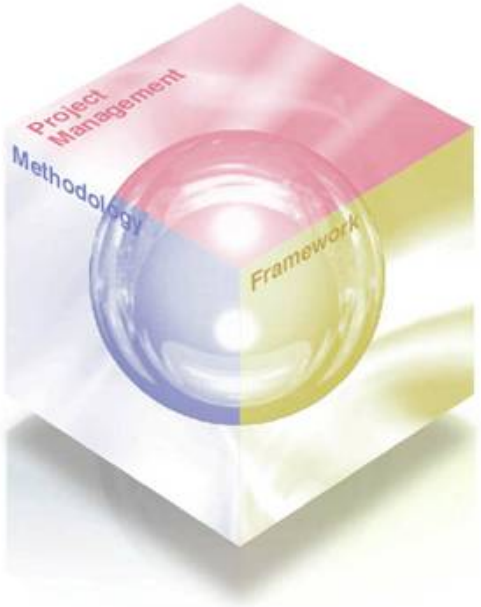
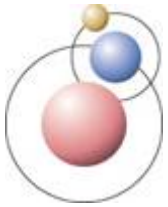
TERASOLUNA® Server/Client Framework for .NET 2.1.0.1
アーキテクチャ説明書(共通編)



株式会社NTTデータ

TERASOLUNA®
BEST SOLUTION FOR SYSTEM DEVELOPMENT

NTT DATA



本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTTデータあるいはNTTデータに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、およびNTTデータの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献：TERASOLUNA Server/Client Framework for .NET アーキテクチャ説明書（共通編）」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTTデータの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTTデータは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTTデータは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求（第三者との間の紛争を理由になされる請求を含む。）に関しても、NTTデータは一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

- Microsoft、Visual Stuido、Windows、.NET Frameworkは、米国Microsoft Corp.の米国及びその他の国における登録商標または商標です。
- TERASOLUNAは、株式会社NTTデータの登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

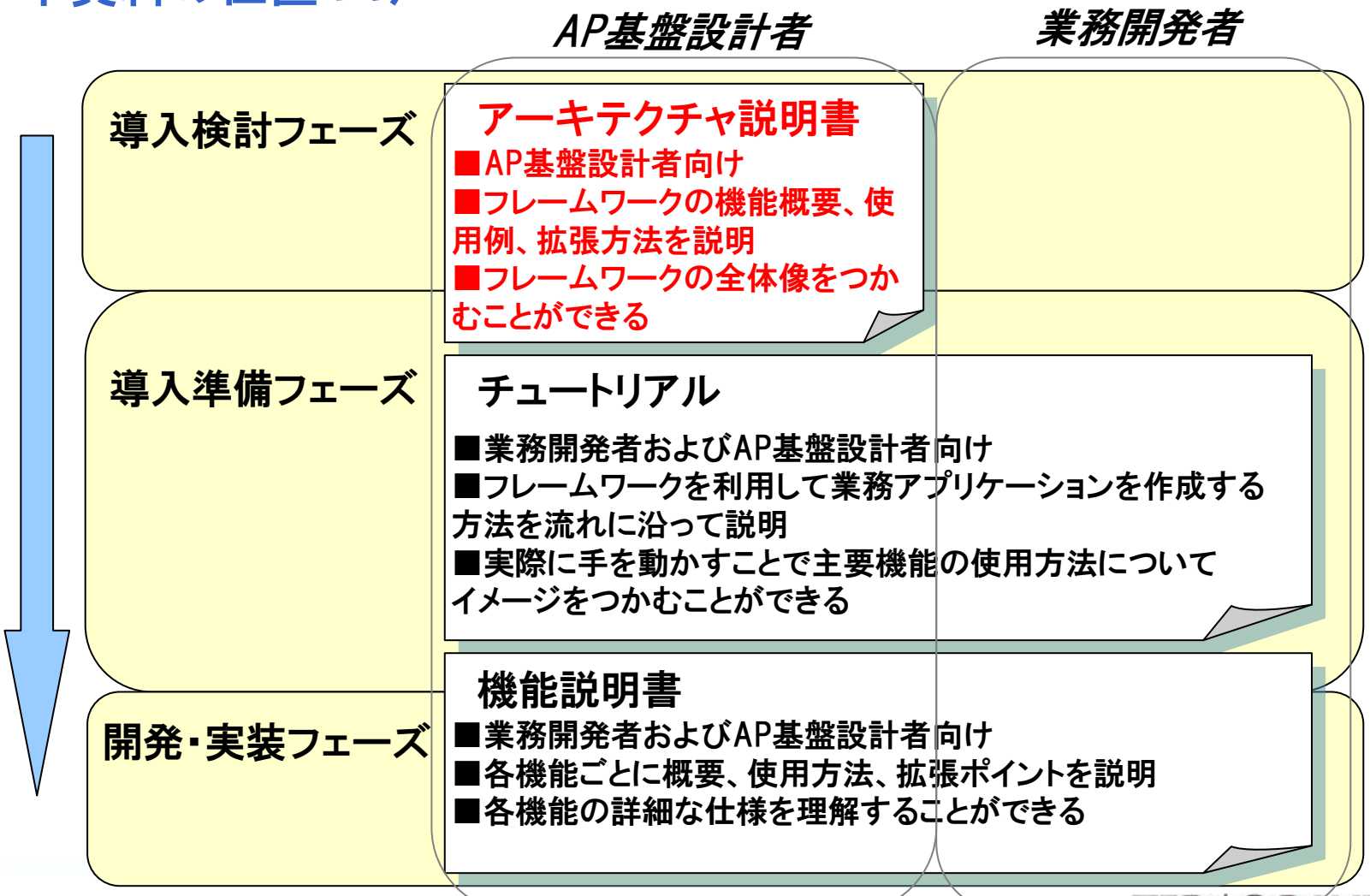
目次

はじめに(1/2)

- 概要
  - 本資料は、「TERASOLUNA Server/Client Framework for .NET 2.1」における共通機能について解説した資料である
    - 共通機能とは
      - Server/Client Framework両方で利用されている機能である
      - 独立したライブラリとして提供しているため、共通機能単体でも利用可能である
  - Server/Client Frameworkの個別機能についてはそれぞれのアーキテクチャ説明書を参照のこと

はじめに(2/2)

本資料の位置づけ



機能概要

共通機能の概要

CM-01 メッセージ管理機能

アプリケーションで扱うメッセージに対し、統一的にアクセスする仕組みを提供

CM-02 入力値検証機能

業務アプリケーションで利用可能な各種入力値検証ルールを提供

データセットに対する入力値検証機能を提供

CM-03 ログ出力機能

アプリケーションで統一的にログを出力する仕組みを提供

CM-04 ビジネスロジック生成機能

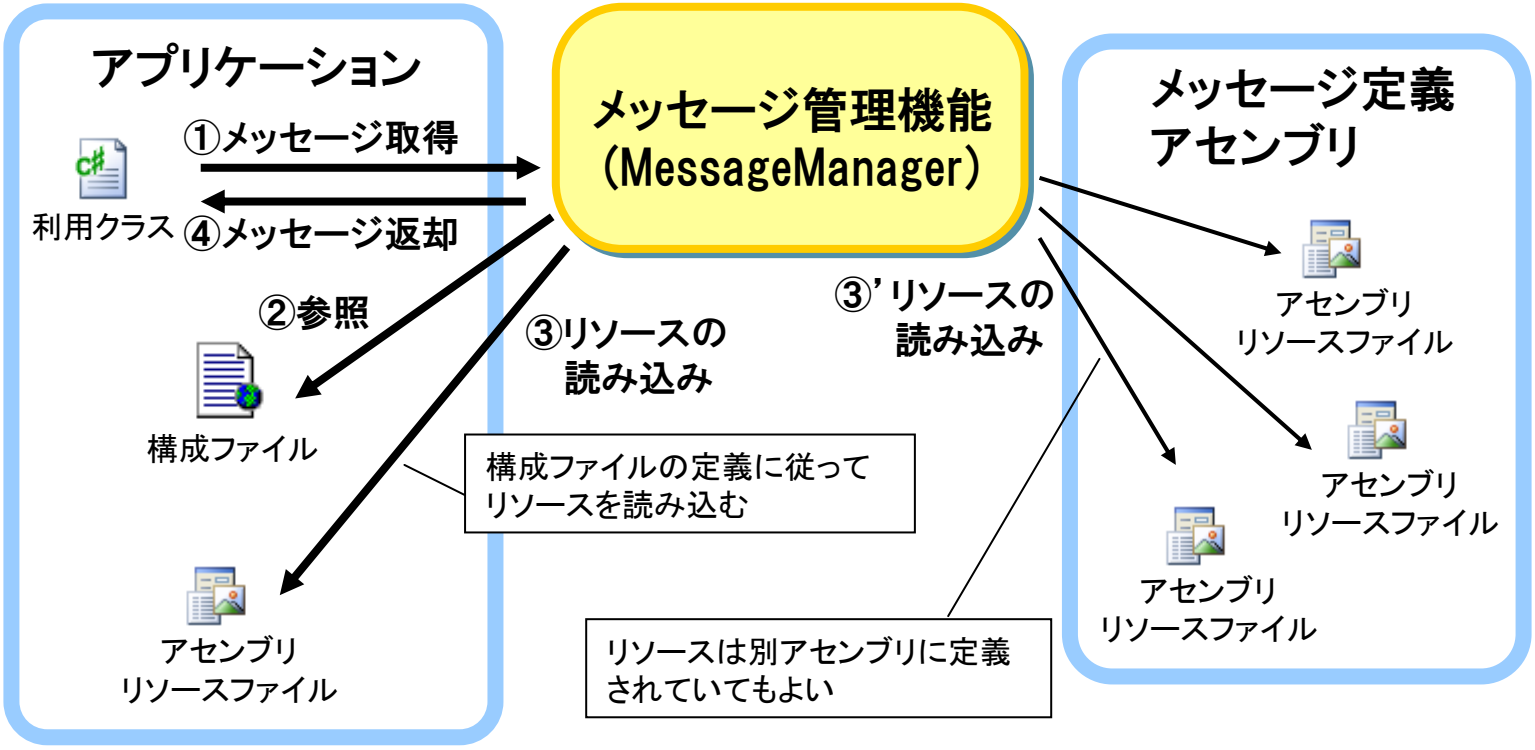
ビジネスロジッククラスのインスタンスを生成する機能を提供

CM-01 メッセージ管理機能 – 概要

- アプリケーションで扱うメッセージに対し、統一的にアクセスする仕組みを提供する
  - 複数のアセンブリリソースファイルから統一的な方法でメッセージを取得可能
    - 業務開発者はフレームワークが提供するMessageManagerクラスのメソッドを呼び出すことで、メッセージを取得する
    - メッセージを取得するクラスとアセンブリリソースファイルは異なるアセンブリでもよい

CM-01 メッセージ管理機能 – 動作イメージ

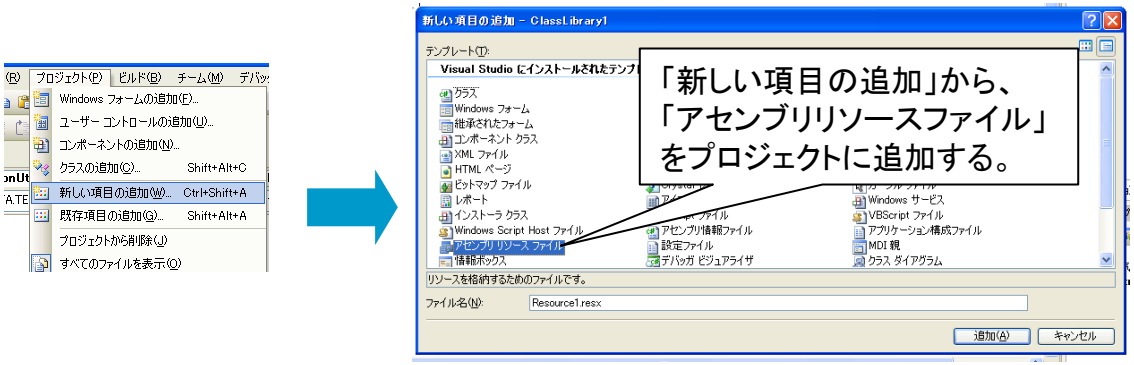
メッセージ管理機能を利用したメッセージ取得の流れ



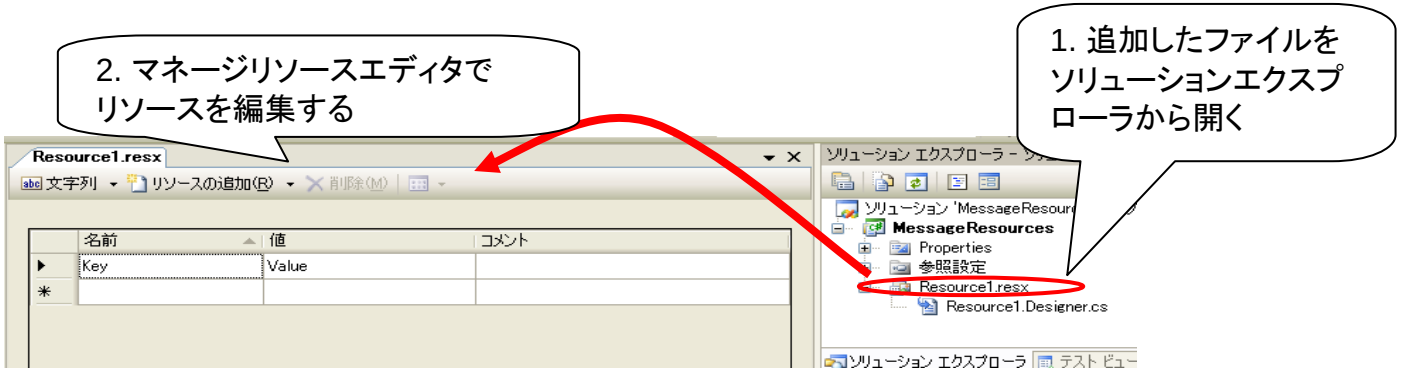
CM-01 メッセージ管理機能 – 使用例(1/3)

メッセージリソースの作成と編集

①プロジェクトにアセンブリリソースファイルを追加する



②追加したファイルをマネージリソースエディタで編集する



CM-01 メッセージ管理機能 – 使用例(2/3)

メッセージリソースの指定

①利用するメッセージリソースの型を、構成ファイルに完全修飾名で記述する

- メッセージリソースは複数指定することができる
- メッセージリソースの型は、メッセージを利用するアセンブリと同一でなくてもよい

構成ファイルの記述例

```
<appSettings>
   <add key="MessageResources.MyMessage01"
        value="MessageResources.Resource01, MessageResources"/>
   <add key="MessageResources.MyMessage02"
        value="MessageResources.Resource02, MessageResources"/>
   <add key="MessageResources.MyMessage00"
        value="CommonResources.CommonResource, CommonResources"/>
</appSettings>
```

リソース1
リソース2
リソース3

複数のリソースを指定した場合、key属性に指定した文字列の辞書順にリソースを検索する。
この例では、まずリソース3が検索され、リソース3でメッセージが見つからなかった場合は、リソース1、リソース2の順に検索される。

add要素
key 属性: "MessageResource."をプレフィックスとする文字列
value属性: メッセージリソースの完全修飾名

CM-01 メッセージ管理機能 – 使用例(3/3)

メッセージの取得

MessageManagerクラスのGetMessageメソッドを用いてメッセージを取得する

メッセージリソースの定義例

名前	値
NULL_EXCEPTION	null 参照です。
ARG_NULL_EXCEPTION	引数 “{0}” が null 参照です。

メッセージを取得するコード例

書式項目を利用してフォーマットされたメッセージを取得することもできる

```
string message1 = MessageManager.GetMessage("NULL_EXCEPTION");
string message2 = MessageManager.GetMessage("ARG_NULL_EXCEPTION", "userName");
```

message1 : null 参照です。
message2 : 引数 “userName” が null 参照です。

CM-01 メッセージ管理機能 – 拡張方法

- 独自のメッセージ管理クラスを利用したい場合

①MessageManager継承クラスを作成する

Initメソッドをオーバーライドし、ResourceManagerListの初期化ロジックを変更することで、利用するメッセージリソースの読み込み方法や、検索順序をカスタマイズすることができる。

②作成したメッセージ管理クラスを利用するための設定を構成ファイルに記述する

構成ファイルの記述例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MessageManagerTypeName"
         value="MyNamespace.MessageManagerEx, MyNamespace"/>
  </appSettings>
</configuration>
```

add要素
key 属性: "MessageManagerTypeName"
value属性: MessageManager継承クラスのセンブリ修飾名

CM-02 入力値検証機能 – 概要(1/2)

- 設定ファイルに検証ルールを記述することで、プログラミングすることなく入力値検証を利用する仕組みを提供する
  - データセットが保持するテーブルの各カラムに対して検証を行う機能を提供
  - 業務アプリケーションで利用可能な各種入力値検証ルールを提供
  - 相関チェックは別途実装する必要がある
  - この機能はイベント処理機能、またはリクエストコントローラ機能から呼び出される

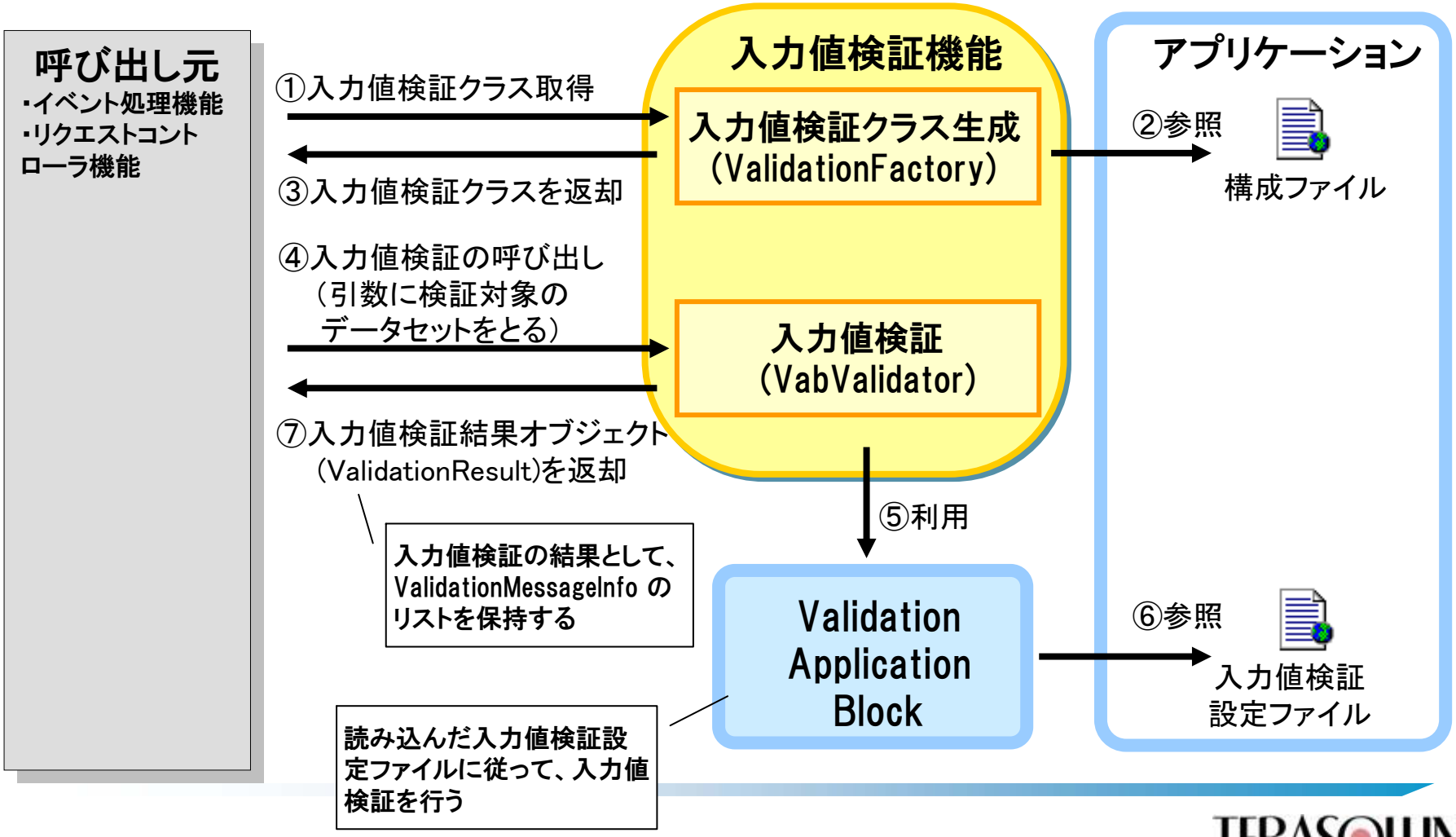
CM-02 入力値検証機能 – 概要(2/2)

提供ルール一覧

項番	ルール	概要
1	正規表現一致チェック	正規表現パターンを指定して、検証対象文字列がパターンとマッチするかを検証する。
2	必須入力チェック	nullまたはホワイトスペース(半角空白、全角空白、改行、タブ文字)でないか検証する。
3	数値文字列チェック	半角数値文字のみで構成されているか検証する。
4	半角英数大文字列チェック	大文字の半角英数文字のみで構成されているか検証する。
5	半角英数文字列チェック	半角英数文字のみで構成されているか検証する。
6	半角文字列チェック	半角文字のみで構成されているか検証する。
7	半角カナ文字列チェック	半角カナ文字のみで構成されているか検証する。
8	全角カナ文字列チェック	全角カナ文字のみで構成されているか検証する。
9	全角文字列チェック	全角文字のみで構成されているか検証する。
10	URL形式チェック	URL形式の文字列であるか検証する。
11	日付形式チェック	日付形式の文字列であるか検証する。
12	int型範囲チェック	int型に変換可能であり、指定した範囲であるか検証する。
13	decimal型範囲チェック	decimal型に変換可能であり、指定した範囲であるか検証する。
14	float型範囲チェック	float型に変換可能であり、指定した範囲であるか検証する。
15	double型範囲チェック	double型に変換可能であり、指定した範囲であるか検証する。
16	日付型範囲チェック	dateTime型に変換可能であり、指定した範囲の日時であるかを検証する。
17	byte列長範囲チェック	指定したエンコーディングでバイト列に展開した際のバイト長が指定した範囲であるか検証する。
18	文字列長チェック	指定した文字数の範囲であるかを検証する。
19	必須文字列チェック	指定した文字列を含んでいるかどうかを検証する。
20	数値チェック	数値に変換可能であり、整数部・小数部がそれぞれ指定した桁数であるか検証する。
21	型チェック	指定した型に変換可能であるか検証する。
22	要素数チェック	コレクション(Listや配列など)であった場合、指定した要素数の範囲であるかを検証する。

CM-02 入力値検証機能 – 動作イメージ

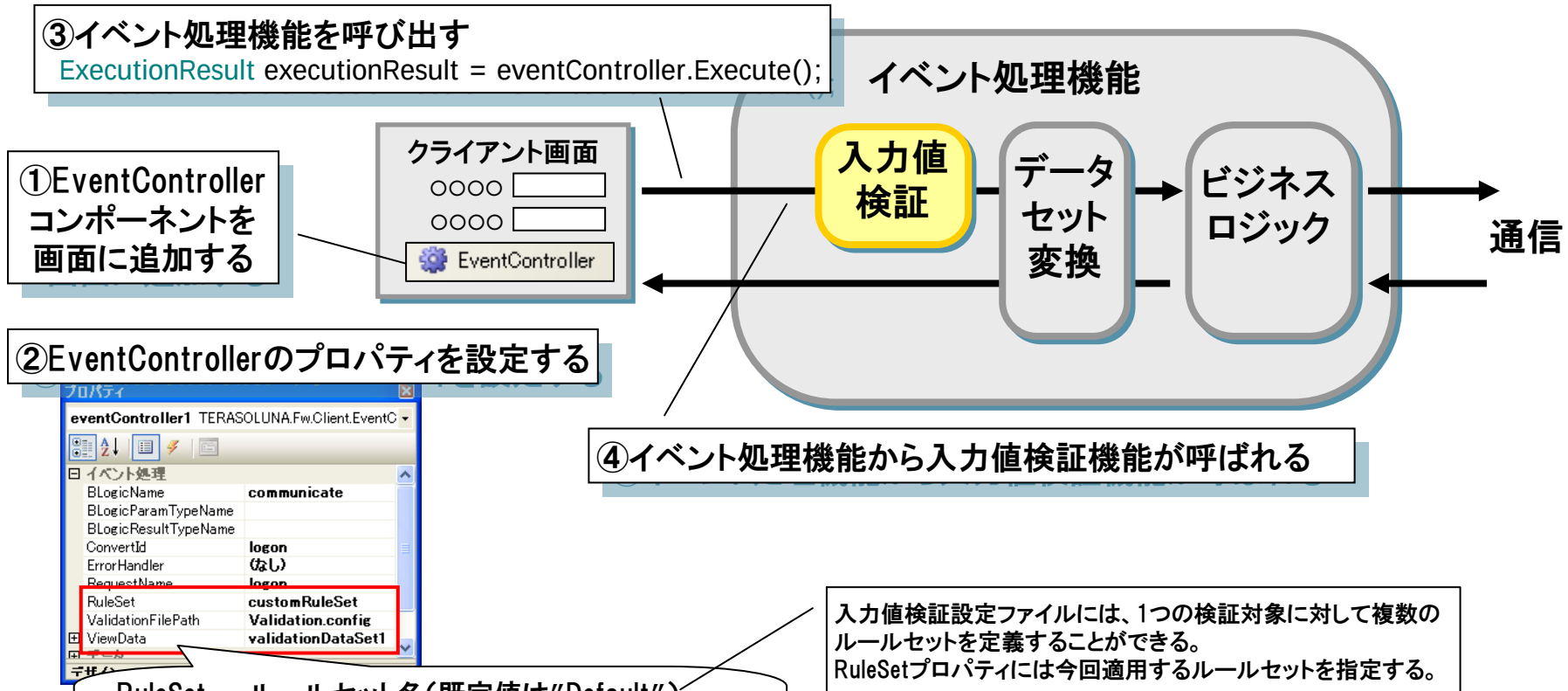
入力値検証機能を利用したフレームワーク内部の動作



CM-02 入力値検証機能 – 使用例(1/3)

- 「イベント処理機能」からの呼び出し例

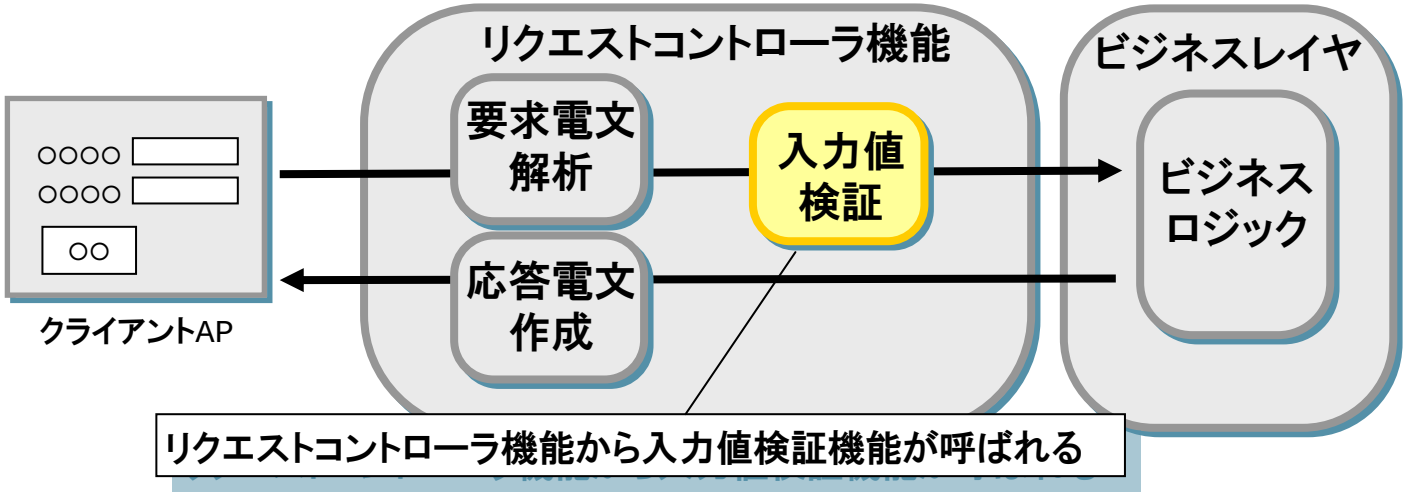
  EventControllerコンポーネントの ValidationFilePath プロパティに入力値検証設定ファイルのパスを指定することで、入力値検証機能を利用できる



CM-02 入力値検証機能 – 使用例(2/3)

- 「リクエストコントローラ機能」からの呼び出し例

  ビジネスロジッククラスのクラス属性に、ValidationFilePathを定義することで、ビジネスロジックの入力値に対する検証を行うことができる



ビジネスロジッククラスの実装例

InputDataSetType・・・入力値検証対象のデータセットの型

```
[ControllerInfo(RequestType=RequestTypeNames.NORMAL, InputDataSetType=typeof(ValidationDataSet))]
[ValidationInfo(ValidationFilePath="Validation.config", RuleSet="customRuleSet")]
public class BLogic01 : IBLogic  { }
```

ValidationFilePath・・・入力値検証設定ファイルのパス
RuleSet・・・ルールセット名(既定値は"Default")

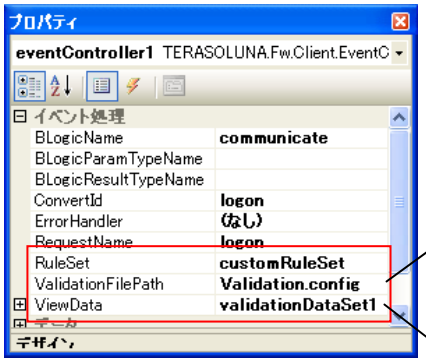
CM-02 入力値検証機能 – 使用例(3/3)

入力値検証設定ファイルの記述例

Table2テーブルのColumn3カラムに格納されている値に対して、全角カナ文字列チェックを行う場合の記述例

「イベント処理機能」からの呼び出し時

EventControllerのプロパティ



「リクエストコントローラ機能」からの呼び出し時

ビジネスロジッククラス

```
[ControllerInfo(RequestType=RequestTypeNames.NORMAL,
InputDataSetType=typeof(ValidationDataSet))]
[ValidationInfo(ValidationFilePath="Validation.config",
RuleSet="customRuleSet")]
public class BLogic01 : IBLogic  { }
```

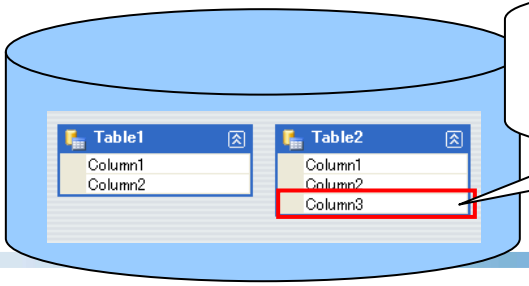
Validation.config(入力値検証設定ファイル)

```
<validation>
  <type assemblyName="Test" name="Test.ViewData.ValidationDataSet+Table2Row">
    <ruleset name="customRuleSet">
      <properties>
        <property name="Column3">
          <validator negated="false" tag="Column1" name="ZenkakuKanaStringValidator"

type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuKanaStringValidator,
                TERASOLUNA.Fw.Common"/>
        </property>
      </properties>
    </ruleset>
  </type>
</validation>
```

RuleSet

入力値検証対象のデータセット



【入力チェックの例】
Table2テーブルのColumn3カラムに格納されている値に対して、全角カナ文字列チェックを行う

CM-02 入力値検証機能 – 拡張方法(1/3)

- 独自の検証ルールを利用したい場合

①Validator※1継承クラスを作成する

Validatorは入力値検証を実施するクラスである。

DoValidateメソッドをオーバーライドし、入力値に対する検証ルールを変更する。

②ValidatorData※2継承クラスを作成する

ValidatorDataは入力値検証設定ファイルの情報解析とValidatorインスタンスの生成・初期化を行うクラスである。

DoCreateValidatorメソッドをオーバーライドし、設定情報からValidatorを生成するロジックを変更する。

※1 Microsoft.Practices.EnterpriseLibrary.Validation.Validator

※2 Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidatorData

CM-02 入力値検証機能 – 拡張方法(2/3)

- 独自の入力値検証機能を利用したい場合
  - ①入力値検証インターフェイス(IValidator※3)を実装したクラスを作成する
    - ValidationFilePathプロパティ
      - 入力値検証設定ファイルのパスを指定する
    - Validate(dataSet)メソッド
      - データセットに対する検証処理を実装する

※3 TERASOLUNA.Fw.Common.Validation.IValidator

CM-02 入力値検証機能 – 拡張方法(3/3)

②構成ファイルに、利用したい入力値検証機能クラス(IValidator実装クラス) の型名を指定する

- 入力値検証のクラスを生成するValidatorFactoryは、構成ファイルに指定された型のインスタンスを生成する
- 構成ファイルに入力値検証機能クラスを指定しない場合は、標準のVabValidatorが使用される
- 構成ファイルに設定できる入力値検証機能クラスは1つのみである

構成ファイルの設定例

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ValidatorTypeName"value="TERASOLUNA.Fw.Common.Validation.VabValidator,TERASOLUNA.Fw.Common.Validation"/>
  </appSettings>
</configuration>
```

appSettings要素に“ValidatorTypeName”をキーとして、IValidator実装クラスの完全修飾名を記述する

CM-03 ログ出力機能 – 概要

- アプリケーションで統一的にログを出力する仕組みを提供する
  - Jakarta Commons Logging 相当の共通ロギングライブラリとして、以下のインターフェイス/クラスを提供

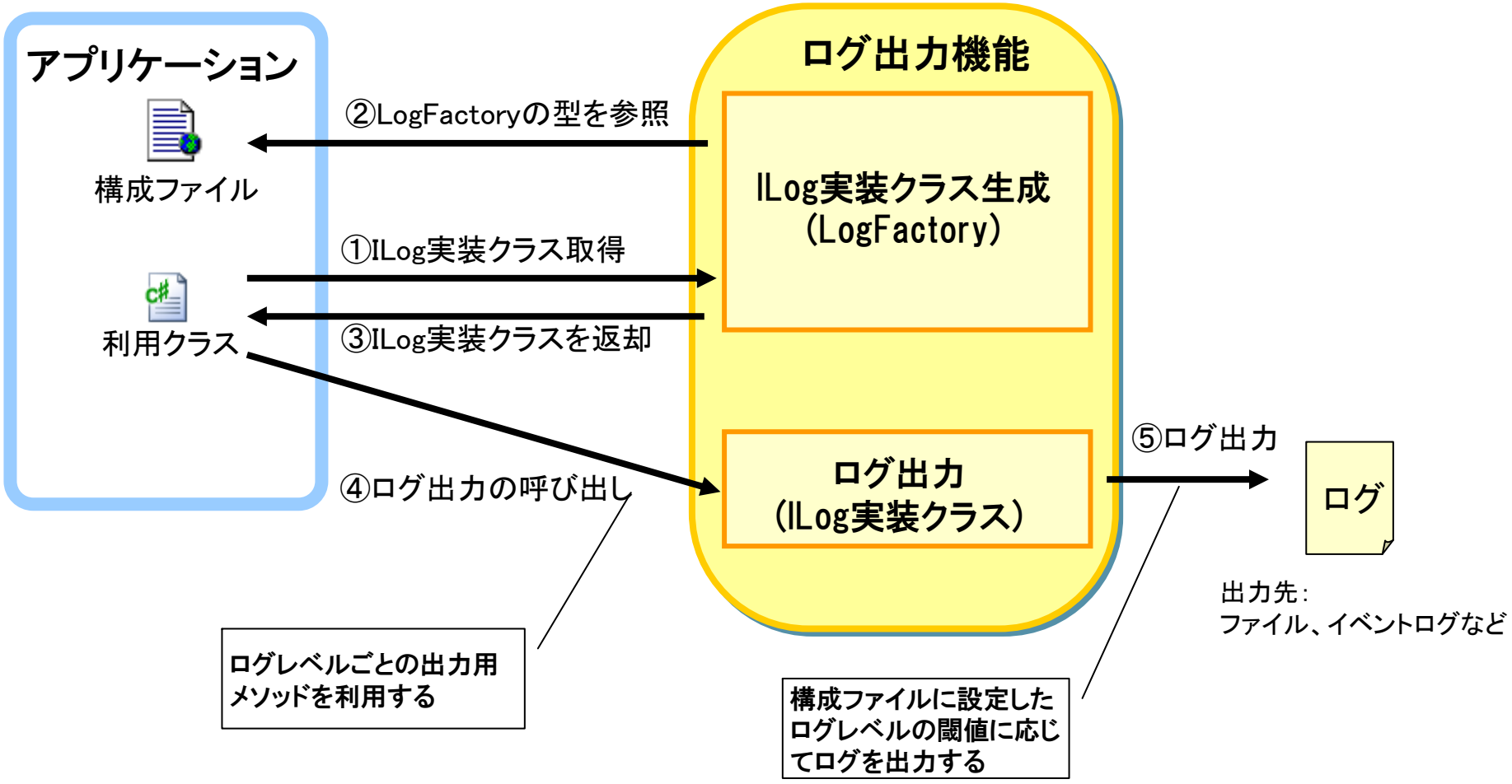
ILog	統一的な方法でログを出力するためのインターフェイス
LogFactory	ILog実装クラスを生成するための静的なメソッドを持つクラス

  - 標準実装として、System.Diagnostics.TraceSourceを利用してログを出力する以下のインターフェイス/クラスを提供

ITraceSourceLog	ILogを継承し、TraceSource用の抽象メソッドを追加したインターフェイス
TraceSourceLogger	TraceSourceを利用してログを出力するためのITraceSourceLog実装クラス
TraceSourceLogFactory	TraceSourceLoggerを生成するためのLogFactory継承クラス

CM-03 ログ出力機能 – 動作イメージ(1/2)

ログ出力機能を利用したログ出力の流れ



CM-03 ログ出力機能 – 動作イメージ(2/2)

ログレベルごとの出力用メソッド一覧

項番	種類	ログレベル	ログ出力用メソッド	出力ログレベル確認用プロパティ
1	致命的なエラー	FATAL	Fatal(object message) Fatal(object message, Exception ex)	IsFatalEnabled
2	エラー	ERROR	Error(object message) Error(object message, Exception ex)	IsErrorEnabled
3	警告	WARN	Warn(object message) Warn(object message, Exception ex)	IsWarnEnabled
4	情報	INFO	Info(object message) Info(object message, Exception ex)	IsInfoEnabled
5	デバッグ	DEBUG	Debug(object message) Debug(object message, Exception ex)	IsDebugEnabled
6	トレース	TRACE	Trace(object message) Trace(object message, Exception ex)	IsTraceEnabled

- ログ出力用メソッド
  引数“message”にはログ出力するメッセージを渡す。
  引数“ex”にはログ出力の原因となった例外を渡す。

- 出力ログレベル確認用プロパティ
  そのログレベルが現在有効かどうかをチェックし、有効ならtrue、無効ならfalseを返す。

CM-03 ログ出力機能 – 使用例(1/2)

ログ出力の実装例

ERRORレベルのメッセージを出力する場合の実装例

```
class Program
{
    // ロガーの取得
    private static ILog _logger = LogFactory.GetLogger(typeof(Program));  ――― ①
    static void Main(string[] args)
    {
        // 設定されているログレベルがERROR以上の場合はログを出力する
        if (_logger.IsErrorEnabled)  ――― ②
        {
            _logger.Error("出力するログメッセージ");  ――― ③
        }
    }
}
```

①LogFactoryクラスのGetLoggerメソッドを呼び出してILog実装クラスを取得
　・引数にはそのクラスの型を渡す
②オン/オフ確認用プロパティでERRORレベルが有効かどうかを確認
③Errorメソッドを呼び出してログメッセージを出力

CM-03 ログ出力機能 – 使用例(2/2)

構成ファイルの設定例

標準実装として提供しているILog実装クラス(TraceSourceLogger)を利用する場合、構成ファイルにTraceSourceの設定をする

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!--使用するLogFactoryを指定する-->
    <add key="LogFactoryTypeName"
         value="TERASOLUNA.Fw.Common.Logging.TraceSourceLog.TraceSourceLogFactory,TERASOLUNA.Fw.Common" />
  </appSettings>
  <system.diagnostics>
    <sources>
      <source name="Default" switchName="DefaultLogSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="DefaultLogFile"/>
          <remove name="Default"/>
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="DefaultLogSwitch" value="Information"/>
    </switches>
    <sharedListeners>
      <add name="DefaultLogFile"
       type="System.Diagnostics.TextWriterTraceListener"
       initializeData="DefaultLog.txt">
      </add>
    </sharedListeners>
  </system.diagnostics>
</configuration>
```

ログレベル

出力先の種類

出力先のファイル名

ログの種類を定義
（複数定義可能）

ログの種類ごとの
ログレベルを定義 → Informationを指定した場合、FATAL、ERROR、WARN、INFOレベルのログが出力される

ログの種類ごとの
出力先を定義 → DefaultLog.txtというファイルにログが出力される

※TraceSourceの詳細な設定方法はMSDNを参照のこと

CM-03 ログ出力機能 – 拡張方法

- TraceSource以外のロギングパッケージ(Logging ABやlog4netなど)を利用したい場合

①個別ロギングパッケージに対応したILog実装クラスを作成する

- ILogインターフェイスに定義されているログ出力用メソッドと、出力ログレベル確認用プロパティを実装する

②LogFactory継承クラスを作成する

- ②で作成した実装クラスのインスタンスを生成するための抽象メソッドを実装する

③構成ファイルに設定を記述する

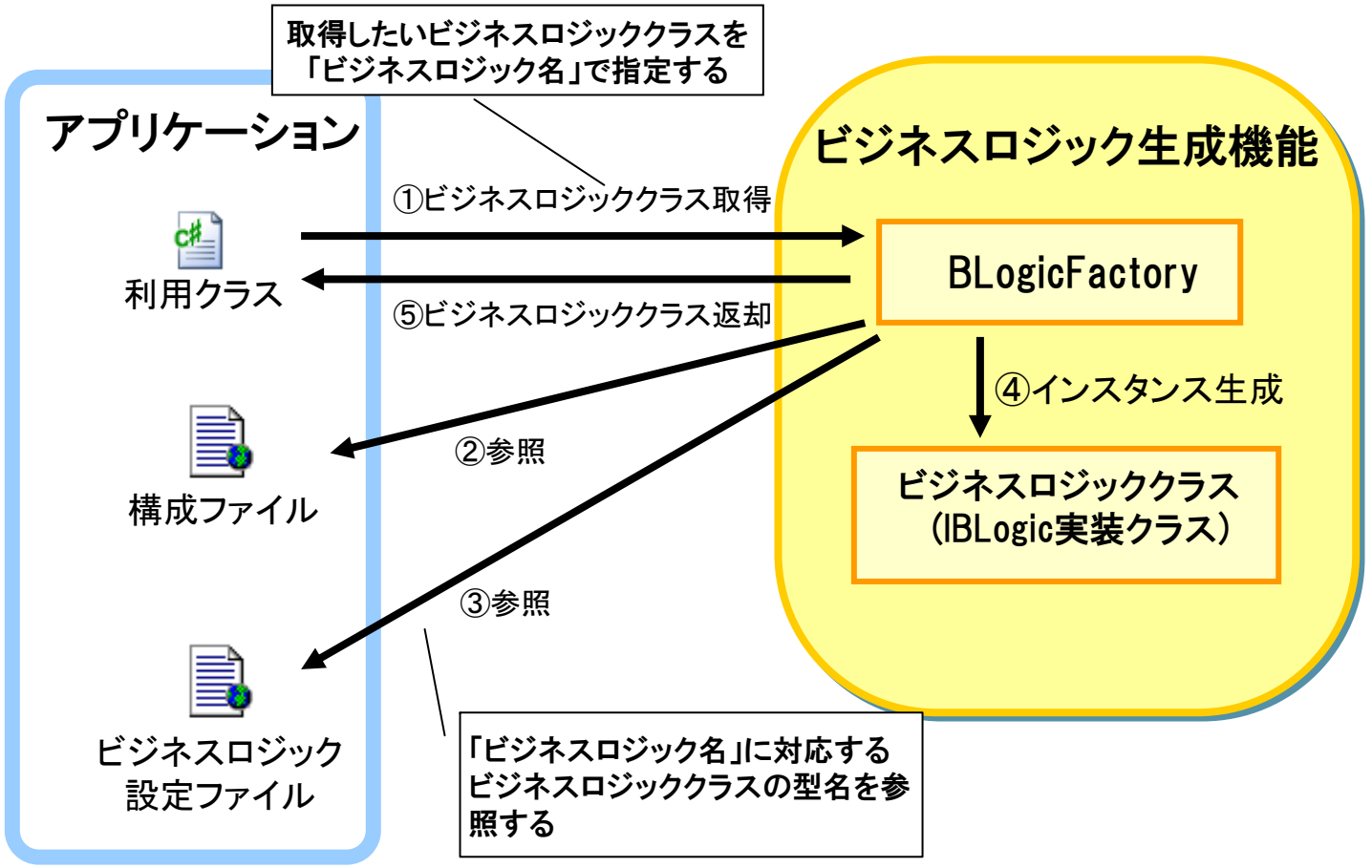
- ③で作成したLogFactory継承クラスを利用するための設定を構成ファイルに記述する

CM-04 ビジネスロジック生成機能 – 概要

- ビジネスロジッククラスのインスタンスを生成する機能を提供する
  - ビジネスロジック設定ファイルに従い、ビジネスロジック名に対応するビジネスロジッククラスのインスタンスを生成する機能を提供
  - ビジネスロジッククラスのインターフェイスとしてIBLogicを提供

CM-04 ビジネスロジック生成機能 – 動作イメージ

ビジネスロジッククラスのインスタンス生成の流れ



CM-04 ビジネスロジック生成機能 – 使用例(1/3)

①IBLogicインターフェイスを実装したビジネスロジッククラスを作成する

```
public interface IBLogic
{
    BLogicResult Execute(BLogicParam param);
}
```

Executeメソッドを実装する

引数

BLogicParam: ビジネスロジック入力情報を格納するクラス

戻り値

BLogicResult: ビジネスロジックの実行結果を格納するクラス

ビジネスロジッククラスの実装例1

```
public class BLogic01 : IBLogic
{
    public BLogic01() { }
    public BLogicResult Execute(BLogicParam param)
    {
      //処理
    }
}
```

ビジネスロジッククラスの実装例2(ジェネリック型)

```
public class BLogic02<T> : IBLogic
{
    public BLogic02() { }
    public BLogicResult Execute(BLogicParam param)
    {
       //処理
    }
}
```

CM-04 ビジネスロジック生成機能 – 使用例(2/3)

②構成ファイルにビジネスロジック設定ファイルへのパスを設定

```
<?xml version="1.0"?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <configSections>
    <section name="blogicConfiguration"
     type="TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection, TERASOLUNA.Fw.Common"/>
  </configSections>
  <blogicConfiguration>
    <files>
      <file path="config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>
</configuration>
```

ビジネスロジック設定ファイルへのパスを設定する

③ビジネスロジック設定ファイルにビジネスロジッククラスを設定

```
<?xml version="1.0" encoding="utf-8" ?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <blogic name="BLogic01" type="TERASOLUNA.Sample.BLogic01, TERASOLUNA.Sample"/>
  <blogic name="BLogic02" type="TERASOLUNA.Sample.BLogic02`1, TERASOLUNA.Sample"/>
</blogicConfiguration>
```

ビジネスロジック名を設定する

ビジネスロジッククラスを設定する

ビジネスロジッククラスがジェネリック型である場合、型パラメータの数を指定する

CM-04 ビジネスロジック生成機能 – 使用例(3/3)

④BLogicFactoryのCreateBLogciメソッドを呼び出し、
IBLogicインターフェイス実装クラスのインスタンスを取得する

1. ビジネスロジッククラスがジェネリック型でない場合

```
string blogicName = "BLogic01";
IBLogic blogic = BLogicFactory.CreateBLogic(blogicName);
```

ビジネスロジック設定ファイルに設定
した「ビジネスロジック名」を渡す

2. ビジネスロジッククラスがジェネリック型である場合

```
string blogicName = "BLogic02";
string genericTypeName = typeof(MyClass).AssemblyQualifiedName;
IBLogic blogic = BLogicFactory.CreateBLogic(blogicName, genericTypeName);
```

型パラメータの型名を渡す