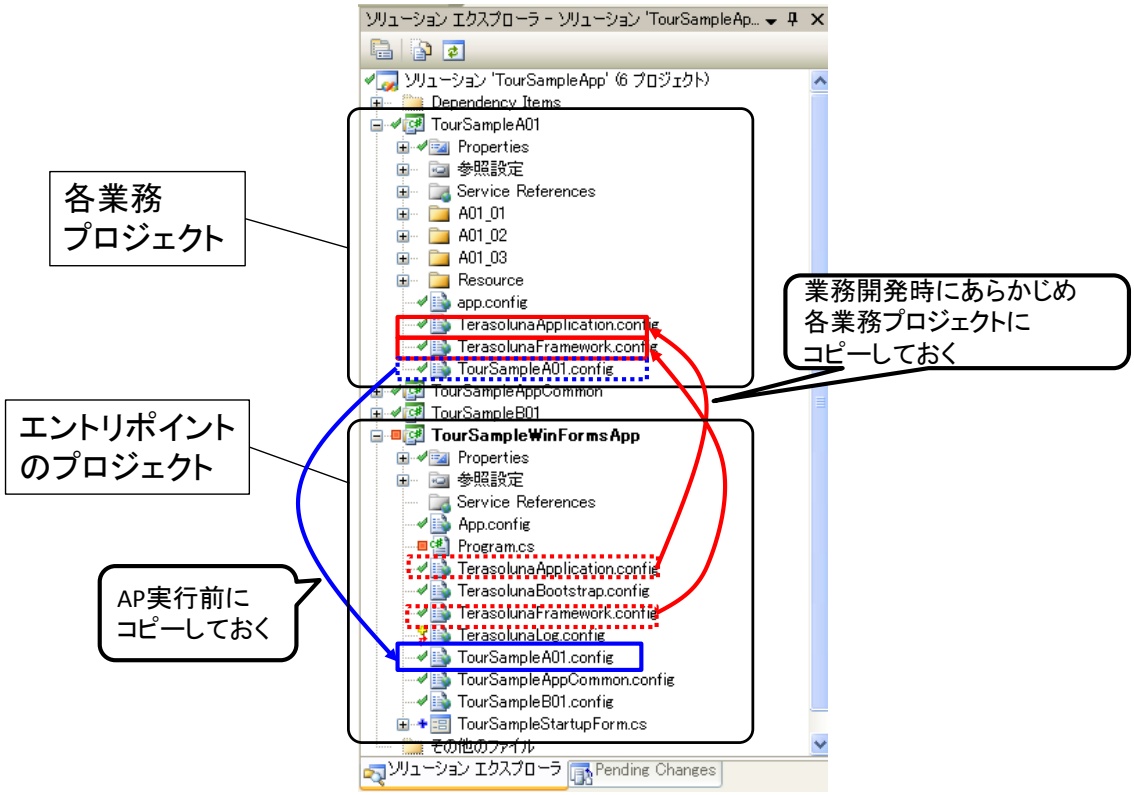

図 11 プロジェクト間での構成ファイルのコピー(Windows Forms アプリケーション)

TERASOLUNA フレームワーク のカスタムプロジェクトテンプレートを利用して作成すれば、ビルド時に自動的に構成ファイルをコピーするビルドイベントのひな形が生成される。これにより、構成ファイルのコピーのし忘れや作業ミスを防止することができる。
ビルドイベントは、各業務プロジェクトを選択し、「プロジェクトメニュー」の「ビルドイベント」タブで確認できる。

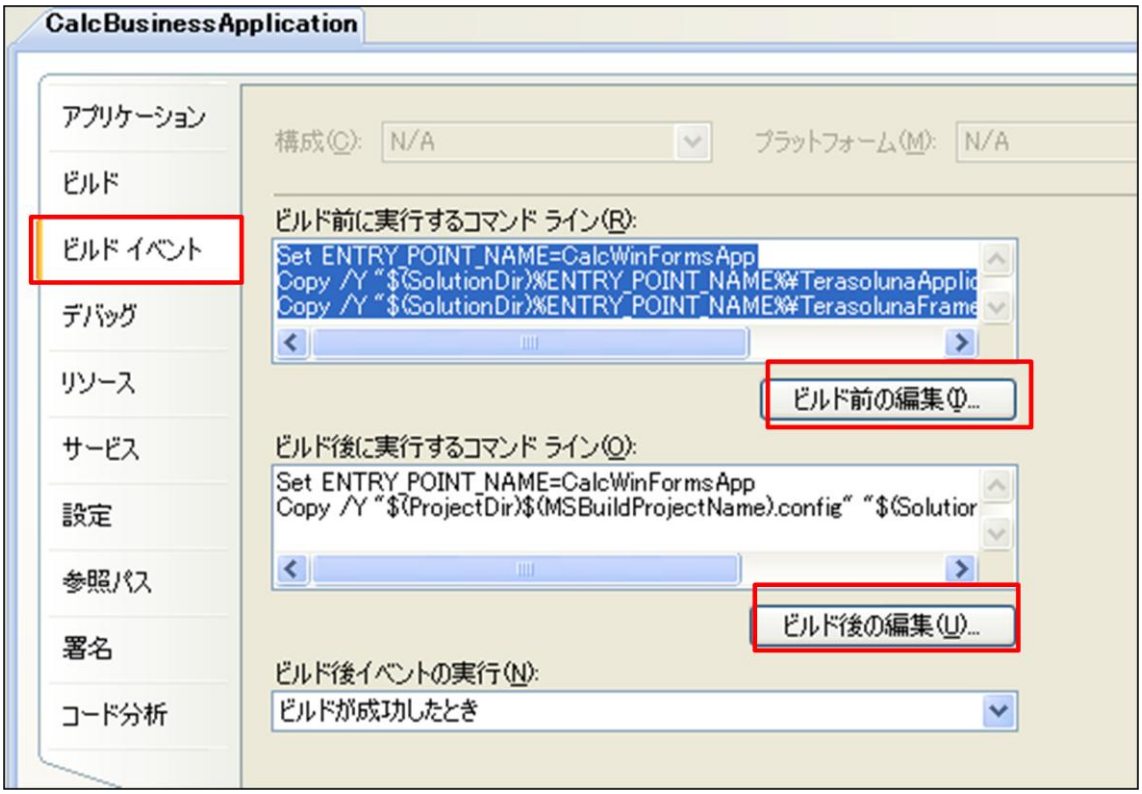

図 12 テンプレートにより生成されたビルドイベント(クライアント AP の業務プロジェクト)

```
Set ENTRY_POINT_NAME=StartupWinFormsApp
Copy /Y "$(SolutionDir)%ENTRY_POINT_NAME%¥TerasolunaApplication.config" "$(ProjectDir)"
Copy /Y "$(SolutionDir)%ENTRY_POINT_NAME%¥TerasolunaFramework.config" "$(ProjectDir)"
```

リスト 1 ビルド前に実行するコマンドライン(クライアント AP の業務プロジェクト)

```
Set ENTRY_POINT_NAME= StartupWinFormsApp
Copy /Y "$(ProjectDir)$(MSBuildProjectName).config" "$(SolutionDir)%ENTRY_POINT_NAME%""
```

リスト 2 ビルド後に実行するコマンドライン(クライアント AP の業務プロジェクト)

上記の各スクリプトのひな形の1行目は、環境変数 ENTRY_POINT_NAME にエントリポイントとなる Visual Studio のプロジェクト名を設定する。下線部分にダミーの文字列が挿入されているが、実際のプロジェクト名に修正する[4]。

[4] 修正しないと、コピー処理が失敗し、ビルドエラーとなる。

各構成ファイルがビルド時に出力されるようにするには、エントリポイントのプロジェクトにある(App.config 以外の)構成ファイルについて、図 13 のようにプロパティエディタで「出力ディレクトリにコピー」を「新しい場合はコピーする」または「常にコピーする」に設定しておく必要がある。なお、Click Once を使ってアプリケーションを配信する場合には、エントリポイントにある(App.config 以外の)構成ファイルについて、図 14 のように、プロパティエディタで「ビルドアクション」を「コンテンツ」に設定しておく必要がある。
なお、これらの設定もカスタムプロジェクトテンプレートにより生成された構成ファイルについては設定済みであるが、エントリポイントにコピーする業務構成ファイルについては、手動での設定が必要である。

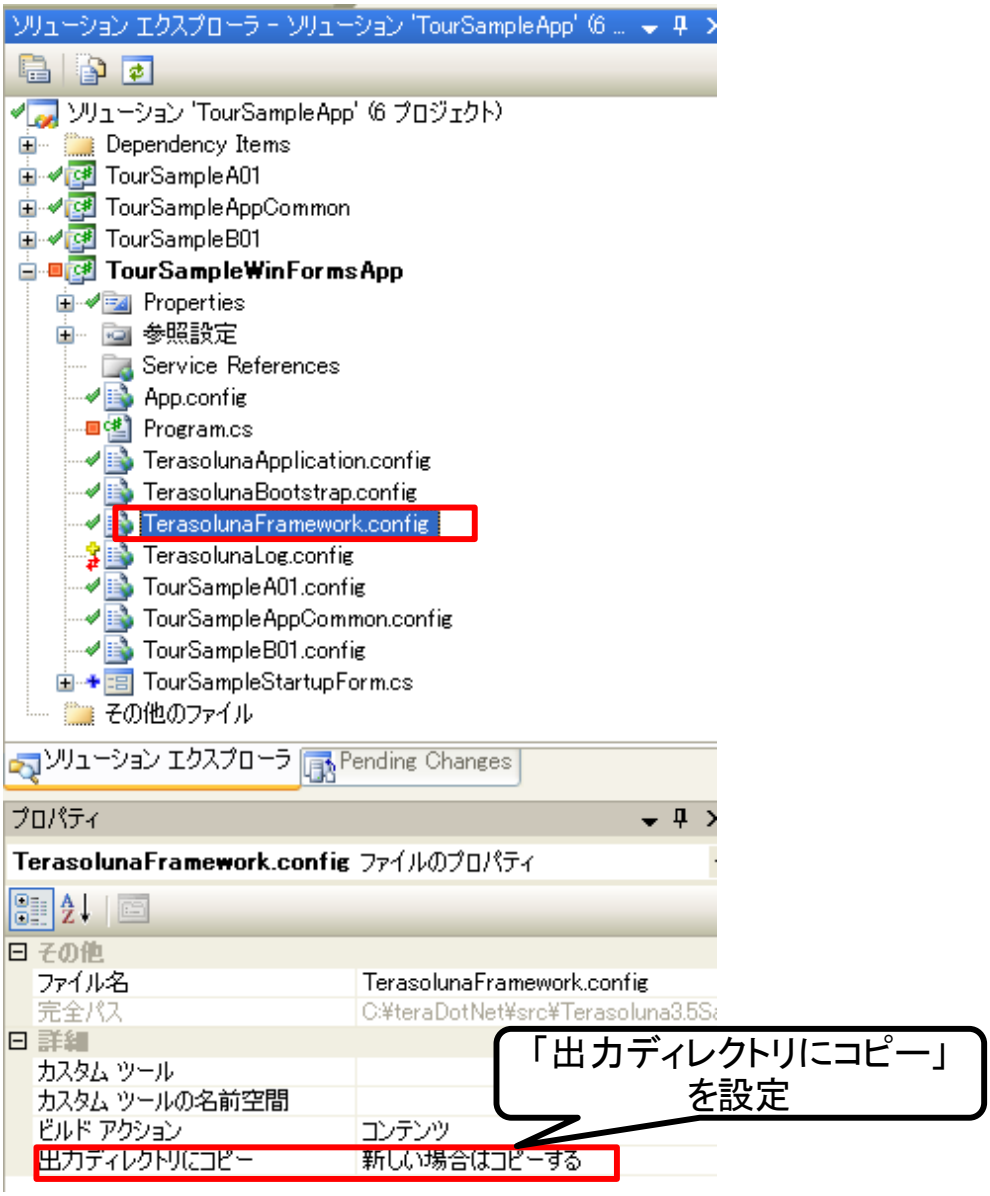

図 13「出力ディレクトリにコピー」の設定(クライアント AP のエントリポイントのプロジェクト)

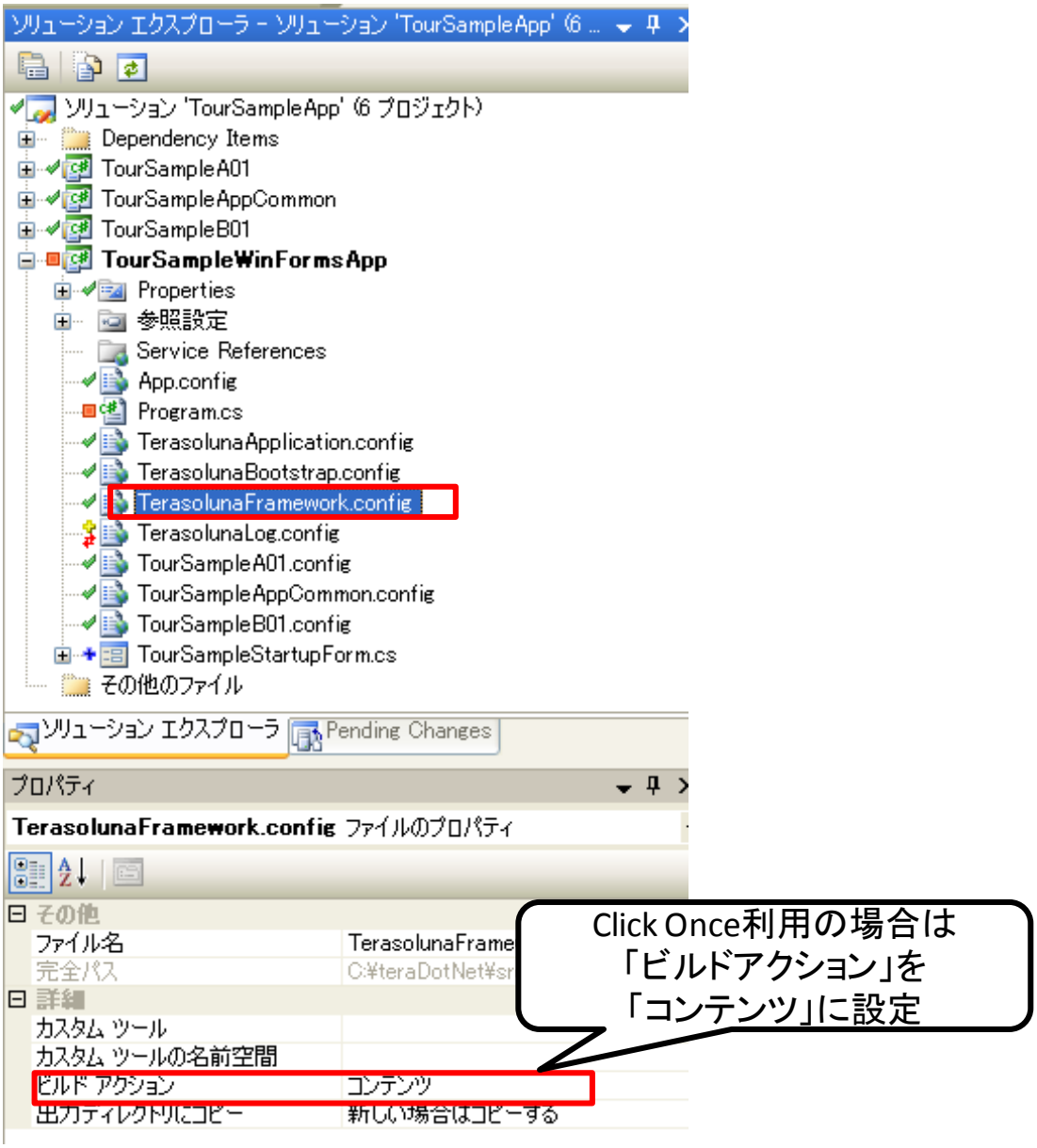

図 14「ビルド アクション」の設定(クライアント AP のエントリポイントのプロジェクト)

◆ 構成ファイルの作成（WCF サービスアプリケーションの場合）

TERASOLUNA フレームワーク を使った WCF サービスアプリケーション(サーバ AP)の場合、以下の構成ファイルが必要となる。
表 3 の通り、TerasolunaFramework.config 、TerasolunaApplication.config 、TerasolunaBootstrap.config、TerasolunaLog.config は、システム全体の設定となるため、VisualStudio 上ではエントリポイントのプロジェクトでマスタを管理する。
業務構成ファイル（”アセンブリ名”.config）は、業務プロジェクトに閉じた設定となるため、対象の業務プロジェクトでマスタ管理する。

表 3 構成ファイルの配置方法(WCF サービスアプリケーションの場合)

項番	構成ファイル	Visual Studio 上でのファイルの配置		
		マスタファイルの配置場所	業務プロジェクトへの配置（デザインモードは使用しない）	エントリポイントのプロジェクトへの配置（実行時モードで使用）
1	FW 構成ファイル (TerasolunaFramework.config)	エントリポイントのプロジェクト	-	○
2	AP 共通構成ファイル (TerasolunaApplication.config)	エントリポイントのプロジェクト	-	○
3	業務構成ファイル (“アセンブリ名”.config)	業務プロジェクト	○ （対象業務のみ）	○ （マスタをコピー）
4	FW 起動構成ファイル (TerasolunaBootstrap.config)	エントリポイントのプロジェクト	-	○
5	ログ構成ファイル (TerasolunaLog.config)	エントリポイントのプロジェクト	-	△[5]

構成ファイルの作成方法としては、TERASOLUNA フレームワーク が提供するカスタムプロジェクトテンプレートを利用することで、プロジェクト作成時に構成ファイルのひな形も一緒に生成する。

WCF サービスアプリケーションの場合は、「WCF サービスアプリケーション」プロジェクトテンプレートを利用する。
プロジェクト作成時に TerasolunaFramework.config 、TerasolunaApplication.config 、TerasolunaBootstrap.config が生成される。

[5] TerasolunaLog.config は、「CM-06 ログ出力機能」の拡張を実施した場合に必要となる。標準提供のログ出力機能を利用する場合は不要である。

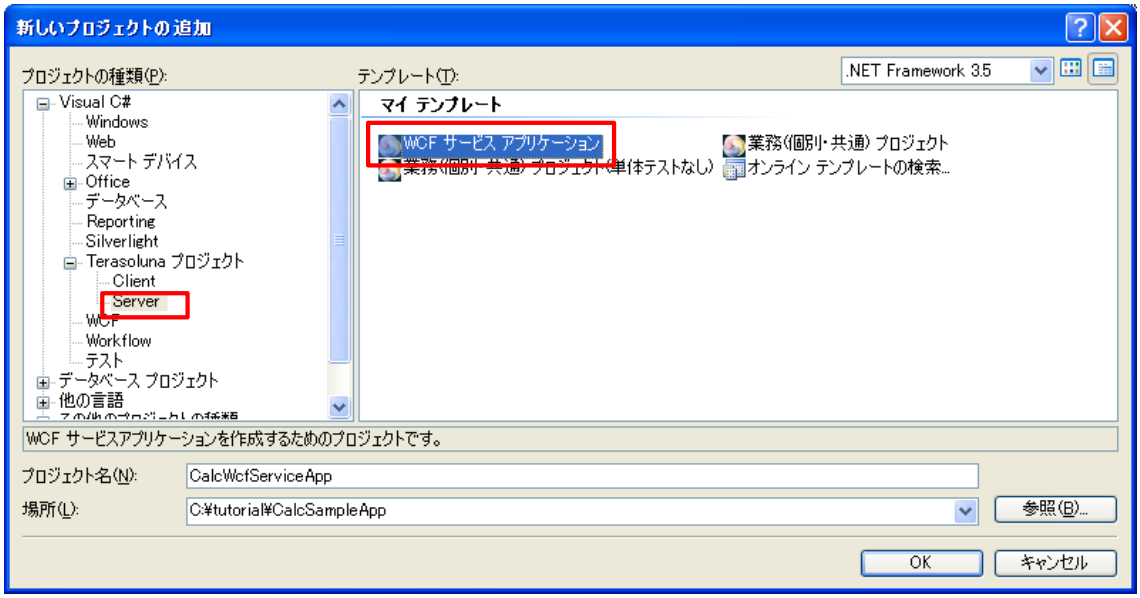

図 15 「WCF サービスアプリケーション」プロジェクトテンプレート

また、業務プロジェクトについては、「業務（個別・共通）プロジェクト」[6]テンプレートを利用し作成する。
プロジェクト作成時に、ファイル名が「"アセンブリ名" + .config」の構成ファイルが生成される。

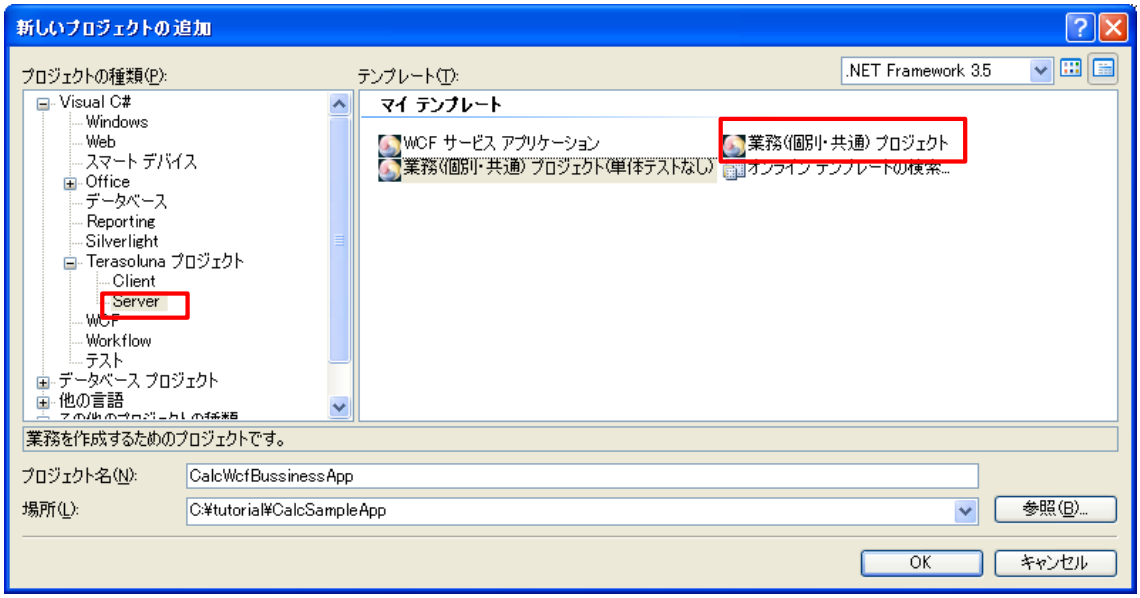

図 16 「業務(個別・共通)プロジェクト」テンプレート

[6] デフォルトのテンプレートでは、単体テストプロジェクトも同時に生成される
単体テストプロジェクトなしのテンプレートも用意している。

作成済みのプロジェクトに構成ファイルを追加したい場合は、TERASOLUNA フレームワークが提供するカスタムアイテムテンプレートにより作成することもできる。

TerasolunaFramework.config は、FW 構成ファイルテンプレートにより作成する。

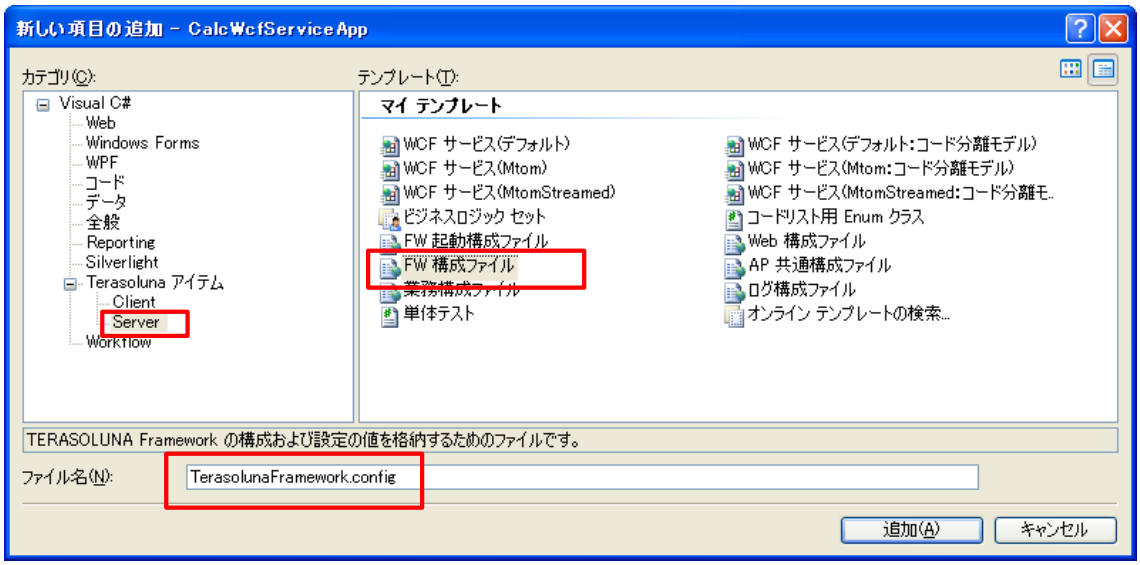

図 17 「FW 構成ファイル」テンプレート

TerasolunaApplication.config は、AP 共通構成ファイルテンプレートにより作成する。

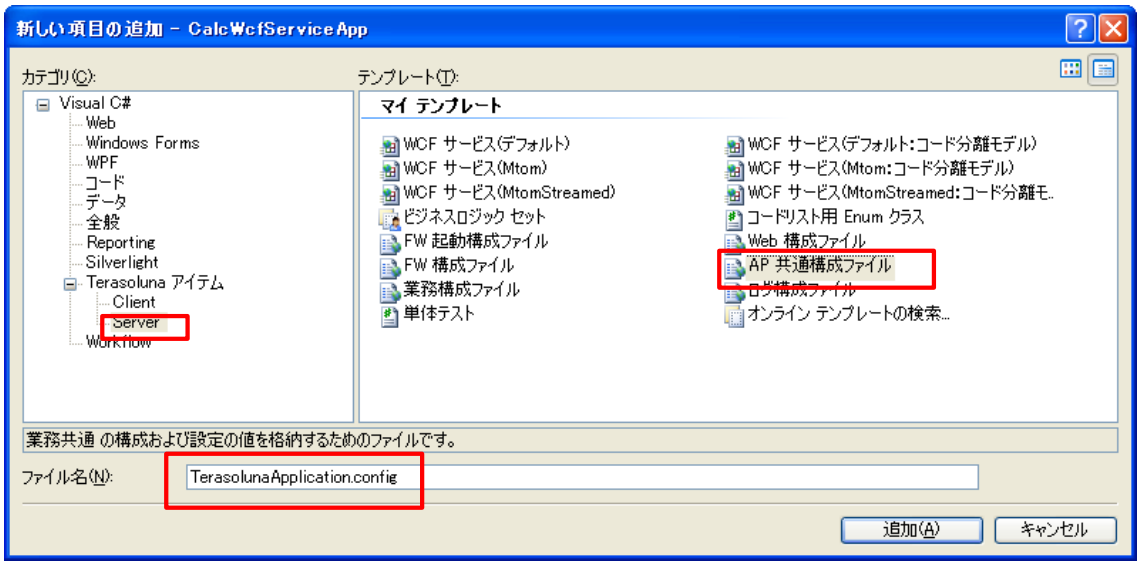

図 18 「AP 共通構成ファイル」テンプレート

TerasolunaBootstrap.config は、FW 起動構成ファイルテンプレートにより作成する。

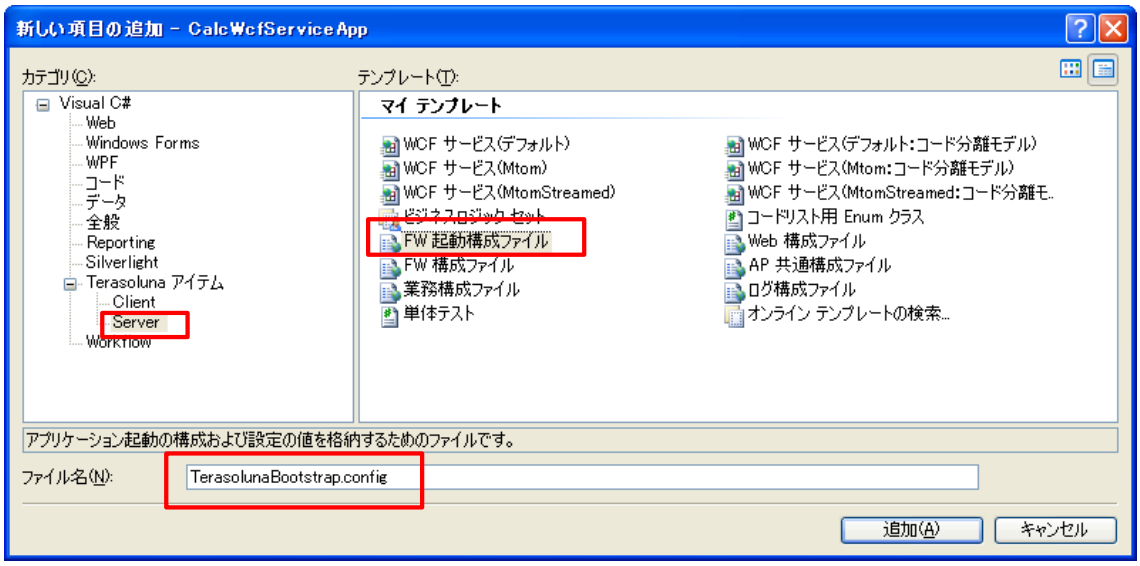

図 19 FW 起動構成ファイル

図 20 に Visual Studio 上での配置イメージを示す。

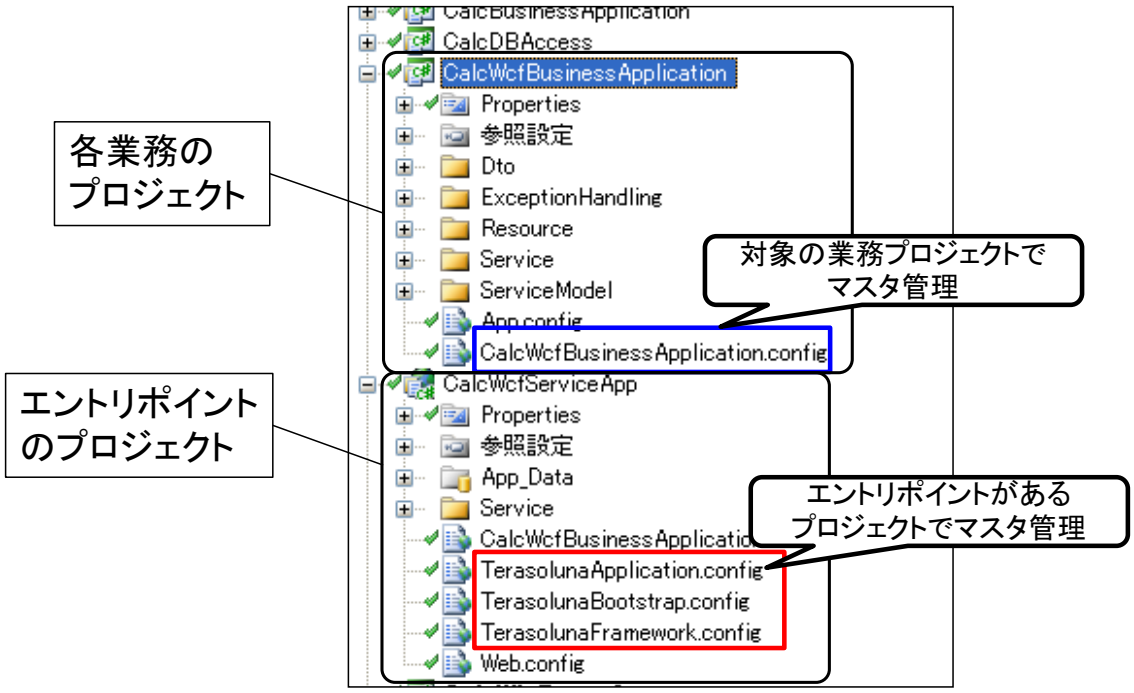

図 20 構成ファイルの配置イメージ(WCF サービスアプリケーション)

TERASOLUNA フレームワークは、サーバ AP 側の Visual Studio デザイナ拡張機能を提

供していないため、「デザインモード」で実行されることはない。このため、表 3 に示したとおり、Windows Forms アプリケーション（クライアント AP）と異なり、TerasolunaFramework.config や TerasolunaApplication.config のコピー作業は不要である。
一方、アプリケーション実行時（「実行時モード」）の場合は、業務構成ファイルも実行可能ファイルと同じ場所に必要となるため、表 3 に示したルールで、エントリポイントとなるプロジェクトに事前にコピーして、ビルド時に出力されるように設定しておく必要がある。

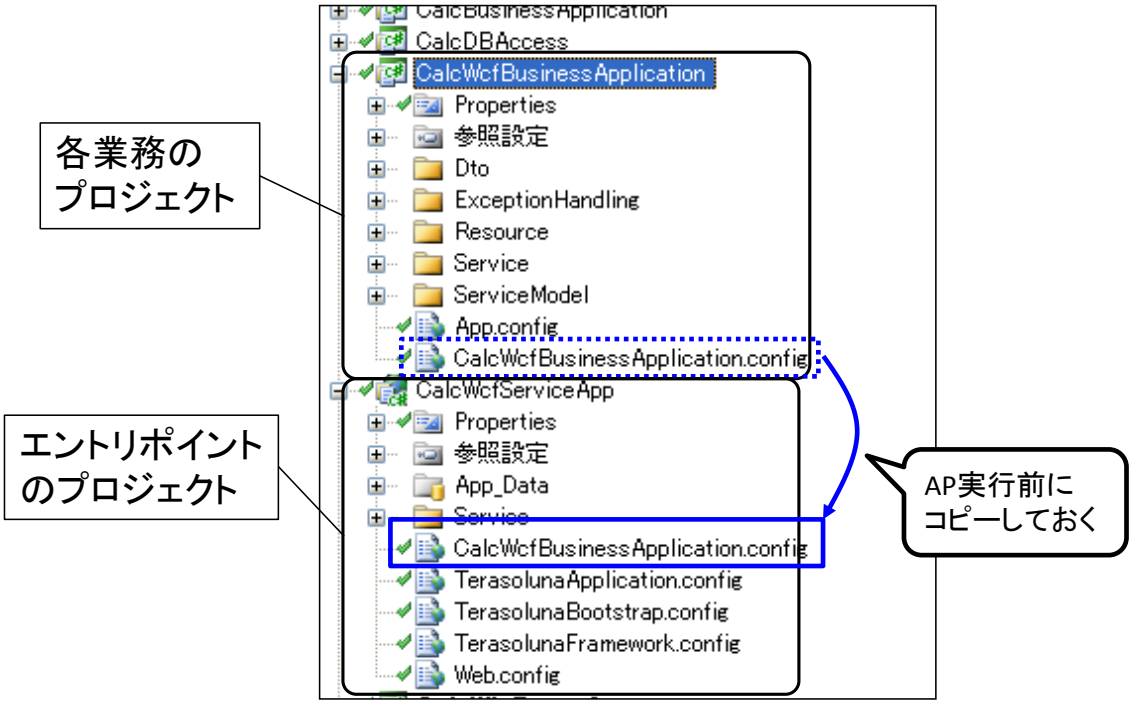

図 21 プロジェクト間での構成ファイルのコピー（WCF サービスアプリケーション）

なお、サーバ側 AP 開発においても、クライアント AP 同様 TERASOLUNA フレームワーク のカスタムプロジェクトテンプレートを利用して作成すれば、ビルド時に自動的にコピーするビルドイベントのひな形が生成される。これにより、構成ファイルのコピーのし忘れや作業ミスを防止することができる。
ビルドイベントは、各業務プロジェクトを選択し、「プロジェクトメニュー」の「ビルドイベント」タブで確認できる。

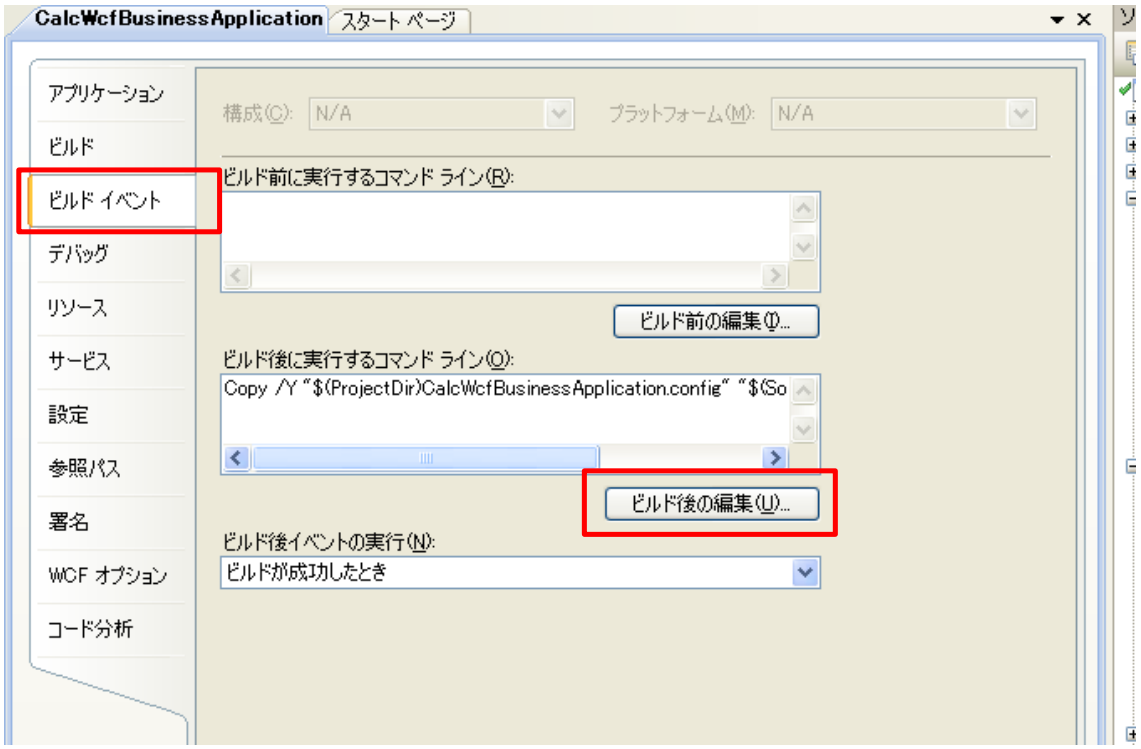

図 22 テンプレートにより生成されたビルドイベント(サーバ AP の業務プロジェクト)

```
Set ENTRY_POINT_NAME= WcfServiceApp
Copy /Y "$(ProjectDir)$(MSBuildProjectName).config" "$(SolutionDir)%ENTRY_POINT_NAME%""
```

リスト 3 ビルド後に実行するコマンドライン(サーバ AP の業務プロジェクト)

上記の各スクリプトのひな形の1行目は、環境変数 ENTRY_POINT_NAME にエントリポイントとなる Visual Studio のプロジェクト名を設定する。下線部分にダミーの文字列が挿入されているが、実際のプロジェクト名に修正する[7]。

各構成ファイルがビルド時に出力されるようにするには、エントリポイントにある各構成ファイルについて、プロパティエディタで「ビルド アクション」を「コンテンツ」に設定しておく必要がある。
(「出力ディレクトリにコピー」は「コピーしない」のままでよい。Windwos Form アプリケーションと設定内容が異なるので注意すること。)
なお、これらの設定もカスタムプロジェクトテンプレートにより生成された構成ファイルについては設定済みであるが、エントリポイントにコピーする業務構成ファイルについては、手動での設定が必要である。

[7] 修正しないと、ビルド時にスクリプトが実施するコピー処理に失敗しエラーが発生する。

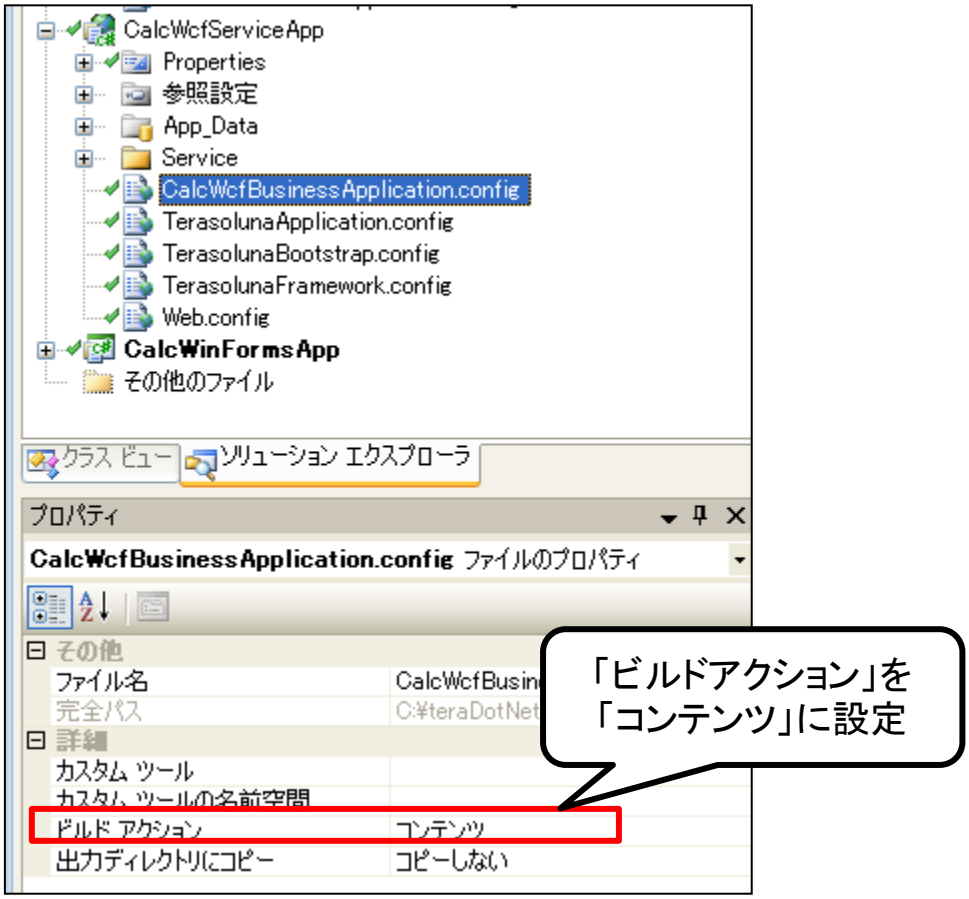

図 23 「ビルド アクション」の設定(サーバ AP のエントリポイントのプロジェクト)

◆ UnityContainer の取得方法

本機能は、UnityManager クラスにより UnityContainer を統合管理しているため、
UnityManager クラスを使って対象の UnityContainer を取得する。
以下に、UnityManager クラスのコンテナ取得用のメソッドを示す。

表 4　UnityManager クラスのコンテナ取得メソッド一覧

項番	メソッド	引数	説明
1	public static IUnityContainer GetFrameworkContainer()	-	「フレームワーク用 UnityContainer」を返却する。
2	public static IUnityContainer GetApplicationContainer(Assembly Name assemblyName)	業務プロジェクトのアセンブリ名	引数で指定されたアセンブリ（業務プロジェクト）に対応する「業務用 UnityContainer」を返却する。
3	public static IUnityContainer GetBootstrapContainer()	-	「FW 起動用 UnityContainer」を返却する。
4	public static IUnityContainer GetLogContainer()	-	「ログクラス用 UnityContainer」を返却する。

◆ DI の利用

Unity AB の DI 機能は、コンストラクタインジェクション、プロパティインジェクション（Setter インジェクション）、メソッドインジェクションの3種類に対応している。
各方式の DI 設定方法の詳細は Unity AB のドキュメント等を参照することとし、ここでは、簡単な例を使い、一般的なプロパティインジェクションを使った DI の設定例を示す。

サンプルプログラムの概要

- Color（色）プロパティと、Run（走る）メソッドを持つ、乗り物(IVehicle)インタフェースを用意。
- 乗用車(Car) クラス、バス(Bus)クラスは、IVehicle インタフェースを実装する。
- 乗り物は、Run メソッド実行時に、ログ(ILog)インタフェースに出力(Write メソッド)する。
- コンソール出力クラス(ConsoleLog)は、ログ(Log)インタフェースを実装する。
- 所有者(Order)クラスは、乗り物（IVehicle オブジェクト）を所有する

サンプルプログラムの対象クラスのクラス図は、図 24 のようになる。

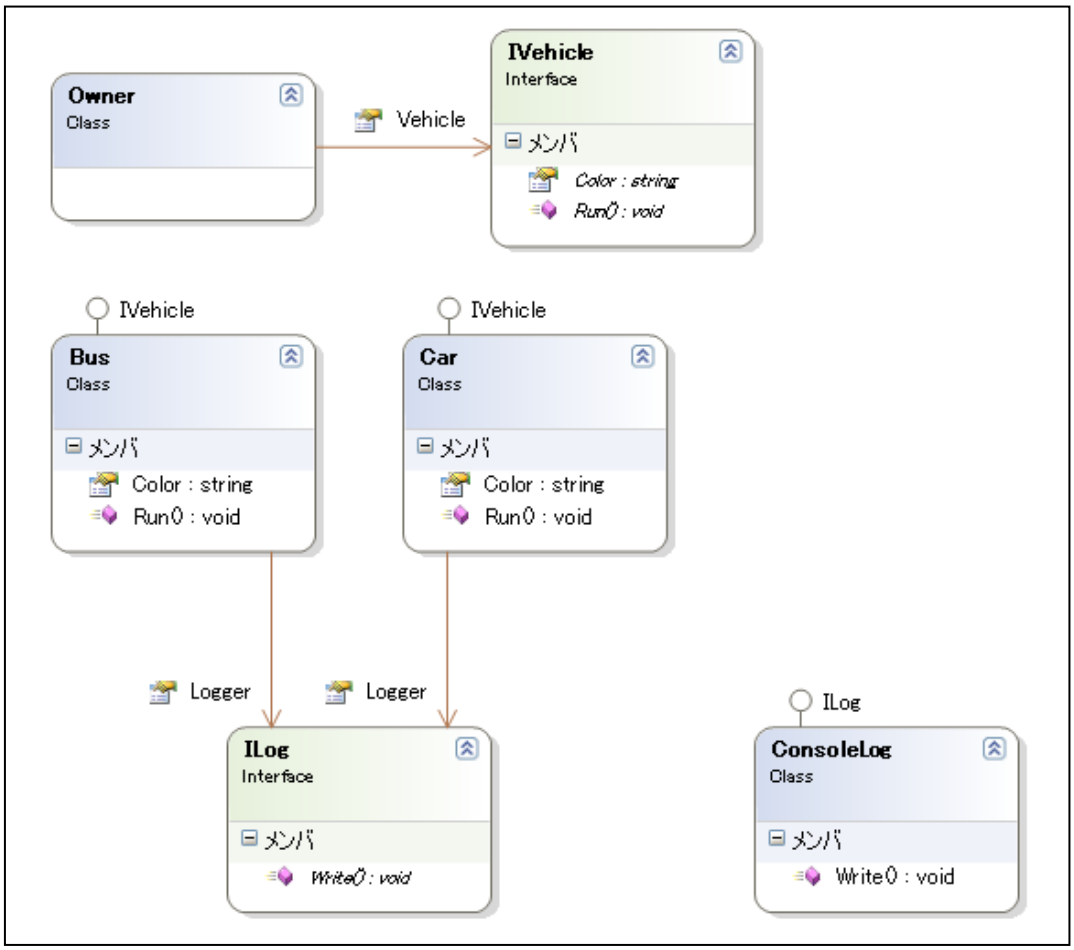

図 24 サンプルプログラムの対象クラス

また、各クラスのコードは、以下の通りになる。
このとき、プロパティインジェクションを使用するため、依存関係を注入したいプロパティに Dependency 属性を付与する。

```
/// <summary>
/// 乗り物のインタフェース
/// </summary>
public interface IVehicle
{
    string Color { get; }
    void Run();
}

/// <summary>
/// 乗用車
/// </summary>
public class Car : IVehicle
{
    [Dependency]
    public ILog Logger {get; set;}
    public string Color {get; set;}
    public void Run()
    {
        Logger.Write("a " + Color + " Car runs");
    }
}

/// <summary>
/// バス
/// </summary>
public class Bus : IVehicle
{
    [Dependency]
    public ILog Logger { get; set; }
    public string Color { get; set; }
    public void Run()
    {
        Logger.Write("a " + Color + " Bus runs");
    }
}
```

Dependency 属性の付与

Dependency 属性の付与

リスト 4 乗り物関連(IVehicle、Car、Order)のソースコード

```
/// <summary>
/// 乗り物の所有者
/// </summary>
public class Owner
{
    [Dependency]
    public IVehicle Vehicle { get; set; }
}
```

Dependency 属性の付与

リスト 5 所有者(Order クラス)のソースコード

```
/// <summary>
/// ログ出力インタフェース
/// </summary>
public interface ILog
{
    void Write(string message);
}

/// <summary>
/// コンソール出力
/// </summary>
public class ConsoleLog : ILog
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

リスト 6 ログ出力関連(Log、ConsoleLog)のソースコード[8]

Unity AB では、各クラスの依存関係を構成ファイルまたはソースコードで記述することができる。図 25 で示す依存関係を構成ファイルで設定した例を以下に示す。詳細な設定方法は、Unity AB のドキュメントを参照すること。

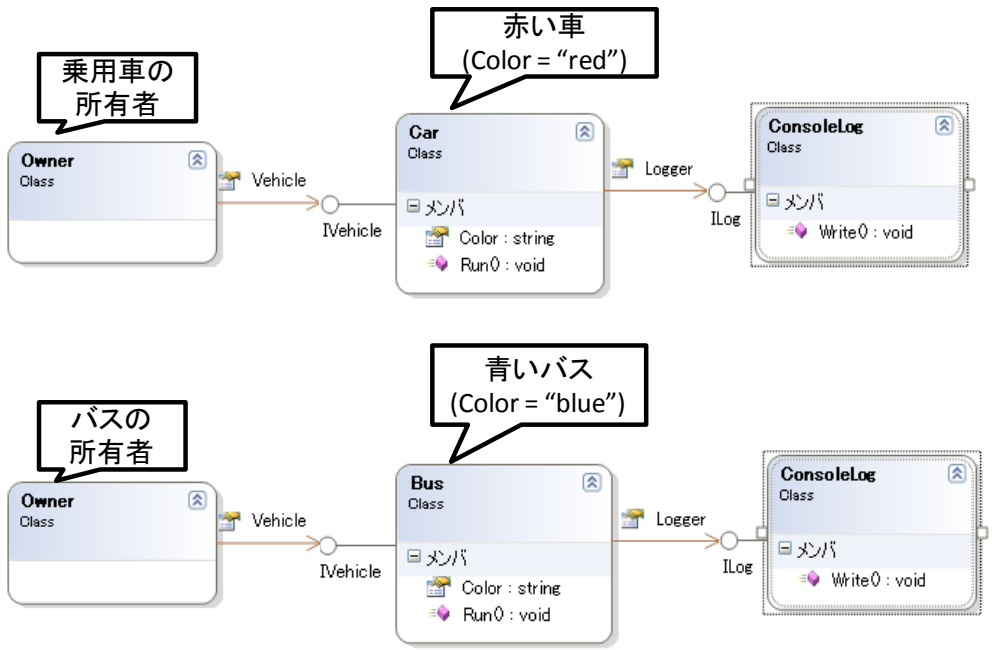

図 25 クラスの依存関係

[8] ここで示すログ出力クラスは、本機能を説明するためのサンプルコードであり、「CM-06 ログ出力機能」とは別のものである。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <!—UnitySectionの設定 -->
    <section name="unity"
             type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                   Microsoft.Practices.Unity.Configuration,
                   Version=1.2.0.0,
                   Culture=neutral,
                   PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <unity>
    <typeAliases>
      ・・・
      <!-- サンプルコードに関するTypeAliasの設定 -->
      <typeAlias alias="string" type="System.String" />
      <typeAlias alias="Owner"
            type="Terasoluna.Laboratories.UnityTest.Owner, Terasoluna.Laboratories" />
      <typeAlias alias="IVehicle"
            type="Terasoluna.Laboratories.UnityTest.IVehicle, Terasoluna.Laboratories" />
      <typeAlias alias="Car"
            type="Terasoluna.Laboratories.UnityTest.Car, Terasoluna.Laboratories" />
      <typeAlias alias="Bus"
            type="Terasoluna.Laboratories.UnityTest.Bus, Terasoluna.Laboratories" />
      <typeAlias alias="ILog"
            type="Terasoluna.Laboratories.UnityTest.ILog, Terasoluna.Laboratories" />
      <typeAlias alias="ConsoleLog"
           type="Terasoluna.Laboratories.UnityTest.ConsoleLog, Terasoluna.Laboratories" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <!-- 所有者のDI設定（乗用車の所有者と、バスの所有者） -->
          <type name="CarOwner" type="Owner">
            <typeConfig>
              <property name="Vehicle" propertyType="IVehicle" >
                <dependency name="RedCar"/>
              </property>
            </typeConfig>
          </type>
          <type name="BusOwner" type="Owner">
            <typeConfig>
              <property name="Vehicle" propertyType="IVehicle" >
                <dependency name="BlueBus"/>
              </property>
            </typeConfig>
          </type>
          <!-- 乗り物のDI設定（赤い乗用車と青いバス） -->
          <type name="RedCar" type="IVehicle" mapTo="Car" >
            <typeConfig>
              <property name="Color" propertyType="string" >
                <value value="red" />
              </property>
              <property name="Logger" propertyType="ILog" >
                <dependency/>
              </property>
            </typeConfig>
          </type>
・・・
```

Unity セクションを定義する。

型名に対するエイリアス名を設定したい場合は、typeAlias を記述

type タグでコンテナで管理したい型を定義

依存関係は、typeConfig タグ配下で記述

依存関係を注入するプロパティは depencency タグで記述。
name を指定した場合、name が一致するインスタンスを注入。

type 属性に指定したインタフェースに対する実装クラスを mapTo 属性で指定

静的な値の代入であれば value タグで記述

```
・・・
          <type name="BlueBus" type="IVehicle" mapTo="Bus" >
            <typeConfig>
              <property name="Color" propertyType="string" >
                <value value="blue" />
              </property>
              <property name="Logger" propertyType="ILog" >
                <dependency/>
              </property>
            </typeConfig>
          </type>
          <!-- ログはコンソール出力-->
          <type type="ILog" mapTo="ConsoleLog" />
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

depencency タグに name がない場合、ILog インタフェースのデフォルトの DI 設定で注入

name 属性がない場合、対象のインタフェースに対するデフォルトの DI 設定として働く

リスト 7 構成ファイルによる DI の設定例

また、構成ファイルの代わりにソースコードで同等の記述をした例を以下に示す。
実際に UnityContainer の Resolve メソッドでインスタンス取得する前に、あらかじめ UnityContainer の RegisterType メソッド等を使って DI 設定をしておく。
UnityContainer が提供するメソッドの詳細については Unity AB のドキュメントを参照のこと。

```
private static void RegisterTypes()
{
    ///実行中のアセンブリに対応する業務個別のUnityContainerを取得
    IUnityContainer container =
      UnityManager.GetApplicationContainer(Assembly.GetExecutingAssembly().GetName());

    /// 所有者のDI設定（乗用車の所有者と、バスの所有者）
    container.RegisterType<Owner>("CarOwner",
        new InjectionProperty("Vehicle", new ResolvedParameter<IVehicle>("RedCar")));
    container.RegisterType<Owner>("BusOwner",
        new InjectionProperty("Vehicle", new ResolvedParameter<IVehicle>("BlueBus")));

    /// 乗り物のDI設定（赤い乗用車と青いバス）
    container.RegisterType<IVehicle, Car>("RedCar",
        new InjectionProperty("Color", "red"),
        new InjectionProperty("Logger", new ResolvedParameter<ILog>())
        );
    container.RegisterType<IVehicle, Bus>("BlueBus",
        new InjectionProperty("Color", "blue"),
        new InjectionProperty("Logger", new ResolvedParameter<ILog>())
    );
    /// ログはコンソール出力
    container.RegisterType<ILog, ConsoleLog>();
}
```

リスト 8 ソースコードによる DI の設定例

以下に、DI 機能を利用する実行プログラムの記述例を示す。
UnityContainer の Resolve メソッド等で、インスタンスを取得することができる。
UnityContainer が提供するメソッドの詳細については Unity AB のドキュメントを参照のこと。

```
class Program
{
    public static void Main(string[] args)
    {
        try
        {
            ///フレームワークの起動
            TerasolunaFramework.Run();
             ///実行中のアセンブリに対応する業務個別のUnityContainerを取得
             IUnityContainer container =
                  UnityManager.GetApplicationContainer(Assembly.GetExecutingAssembly().GetName());
            ///乗用車の所有者を取得(CarOwnerは、構成ファイルで指定したname属性)
            Owner carOwner = container.Resolve<Owner>("CarOwner");
            carOwner.Vehicle.Run();
            ///バスの所有者を取得(BusOwnerは、構成ファイルで指定したname属性)
            Owner busOwner = container.Resolve<Owner>("BusOwner");
            busOwner.Vehicle.Run();
        }
        finally
        {
            ///フレームワークの終了
            TerasolunaFramework.Exit();
        }
    }
}
```

リスト 9 サンプルプログラムの実行例

上記、サンプルプログラムの実行結果は、以下の通り。
出力内容の下線部をみると、依存関係が注入されていることが分かる。

```
a red car runs
a blue bus runs
```

リスト 10 サンプルプログラムの実行結果

◆ インスタンスの生存期間の管理

UnityContainer は、デフォルトまたは DI 設定時に指定された生存期間(lifetime)に基づき、インスタンスを管理する。生存期間は、Unity が標準提供する LiftimeManager 継承クラスによってどうやってインスタンスを保持・破棄するかを制御する。
UnityContainer を使ってインスタンス管理する場合には、生存期間にも十分考慮すること。

表 5 Unity AB がサポートする LifetimeManager

項番	LifetimeManger の種類	説明
1	TransientLifeTimeManager	UnityContainer の Resolve メソッド等の呼び出しや他のオブジェクトへの DI 時に毎回新しいインスタンスを生成する LifetimeManager。 RegisterType メソッドや<types>タグの設定で、lifetime の設定をしない場合は、デフォルトでこの LifetimeManager を使用する。
2	ContainerControlledLifeTime Manager	コンテナ内でシングルトンインスタンスとして管理する LifetimeManager。 UnityContainer の Resolve メソッド等の呼び出しや他のオブジェクトの DI 時に、毎回同じインスタンスを返却する。コンテナがガベージコレクトまたは破棄(Dispose)されるときに、ガベージコレクション対象になる。RegisterInstance メソッドや<instances>タグの設定で、lifetime の設定をしない場合は、デフォルトでこの LifetimeManager を使用する。 RegisterType メソッドや<types>タグの設定でシングルトンオブジェクトにしたい場合は、この LifetimeManager を明示的に指定する必要がある。 なお、コンテナ単位にインスタンス化されるため、アプリケーション全体では、同時に複数のシングルトンインスタンスが存在していることがあるので注意すること(コンテナを間違えると、別のシングルトン・インスタンスが取得されてしまう)。
3	ExternallyControlledLifetime Manager	コンテナは弱い参照(weak reference)として、シングルトンインスタンスを保持する LifetimeManager。このため、強い参照(strong refercence)で保持する他のオブジェクトがなくなるとガベージコレクションの対象となる。
4	PerThreadLifetimeManager	スレッドごとにシングルトンインスタンスを管理する LifetimeManager。 Resolve メソッド等の呼び出しや他のオブジェクトの DI 時には、スレッドごとに異なるインスタンスを返却する。

LifetimeManager の詳細な説明は、Unity AB のドキュメントを参照することとして、ここでは、LifetimeManager を使ったサンプルプログラムを以下に示す。

LifetimeManager は、ソースコード(RegistType メソッド等の引数)または構成ファイルの DI 設定により指定する。
ここでは、構成ファイルを使った LifetimeManager の記述例を示す。
/configuration/unity/containers/container/types/type タグで DI の設定をする際、lifetime タグで、LifetimeManager の型を指定する。

この例では、同じインスタンス生成対象クラス(LifetimeManagerTestClass)に対して、異なる LifetimeManager を指定したものを定義している。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
・・・
  <unity>
    <typeAliases>
      <!-- typeAliasの設定 -->
      <!—LifetimeManagerの定義 -->
      <typeAlias alias="singleton"
                 type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager,
                      Microsoft.Practices.Unity" />
      <typeAlias alias="external"
                 type="Microsoft.Practices.Unity.ExternallyControlledLifetimeManager,
                      Microsoft.Practices.Unity" />
      <typeAlias alias="perThread"
                 type="Microsoft.Practices.Unity.PerThreadLifetimeManager,
                      Microsoft.Practices.Unity" />
      <!-- インスタンス生成対象クラス-->
      <typeAlias alias="LifetimeManagerTestClass"
            type="Terasoluna.Laboratories.UnityTest.LifetimeManagerTestClass,
                                Terasoluna.Laboratories" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <!-- lifetimeタグでLifetimeManagrを指定-->
          <type name="SingletonObject" type="LifetimeManagerTestClass">
            <lifetime type="singleton"/>
          </type>
          <type name="ExternalObject" type="LifetimeManagerTestClass">
            <lifetime type="external"/>
          </type>
          <type name="PerThreadObject" type="LifetimeManagerTestClass">
            <lifetime type="perThread"/>
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 11 構成ファイルによる LifetimeManager の設定例

異なる LifetimeManager が指定されたインスタンスについて、インスタンスの生存期間を確認するサンプルプログラムを以下に示す。

```
    public class UnityLifetimeManagerTest
    {
        public static void Test()
        {
            ///実行中のアセンブリに対応する業務個別のUnityContainerを取得
            IUnityContainer container =
                UnityManager.GetApplicationContainer(Assembly.GetExecutingAssembly().GetName());
            /// defaultインスタンスの取得(TransientLifeTimeManager)
            Console.WriteLine("###TransientLifetimeManagerのテスト###");
            LifetimeManagerTestClass default1 = container.Resolve<LifetimeManagerTestClass>();
            LifetimeManagerTestClass default2 = container.Resolve<LifetimeManagerTestClass>();
            ///同一インスタンスかのチェック
            bool result = Object.ReferenceEquals(default1, default2);
            Console.WriteLine(" Object.ReferenceEquals(default1, default2): " + result);

            /// Singletonインスタンスの取得(ContainerControlledLifetimeManager)
            Console.WriteLine("###ContainerControlledLifetimeManagerのテスト###");
            LifetimeManagerTestClass singleton1 =
                container.Resolve<LifetimeManagerTestClass>("SingletonObject");
            LifetimeManagerTestClass singleton2 =
                container.Resolve<LifetimeManagerTestClass>("SingletonObject");
            result = Object.ReferenceEquals(singleton1, singleton2);
            Console.WriteLine(" Object.ReferenceEquals(singleton1, singleton2): " + result);

            /// Externalインスタンスの取得(ExternallyControlledLifetimeManager)
            Console.WriteLine("###ExternallyControlledLifetimeManagerのテスト###");
            LifetimeManagerTestClass external1 =
                container.Resolve<LifetimeManagerTestClass>("ExternalObject");
            LifetimeManagerTestClass external2 =
                container.Resolve<LifetimeManagerTestClass>("ExternalObject");
            result = Object.ReferenceEquals(external1, external2);
            Console.WriteLine(" Object.ReferenceEquals(external1, external2): " + result);
            Console.WriteLine(" external1.hashCode=" + external1.GetHashCode());
            Console.WriteLine(" external2.hashCode=" + external2.GetHashCode());
            external1 = null;     //強い参照を削除
            external2 = null;     //強い参照を削除
            GC.Collect();          //GCの強制実行
            Console.WriteLine("#GC強制実行");
            LifetimeManagerTestClass external3 =
              container.Resolve<LifetimeManagerTestClass>("ExternalObject");
            Console.WriteLine(" external3.hashCode=" + external3.GetHashCode());

            /// PerThreadインスタンスの取得(PerThreadLifetimeManager)
            Console.WriteLine("###PerThreadLifetimeManagerのテスト###");
            LifetimeManagerTestClass perThread1 =
                container.Resolve<LifetimeManagerTestClass>("PerThreadObject");
            LifetimeManagerTestClass perThread2 =
                container.Resolve<LifetimeManagerTestClass>("PerThreadObject");
            result = Object.ReferenceEquals(perThread1, perThread2);
            Thread.CurrentThread.Name = "Thread1";
            Console.WriteLine("#CurrentThread:" + Thread.CurrentThread.Name);
            Console.WriteLine(" Object.ReferenceEquals(perThread1, perThread2): " + result);
            Console.WriteLine(" perThread1.hashCode=" + perThread1.GetHashCode());
            Console.WriteLine(" perThread2.hashCode=" + perThread2.GetHashCode());
            Action<IUnityContainer> action =
                    new Action<IUnityContainer>(ResolvePerThreadObjectAsync);
            action.BeginInvoke(container, null, null);   //別スレッドで実行
        }
・・・
```

```
・・・
        //非同期実行用処理
        private static void ResolvePerThreadObjectAsync(IUnityContainer container)
        {
            Thread.CurrentThread.Name = "Thread2";
            Console.WriteLine("#CurrentThread:" + Thread.CurrentThread.Name);
            LifetimeManagerTestClass perThread3 =
            container.Resolve<LifetimeManagerTestClass>("PerThreadObject");
            Console.WriteLine(" perThread3.hashCode=" + perThread3.GetHashCode());
        }
    }
```

リスト 12 サンプルプログラムの実行例

以下にサンプルプログラムの実行結果を示す。
TransientLifetimeManager の場合、コンテナから最初に取得したインスタンス(default1)と次に取得したインスタンス(default2)が異なるインスタンスであるのに対し、ContainerControlledLifetimeManager の場合、コンテナから最初に取得したインスタンス(singleton1)と次に取得したインスタンス(singleton2)が同一のオブジェクトであることが分かる。
また、ExternallyControlledLifetimeManger の場合、コンテナから最初に取得したインスタンス(external1)と次に取得したインスタンス(external2)は同一であるが、external1、external2 の(強い)参照を null にして GC を強制実行後に取得した external3 は、ハッシュコードを見ると、external1、external2 が参照していたインスタンスと異なるインスタンスであることが分かる。
PerThreadLifetimeManager では、メインスレッドが取得したインスタンス(perThread1,perThread2)は同じインスタンスであるのに対し、デリゲートを非同期実行して取得したインスタンス(perThread2)は、ハッシュコードを見ると異なるインスタンスであるのが分かる。

```
###TransientLifetimeManager のテスト###
  Object.ReferenceEquals(default1, default2): False
###ContainerControlledLifetimeManager のテスト###
  Object.ReferenceEquals(singleton1, singleton2): True
###ExternallyControlledLifetimeManager のテスト###
  Object.ReferenceEquals(external1, external2): True
  external1.hashCode=39785641
  external2.hashCode=39785641
#GC 強制実行
  external3.hashCode=45523402
###PerThreadLifetimeManager のテスト###
#CurrentThread:Thread1
  Object.ReferenceEquals(perThread1, perThread2): True
  perThread1.hashCode=35287174
  perThread2.hashCode=35287174
#CurrentThread:Thread2
  perThread3.hashCode=12547953
```

リスト 13 サンプルプログラムの実行結果

◆ AOP の利用

AOP は、Unity AB のインターセプタ機能を使って実現する。Unity AB のインターセプタは、以下の3つをサポートしている。

表 6 Unity AB がサポートするインターセプタ

項番	インターセプタの種類	説明
1	TrasparentProxyInterceptor	.NET の TransparentProxy/RealProxy により作成されたプロキシ（代理オブジェクト）を利用するインターセプタ。対象クラスの全てのメソッドを AOP 適用対象としたい場合に利用する。基本的に MarshalByRefObject 継承クラスである必要がある。インタフェースを対象とした場合にはそのインタフェースのメソッドのみ適用される。透過プロキシを利用するため通常のメソッドコールより呼び出しが遅くなる。
2	InterfaceInterceptor	動的にコード生成したプロキシを利用するインターセプタ。インターセプトしたいメソッドが1つのインタフェースに限定される時に使用する。
3	VirtualMethodInterceptor	対象クラスをインスタンス化する代わりに、対象クラスの virtual メソッドの処理に、インターセプトしたい処理をフックさせるようオーバーライドした継承クラスを動的にコード生成しインスタンス化するインターセプタ。virtual メソッドのみインターセプトすることができる。

インターセプタ機能の詳細については、Unity AB のドキュメントを参照することとし、ここでは、VirtualMethodInterceptor を利用したサンプルコードをもとに AOP の利用方法を説明する。

サンプルプログラムの概要

- AOP を使って、ClassA クラスのメソッド実行の前後に、コンソールログを出力。
- VritualMethodIntereceptor を使用し、virtual メソッドに対して処理をインターセプト。
- AOP の処理対象は、以下のとおり。
  - 「DoSomething」メソッド
  - メソッド名が「Execute」で始まり、string 型の引数を1つ持つメソッド

(1) CallHandler の実装

CallHandler は、AOP の用語で言うところの Advice にあたるもので、クラスから分離させた共通的な処理である。CallHandler によりメソッドやプロパティ呼び出しの前後に処理を織り込むことができる。CallHandler を作成するためには、Microsoft.Practices.Unity.InterceptionExtension.ICallHandler インタフェースを実装し、インターセプト時に織り込みたい処理を記述する。ICallHandler インタフェースの API や作成方法についての詳細は、Unity AB のドキュメントを参照のこと。
以下に、ICallHandler の実装例を示す。この例では、ICallHandler の Invoke メソッドを実装し、対象の呼び出しメソッドの前後でコンソール出力をしている。

```
public class SampleLogCallHandler : ICallHandler
{
    public IMethodReturn Invoke(
            IMethodInvocation input, GetNextHandlerDelegate getNext)
    {
        ///ログの出力
        string methodName = input.MethodBase.Name;
        Console.WriteLine("<<SampleLogCallHandler>>" + methodName + "メソッド実行");
        ///対象の呼び出しメソッドを実行
        IMethodReturn methodReturn = getNext().Invoke(input, getNext);
        ///ログの出力
        Console.WriteLine("<<SampleLogCallHandler>>" + methodName + "メソッド終了");
        return methodReturn;
    }

     public int Order { get; set; }
}
```

リスト 14 ICallHandler インタフェースの実装例

(2) AOP の実施範囲の設定

Unity AB では対象のクラスに ICallHandler の処理を織り込む適用範囲(AOP の用語でいうところの Pointcut にあたる)を設定する方法として、以下の2つがある。

- メソッド等にカスタム属性を付与し明示的に適用範囲を指定する。
  - HandlerAttribute クラスを継承し作成したカスタム属性を、処理を織り込みたいメソッド等に付与する。
- コンテナ全体に適用するポリシーを対象クラスの外側から設定する。
  - IMatchingRule インタフェース実装クラスにより適用範囲を指定して、コンテナ全体の適用ポリシーを設定する。

【メソッド等にカスタム属性を付与するケース】
UnityAB のインターセプタ機能では、AOP 対象メソッドに、Microsoft.Practices.Unity.InterceptionExtension.HandlerAttribute クラスを継承し作成したカスタム属性を付与することで、ICallHandler 実装クラスの処理を織り込むことができる。
以下に、HandlerAttribute 継承クラスの実装例を示す。HandlerAttribute クラスの CreateHandler メソッドをオーバーライドして SamleCallHandler のインスタンスを返却する処理を実装する。このサンプルでは、SampleCallHandler を適用するカスタム属性を実装している。

```
    /// <summary>
    ///   SampleLogCallHandlerを適用するHandlerAttribute
    /// </summary>
    public class SampleLogCallHandlerAttribute : HandlerAttribute
    {
        public override ICallHandler CreateHandler(Microsoft.Practices.Unity.IUnityContainer container)
        {
            return new SampleLogCallHandler();
        }
    }
```

リスト 15 HandlerAttribute インタフェースの実装例

次に、対象メソッドに、作成したカスタム属性を付与する。以下に、HandlerAttribute を継承したカスタム属性を付与した例を示す。
このサンプルでは、DoSomething メソッドに SampleCallHandler 属性を付与している。

```
[SetInterceptor(typeof(VirtualMethodInterceptor), Name="Test")]
public class ClassA
{
    /// HandlerAttributeの付与
    [SampleLogCallHandler]
    public virtual void DoSomething()
    {
        Console.WriteLine("DoSomething実行");
    }
 ・・・
}
```

リスト 16 HandlerAttribute を継承したカスタム属性の付与

なお、上記サンプルコードのクラスに付与された SetInterceptor 属性については、後述する「(3)AOP 対象クラスの実装」を参照のこと。

【コンテナ全体に適用するポリシーを設定するケース】
Unity AB のインターセプタ機能では、インターセプト対象クラスの外側から、コンテナ全体に適用するポリシーを定義して AOP の適用範囲を設定することができる。
ポリシーとして、AOP の適用範囲を決定するための適用ルールと AOP で織り込む処理(前述の ICallHandler インタフェース実装クラス)を設定する。
適用ルールは、UnityAB が標準提供する Microsoft.Practices.Unity.InterceptionExtension.IMatchingRule 実装クラスを利用するか、IMatchingRule インタフェースを実装して独自のルールを実装する。
実装方法の詳細は、Unity AB のドキュメントを参照するとして、ここでは、メソッド名が Execute から始まり(ワイルドカード「*」を利用)、string 型の引数をとるメソッドに前述の SamleLogCallHandler の処理を織り込む例を示す。

```
///UnityContainerを取得
IUnityContainer container =
            UnityManager.GetApplicationContainer(Assembly.GetExecutingAssembly().GetName());
 ・・・

///AOPの適用範囲の設定
///UnityAB標準のMethodSignatureMatchingRuleを利用
///シグニチャがExecuteXXXX(string)に、SampleLogCallHandlerが適用される
Interception interception = UnityContainerUtility.GetRequiredConfigure<Interception>(container);

interception.AddPolicy("ExecuteMethodLogPolicy")
            .AddMatchingRule(
              new MethodSignatureMatchingRule("Execute*", new string[] { "System.String" }, true))
            .AddCallHandler(new SampleLogCallHandler());
```

リスト 17 AOP の適用ポリシーの実装例

(3) AOP 対象クラスの実装

UnityAB の標準機能では、ソースコードまたは構成ファイルで AOP 対象のクラスを設定する。
本機能では、UnityAB を機能拡張し、SetDefaultInterceptor（または SetInterceptor）属性を付与するだけで AOP 対象クラスとすることができる。

表 7　SetDefaultInterceptor 属性の使用方法

項番	パラメータ	必須	内容
1	Type　interceptionType （第1引数）	○	以下のインターセプタクラスの型オブジェクトを指定する。 ・TrasparentProxyInterceptor ・InterfaceInterceptor ・VirtualMethodInterceptor
2	Type TypeToIntercept	―	AOP 対象クラスの型オブジェクトを指定する。 対象クラスが実装しているインタフェースに対してインターセプタを適用する場合に使用する。 指定しない場合は、対象クラスの型オブジェクトになる。

表 8　SetInterceptor 属性の使用方法

項番	パラメータ	必須	内容
1	Type　interceptionType （第1引数）	○	以下のインターセプタクラスの型オブジェクトを指定する。 ・TrasparentProxyInterceptor ・InterfaceInterceptor ・VirtualMethodInterceptor
2	string name （第2引数）	―	SetInterceptor 属性を区別する名前 UnitContainer の Resolve メソッドの引数に名前を指定すると、Name が一致する SetInterceptor 属性が適用されたクラスのインスタンスが生成される。
3	Type TypeToIntercept	―	AOP 対象クラスの型オブジェクトを指定する。 対象クラスが実装しているインタフェースに対してインターセプタを適用する場合に使用する。 指定しない場合は、対象クラスの型オブジェクトになる。

SetIntercpetor と SetDefaultInterceptor の違いは name 指定ができるかどうかのみで、基本的には同じ機能である。SetDefaultInterceptor の場合は、name 指定ができないので、UnityContainer の Resolve メソッドを引数なし（デフォルト）で実行した場合に生成されるインスタンスにインターセプタが適用される。
SetInterceptor 属性は、UnityContainer の Resolve メソッドの引数に name 指定した場合に生成されるインスタンスにインターセプタが適用される。

以下に、SetInterceptor 属性を使った AOP の対象となるクラスの記述例を示す。
この例では、インターセプタとして VirtualMethodInterceptor を設定している。
このため、ClassA クラスの ExecuteA002 メソッドは「Execute」で始まるメソッドであるので、適用ルールに合致するが、virtual 修飾子が付与されていないため、AOP は適用されない。
また、SetInterceptor 属性に name を”Test”という文字列を指定している。これにより、UnityContainer の Resolve メソッドの引数を”Test”として実行したときに VirtualMethodInterceptor が適用されたインスタンスを生成する。

```
///SetInterceptor属性の付与
[SetInterceptor(typeof(VirtualMethodInterceptor), "Test")]
public class ClassA
{
    /// HandlerAttributeの付与
    [SampleLogCallHandler]
    public virtual void DoSomething()
    {
        Console.WriteLine("DoSomething実行");
    }

    ///この例ではVirtualMethodInterceptorを適応するので
    ///virtual修飾子を付与
    public virtual void ExecuteA001(string input)
    {
        Console.WriteLine("ExecuteA001出力:" + input);
    }
    ///シグニチャが一致しないので
    ///SampleLogCallHandlerは適用されない
    public virtual void WriteMessage(string input)
    {
        Console.WriteLine("WriteMessage出力:" + input);
    }
    ///virtualを付与しないとシグニチャが一致しても、
    ///SampleLogCallHandlerは適用されない
    public void ExecuteA002(string input)
    {
        Console.WriteLine("ExecuteA002出力:" + input);
    }
}
```

リスト 18 SetInterceptor 属性を付与した AOP 対象クラスの記述例

(4) 構成ファイルの記述

TerasolunaFramework.config や TerasolunaApplication.config など、AOP を利用したい UnityContainer に対応する構成ファイルについてそれぞれ、AOP を有効化する設定を記述する必要がある。
まず、UnityAB 標準のインターセプタの機能を有効にするためには、/configuration/unity/containers/container/extensions/add タグで、以下の UnityContainerExtension 継承クラスを記述する。
UnityContainerExtension クラスについては、後述の「Extension クラスによるフレームワーク機能の統合」を参照のこと。

- Microsoft.Practices.Unity.InterceptionExtension.Interception クラス
  - UnityAB が標準で提供しているインターセプタを有効にするための Extension
- Terasoluna.Unity.SetInterceptorExtension クラス
  - SetInterceptor 属性を有効にするための Extension

以下に、設定例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="unity"
         type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                    Microsoft.Practices.Unity.Configuration,
                    Version=1.2.0.0,
                    Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <unity>
    <typeAliases>
      ・・・
    </typeAliases>
    <containers>
      <container>
        <types>
          ・・・
        </types>
        <extensions>
          <!-- AOPの設定 -->
          <add type="Microsoft.Practices.Unity.InterceptionExtension.Interception,
                      Microsoft.Practices.Unity.Interception,
                      Version=1.2.0.0,
                      Culture=neutral,
                      PublicKeyToken=31bf3856ad364e35" />
          <add type="Terasoluna.Unity.SetInterceptorExtension,
                     Terasoluna" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 19 AOP の設定例

(5) 実行例

AOP を利用する実行プログラムの記述例を以下に示す。

```
class Program
{
    public static void Main(string[] args)
    {
        try
        {
            TerasolunaFramework.Run();
            IUnityContainer container =
                UnityManager.GetApplicationContainer(Assembly.GetExecutingAssembly().GetName(
                ));
            Interception interception =
                UnityContainerUtility.GetRequiredConfigure<Interception>(container);
            interception.AddPolicy("ExecuteMethodLogPolicy")
            .AddMatchingRule(
                new MethodSignatureMatchingRule("Execute*", new string[] { "System.String" }, true))
            .AddCallHandler(new SampleLogCallHandler());

            ///SetInterceptor属性にしたnameを指定
            ClassA classA = container.Resolve<ClassA>("Test");
            Console.WriteLine("####name=Testで生成したClassAインスタンス####");
            ///DoSomethingメソッド実行
            classA.DoSomething();
            ///ExecuteA001メソッド実行
            classA.ExecuteA001("あいうえお");
            Console.WriteLine();
            ///WriteMessageメソッド実行
            classA.WriteMessage("さしすせそ");
            Console.WriteLine();
            ///ExecuteA002メソッド実行
            classA.ExecuteA002("たちつてと");
            Console.WriteLine();

            ///SetInterceptor属性にしたnameを指定しない（デフォルト）
            Console.WriteLine("####name指定なしで生成したClassAインスタンス####") ;
            ClassA classA2 = container.Resolve<ClassA>();
            ///DoSomethingメソッド実行
            classA2.DoSomething();
            Console.WriteLine();
            ///ExecuteA001メソッド実行
            classA2.ExecuteA001("あいうえお");
        }
        finally
        {
            TerasolunaFramework.Exit();
        }
    }
}
```

リスト 20 サンプルプログラムの実行例

サンプルコードの実行結果を以下に示す。
カスタム属性の付与により DoSomething メソッドにログが出力されているのが分かる。
また、適用ポリシーに基づき ExecuteA001 メソッドにログが出力されているのが分かる。
一方、ExecuteA002 メソッドは、virtual メソッドでないため AOP が適用されていない。
SetInterceptor 属性に指定した name を指定しないでインスタンス生成した場合にも、AOP が適用されていない。

```
####name=Test で生成した ClassA インスタンス####
<<SampleLogCallHandler>>DoSomething メソッド実行
DoSomething 実行
<<SampleLogCallHandler>>DoSomething メソッド終了
<<SampleLogCallHandler>>ExecuteA001 メソッド実行
ExecuteA001 出力:あいうえお
<<SampleLogCallHandler>>ExecuteA001 メソッド終了

WriteMessage 出力:さしすせそ

ExecuteA002 出力:たちつてと

####name 指定なしで生成した ClassA インスタンス####
DoSomething 実行

ExecuteA001 出力:あいうえお
```

リスト 21 サンプルコードの実行結果

◆ Extension クラスによるフレームワーク機能の統合

Unity AB では、Extension クラスにより、Unity AB 標準のインスタンス生成処理にカスタム処理を追加することができる。
Extension クラスは、Microsoft.Practices.Unity.UnityContainerExtension クラスを継承し作成する。Enterprise Library[9]では、Extension クラスを利用した Application Block の統一的なオブジェクト生成・取得方法を提供している。各 Application Block 固有の Extension クラスが用意されており、Extention クラスで実装オブジェクトの情報を事前に登録しておくことで、UnityContainer から Application Block の実装オブジェクトを取得することができるため、Application Block の拡張に柔軟に対応できるようになっている。
TERASOLUNA フレームワークも Enterprise Library と同様に、各フレームワーク機能が、標準機能を実装するクラス情報を事前に登録しておく固有の Extension クラスを用意しており、この Extension クラスを UnityContainer に追加することで、フレームワークの標準機能が利用できるようになっている。
このため、各開発プロジェクトの要件に合わせて TERASOLUNA フレームワーク の機能を拡張したい場合は、標準実装クラスの代わりに機能拡張した実装クラスの情報を登録するための Extension クラスを作成し、UnitContainer に追加する Extension クラスを差し替えることで簡単に機能拡張することができる。
また、Extension クラスによる機能拡張は、構成ファイルで実装クラス情報の設定を1つ1つ記述していくよりも設定ファイルの記述量を大幅に削減でき、保守性や可読性の面でも有利である。

以下に、UnityContainerExtension の実装例として、「CL-02 画面遷移機能」の Extension クラスである FormForwardExtension クラスの実装を示す。

---

9 マイクロソフトが推進するオープンソース・ライブラリ。TERASOLUNA も一部その機能を利用している。
[http://www.codeplex.com/entlib]

```
namespace Terasoluna.Windows.Forms.FormForward
{
    public class FormForwardExtension : UnityContainerExtension
    {
・・・
        protected override void Initialize()
        {
・・・
            ///フレームワークの機能を構成するインタフェースに対する実装クラスをDIする
            Container.RegisterType<FormForwardManager>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormForwardFlowController, FormForwardFlowController>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormFactory, AttributeFormFactory>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormOpener, FormOpener>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormInfoManager, FormInfoManager>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormDataCopyManager, FormDataCopyManager>(
                new ContainerControlledLifetimeManager());
            Container.RegisterType<IFormForwardGroupScopeManager,
                    FormForwardGroupScopeManager>(
                new ContainerControlledLifetimeManager());
        }
    }
}
```

リスト 22 Extension クラスの実装例(FormFowardExtension クラス)

リスト 22 を見ると分かるように、UnityContainerExtension.Initialize メソッドをオーバーライドして、フレームワーク機能を構成するインタフェースに対する実装クラスを登録している。
各開発プロジェクトの要件に合わせて TERASOLUNA フレームワーク の機能を拡張したい場合も同様に、UnityContainerExtension.Initialize メソッドで標準実装クラスの代わりに機能拡張した実装クラスを登録するように Extension クラスを作成する。なお、Extension クラスによる各フレームワークの機能拡張の手順は、各フレームワーク機能説明書を参照すること。

次に、Extension クラスを UnityContainer へ追加登録するには、以下の2つの方法がある

- 構成ファイル上で、/configuration/unity/containers/container/extensions/add タグを使って Extension クラスを設定する。
- ソースコード上で、UnityContainer.AddExtension(または AddNewExtension)メソッドを実行し、Extension クラスを追加する。

ここでは、構成ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
        <extensions>
          <!-- 画面遷移機能のデフォルト設定 -->
          <add type="Terasoluna.Windows.Forms.FormForward.FormForwardExtension,
                    Terasoluna.Windows.Forms" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 23 構成ファイルの記述例（FormForwardExtension クラス）

■ 内部構成

◆ 構成クラス

インスタンス管理機能を構成するクラスを以下に示す。

表 9 構成クラス一覧

項番	クラス名	説明
Terasoluna.Unity		
1	UnityManager	TERASOLUNA フレームワーク が使用するインスタンスを管理するクラス。
2	IUnityContainerBuilder	UnityContainer を生成するインタフェース。
3	RuntimeUnityContainerBuilder	実行時に利用する IUnityContainerBuilder の実装クラス。
4	IUnityContainerLoader	UnityContainer をロードするインタフェース。
5	RuntimeUnityContainerLoader	実行時に利用する IUnityContainerLoader の実装クラス。
6	EmptyUnityContainerLoader	IUnityContainerLoader の空実装クラス。
7	IUnityContainerImportManager	IUnityContainerImporter を管理し、分割した Unity の設定ファイルをマージするインタフェース
8	UnityContainerImportManager	IUnityContainerImportManager の実装クラス。
9	IUnityContainerImporter	構成ファイルから Unity セクションの情報を UnityContainer へロードするクラス。
10	ConfigUnityContainerImporterBase	IUnityContainerImporter の基底クラス。
11	AppConfigUnityContainerImporter	アプリケーション構成ファイルにある Unity の設定情報を取り込む IUnityContainerImporter 実装クラス。
12	AssemblyConfigUnityContainerImporter.cs	アセンブリ名から特定した設定ファイルにある Unity の設定情報を取り込む IUnityContainerImporter 実装クラス。
13	FileConfigUnityContainerImporter	パスで指定した設定ファイルにある Unity の設定情報を取り込む IUnityContainerImporter 実装クラス。
14	RegisteredTypeManager	UnityContainer に登録した型を管理するクラス
15	SetDefaultInterceptorAttribute	IInterceptor の実装クラスを定義する属性クラス。
16	SetInterceptorAttribute	IInterceptor の実装クラスを定義する属性クラス。

項番	クラス名	説明
17	SetInterceptorExtension	SetInterceptor 属性を有効にする UnityContainer の機能を拡張するためクラス。
18	SetInterceptorBuilderStrategy	UnityContainer のインスタンス生成のパイプライン処理で、SetInterceptor 属性を処理するストラテジクラス。
19	UnityConfigurationUtility	Unity AB の設定ファイルを読み込むユーティリティクラス。
20	UnityContainerUtility	IUnityContainer に対する拡張機能を提供するユーティリティクラス。
21	UnityContainerExtensionUtility	UnityContainerExtension に関するユーティリティクラス。
22	UnityInterceptionUtility	AOP 機能に関するユーティリティクラス。
23	SaveStackTraceCallHandler	例外を再スローした場合に StackTrace が消失しないように、AOP で StactTrace を退避する CallHandler

■ 拡張ポイント

(1) Extension クラスによるインスタンス生成機能の拡張

Unity AB の内部では、ObjectBuilder2 という Applicaton Block を使用してインスタンス生成を行っている。ObjectBuilder2 は、インスタンス生成のメカニズムとして、「ステージ」と呼ばれるパイプラインを構築する。Unity AB は、ObjectBuilder2 のデフォルトのパイプラインを拡張し、7つのステージ(Setup、TypeMapping、Lifetime、PreCreation、Creation、Initialization、PostInitialization)[10]を用意している。「ステージ」には、オブジェクトの生成、破棄を管理する「ストラテジ」が複数存在し、各「ステージ」に登録されている「ストラテジ」が順番に実行されていくことでインスタンス生成を行う。

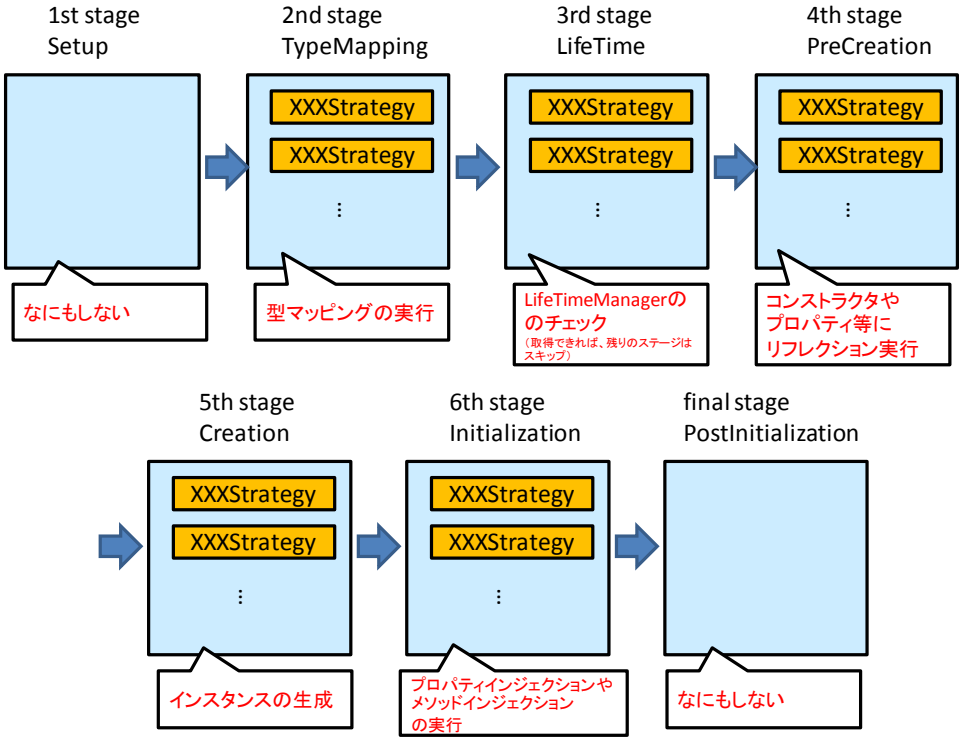

図 26 UnityAB のインスタンス生成パイプラインのイメージ

UnityAB では、前述の Extension クラスを使用して独自のストラテジをいずれかのステージに追加することでインスタンス生成パイプラインに独自の処理を追加することできる。

Extension によるインスタンス生成機能拡張方法の詳細については、Unity AB のドキュメントや Unity QuickStarts のサンプルコード等を参照すること。

ここでは本機能が提供する SetInterceptorExtension クラスを例に拡張方法の概要を説明する。SetInterceptorExtension クラスは UnityContainerExtension 継承クラスで、TERASOLUNA フレームワークが独自に定義した SetInterceptor 属性が付与されたクラスについてインターセプタを適用したインスタンスを生成するようにインスタンス生成機能ををを拡張している。

SetInterceptorExtension クラスの内部では、UnityContainerExtensioin.Intialize メソッドを

---

[10] UnityBuildState 列挙型の定義を参照。

オーバーライドして、Unity AB のインスタンス生成パイプラインに、BuiderStrategy の継承クラスした独自のストラテジ（Terasoluna.Unity.SetInterceptorBuilderStrategy）クラスを作成し、Setup ステージに追加している。

```
public class SetInterceptorExtension：UnityContainerExtension
{
    protected override void Initialize()
    {
        ・・・
        ///Unityのインスタンス生成パイプラインに、独自のBuildStrategyを追加する。
        SetInterceptorBuilderStrategy strategy = new SetInterceptorBuilderStrategy(this);
        Context.Strategies.Add(strategy, UnityBuildStage.Setup);
        ・・・
    }
}
```

リスト 24 Extension クラスの実装例（SetInterceptorExtension クラス）

BuilderStrategy 継承クラスを Unity のインスタンス生成パイプラインに追加することで、標準のインスタンス生成に処理を追加することができる。

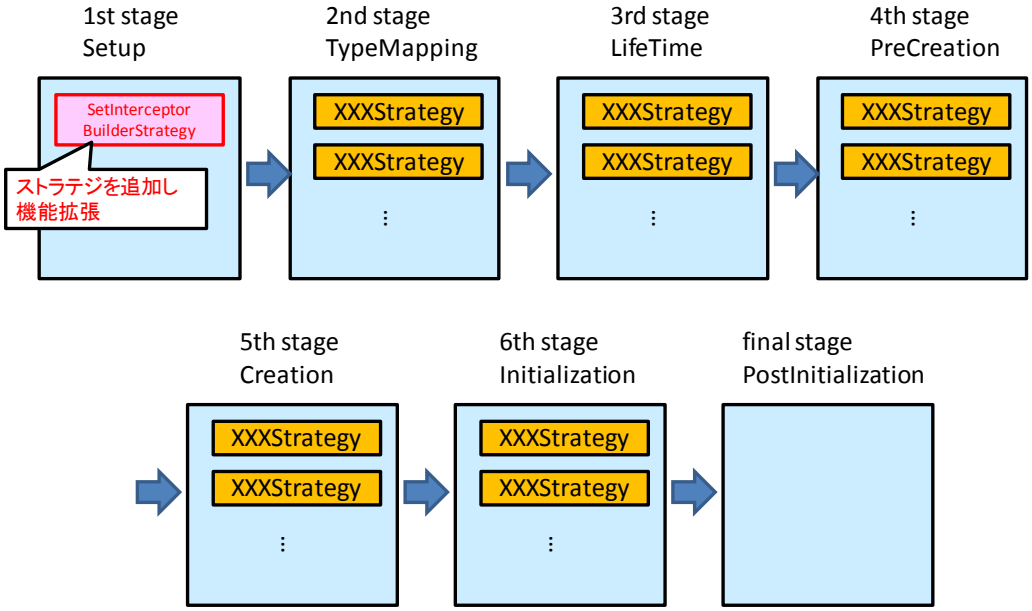

図 27 SetInterceptorBuilderStrategy の追加による機能拡張イメージ

SetInterceptorBuilderStrategy クラスでは、BuiderStrategey.PreBuildUp メソッドをオーバーライドし、インスタンス生成の前処理として SetInterceptor 属性があればインターセプタを適用するように実装している。
各プロジェクトでインスタンス生成機能拡張が必要な場合も同様に、BuilderStrategy クラスを継承し拡張したい処理を実装し、適切なステージに追加することでインスタンス生成機能を拡張することができる。

```
public class SetInterceptorBuilderStrategy : BuilderStrategy
{
        ・・・
        /// <summary>
        ///  ビルド前処理を実行する。
        /// </summary>
        public override void PreBuildUp(IBuilderContext context)
        {
            ///SetInterceptorAttribute属性を見てインターセプターの登録処理する
            ・・・
        }
}
```

リスト 25 BuilderStrategy 継承クラスの実装例(SetInterceptorBuilderStrategy)

また、Extension クラスを有効にするためには、構成ファイルやソースコードにより
UnityContainer に Extension クラスを追加する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
        <extensions>
          <!-- SetInterceptor属性を有効にするためのUnity拡張機能を設定 -->
          <add type="Terasoluna.Unity.SetInterceptorExtension, Terasoluna" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 26 Extension クラスの設定例(SetInterceptorExtension クラス)

(2) 構成ファイルの複数分割・切り出し

通常、1つの UnityContainer に対して1つの構成ファイルが紐づいているが、これをさらに複数の構成ファイルに分割して別ファイルへ切り出したい場合には、対象のコンテナに対する構成ファイルに、Terasoluna.Unity.IUnityContainerImporter クラスの DI 定義を追加する。
例えば、TerasolunaApplication.config の設定内容を別ファイルへ切り出したい場合には、TerasolunaApplication.config に、IUnityContainerImporter の DI 設定を記述する。
IUnityContainerImporter インタフェースは、Unity の設定を読み込むためのインタフェースである。本機能のデフォルトの動作では、各 UnityContainer に対応する特定の構成ファイルを読み取る IUnityContainerImporter 実装クラスが設定をロードするようになっている。ここで、IUnityContainerImporter の DI 設定を追加すると、その IUnityContainerImporter が読み込む設定内容もマージする。
IUnityContainerImporter の実装クラスとして、FileConfigUnityContainerImporter クラスが提供されているのでこれを利用する。
FileConfigUnityContainerImporter は、ConfigurationPath プロパティで指定したパスに存在する構成ファイルを読み取る。
以下に、IUnityContainerImporter の設定例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <typeAliases>
      <!-- typeAliasの設定 -->
      <typeAlias alias="string" type="System.String" />
      <typeAlias alias="Importer" type="Terasoluna.Unity.IUnityContainerImporter, Terasoluna" />
      <typeAlias alias="FileConfigUnityContainerImporter"
       type="Terasoluna.Unity.FileConfigUnityContainerImporter, Terasoluna" />
    <containers>
      <container>
        <types>
          <!-- IUnityContainerImporterのDI定義を記述することで、複数ファイルへ分割可能 -->
          <type type="Importer" name="UnityConfig2" mapTo="FileConfigUnityContainerImporter">
            <typeConfig>
              <property name="ConfigurationPath" propertyType ="string">
                <!-- 分割した設定ファイルの相対パスを記述 -->
                <value value="Terasoluna.Laboratories2.config"/>
              </property>
            </typeConfig>
          </type>
          <type type="Importer" name="UnityConfig3" mapTo="FileConfigUnityContainerImporter">
            <typeConfig>
              <property name="ConfigurationPath" propertyType ="string">
                <!-- 分割した設定ファイルの相対パスを記述 -->
                <value value="UnityTest¥Terasoluna.Laboratories3.config"/>
              </property>
            </typeConfig>
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 27 構成ファイルの分割の切り出しの設定例

■ 関連機能

「CM-01 アプリケーション起動・終了機能」