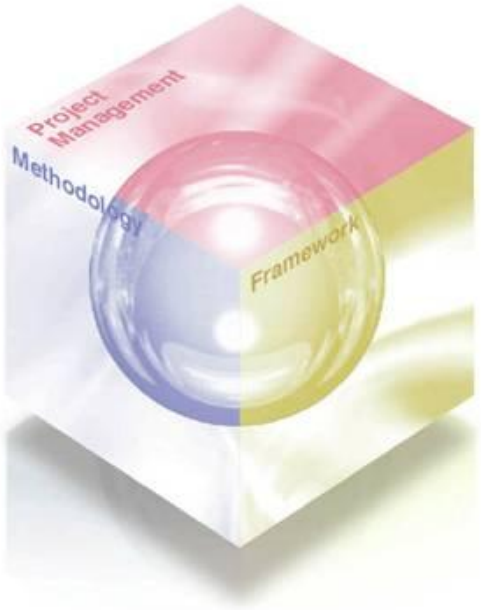
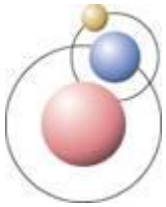
TERASOLUNA® Server/Client Framework for .NET 3.0.0.0
アーキテクチャ説明書



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 Framework for .NET(Rich版) アーキテクチャ説明書」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

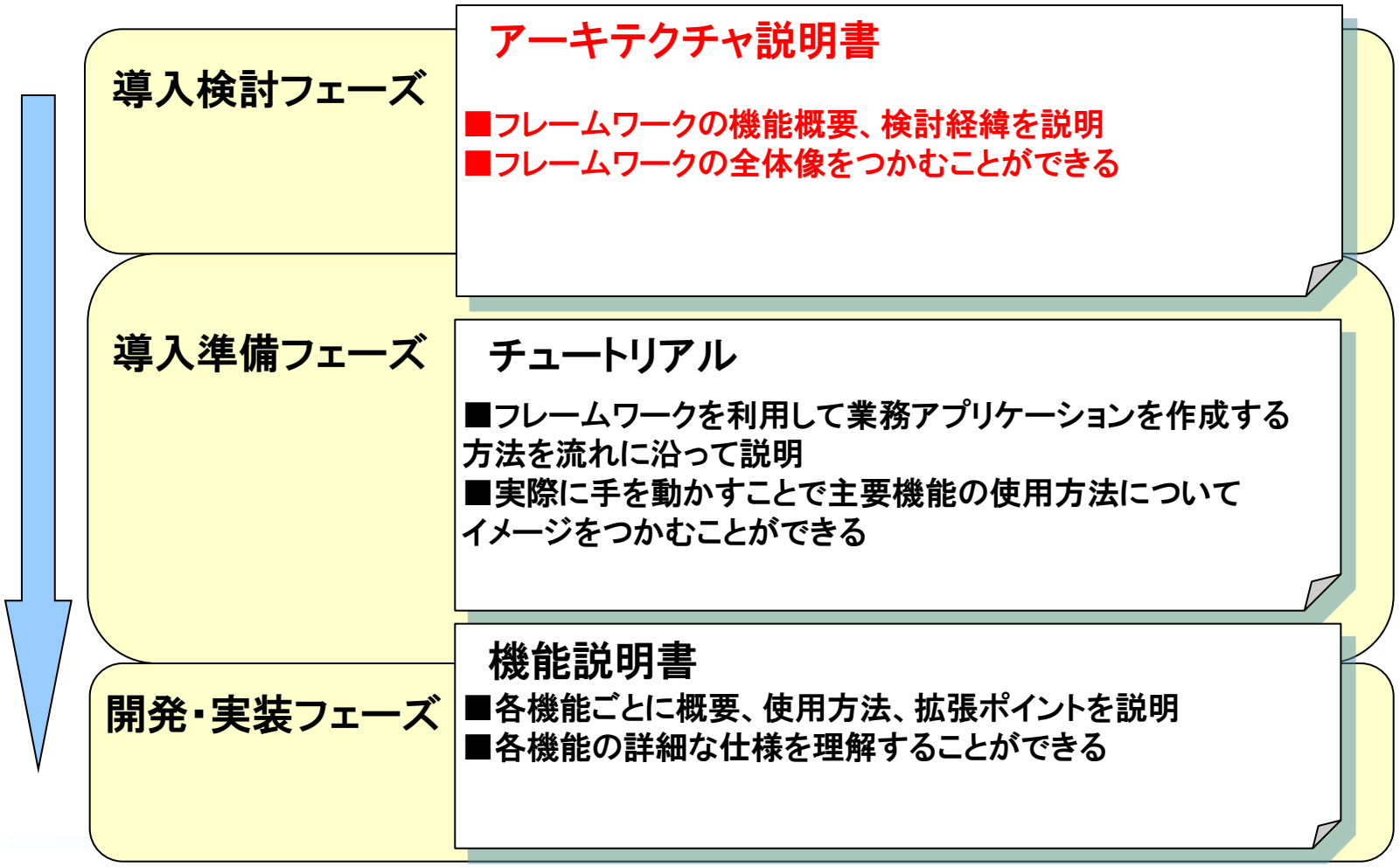
- はじめに
- 本資料の位置づけ
- 動作確認済み環境
- 全体アーキテクチャ概要
- クライアントフレームワーク機能概要
- サーバフレームワーク機能概要
- TERASOLUNAフレームワークの拡張ポイント
- TERASOLUNAフレームワークによるアプリケーションのソリューション構成

はじめに

- 本資料は、「TERASOLUNA Server/Client Framework for .NET 3.0.0.0」を解説した資料である

- 本フレームワークは、.NET Framework上で動作する、リッチクライアントアプリケーション構築のためのフレームワークである

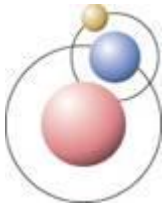
本資料の位置づけ



動作確認済み環境

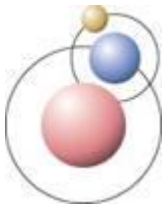
- 動作確認済み環境
  - クライアントフレームワーク
    - 対応OS
      - Windows XP Professional SP3
      - Window Vista Business SP2
      - Windows 7 Enterprise
    - .NET Framework
      - .NET Framework 3.5SP1
  - サーバフレームワーク
    - 対応OS、Web/APサーバ
      - WindowsServer2003 R2 Enterprise SP2 上のIIS6.0
      - WindowsServer2008 Enterprise SP1上のIIS7.0
      - WindowsServer2008 R2 Enterprise上のIIS7.5
    - .NET Framework
      - .NET Framework 3.5SP1

※上記は、動作検証を実施した環境であり、その他環境での利用を制限するものではありません。



全体アーキテクチャ概要

TERASOLUNA



全体アーキテクチャ概要

- ここでは、フレームワーク開発コンセプト、全体アーキテクチャを説明し、リッチクライアント版フレームワーク全体の動作を俯瞰的に理解することを目的とする

全体アーキテクチャ概要 – 目次

- フレームワークの開発コンセプト
- フレームワークの静的な構造
  - アーキテクチャ全体概要図
  - レイヤ構成図
  - テクノロジーマッピング
  - DLL構成
- フレームワークの振る舞い
  - フレームワークの動作概念図
  - シーケンス図

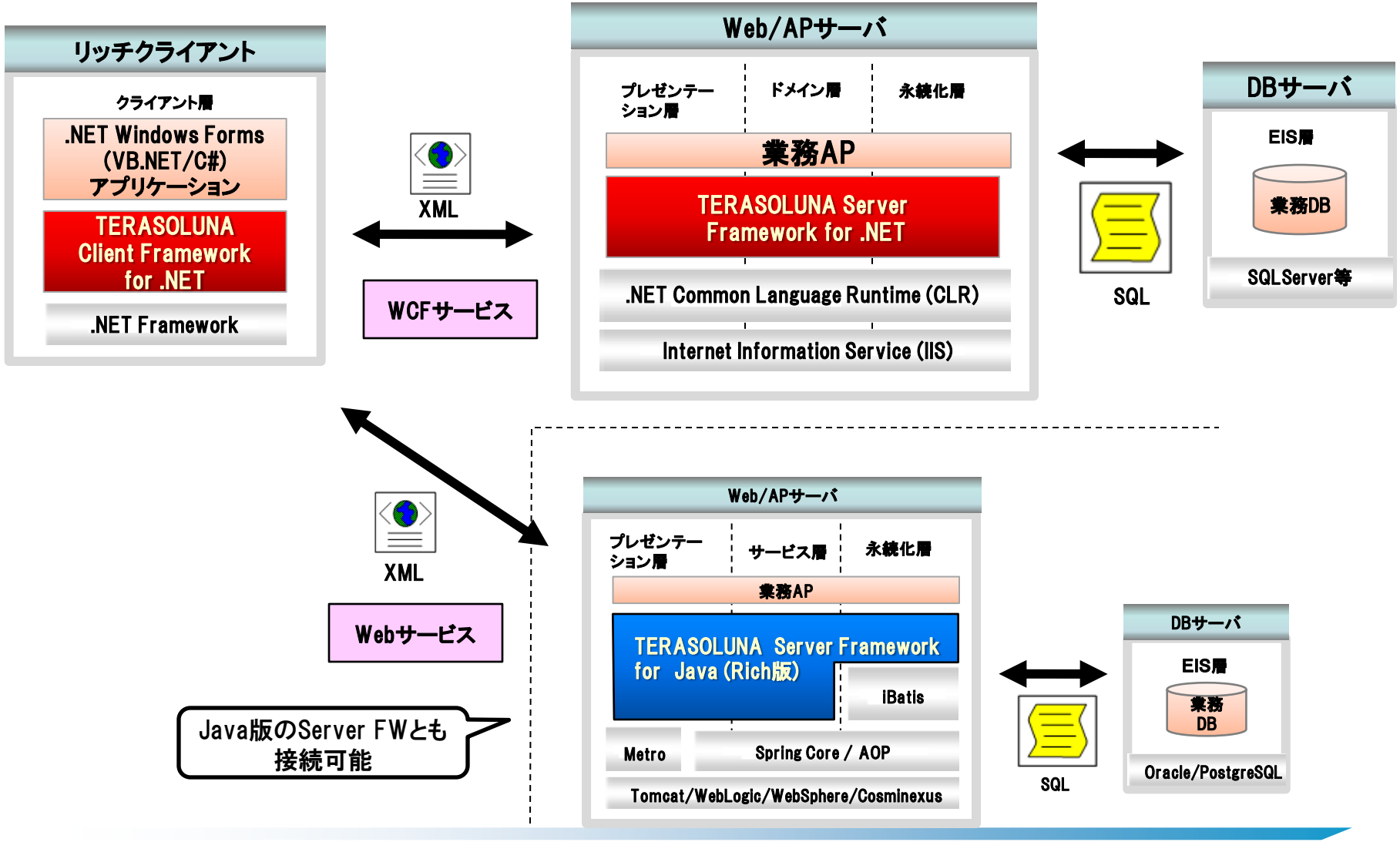
フレームワークの開発コンセプト

- **業務AP開発の生産性向上**
  - テンプレート、スニペット、インテリセンスの活用
    - 雛型やコード補完によりコーディング作業をよりスピーディに実施可能。
  - 双方向データバインドの活用
    - データソースウィンドウの機能を活用し、画面作成が容易に。
    - UIコントロール⇔データクラス間のコピー処理がノンコーディングに。
  - 設定ファイルの削減
    - 業務開発者向けには、XML形式などテキストベースの設定ファイルを排除し、Visual StudioのGUIベースのエディタや、カスタム属性を活用。
    - Visual Studioのデザイナ機能を拡張し、プロパティの入力支援機能も提供。
    - CoCを活用し記述量そのものを削減。
      - 記述量が肥大しがちなデータクラス間のマッピング処理は、プロパティ名を一致させることで自動コピーが可能。
- **長く使い続けられるフレームワーク**
  - 標準的な.NETの開発スタイルの維持
    - .NET標準の技術やEnterprise Libraryといったテクノロジを利用し、Microsoftの方向性とかい離しないテクノロジを選択。
    - 今後、TERASOLUNAフレームワークがバージョンアップし内部アーキテクチャや採用するテクノロジが変わったとしても、標準的な.NETの開発スタイルを大きく変えないようにすることで、技術者の確保や学習を容易に。

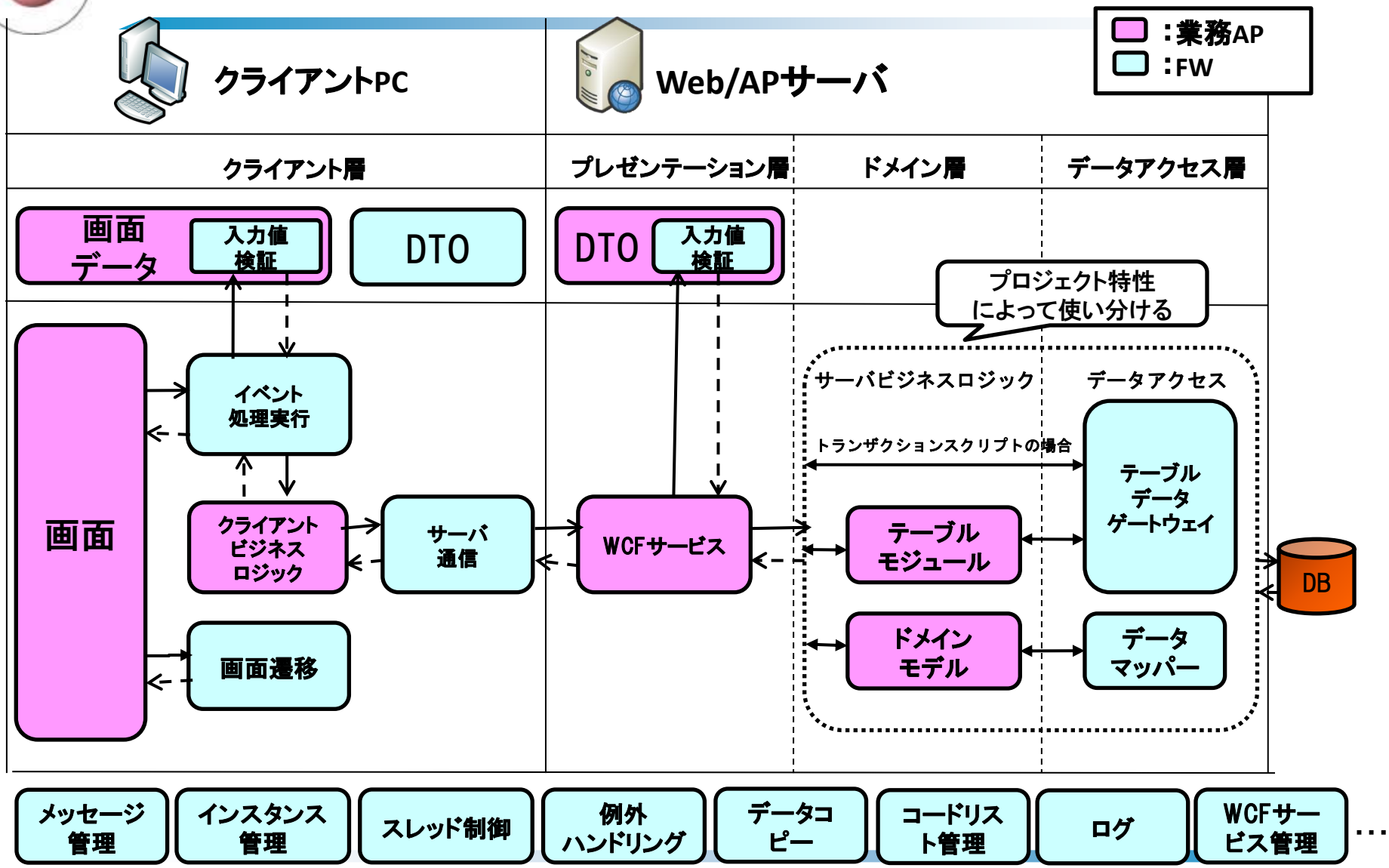
フレームワークの開発コンセプト

- 柔軟性の高いフレームワーク
  - **WCFサービスによる標準的な通信技術を採用**
    - WCFを採用し、実装スタイルを変えることなく、WCFサービスをはじめとしたさまざまな通信プロトコルを設定可能。
    - 言語やプラットフォームに依存せずシステム接続が可能。
      - TERASOLUNA Server Framework for Java (Rich)との接続も可能。
  - **POCOベースの業務AP開発**
    - 「画面データ」やDTO、ビジネスロジックなどPOCO(Plain Old CLR Object)ベースで実装可能。再利用性や共通化の促進につなげることが可能。
  - **機能拡張の容易性**
    - FWを構成する各コンポーネントにはインタフェースが定義されているので、インスタンス管理機能(DI/AOP)を使って、容易に差し替え可能。
    - プロジェクトの特性に合わせて、各機能を取捨選択して利用したり、.NETの新たなテクノロジーが出現しても容易に取り込むことが可能。

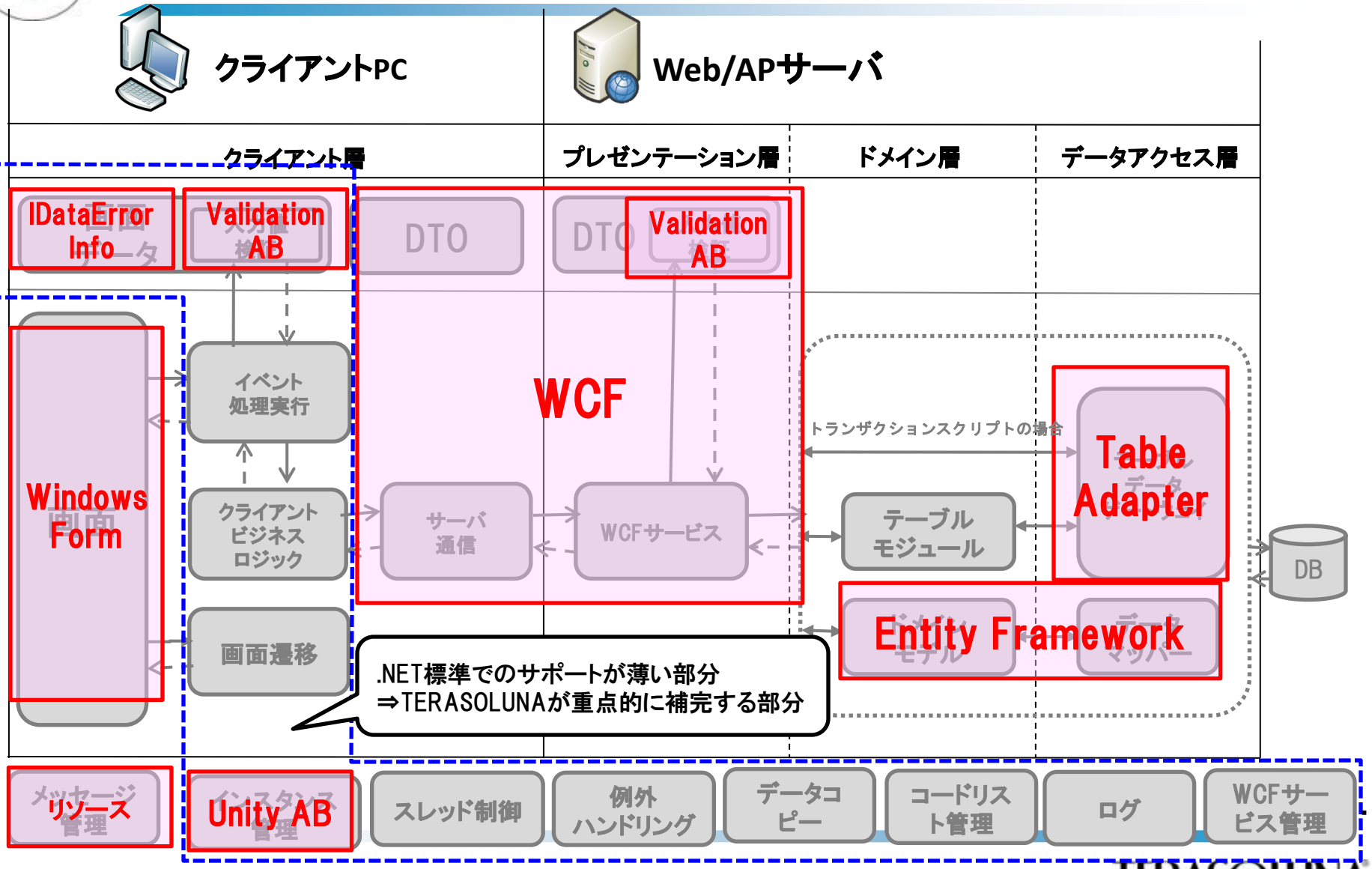
アーキテクチャ全体概要図

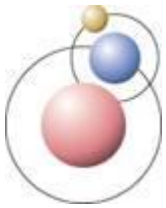


レイヤ構成図



テクノロジーマッピング





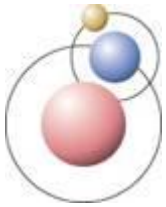
テクノロジーマッピング

- クライアントAPの制御まわりの機能を重点開発
  - TERASOLUNAフレームワークとしては、.NET標準のサポートが薄い部分を重点的にサポート
  - サーバ側はWCFやADO.NETなど、生産性の高い.NET標準機能の組み合わせをベースに、開発生産性向上につながる各種機能を提供する
- Unity Application Block(Unity AB)の採用
  - UnityABは、Microsoftのpatterns & practicesチームによって開発されたDI/AOPコンテナ
  - フレームワークの共通基盤にDI/AOPコンテナを据えて拡張性の高いフレームワークを目指す

テクノロジーマッピング

- プレゼンテーションテクノロジーはWindows Formsを採用
  - Visual Studio 2008(.NET 3.5SP1)の時点での開発生産性や業務APとしての機能の充足性を考慮しWindows Formsを選択
    - ただしWPF/Silverlight等の新しいテクノロジーへの対応を考慮し、機能追加しやすいアーキテクチャにしておく

- 通信はWCF(Windows Communication Foundation)を採用
  - 前バージョン(ver2.1)では、独自のXML電文仕様だったがSOAP、REST等各種標準に対応
    - TERASOLUNA Server Framework for Java (Rich)だけでなく、言語やプラットフォームに依存せずに他システムとの接続の機会を広げることが可能となった

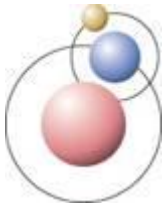


テクノロジーマッピング

- データアクセス機能は、ADO.NETをそのまま利用
  - プロジェクトの特色に合わせてテクノロジーを選択できるようTERASOLUNAフレームワークとしては特定のテクノロジーに依存しない

DLL構成 – 全体共通FW

	DLL名	依存ライブラリ	責務	デフォルト名前空間	サブ名前空間
1	Terasoluna	Unity AB	クライアント/サーバ 全体共通機能	Terasoluna	BizLogic Collections ComponentModel Configuration Data DataCopy ExceptionHandling Initialization Logging ObjectModel Reflection SelectableValues ServiceModel Text Threading TreeModel Unity
2	Terasoluna.Validation	Terasoluna.dll EnterpriseLibrary(Validation AB)	ValidationABに依存した クライアント/サーバ 全体共通機能 (入力値検証機能)	Terasoluna.Validation	–



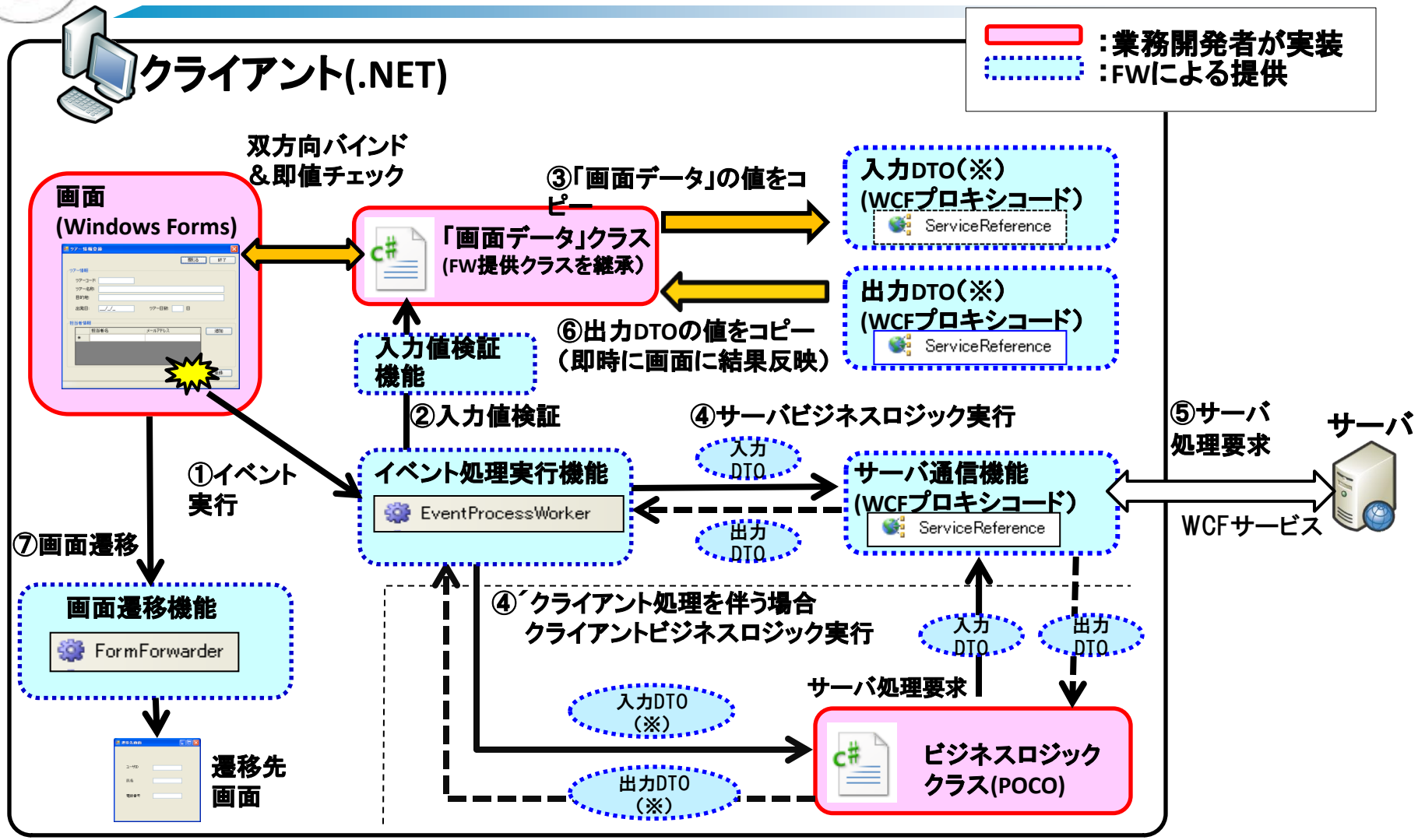
DLL構成 – クライアントFW

	DLL名	依存ライブラリ	責務	デフォルト名前空間	サブ名前空間
1	Terasoluna.Windows	Terasoluna.dll Unity AB	特定のプレゼンテーション技術に依存しないクライアント共通機能	Terasoluna.Windows	ViewModel
2	Terasoluna.Windows. ViewModel.Validation	Terasoluna.dll Terasoluna.Windows.dll EnterpriseLibrary(Validation AB) Unity AB	Validation ABに依存したクライアント共通機能(「画面データ」)	Terasoluna.Windows.ViewModel.Validation	Events
3	Terasoluna.Windows. Forms	Terasoluna.dll Terasoluna.Windows.dll Terasoluna.Windows.ViewModel.Validation.dll	Windows Formsに依存したクライアント機能	Terasoluna.Windows.Forms	BizLogic Controls Events ExceptionHandling FormForward Initializers MessageNotification Threading
4	Terasoluna.Design	Terasoluna.dll Terasoluna.Windows.dll Terasoluna.Windows.Forms.dll Unity AB	デザイナ機能拡張による入力支援機能	Terasoluna.<Any>.Design	BizLogic/Design DataCopy/Design Design/ComponentModel Design/Configuration Design/Controls Design/Unity Windows/Forms/Controls/Design Windows/Forms/Events/Design

DLL構成 – サーバFW (Rich)

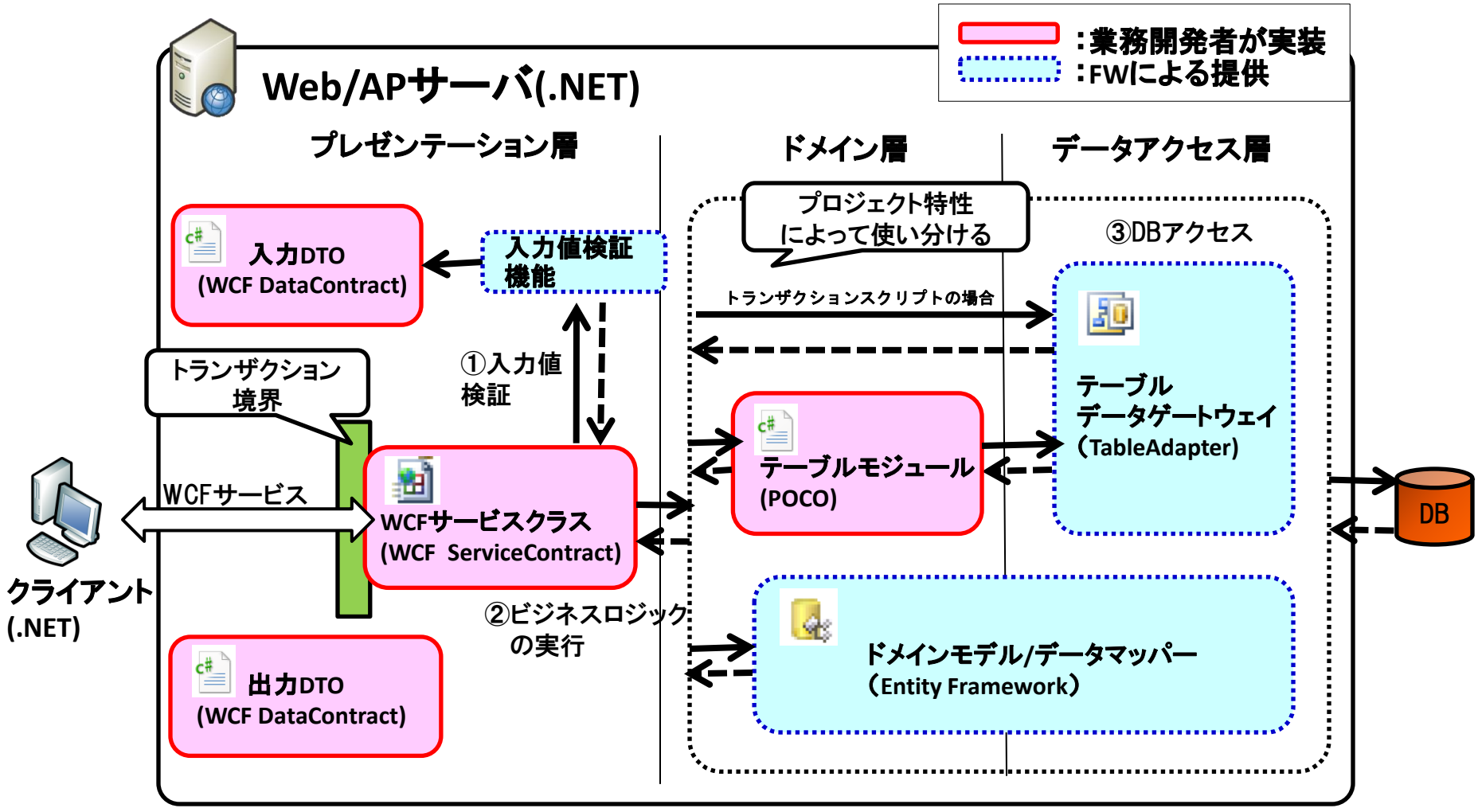
	DLL名	依存ライブラリ	責務	デフォルト名前空間	サブ名前空間
1	Terasoluna.Server	Terasoluna.dll	特定の技術に依存しないサーバ共通機能	Terasoluna.Server	Configuration
2	Terasoluna.Server.ServiceModel	Terasoluna.dll Terasoluna.Server.dll Unity AB	WCFに依存したサーバ機能	Terasoluna.Server.ServiceModel	ExceptionHandling

フレームワークの動作概念図(クライアント)

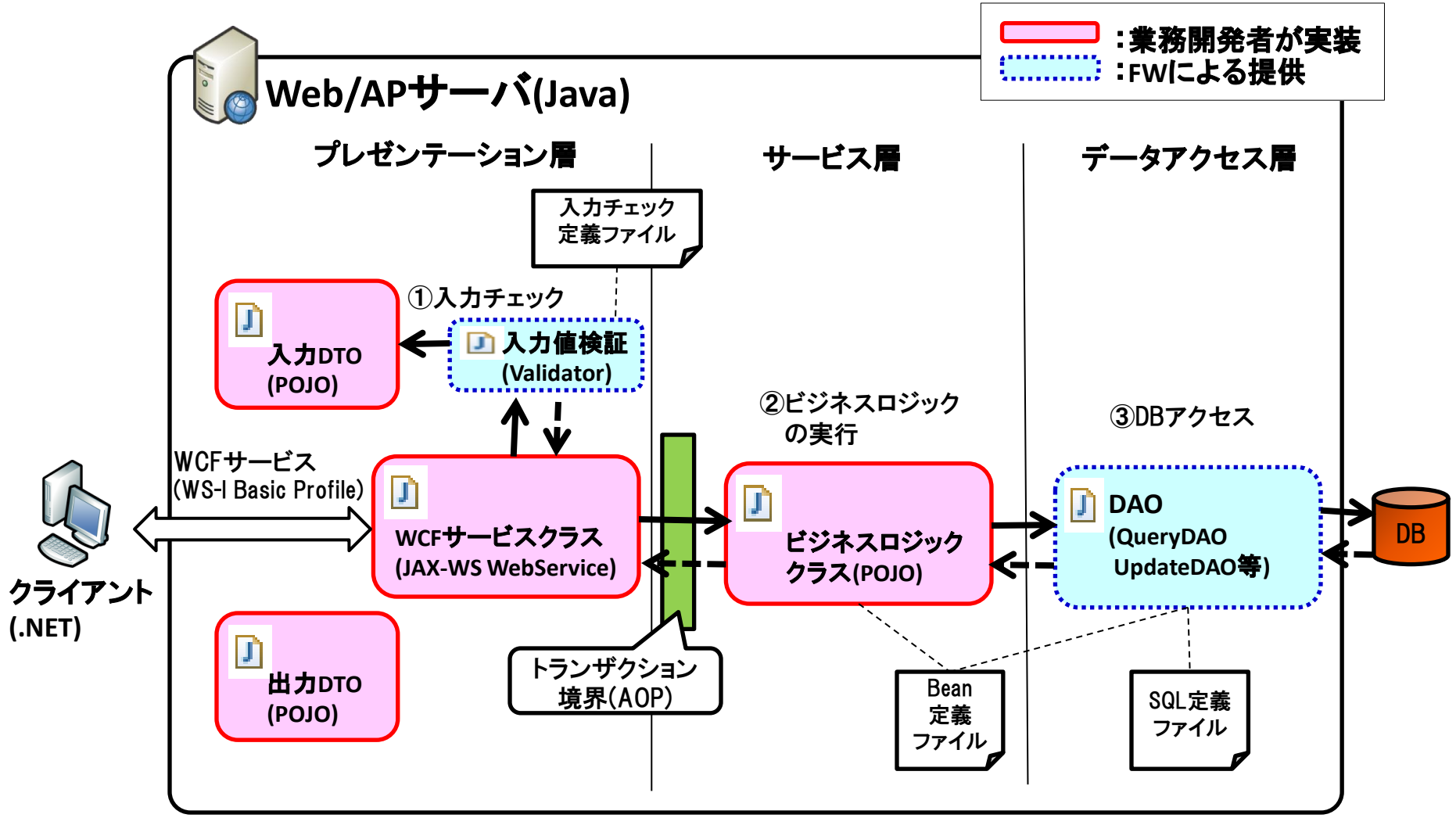


(※)④´の場合、WCFプロキシコードとは別に開発者が実装したビジネスロジック用の入出力DTOを使用するケースもある

フレームワークの動作概念図(サーバ)



【参考】TERASOLUNA Server Framework for Java (Rich) の動作概要図



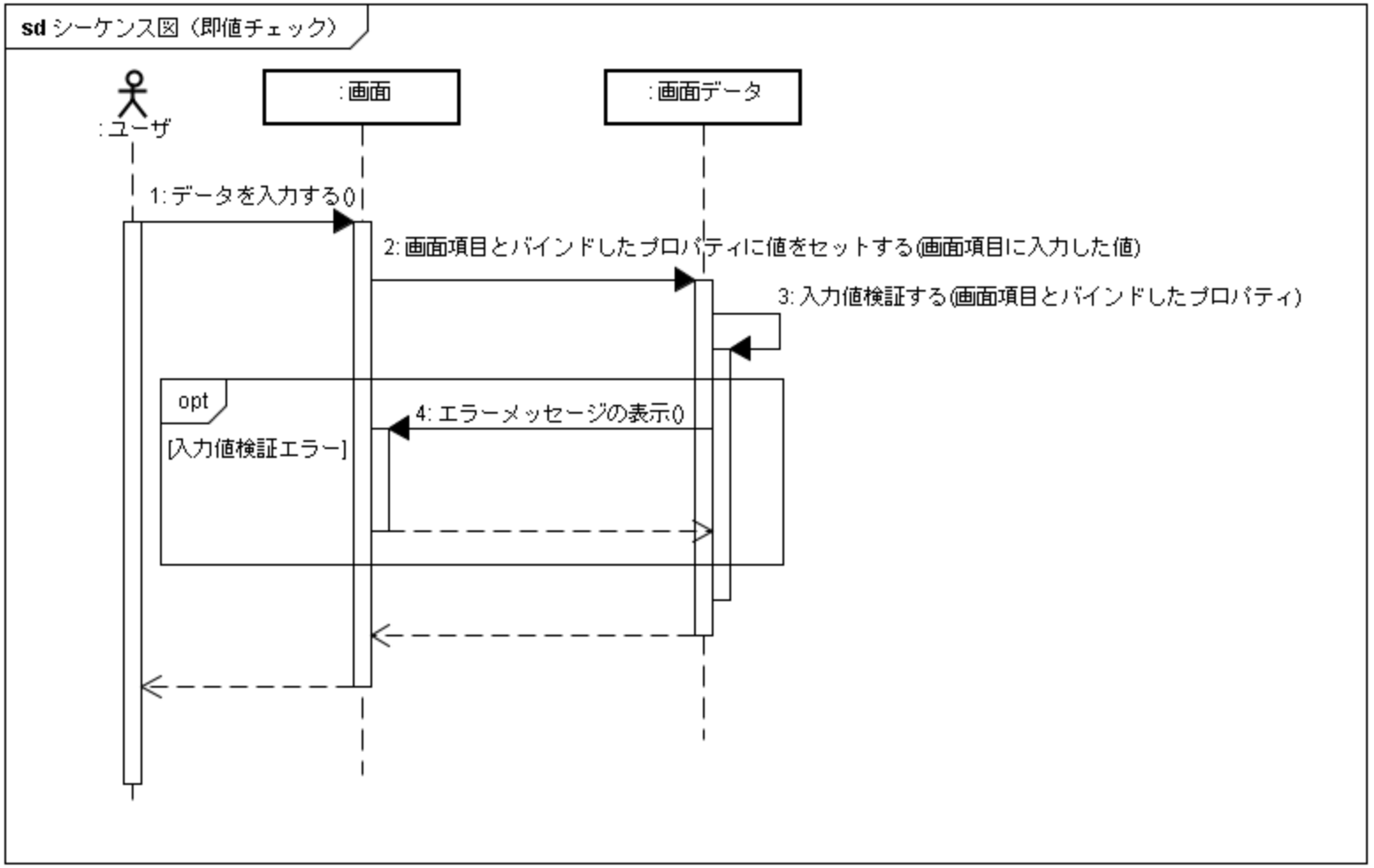
シーケンス図

- クライアント処理
  - 以下の4つの代表的な機能に分けて記述
    - 即値チェック
      - ロストフォーカス時に入力値検証
    - イベント処理
      - 入力値検証、「画面データ」⇔DTOのコピー、ビジネスロジック実行、サーバ通信処理といった定型的な一連の業務処理を実施
    - 画面遷移処理(画面切り替え)
      - 指定した遷移方式により画面(Form)を切り替えて遷移
      - 画面間のデータコピー等も実施
    - 画面遷移処理(パネル切り替え)
      - 画面に配置したパネル(Control)を切り替えて遷移
      - パネルが配置された大元の画面は切り替わらない

シーケンス図

- サーバ処理
  - ビジネスロジック実行やDBアクセス処理など1リクエスト内の一連の処理を記述

シーケンス図（即値チェック）

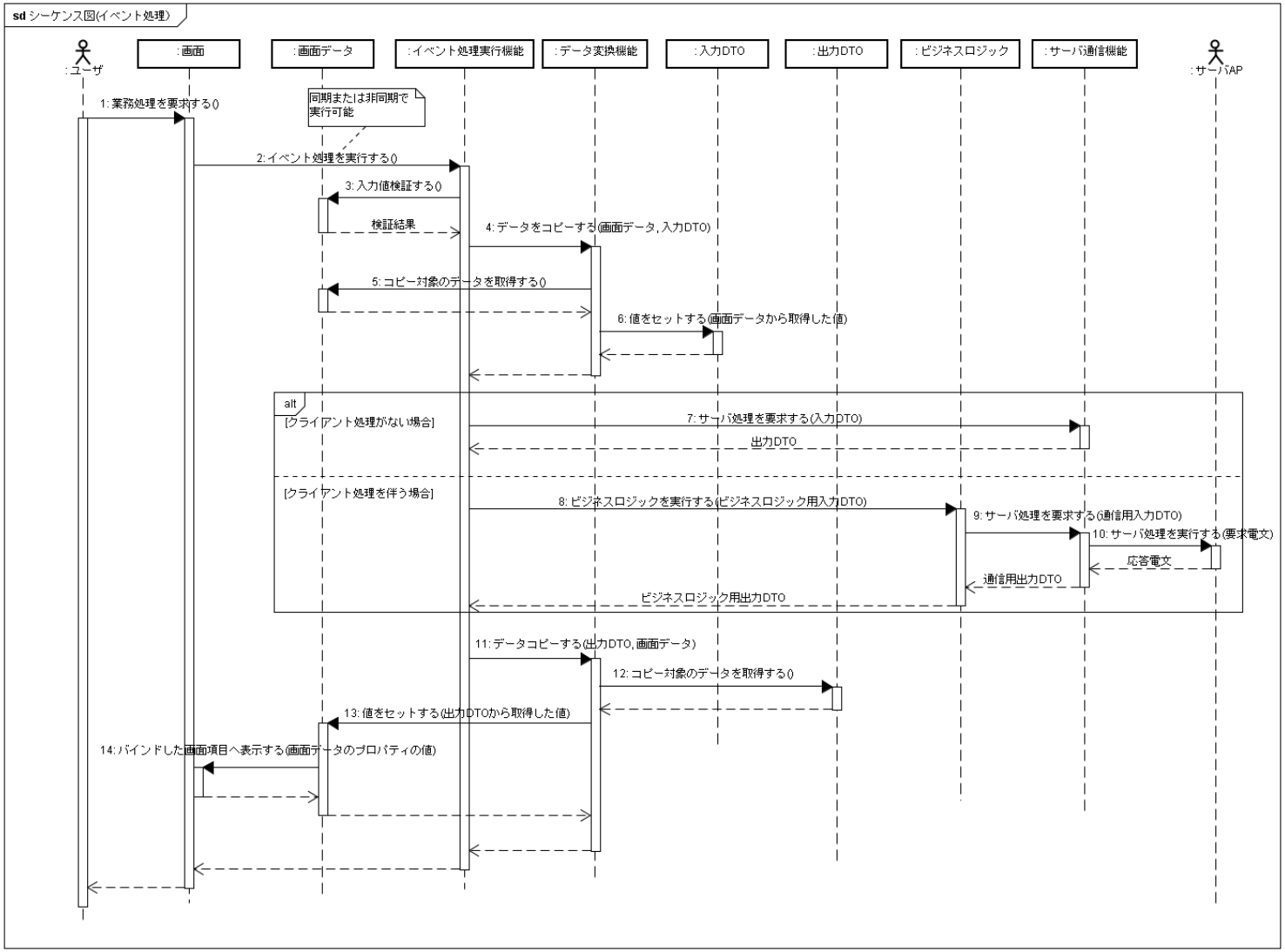


シーケンス図(即値チェック)解説

1. ユーザは、テキストボックスなどの画面項目にデータを入力する。
2. 双方向バインドにより、画面項目のロストフォーカス時に、「画面データ」のバインド設定したプロパティに即時に入力データがセットされる。
3. 「画面データ」は、セットされたプロパティの値を検証し、入力内容が正しいことを即時に確認する。

   即値チェックは実施しないようにすることも可能である。また、即値チェックでは単項目チェックのみ実施可能である。相関項目チェックなどのカスタム入力チェックは、イベント処理時に実施する。
4. 入力値検証エラーの場合、バインドした各画面項目に詳細なエラーメッセージを表示する。デフォルトでは、ErrorProviderを使って各画面項目へエラー表示する。なお、エラーメッセージが表示された状態でも、処理は続行し、次の画面項目に入力可能である。また、ユーザが正しいデータを再入力すると、再度入力値検証処理が実行されてエラー表示は消える。

シーケンス図(イベント処理)



シーケンス図(イベント処理)解説

1. ユーザは、画面上のボタン押下等により、業務処理を要求する。
2. 「イベント処理実行機能」は、同期または非同期でイベント処理を実行する。
   この時ユーザ誤動作によるイベント多重実行を防ぐ仕組みとして、イベント処理開始～完了まで、イベント発生元画面や画面上のコントロールを操作できないよう画面ロックを実施する。
3. 「イベント処理実行機能」は、「画面データ」を検証し、入力値が正しいことを確認する。
   この時、即値チェックでは実施しなかった相関項目チェックなどのカスタム入力チェックも検証する。

4～6.

「イベント処理実行機能」は、「データコピー機能」を利用し、「画面データ」の値を「入力DTO」にデータコピーする。デフォルトでは、「画面データ」と、クラス階層構造及び名称が一致するDTOのプロパティに値をデータコピーする。

7. クライアント処理がない場合、「イベント処理実行機能」は、「サーバ通信機能」を直接呼び出しサーバビジネスロジックを実行する。 クライアントサーバ通信は、通常SOAPによるWCFサービスを使用する。「入力DTO」は、XMLへシリアライズしてサーバへ送信する。サーバビジネスロジック実行後、処理結果を受信すると、XMLへデシリアライズし「出力DTO」を取得する。

8～10. クライアント処理を伴う場合、「イベント処理実行機能」は「ビジネスロジッククラス」を呼び出す。
   なおサーバ通信処理を実施するには、「ビジネスロジッククラス」より「サーバ通信機能」を呼び出す。

11～13.

「イベント処理実行機能」は、「データコピー機能」を利用し、「出力DTO」の値を「画面データ」にデータコピーする。 デフォルトでは、 DTOとクラス階層構造及び名称が一致する「画面データ」のプロパティに値をデータコピーする。

14. 双方向バインドにより、「画面データ」へコピーされた値が即時に画面に表示される。

シーケンス図(イベント処理)解説

(異常系などの代替フロー処理　シーケンス図は省略)

*a.　任意のクライアント処理でユーザによるキャンセルが発生した場合(イベントの非同期実行時のみ)

*a1.「イベント処理実行機能」は、非同期処理結果待ち状態から解放し、イベント処理を中断する。

*a2.「イベント処理実行機能」は、画面ロックの解除などの後処理を実施し キャンセルされた旨をダイアログ表示する。(処理終了)

*b.　任意のクライアント処理でシステムエラーが発生した場合

*b1.「イベント処理実行機能」は、イベント処理を中断し画面ロックの解除などの後処理を実施する。

*b2. エラーログを出力し、システムエラーが発生した旨をダイアログ表示する。(処理終了)

3a.　入力値検証エラーが発生した場合

3a1.「イベント処理実行機能」は、入力値検証エラーの旨をダイアログ表示する。

3a2.「イベント処理実行機能」は、イベント処理を中断し画面ロックの解除などの後処理を実施する。

3a3. 即値チェックの場合と同様に、「画面データ」とバインドした各画面項目に詳細なエラーメッセージを表示する。デフォルトでは、 ErrorProviderを使って各画面項目にエラーが表示される(処理終了)

7a. サーバエラー電文(SOAP Fault)により入力チェックエラーまたは業務エラーが返却された場合

7a1.「イベント処理実行機能」は、エラーメッセージをダイアログ表示する(処理終了)

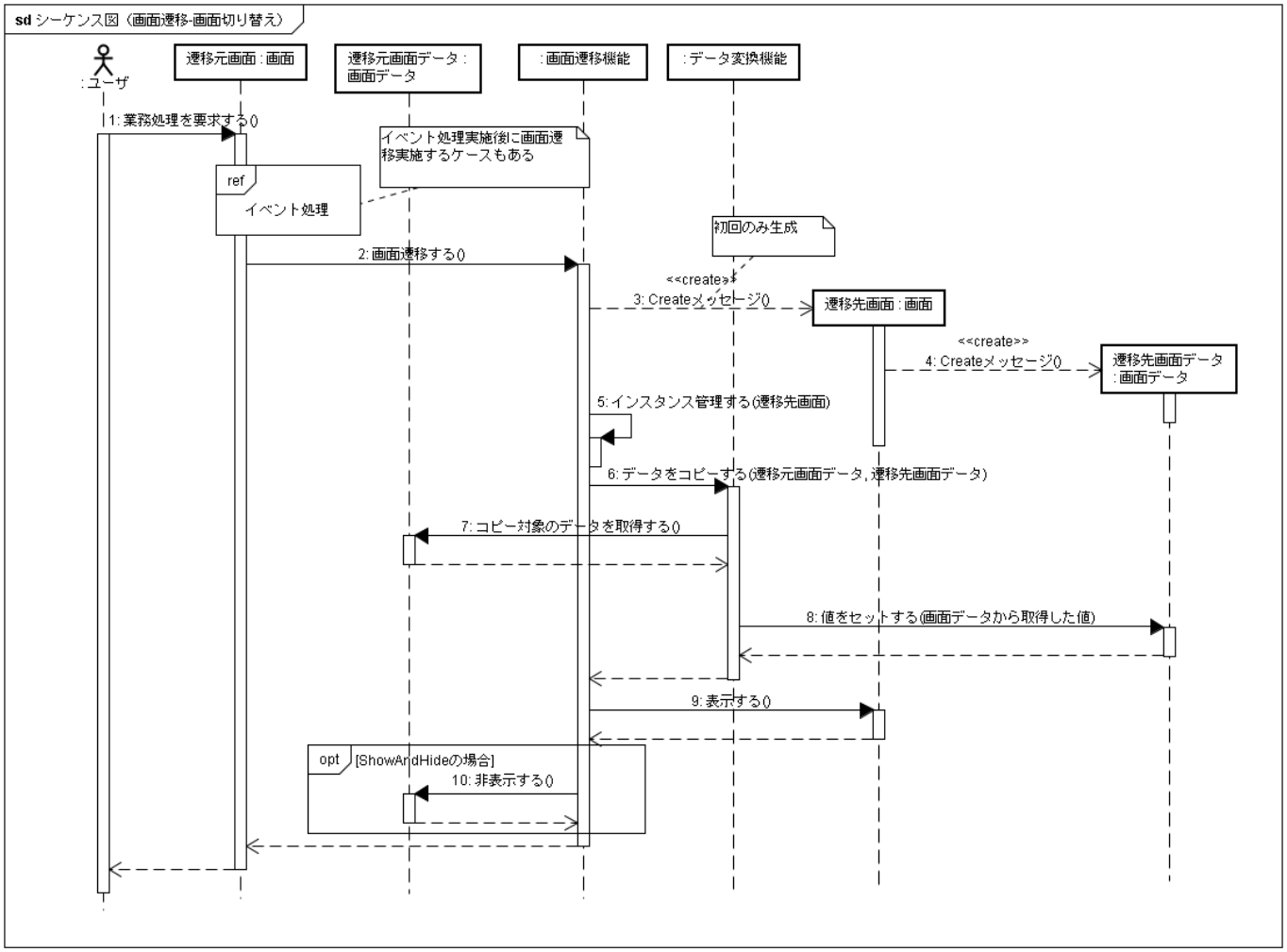
7b. サーバエラー電文(SOAP Fault)によりシステムエラーが返却された場合

*bと同様の処理

8a.　クライアントビジネスロジック実行時に業務エラーが発生した場合

7aと同様の処理

シーケンス図(画面遷移 - 画面切り替え)



シーケンス図(画面遷移 - 画面切り替え)解説

1. ユーザは、画面上のボタン押下により、業務処理を要求する。
2. 「画面遷移機能」は、画面遷移処理を実行する。
   画面遷移処理は、イベント処理の実行結果に応じて実施するケースもある。

3～5.

「画面遷移機能」は、遷移先画面のインスタンスを生成する。また、遷移先画面に対応する「画面データ」のインスタンスを生成する。生成した画面のインスタンスは管理し、インスタンスがすでに存在する場合にはそのインスタンスを取得する。

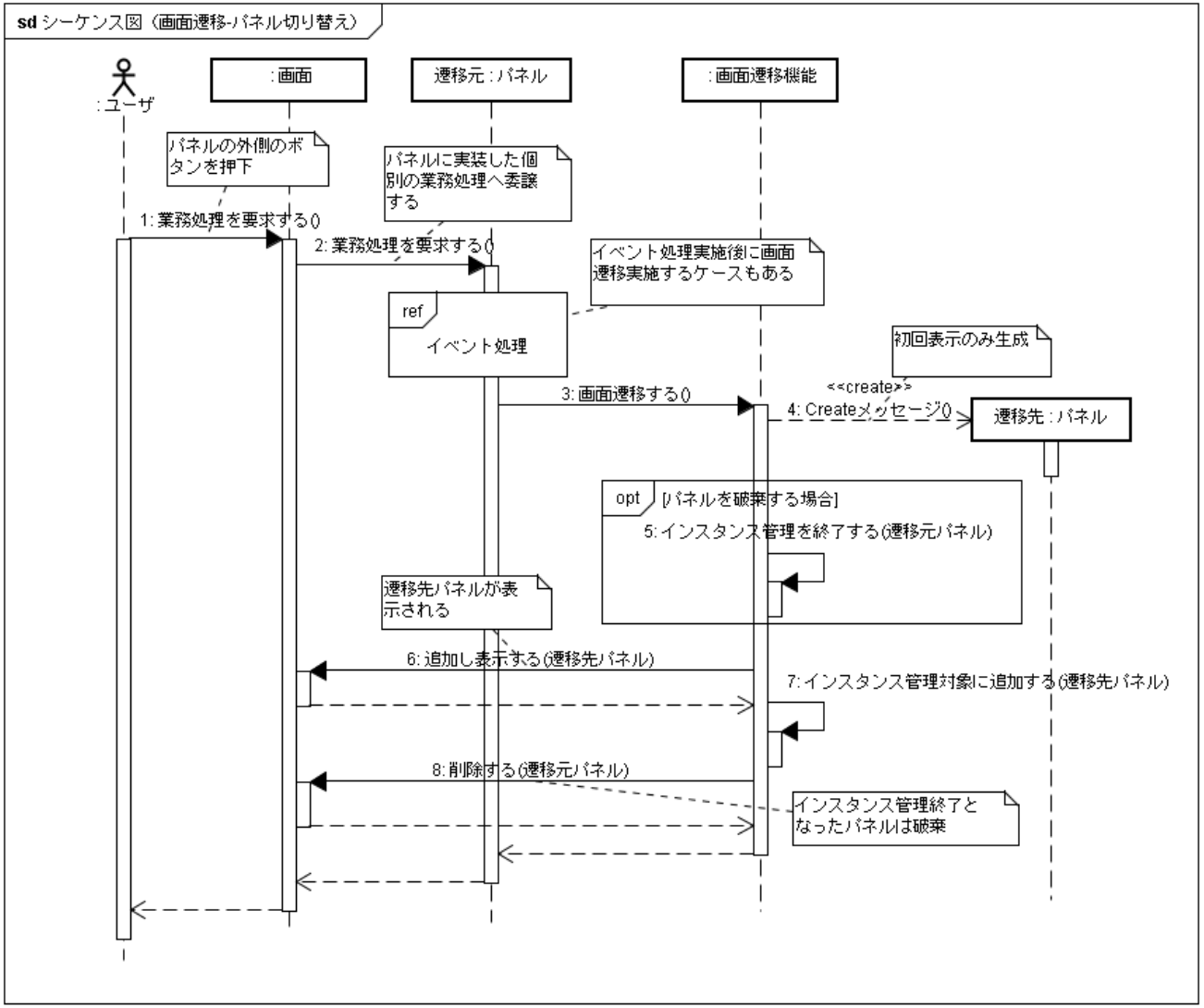
6～8.

「画面遷移機能」は、「データコピー機能」を利用し、遷移元「画面データ」の値を遷移先の「画面」にデータコピーする。

9～10.

開発時に指定した遷移方式に応じて遷移先画面をモードレスまたはモーダルで表示する。遷移方式によっては、遷移元画面を非表示にする。

シーケンス図(画面遷移 - パネル切り替え)



シーケンス図(画面遷移 - パネル切り替え)解説

1～2.

ユーザは、画面上のボタン押下により、業務処理を要求する。パネル切り替えの場合、パネルの外側の大元の画面に配置された共通のボタンを押下するケースが想定される。このようなケースでは、共通ボタンに対応する各パネル固有の業務処理を各パネルの実装に委譲する。

3. 「画面遷移機能」は、画面遷移処理を実行する。

画面遷移処理は、イベント処理の実行結果に応じて実施するケースもある。

4. 「画面遷移機能」は、遷移先パネルのインスタンスを生成する。「画面遷移機能」は、一度遷移したパネルをインスタンス管理するため、表示後に破棄されていないパネル(Hide()状態のパネル)のインスタンスについては、新たにインスタンスを生成しない。

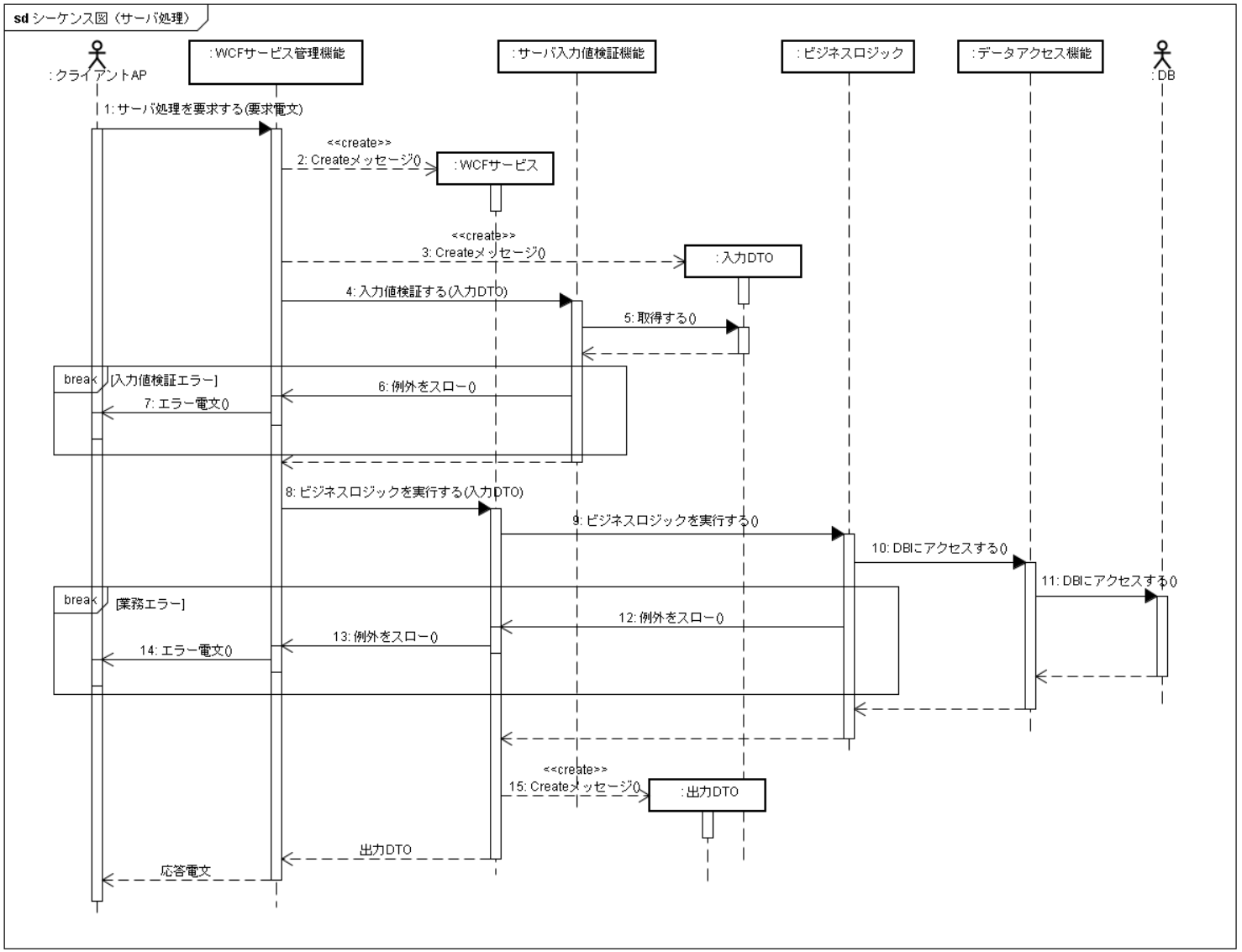
5. 遷移元パネルを破棄する場合、インスタンス管理の終了を指示する。

6, 7.

「画面遷移機能」は、大元の画面に遷移先パネルを追加し表示する。

8. 「画面遷移機能」は、大元の画面から遷移元パネルを削除することで、現在表示中のパネルを非表示にする。この時、5.でインスタンス管理を終了したパネルは破棄される。

シーケンス図（サーバ処理）



シーケンス図(サーバ処理)解説

1. クライアントAPは、サーバ処理を要求する。
2. 「WCFサービス管理機能」は、指定されたエンドポイントのURLをもとに対象の「WCFサービスクラス」を作成する。
3. 「WCFサービス管理機能」は、要求電文より「入力DTO」を作成する。

4, 5.
「サーバ入力値検証機能」は、「入力DTO」を検証し、入力値が正しいことを確認する。

6, 7.
入力値検証エラーの場合は例外をスローする。例外がスローされると例外情報をもとにエラー電文を生成し、クライアントAPへ返却する。(処理終了)

8, 9.
「WCFサービス管理機能」は、「WCFサービスクラス」を呼び出しビジネスロジックを実行する。
プロジェクト特性に応じて、ビジネスロジックをドメイン層に配置する。

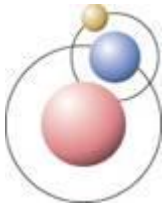
10, 11.
ビジネスロジックより、「データアクセス機能」を呼び出し、データベースへアクセスする。

12～14.
ビジネスロジック内で業務エラーが発生した場合は、例外をスローする。外がスローされると例外情報をもとにエラー電文を生成し、クライアントAPへ返却する。(処理終了)

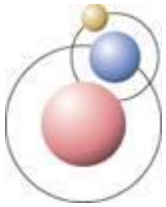
15. 「WCFサービス管理機能」は、ビジネスロジックの処理結果である「出力DTO」をもとに、応答電文を生成し、クライアントAPへ返却する。

*a. 任意サーバ処理でシステムエラーが発生した場合(シーケンス図省略)

*a1. スローされた例外情報をもとにエラー電文を生成し、クライアントAPへ返却する。(処理終了)



クライアントフレームワーク機能概要

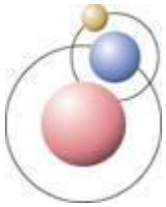


クライアントフレームワーク機能概要

- ここでは、クライアントフレームワークのアーキテクチャに関連する主要な機能の検討経緯や動作概念を説明し、各機能の理解を深めることを目的とする

クライアントフレームワーク機能概要 – 目次

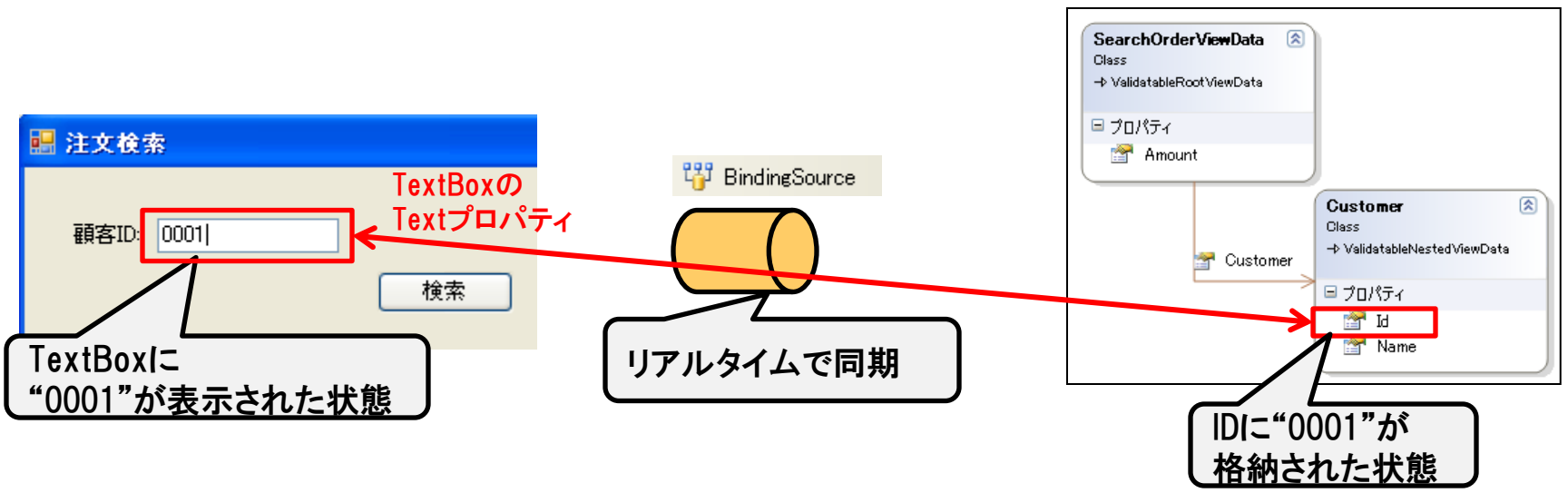
- 画面データ機能
- イベント処理実行機能
- 通信機能
- 画面遷移機能
- クライアントエラーハンドリング機能



画面データ機能

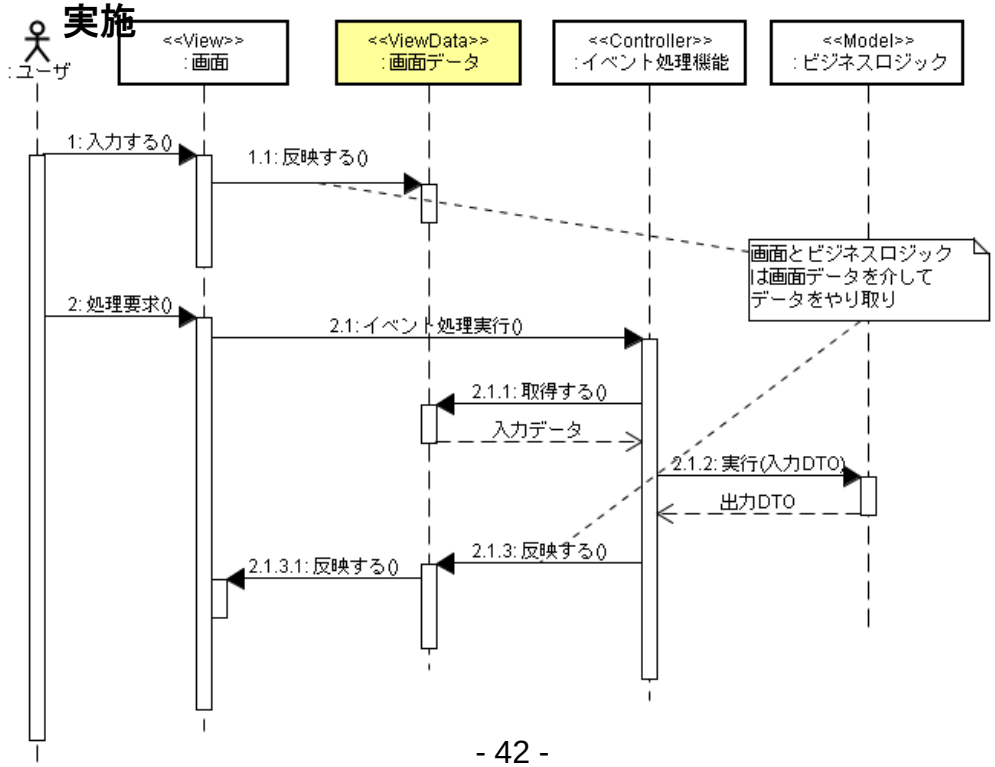
画面データ機能の責務①

- MVCの分離
  - 「画面データ」は、UIコントロールと双方向バインドするデータソース
    - UIコントロールのプロパティと「画面データ」のプロパティの値をリアルタイムで双方向に同期



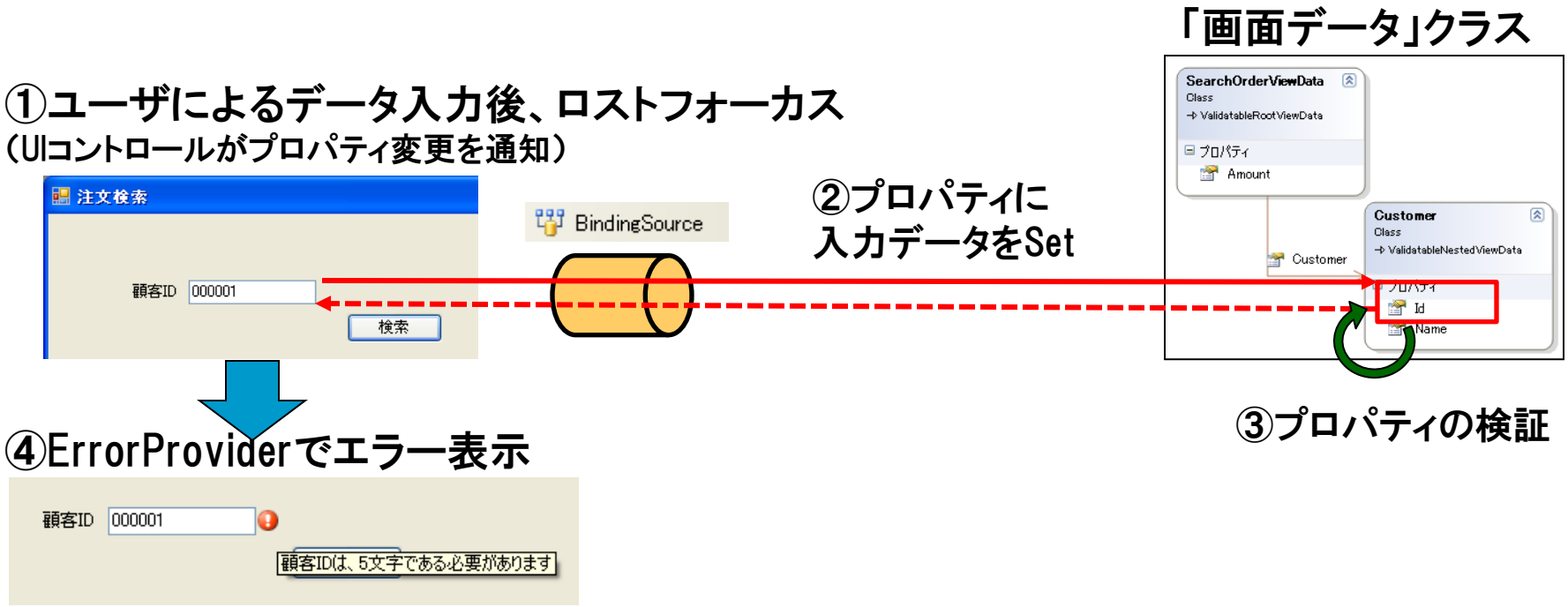
画面データ機能の責務①

- MVCの分離
  - ビジネスロジック(Model)と画面(View)の依存関係を断ち切る
    - 画面上のコントロールを直接扱う必要がない
      - 例えばTextBoxならTextプロパティに直接アクセスしない
    - ビジネスロジックで扱うデータ(DTO)さえ知っていればよい
      - 「画面データ」⇔DTOの変換は「イベント処理機能」を介して「データコピー機能」が実施



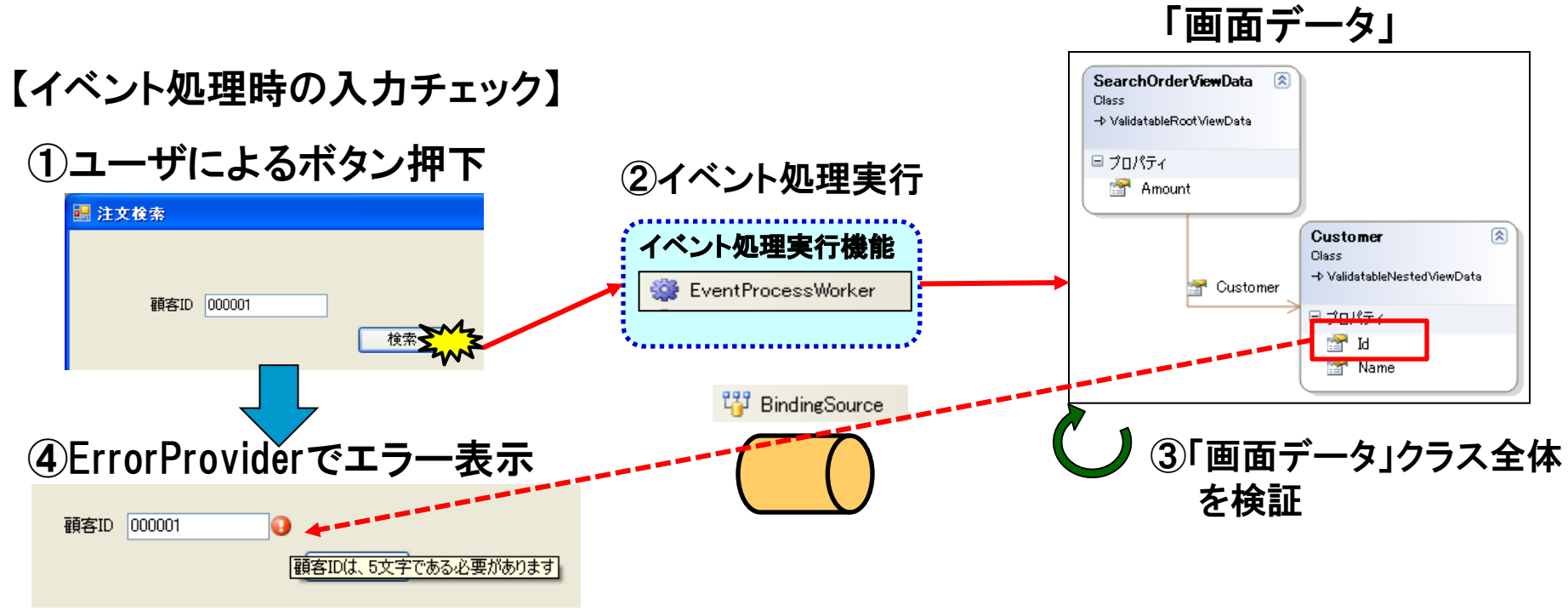
画面データ機能の責務②

- ロストフォーカス時の入力値検証(即値チェック)
  - ロストフォーカス時にバインドされている「画面データ」のプロパティをチェックすることで、ユーザにリアルタイムでエラーを通知



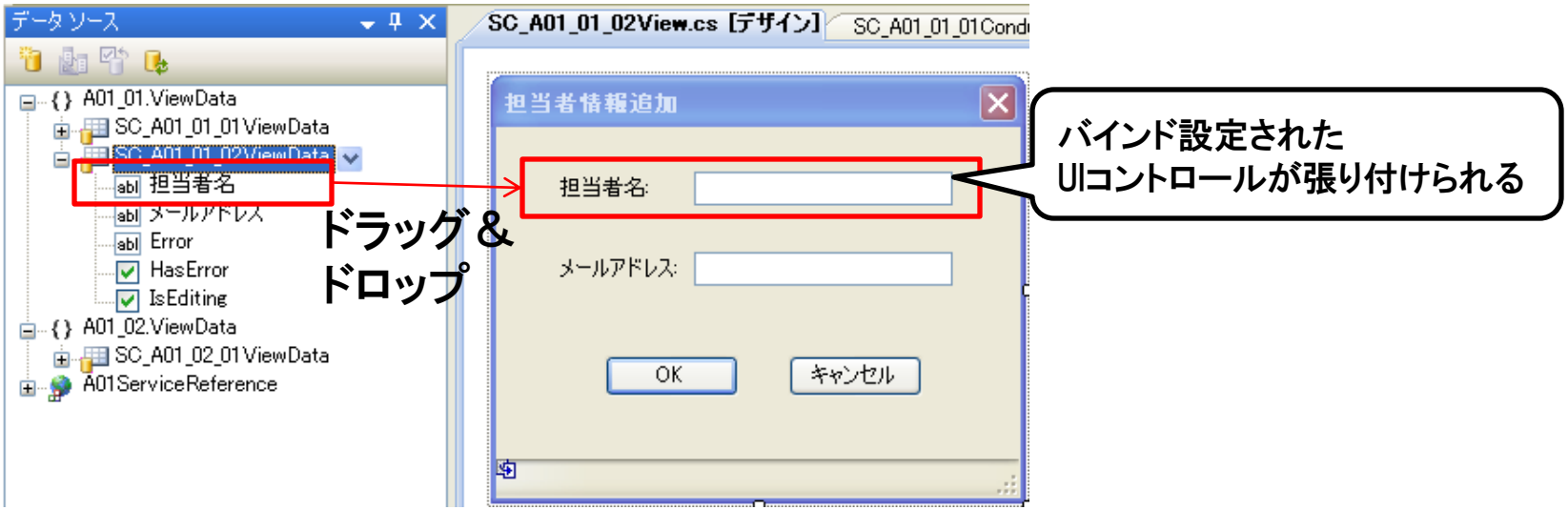
画面データ機能の責務③

- イベント処理実行時の入力値検証
  - イベント処理実行時に、「画面データ」全体を検証することにより、ユーザが入力した値を検証



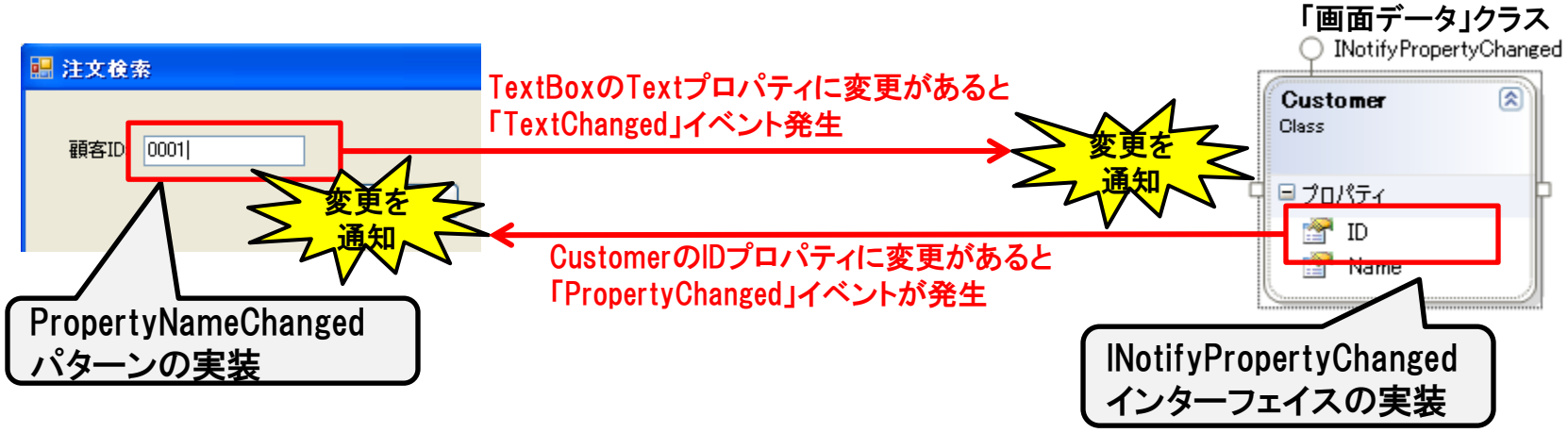
画面データ機能の検討経緯（双方向バインド）

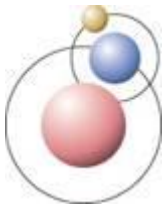
- 双方向バインドによる開発生産性の向上
  - Visual Studioのデザイナ機能によりノンコーディングで、UIコントロールと「画面データ」の同期が可能
    - 「データソースウィンドウ」からコントロールを張り付けることで簡単に画面の作成やバインド設定が可能



画面データ機能の検討経緯（双方向バインド）

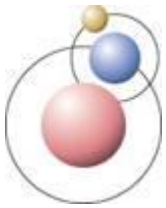
- 双方向バインドは便利であるが、UIコントロールと「画面データ」が相互にプロパティの変更を通知するように以下の実装が必要
  - UIコントロールは、「*PropertyName*Changedパターン」を適用
    - 「プロパティ名」+”Changed”という名前のイベントを定義
  - 「画面データ」は、INotifyPropertyChangedインタフェースを実装
    - プロパティが変更されるとイベントを通知するように実装





画面データ機能の検討経緯(双方向バインド)

- UIコントロールの*PropertyName*Changedパターンの実装
  - 標準コントロールでは実装済み(カスタムコントロールを利用しない限り、対応の必要なし)
    - TextBox.Textプロパティ ⇒ TextChangedイベント
  - 業務開発者への実装作業負担は少ないため、TERASOLUNAフレームワークでの対処はしない
    - アーキテクトや業務共通開発者が、カスタムコントロールを作成し、バインド可能なプロパティを追加する場合には、当パターンにより実装する必要がある
      - MSDNライブラリ「方法：PropertyNameChanged パターンを適用する」
        - http://msdn.microsoft.com/ja-jp/library/ms229615.aspx

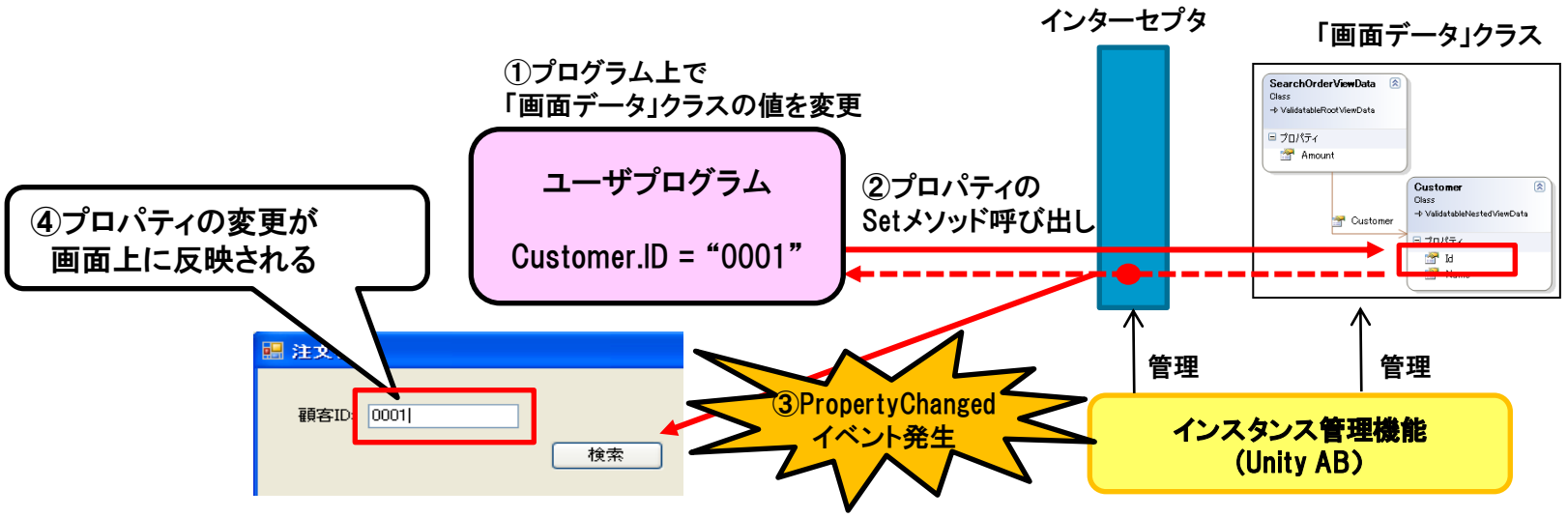


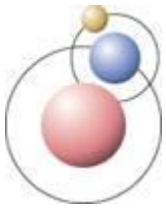
画面データ機能の検討経緯（双方向バインド）

- 「画面データ」クラスのINotifyPropertyChangedの実装
  - 各「画面データ」の各プロパティのSetメソッド実行時に、PropertyChangedイベントを呼び出す処理を実装する
  - 「画面データ」クラスは、業務開発者が作成するため、実装負担が大きく、フレームワークで対処する必要がある
- TERASOLUNAフレームワークにおける対処
  - フレームワークがINotifyPropertyChangedインタフェースを実装し、AOPで自動的にPropertyChangedイベントの呼び出し処理を実施
  - 業務開発者によるINotifyPropertyChangedの実装が不要
    - 双方向バインドの実装負担が大幅に軽減

PropertyChangedイベントの動作イメージ

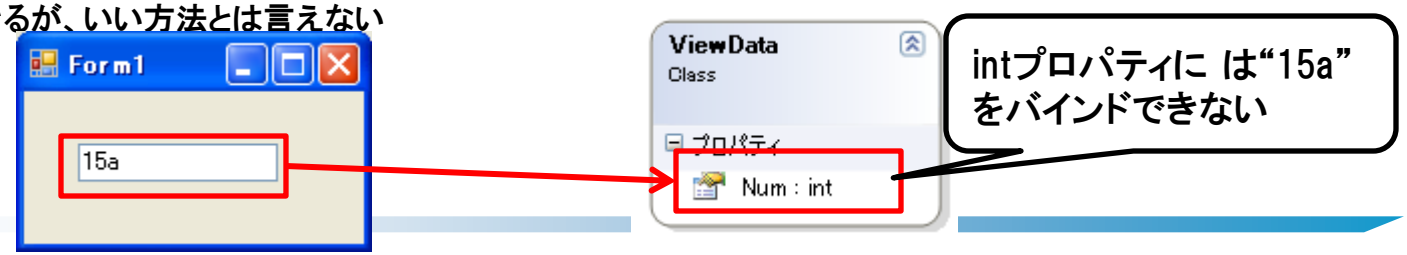
- 「インスタンス管理機能」のAOP機能を利用
  - Unity ABのインターセプタ機能
  - 各プロパティのSetメソッド実行直後にAOPで自動的にPropertyChangedイベントを呼び出す





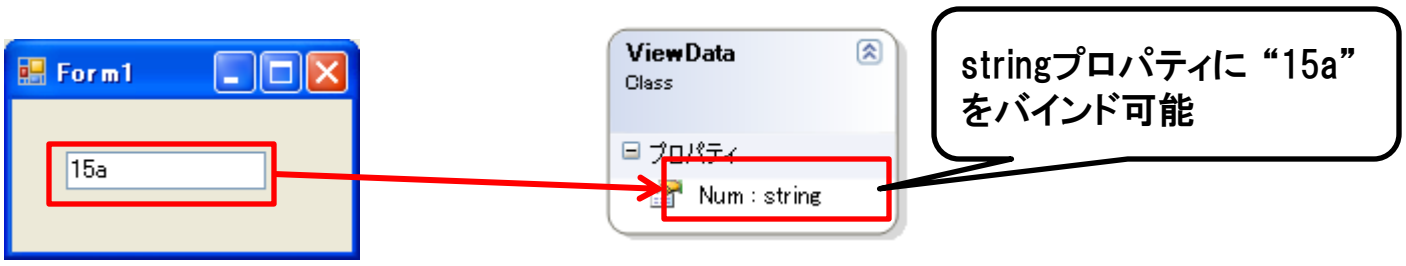
「画面データ」の検討経緯(入力値検証)

- なぜ、「画面データ」とDTOを区別するのか？
  - DTO(WCFのDataContract含む)もデータバインド可能
    - DTOを直接画面とバインドしたほうが、開発生産性は上がるはず？
- View層(画面データ)とModel層(DTO)の分離
  - Model層(ビジネスロジック)が、View層に従属する「画面データ」に、依存しないようにするべき
- ロストフォーカス時にデータバインドに失敗した場合の問題
  - ユーザが画面入力を誤ってデータソースとなる項目の型と違うデータを入れてしまうと、型の不一致によりデータバインドに失敗する
    - (例)TextBoxに”15a”を入力しても、「画面データ」のint型プロパティに値は反映されない
  - Windows Formsのデフォルトの設定では、無効な値を入力した場合にロストフォーカスできない。また「×」ボタンでも画面が閉じられない
    - (参考)FormのAutoValidateプロパティ(暗黙的な検証)のデフォルト値が「EnablePreventFocusChange」であるため(ロストフォーカス時に暗黙的な検証が実行され、無効な値の場合にロストフォーカスできない)。
    - (参考)Form_ClosingイベントでCancelEventArgs.Cancel = falseに上書きすれば、一応強制的に閉じることはできるが、いい方法とは言えない



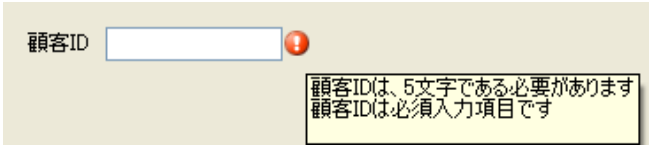
「画面データ」の検討経緯(入力値検証)

- データバインド失敗時の問題の対処
  - 「画面データ」のプロパティは文字列型(System.String)
    - 文字列型であれば、ユーザが誤って入力したとしても、型の不一致が起こらずデータバインドに失敗しないため、ロストフォーカスが可能
      - 最終的には誤った入力を防止する仕組みは必要
    - DTOはビジネスロジックで扱う忠実なデータ型が含まれる
      - 数値型(System.Int32)、日付型(System.DateTime)・・・
    - DTOとは異なる、文字列型ベースのデータバインド専用データクラス(＝「画面データ」クラス)が必要

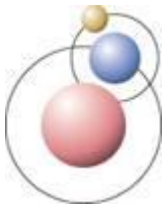


「画面データ」の検討経緯（入力値検証）

- データバインド失敗時の問題の対処
  - 即値チェック
    - 一時的に誤った入力を許容する代わりに、ロストフォーカス時にリアルタイムで入力値検証エラー情報をユーザに通知する
      - System.ComponentModel.IDataErrorInfoインタフェースの実装
      - ErrorProviderにより画面項目ごとにエラーメッセージを表示
      - 項目単位で検証可能なチェック（単項目チェック）を実施



  - イベント処理実行時の入力チェック
    - ボタンクリック時など、ユーザがシステムに処理要求するタイミングで入力チェック（後述の「イベント処理実行機能」を参照）
      - 単項目だけでなく、インスタンス全体の検証（相関項目チェックなどのカスタム入力チェック）も実施
      - 入力値検証に失敗した場合は、イベント処理を中断
      - 即値チェックと同様にErrorProviderで表示

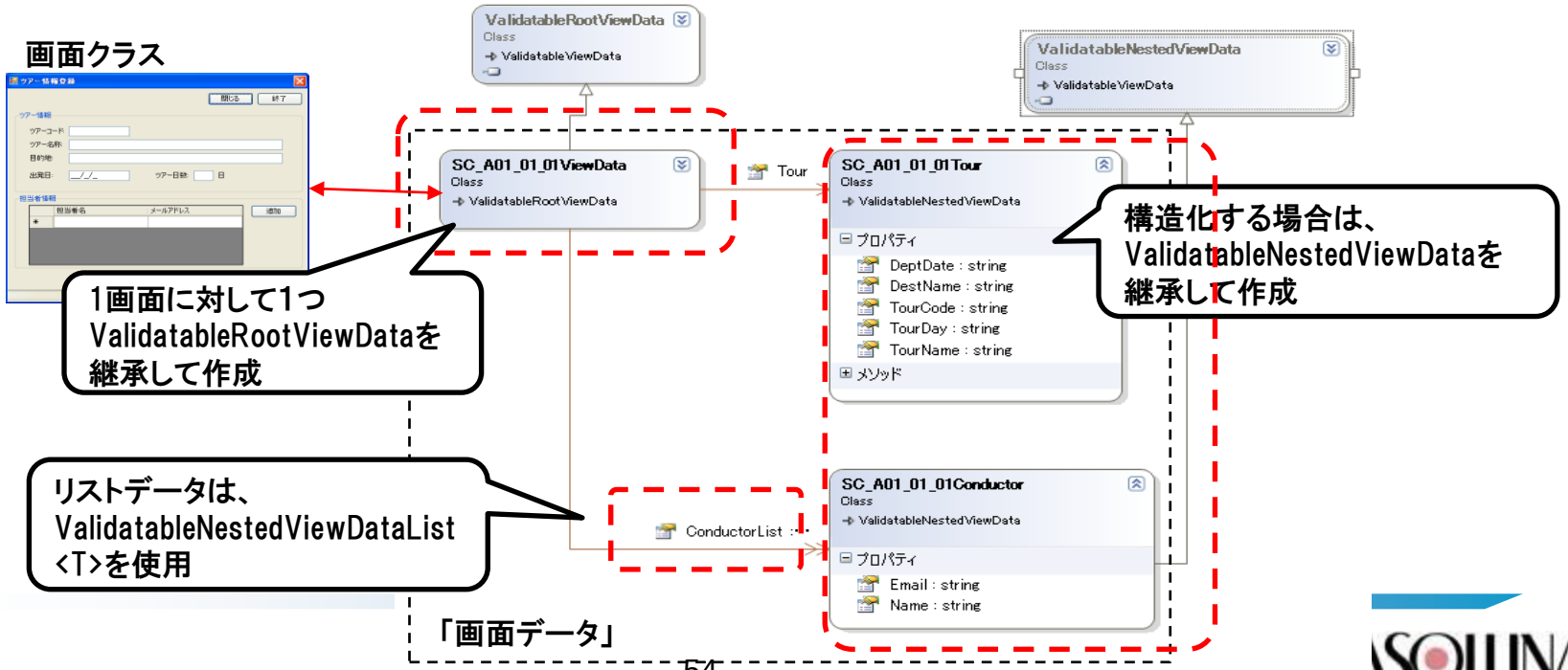


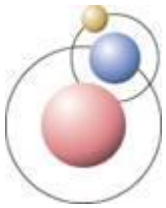
「画面データ」の検討経緯(入力値検証)

- 入力値検証の実装スタイルの統一
  - 「画面データ」の責務として入力値検証機能を持たせることで、入力チェック処理を一か所にまとめることができる
    - Validatingイベント等で検証処理を実装すると、画面クラスのコードが煩雑になり、同様の検証ロジックが各画面クラスに散在してしまう
  - ValidationABをベースとした入力値検証処理を「画面データ」の各プロパティ等に宣言的に記述
    - 単項目チェックはカスタム属性で付与
    - カスタム入力チェックはメソッドとして実装しカスタム属性を付与
  - IDataErrorInfoとErrorProviderによる入力チェックエラー表示
    - 「画面データ」に記述した入力値検証処理が、ロストフォーカス時やイベント処理実行時に実行
    - IDataErrroInfoのインデクサ(this[]プロパティ)で、入力値検証エラーのメッセージを返却することで、ErrorProviderが表示される

「画面データ」クラスの実装イメージ

- 「画面データ」の実装には、画面クラスに1対1に対応する「画面データ（ルート）」、内部に保有される「画面データ（ネスト）」、そのコレクションである「画面データリスト」、の3パターンがある。
  - 「画面データ（ルート）」は、ValidatableRootViewDataクラスを継承し実装
  - 「画面データ（ネスト）」はValidatableNestedViewDataを継承
  - 「画面データリスト」はValidatableNestedViewDataList<T>を利用





「画面データ」クラスの実装イメージ

- 単項目入力値検証処理は、カスタム属性ベースで実装
  - フレームワークがValidationABを拡張した機能を提供
  - プロパティに対して宣言的に記述でき見通しが良い
  - 属性なので、インテリセンスによるコード補完が可能
  - TERASOLUNAフレームワーク提供のスニペットにより、コードの自動生成が可能

```
[DefaultRuleset("RS01")]
public class SC_B01_01_01ViewData : ValidatableRootViewData
{
    [DisplayName("ユーザID")]
    [RequiredValidator(Tag="ユーザID", Ruleset="RS01")]
    public virtual string UserId { get; set; }

    [DisplayName("パスワード")]
    [RequiredValidator(Tag = "パスワード", Ruleset = "RS01")]
    public virtual string Password { get; set; }
}
```

プロパティに対して実施するバリデータをカスタム属性で定義

「画面データ」クラスの実装イメージ

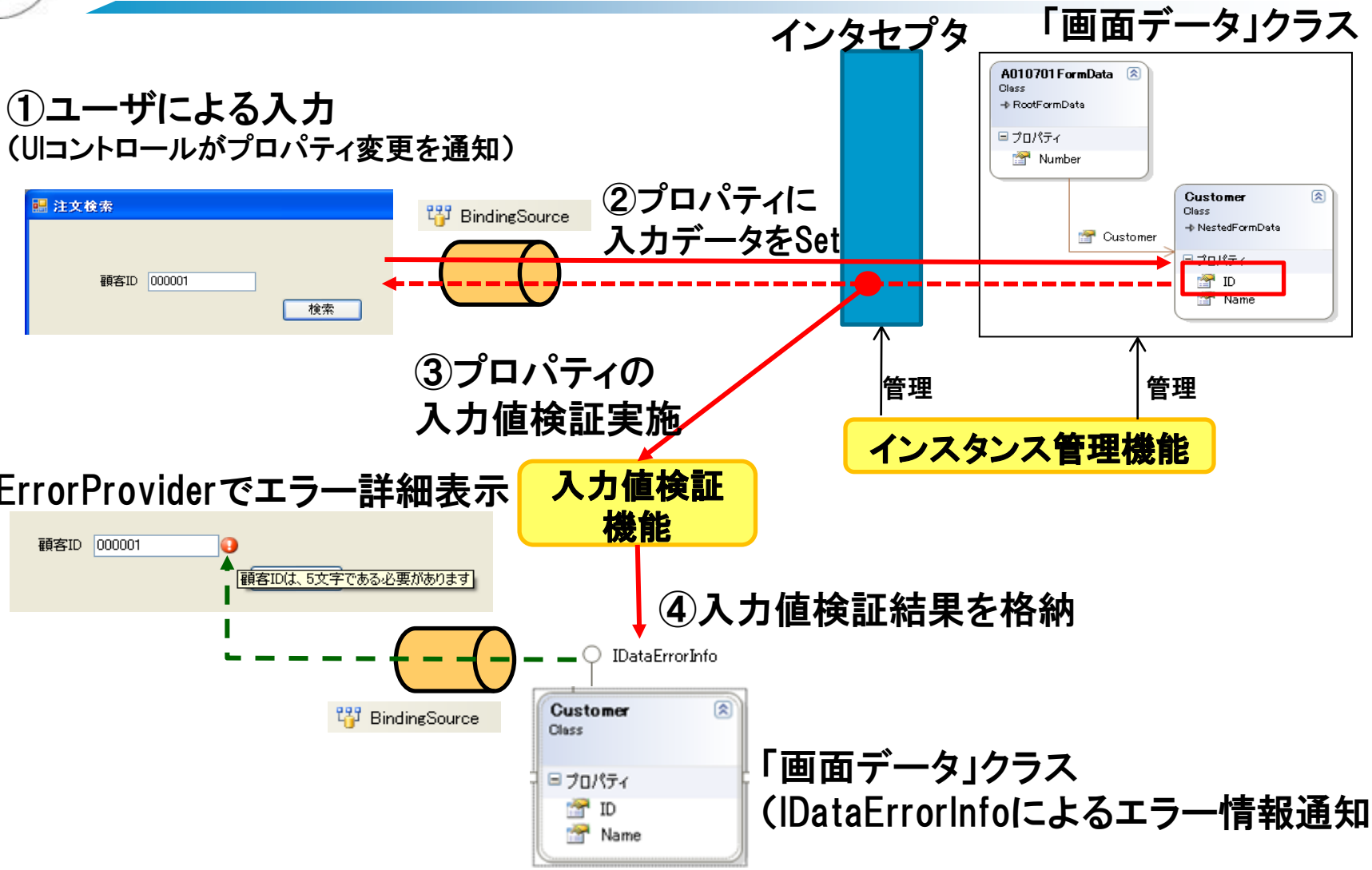
- 相関チェックなどのカスタム入力値検証処理は、メソッドで実装
  - Validation ABのSelfValidaionメソッドを利用

```
[HasSelfValidation]
・・・
public class SampleViewData : ValidatableRootViewData
{
    [SelfValidation(Ruleset = "RS01")]
    public void CustomValidator01(ValidationResults results)
    {
        ///TextFieldとNumberFieldのどちらか入力されていることをチェック
        if (!string.IsNullOrEmpty(Id) || !string.IsNullOrEmpty(Num))
        {
            ValidationResult result =
                new ValidationResult(Resource1.MSG0002, this, "TextField", null,
                                     EventSpecificValidator.DefaultInstance);
            ValidationResult result2 =
                new ValidationResult(Resource1.MSG0002, this, "NumberField", null,
                                                 EventSpecificValidator.DefaultInstance);
            results.AddResult(result);
            results.AddResult(result2);
        }
    }
}
```

HasSelfValidaion属性

SelfValidation属性を付与し、ValidationResultsを引数として、戻り値がvoidのメソッドを定義して、入力値検証処理を実装

即値チェックの動作イメージ



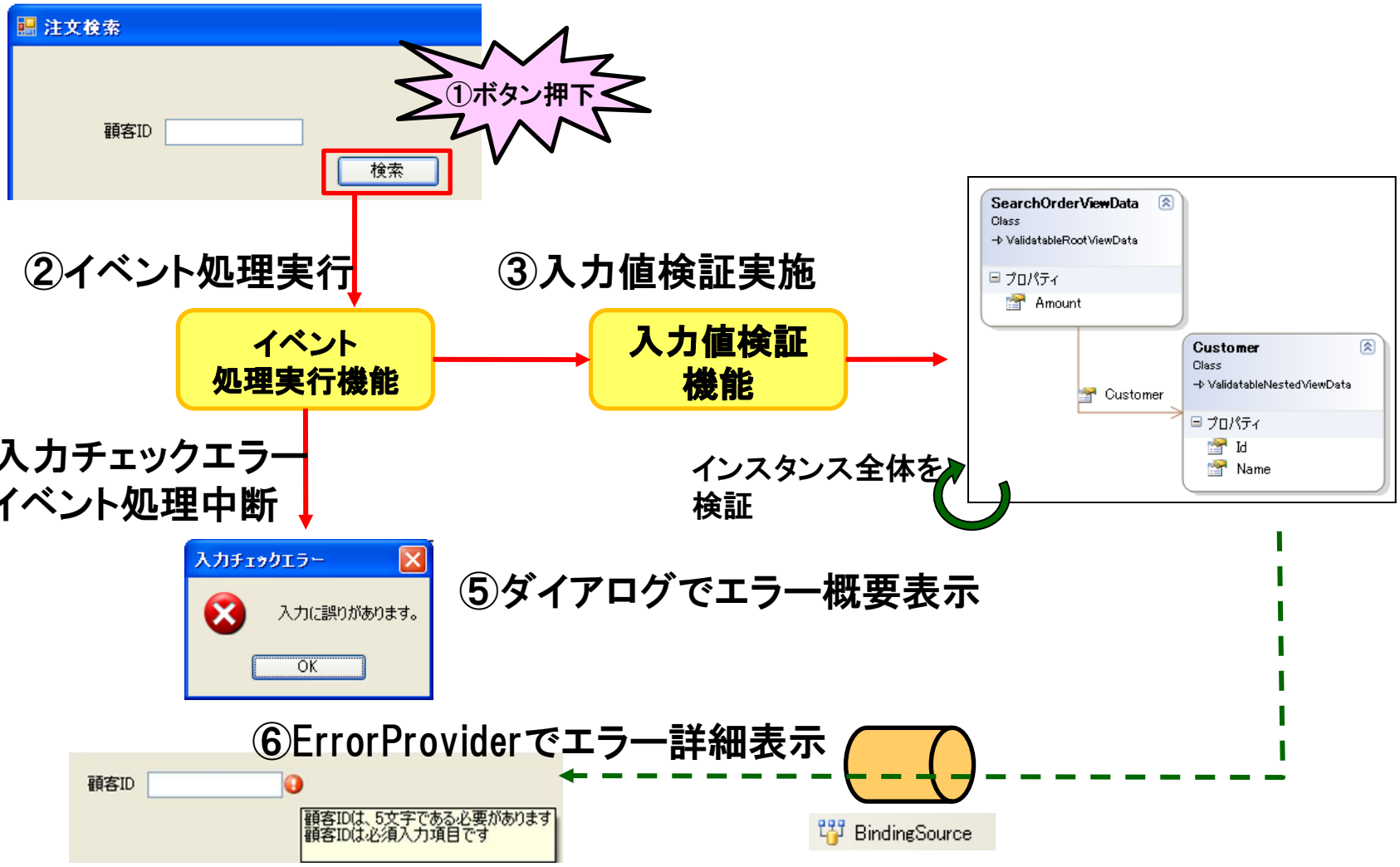
即値チェックの設定

- DefaultRuleset属性の導入
  - ValidationABを拡張し即値チェックで実施する入力値検証処理のルールセット(入力値検証のパターン)名を指定
  - 「画面データ(ルート)」に1つだけ設定可能
    - つまり、1画面に対して1つだけ設定可能

「画面データ」クラスに対するカスタム属性で記述

```
[DefaultRuleset("RS01")]
[RulesetMapping("RS01", "CRS01", "Tour")]
public class SC_A01_01_01ViewData : ValidatableRootViewData
{
    public virtual Tour Tour { get; set; }
}
```

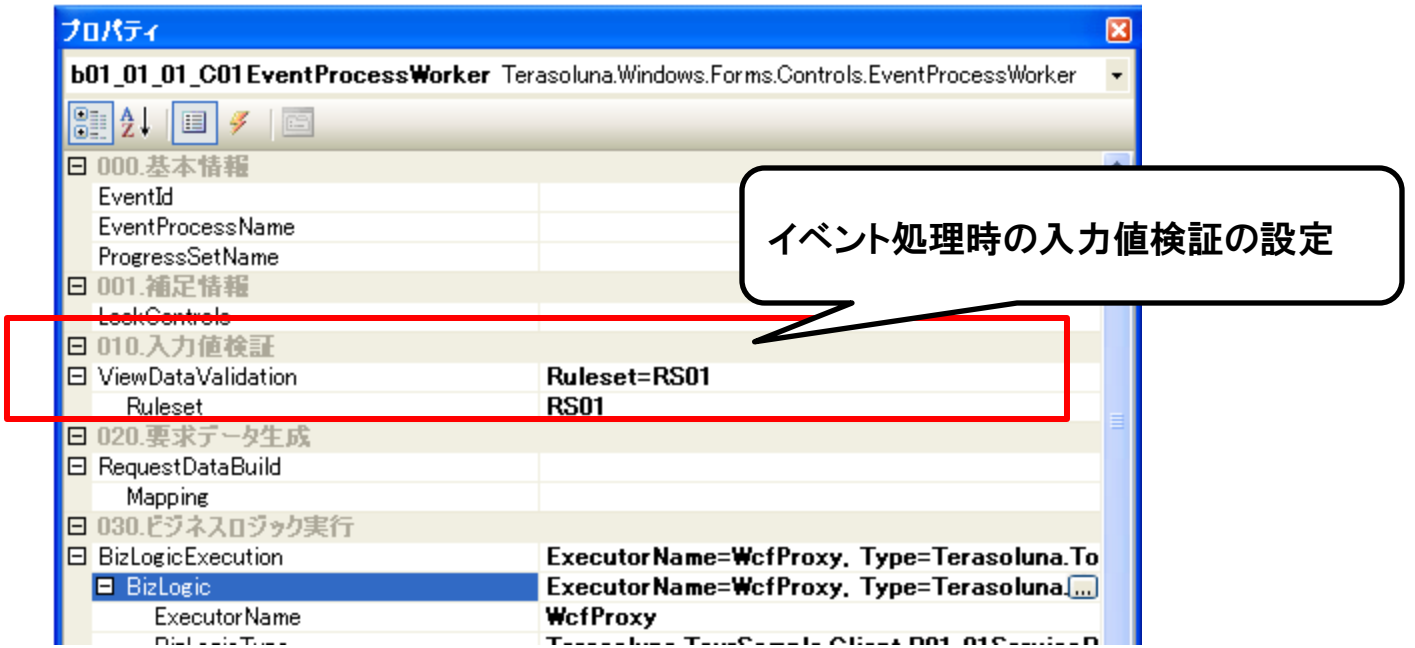
イベント処理実行時の入力値検証の動作イメージ

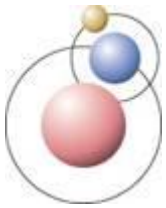


イベント処理実行時の入力値検証チェックの設定

- EventProcessWorkerクラスのプロパティで設定
  - 後述の「イベント処理実行機能」を参照のこと

EventPrcessWorkerのプロパティ





「画面データ」の検討経緯（入力値検証）

- 「画面データ」は、POCOベースの実装

（「画面データ」クラスはTERASOLUNAフレームワーク依存のクラスであるので厳密にはPOCOではないが、DataSetから脱却したという意味で「POCOライク」）

- ◆相互運用性の考慮
  - 通信基盤としてWCFを採用したため、DTOは、言語やプラットフォームに依存せず接続可能にする必要があるため、POCO
- ◆DTO⇔「画面データ」のデータコピー処理の考慮
  - DTOがPOCOなら「画面データ」もPOCOのほうが、自動的にデータコピー処理しやすい
    - リフレクションによるデータコピー
    - 後述の「データコピー機能」の説明を参照

「画面データ」の検討経緯（入力値検証）

- クラスの構造化・共通化
  - DataSetのようなテーブル構造のデータではなく、通常のオブジェクトなので、共通化や構造化が容易
  - 「画面データ」クラスの共通化が進めば、業務ロジックの共通化も促進

- 入力値検証処理の構造化・共通化への対応
  - 検証処理ロジックを「画面データ」クラスに閉じ込めているため、検証処理の共通化や構造化が図れる

「画面データ」の検討経緯（入力値検証）

- ルールセット名のマッピング定義の考慮
  - 「画面データ」クラスに対してどの入力チェックの組み合わせを検証するかを識別する名前として「ルールセット名」を使用
  - 構造化や共通化が進むと、ルート階層の「画面データ」（「画面データ」（ルート））のルールセット名に対して、ネストした「画面データ」（「画面データ」（ネスト））のどのルールセット名を対応させるかといったルールセット名のマッピング定義が必要

「画面データ」の検討経緯（入力値検証）

- 「画面データ」クラスの性質上、画面クラス単位に入力チェック処理を設計する
  - 「画面データ」（ルート）が、画面と1対1に対応する
- 「画面データ」（ルート）で、ルールセットのマッピングルールを宣言的に定義したい

「画面データ」の検討経緯（入力値検証）

- RulesetMapping属性の導入
  - ValidationABを拡張し、ルート階層の「画面データ（ルート）」で全ての「画面データ（ネスト）」のルールセット名のマッピングを定義
    - [RulesetMapping(“「画面データ（ルート）」のルールセット名”,
      “「画面データ（ネスト）」のルールセット名”,
      “「画面データ（ネスト）」までのプロパティパス”)]

```
[DefaultRuleset(“RS01”)]
[RulesetMapping(“RS01”, “CRS01”, “Tour”)]
public class SC_A01_01_01ViewData : ValidatableRootViewData
{
    public virtual Tour Tour { get; set; }
}
```

クラスに対するカスタム属性で記述

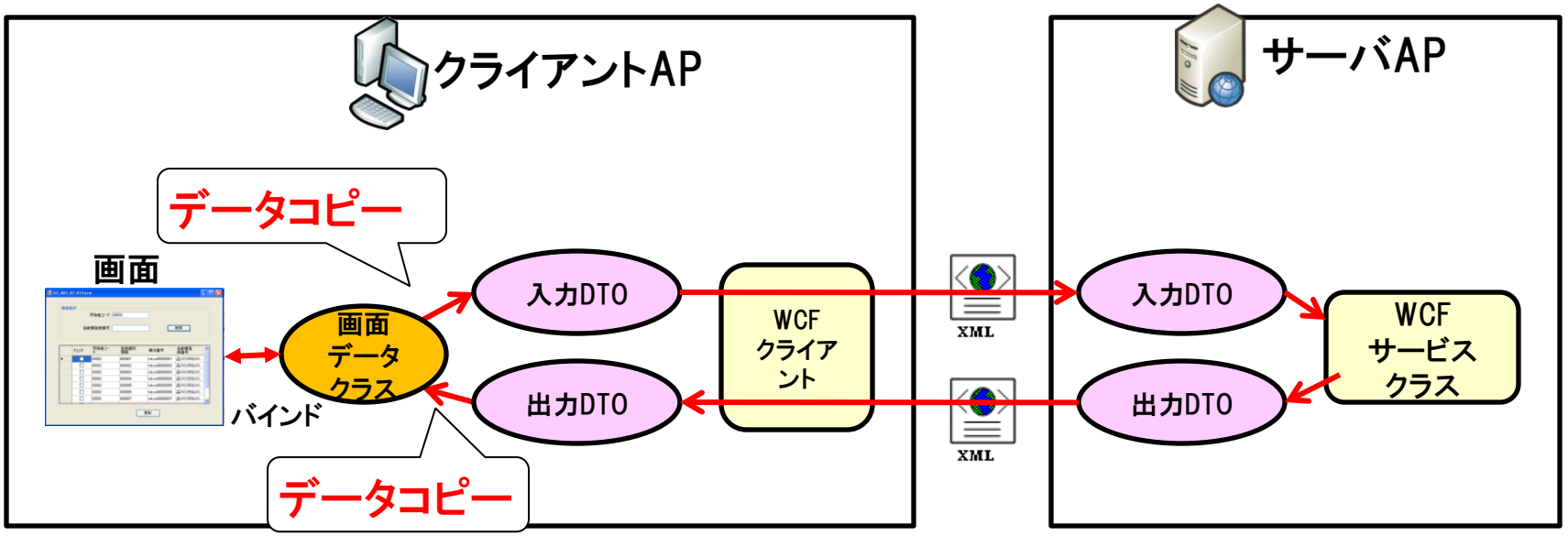
ネストした「画面データ」

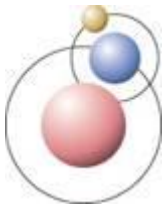
「画面データ」の検討経緯（入力値検証）

- （参考）ObjectValidator属性、ObjectCollectionValidator属性
  - [ObjectValidator(“ネストデータのルールセット名”,
    Ruleset = “現在のクラスのルールセット名”)]
  - Validation ABが標準で提供する機能
    - RulesetMapping属性と同様にオブジェクト間のルールセットの対応関係を定義する
  - 「画面データ」クラスに関しては、RulesetMapping属性を利用する
    - ObjectValidator属性やObjectCollectionValidatorは属性は使用しない
    - 画面に対する入力値検証としてコードの見通しをよくしたい
      - ObjectValidatorだと、各プロパティにカスタム属性を定義するため、階層化が進むとコードの見通しが悪くなる
  - サーバ入力値検証（サーバ入力DTOの検証）については、ObjectValidator、ObjectCollectionValidatorを利用する

「画面データ」の検討経緯（DTOとのデータコピー）

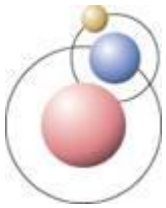
- DTOと「画面データ」を区別したことによるデメリット
  - ビジネスロジック実行直前に、「画面データ」から入力DTOへプロパティのデータコピー処理が必要
  - ビジネスロジック実行後に、出力DTOから「画面データ」へプロパティのデータコピー処理が必要





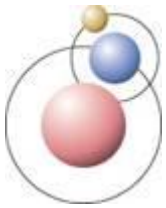
「画面データ」の検討経緯（DTOとのデータコピー）

- 「画面データ」⇔DTOのデータコピー処理の問題
  - 単調で膨大なコピー作業
    - データ構造やプロパティ名がほぼ同一のデータクラス間でプロパティを1つ1つコピーする必要がある。
  - プロパティごとに型変換の考慮が必要
    - 「画面データ」クラスは全て文字列型ベースであるが、DTOは、数値型や日付型といったサーバロジックで扱う実際の型が入っているため、型変換処理を行ったり、Nullやデフォルト値などの考慮も必要である。
  - 従来のFWでは、設定ファイルが膨大に
    - データコピー機能を持つ従来のFWではXMLベースの設定ファイルで行うケースが多い
      - コンパイルエラーによるミス検知ができない、テキストベースの設定ファイルを大量に記述する必要がある



データコピー機能

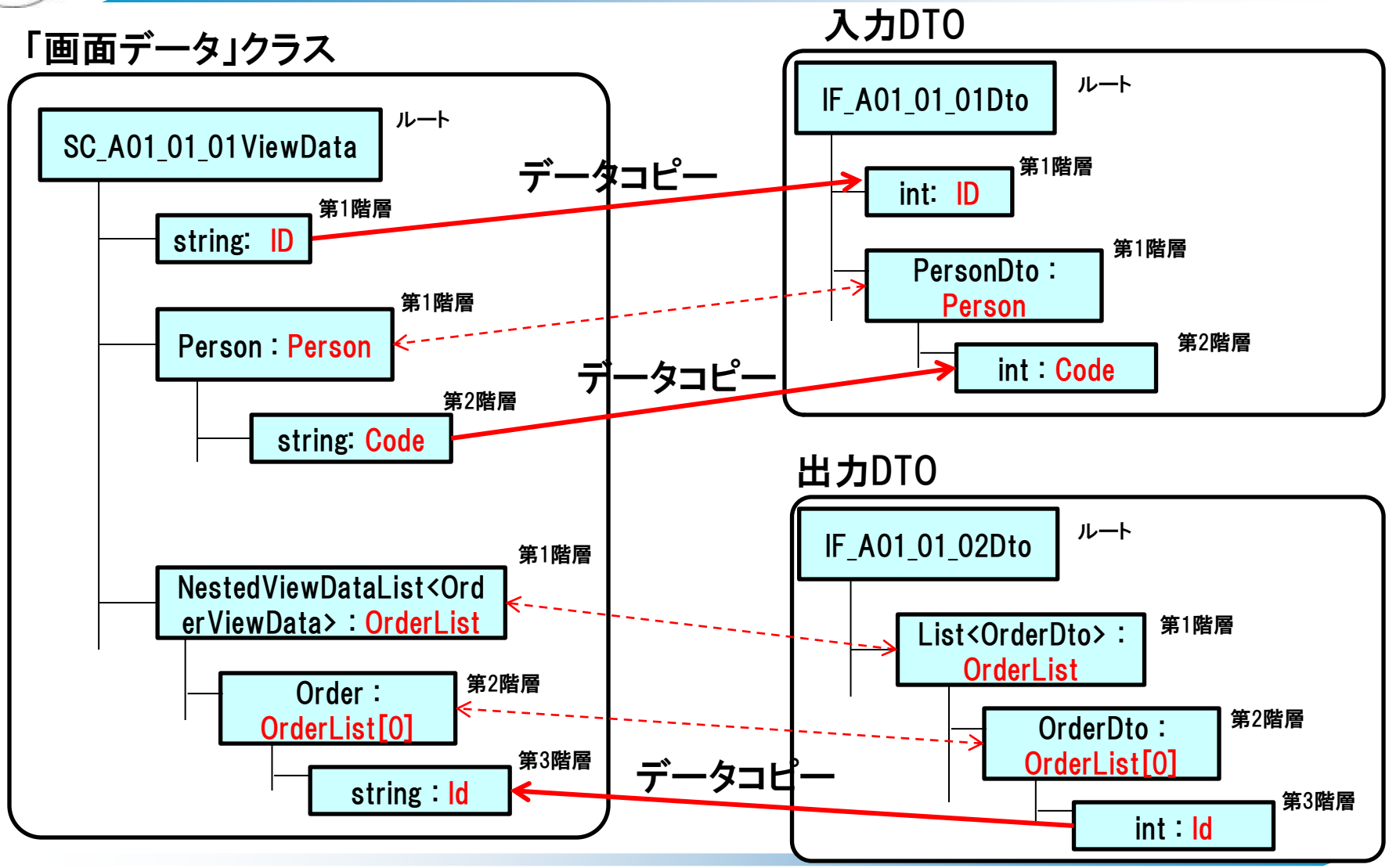
- 「データコピー機能」
  - TERASOLUNAフレームワークの全体共通機能として提供
    - 「画面データ」、DTO等のPOCO、DataSetなど、さまざまなデータクラス間で利用可能な汎用的な機能
  - CoC(Convention over Configuration)によるマッピング設定の作業量削減
    - クラスの階層レベルとプロパティ名が一致していれば、自動的にデータコピー
  - 階層やプロパティ名が一致していない場合でも個別にマッピング設定可能
    - 必要に応じて細かいマッピング設定だけすればよい
  - 自動コピーさせない項目、自動コピーさせたい項目だけを明示的に指定することも可能

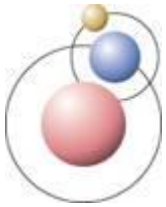


データコピー機能

- 全てディープコピー
  - シャローコピーでのAPの予期せぬ誤動作を防止
- 後述のイベント処理実行機能や画面遷移機能から、データコピー機能が自動的に呼ばれる
  - 個別のマッピング設定は、GUIベースの設定が可能
    - Visual Studioのプロパティエディタで設定
    - 従来のXMLベースの設定ファイルは排除
- ユーティリティメソッドを呼び出しコーディングすることも可能
  - DataCopyManagerのCopyメソッド
  - 「イベント処理実行機能」、「画面遷移機能」などでCoCによる自動コピーやマッピングの定義が難しい場合や、画面内のイベントハンドラ、サーバビジネスロジックでの利用など、さまざまな場面で利用可能

「画面データ」⇔DTOのデータコピーイメージ

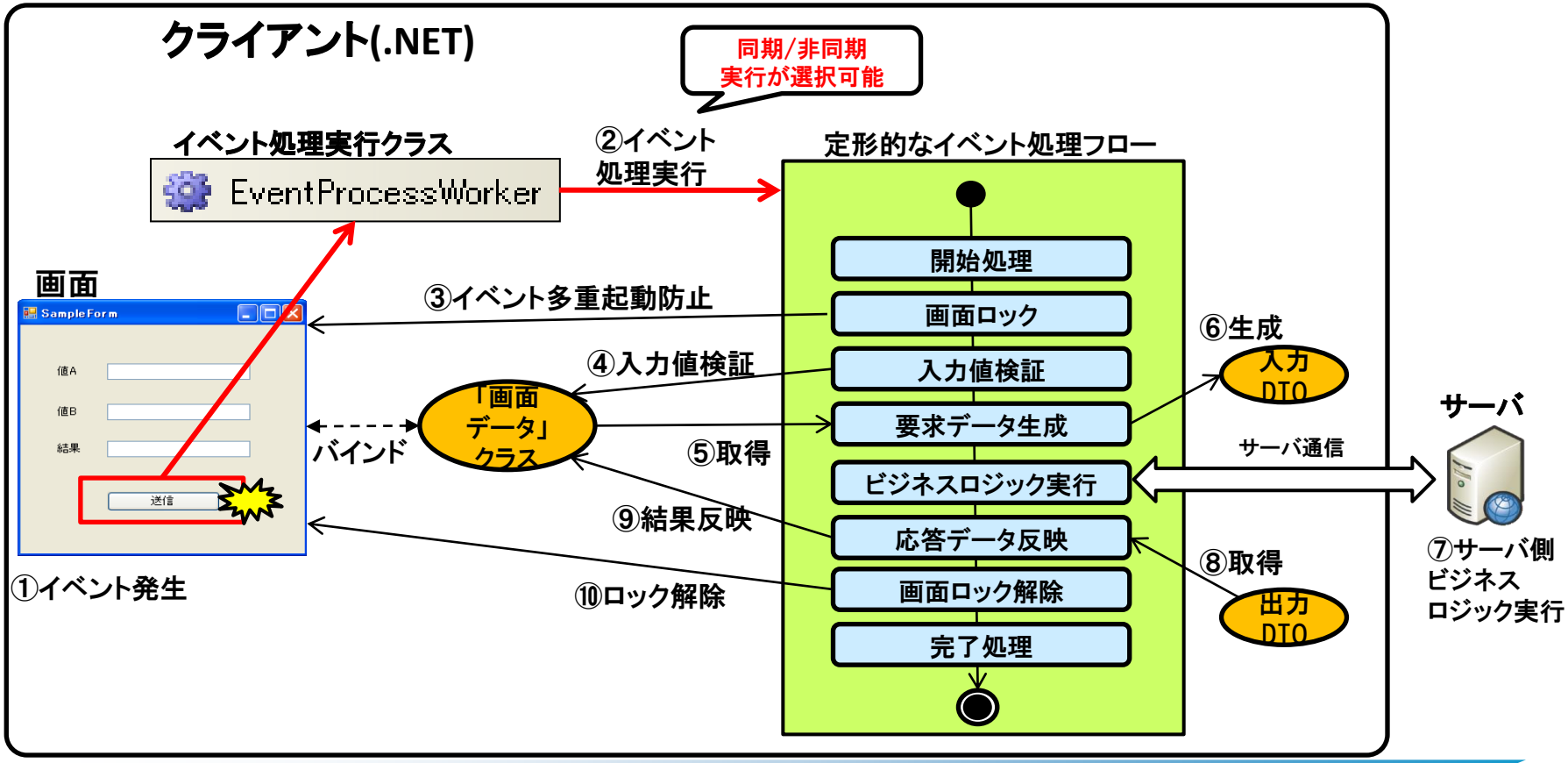




イベント処理機能

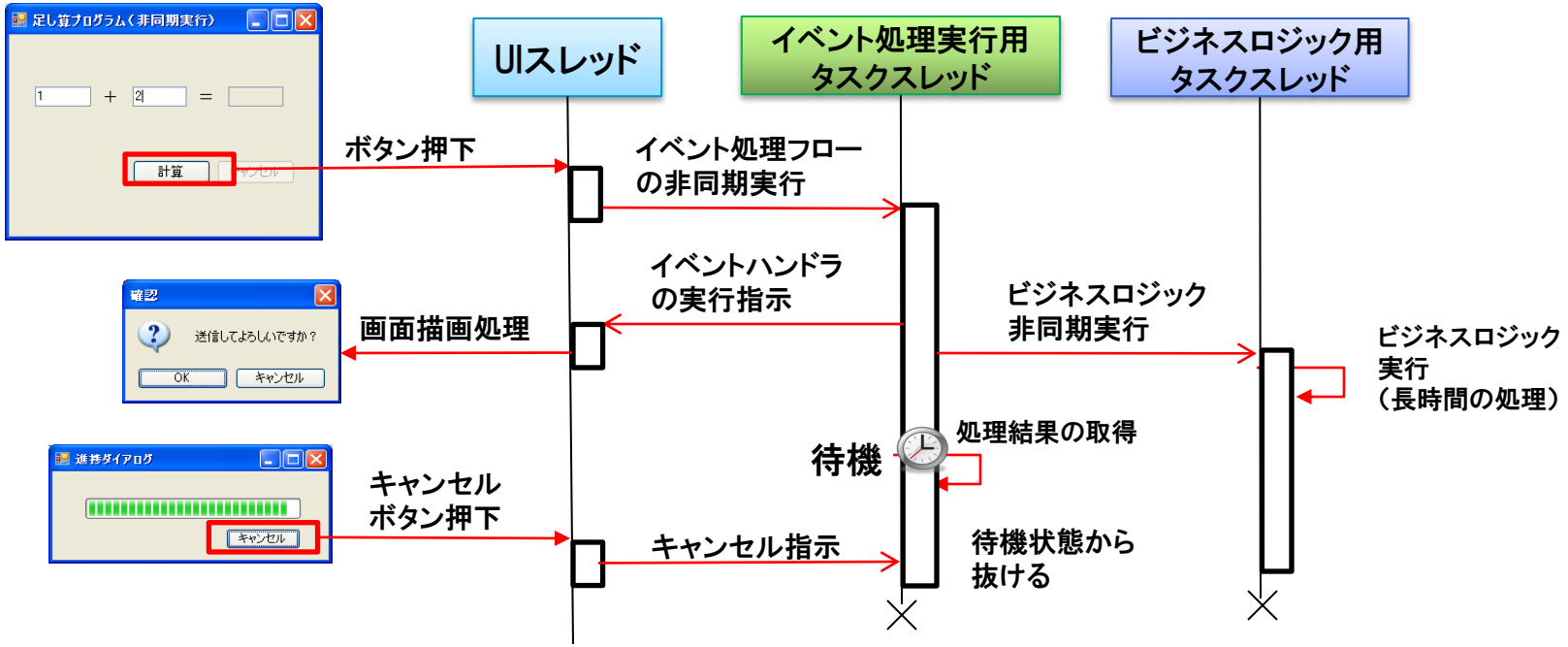
イベント処理実行機能の責務①

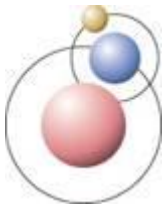
- 業務処理の定型化
  - 業務APの定型的な処理の流れを制御する



イベント処理実行機能の責務②

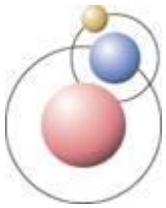
- 業務処理の非同期実行
  - 非同期処理にかかる開発者の実装コストを大幅に軽減
    - 画面描画処理（UIスレッドによる処理）とビジネスロジック（タスクスレッドによる処理）の実装の分離
    - キャンセル処理の実現





イベント処理実行機能の検討経緯(イベント処理フロー)

- クライアント処理の実装時の問題
  - .NETの標準的な実装パターンでは、画面のイベントハンドラメソッド内に全てのクライアント処理を実装してしまう恐れがある
    - ビジネスロジック部分のテストがしにくい
    - コードの見通しが悪く、保守しにくい
- 一般的な業務APの処理の流れは定型的
  - 入力値検証
  - 画面入力からビジネスロジックの入力データを生成
  - ビジネスロジックを実行
  - ビジネスロジックの出力データを画面に表示



イベント処理実行機能の検討経緯(イベント処理フロー)

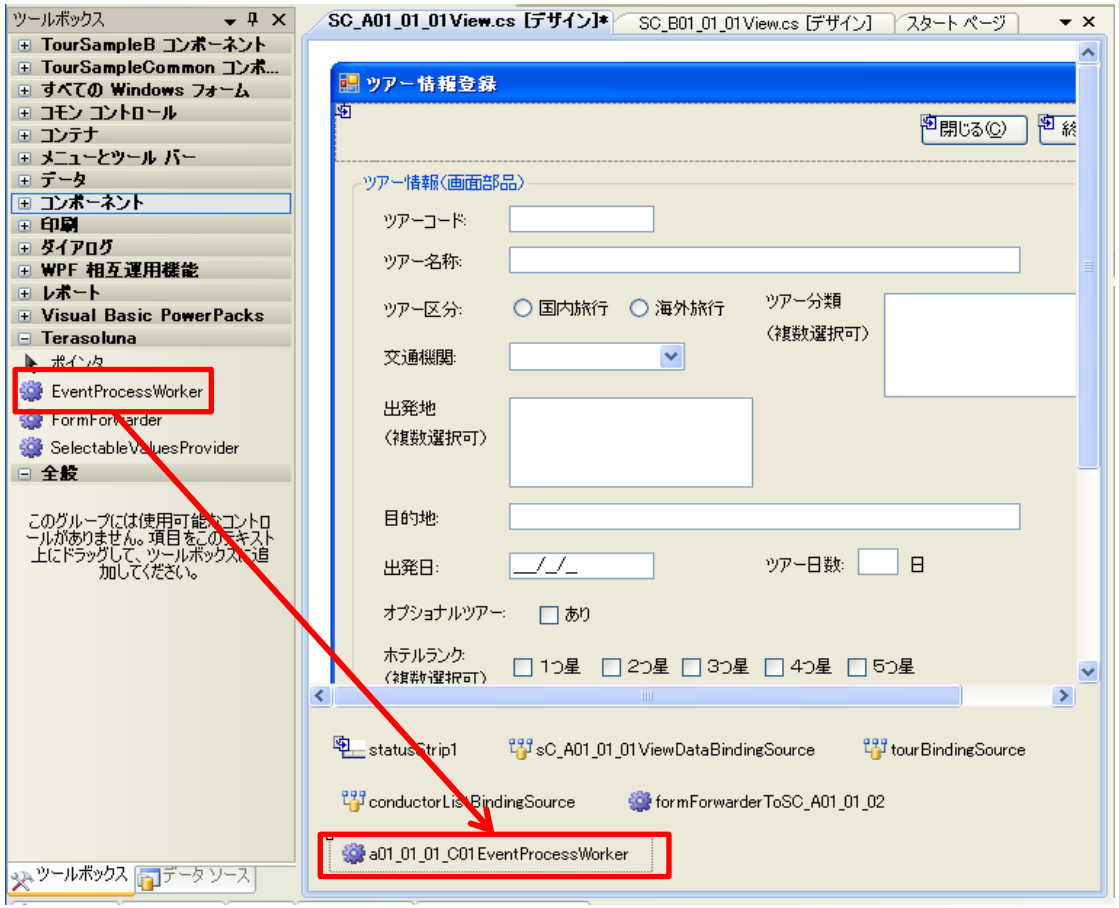
- 定型的なイベント処理フローの提供
  - フレームワークが、ひな型となるイベント処理フローを定義し、業務開発者が個別に処理を実装する場所を縛ることで、均一で保守しやすいコードになる
  - 処理の流れを定型化することで、フレームワークで集約的に例外処理ができるため、実装がシンプルになる
    - 後述のクライアントエラーハンドリング機能を参照

イベント処理実行機能の検討経緯(イベント処理フロー)

- 画面コンポーネントによる開発スタイル
  - 業務開発者は画面部品として画面に貼り付けて開発
    - Componentクラスを継承した画面部品(EventProcessWorkerクラス)を提供
    - プロパティで定型的な動作を設定
      - 実行するイベント処理フロー名
      - 実行する入力値検証処理パターン(ルールセット名)
      - 実行するビジネスロジック(クラス名、メソッド名等)
      - 「画面データ」⇔DTOのデータコピーの個別設定
  - Visual Studioのエディタ機能を拡張したプロパティ入力支援の提供
    - イベント処理フローの候補リストをプルダウン表示
      - 業務開発者は、イベント処理フローを簡単に選択可能
    - 実行対象のビジネスロジックを選択する専用エディタ
      - ビジネスロジッククラスやWCFクライアントの候補を表示

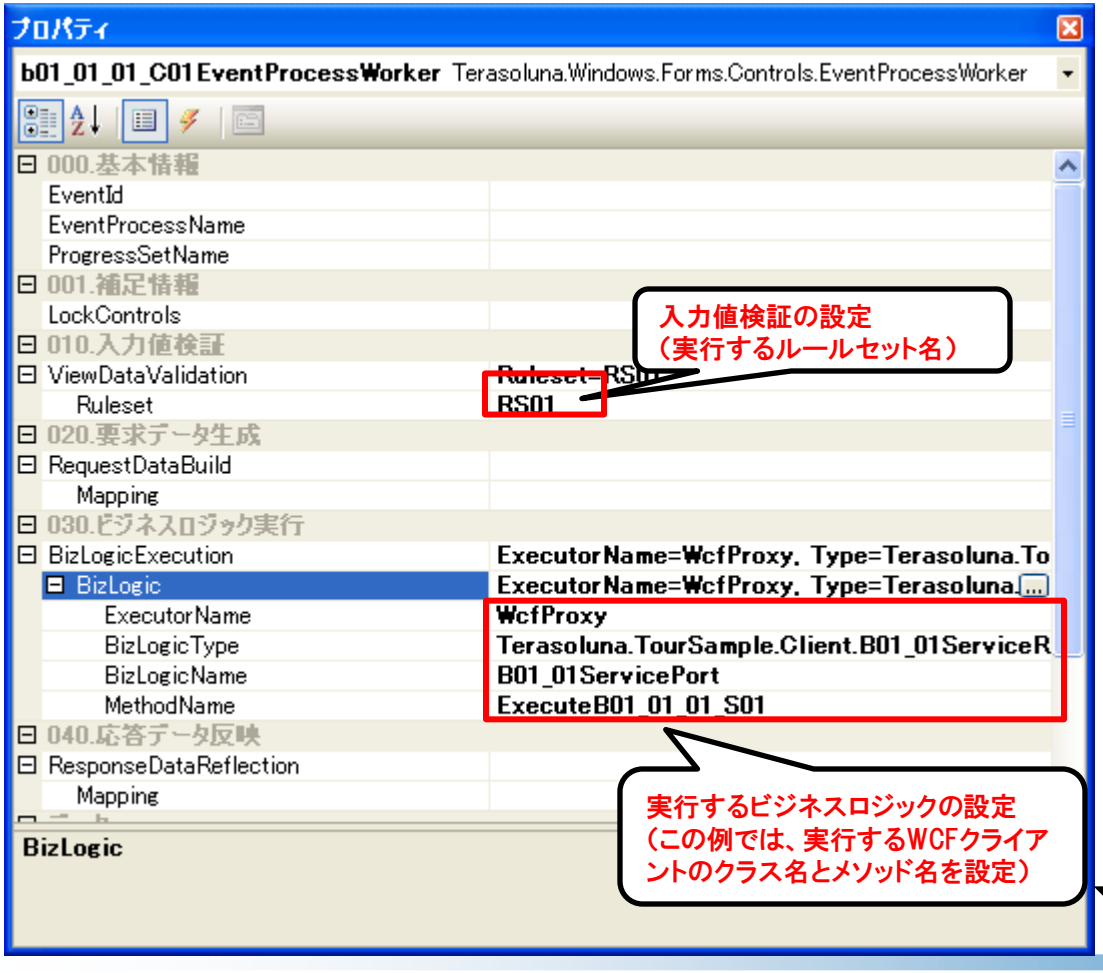
EventProcessWorkerの利用イメージ

- EventProcessWorkerを画面部品として貼り付け

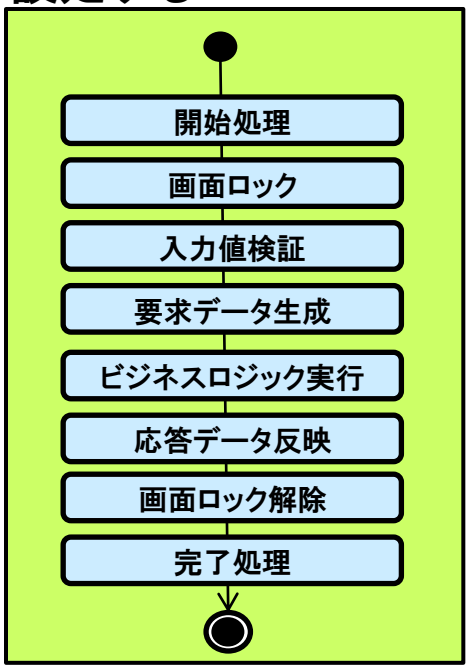


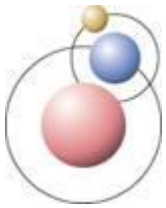
EventProcessWorkerの利用イメージ

EventProcessWorkerクラスのプロパティ



業務開発者はイベント処理フローの流れに沿ってプロパティエディタで設定する



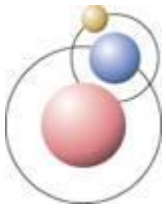


EventProcessWorkerの利用イメージ

- EventProcessWorkerの実行
  - 同期実行の場合はRunWorker()メソッド
  - 処理結果はメソッドの戻り値として取得可能

```
public partial class SC_B01_01_01View：Form
{
・・・
      private void loginButton_Click(object sender, EventArgs e)
      {
         ///イベント処理実行
         EventProcessResult result =  b01_01_01_C01EventProcessWorker.RunWorker();

         if (result.IsSuccess)
         {
            ///成功時はFormForwarderを使って画面遷移
            formForwarderToSC_B01_01_02.Forward();
         }
      }
}
```


TERASOLUNA



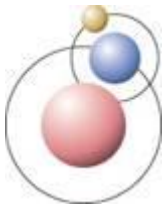
EventProcessWorkerの利用イメージ

- EventProcessWorkerの実行
  - 非同期実行の場合はRunWorkerAsync()メソッド
  - 処理結果はCompletedイベントでハンドリング

```
public partial class SC_A01_01_01View : Form
{
・・・
      private void registTourButton_Click(object sender, EventArgs e)
      {
          ///イベント処理実行
          a01_01_01_C01EventProcessWorker.RunWorkerAsync();
      }
・・・
      /// <summary>
      ///イベント処理フロー完了時(Completedイベント)
      /// </summary>
      private void a01_01_01_C01EventProcessWorker_Completed(object sender,
                                                  Terasoluna.Windows.UI.Events.EventProcCompletedEventArgs e)
      {
          if (e.Result.IsSuccess)
          {
              ///成功時は、ダイアログを表示（メッセージ通知機能）
              MessageNotifier.ShowInformationMessage(this, ResourceCommon.INFO_COMMON_0001);
          }
      }
}
```

イベント処理実行機能の検討経緯(イベント処理フロー)

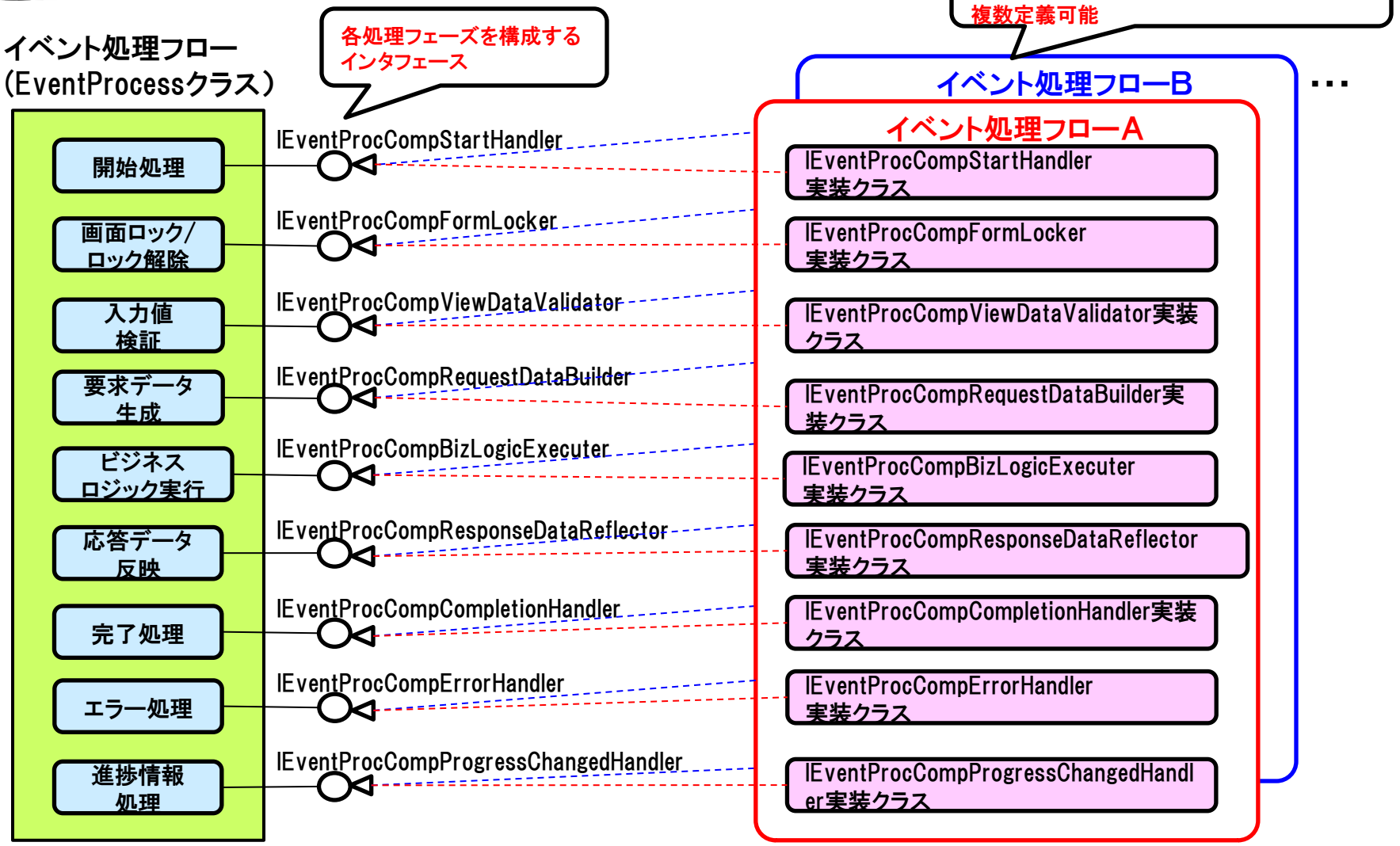
- イベント処理フローの拡張性の考慮
  - 定型的な処理も業務APの特性によって異なる
    - アーキテクトがイベント処理フローで実施する共通的な処理を自由に差し替え可能にする柔軟性が必要である
    - イベント処理フローは複数定義できる必要がある
    - FW標準提供およびアーキテクトが追加定義したイベント処理フローパターンの中から、業務開発者が処理内容を簡単に設定できるようにする
  - 各業務の実装の自由度は高いほうがよい
    - フレームワークで縛りつつも、ある程度の自由度はほしい
    - ひな型となるイベント処理フローに、業務開発者が処理を差し込めるポイントをきめ細かく定義
    - できるだけ少ない手順で簡単に処理を差し込めるとよい



イベント処理実行機能の検討経緯(イベント処理フロー)

- DIによるイベント処理フローの定義
  - 「インスタンス管理機能」を利用
    - Unity ABのDI(Dependency Injection)機能
  - イベント処理のデフォルト実装を差し替え可能に
    - イベント処理フローを構成する各処理フェーズに対応するインタフェースとデフォルト実装を提供
    - アーキテクトは、インタフェースを実装しデフォルト実装を差し替えたプロジェクト独自のイベント処理フローを定義可能
    - Extensionクラス(UnityContainerExntension継承クラス)により機能を追加
      - 後述の「TERASOLUNAフレームワークの拡張ポイント」を参照

イベント処理フローのクラス構造



イベント処理フロー定義のイメージ

【アーキテクト】
Extensionクラスの実装

```
public class SampleEventProcessExtension : ValidatableEventProcessExtension
{
    protected override void SetupEventProcesses()
    {
        base.SetupEventProcesses();
        ・・・
        ///イベント処理フローの追加定義
        ///イベント処理フローを実現するコンポーネントをDIで組み合わせる
        RegisterEventProcess<EventProcess>("NewEventProcess",
            ///画面ロック
            new PropertyInstanceMapping(EventProcPropFormLocker, "NewFormLock"),
            ///入力値検証
            new PropertyInstanceMapping(EventProcPropFormLocker, "NewViewDataValidator"),
            ///完了処理
            new PropertyInstanceMapping(EventProcPropCompletionHandler, "NewCompletionHandler"),
        );
    }
・・・
}
```

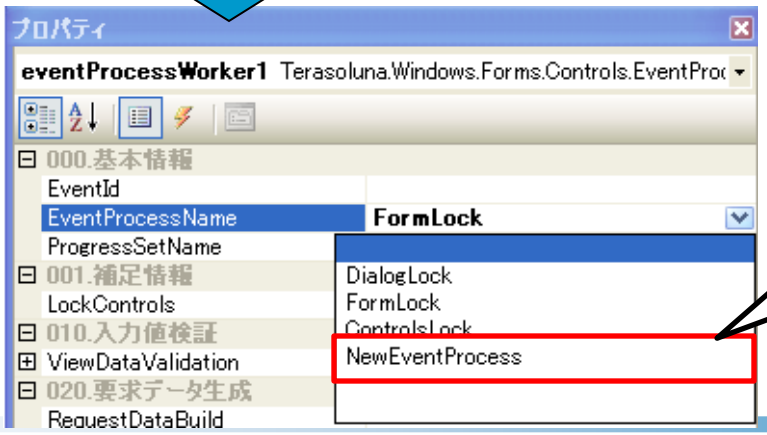
Extesnsionクラスでイベント処理フローをカスタマイズ

“NewEventProcess”という名前の独自のイベント処理フローを追加定義

```
<extensions>
    <!-- イベント処理フロー定義のプロジェクト拡張設定 -->
    <add type="TourSampleAppCommon.Event.SampleEventProcessExtension,
          TourSampleAppCommon" />
</extensions>
```

ExtensionをAP共通構成ファイルに設定

【業務開発者】
EventProcessWorkerのプロパティ設定



イベント処理フローが追加で表示される

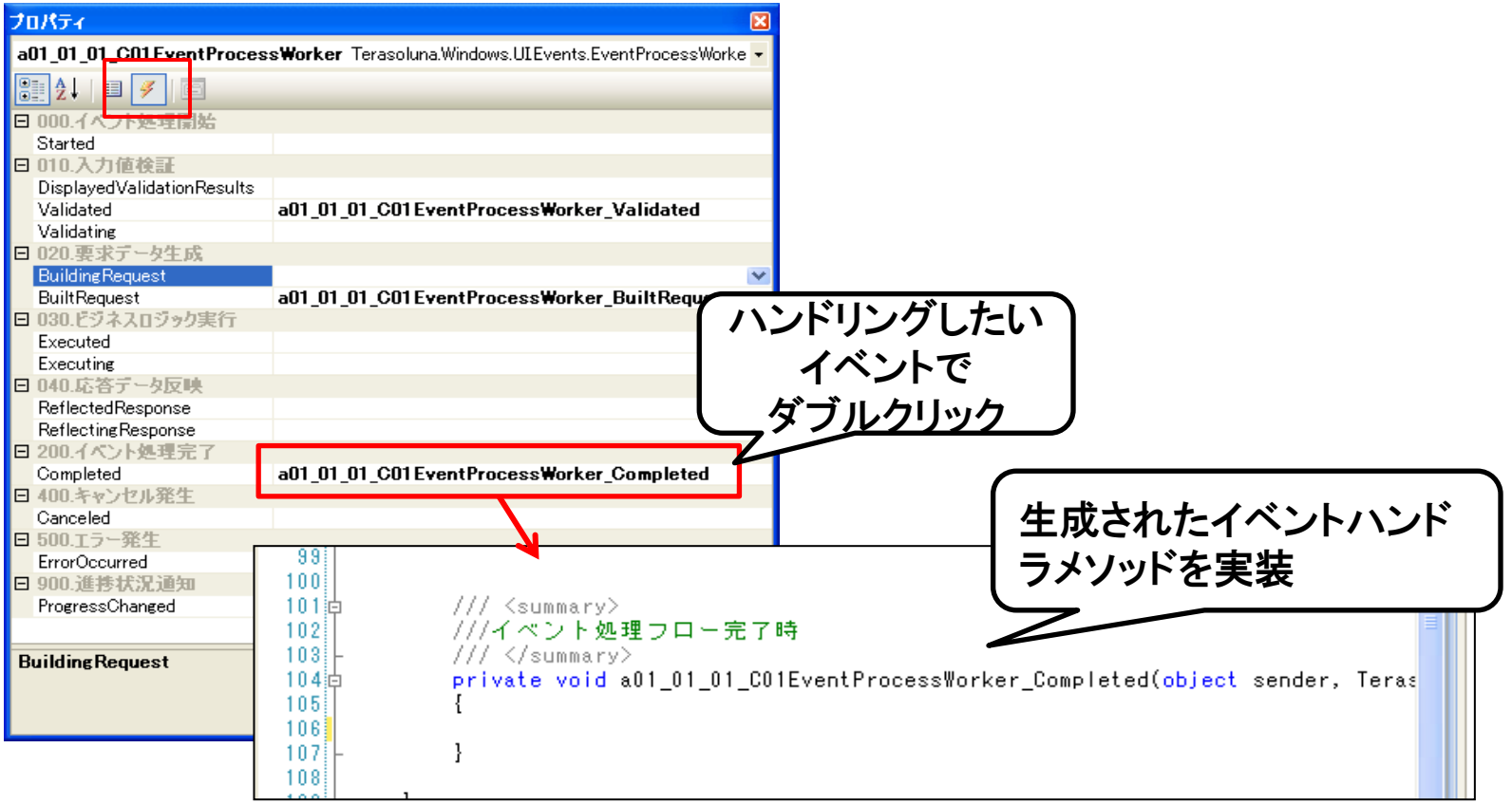
イベント処理実行機能の検討経緯(イベント処理フロー)

- イベントハンドラによる業務処理の実装
  - イベント処理フローが進むごとに.NETのeventが発生
    - 業務開発者はイベントハンドラを実装することで処理を差し込める
    - イベントの発生ポイントがきめ細かく定義されている
      - 処理を差し込めるポイントが多いので、処理フローが定型化されていても、ある程度自由度の高い実装が可能
    - 各業務で想定されるイベント処理実装の例
      - 入力チェックエラー時に、システム共通では実装できない個別のエラー処理を実施
      - ビジネスロジック処理直前に、実行確認のダイアログを表示
      - ビジネスロジック処理成功時に、処理完了のダイアログを表示
      - ビジネスロジック処理結果に応じて画面遷移

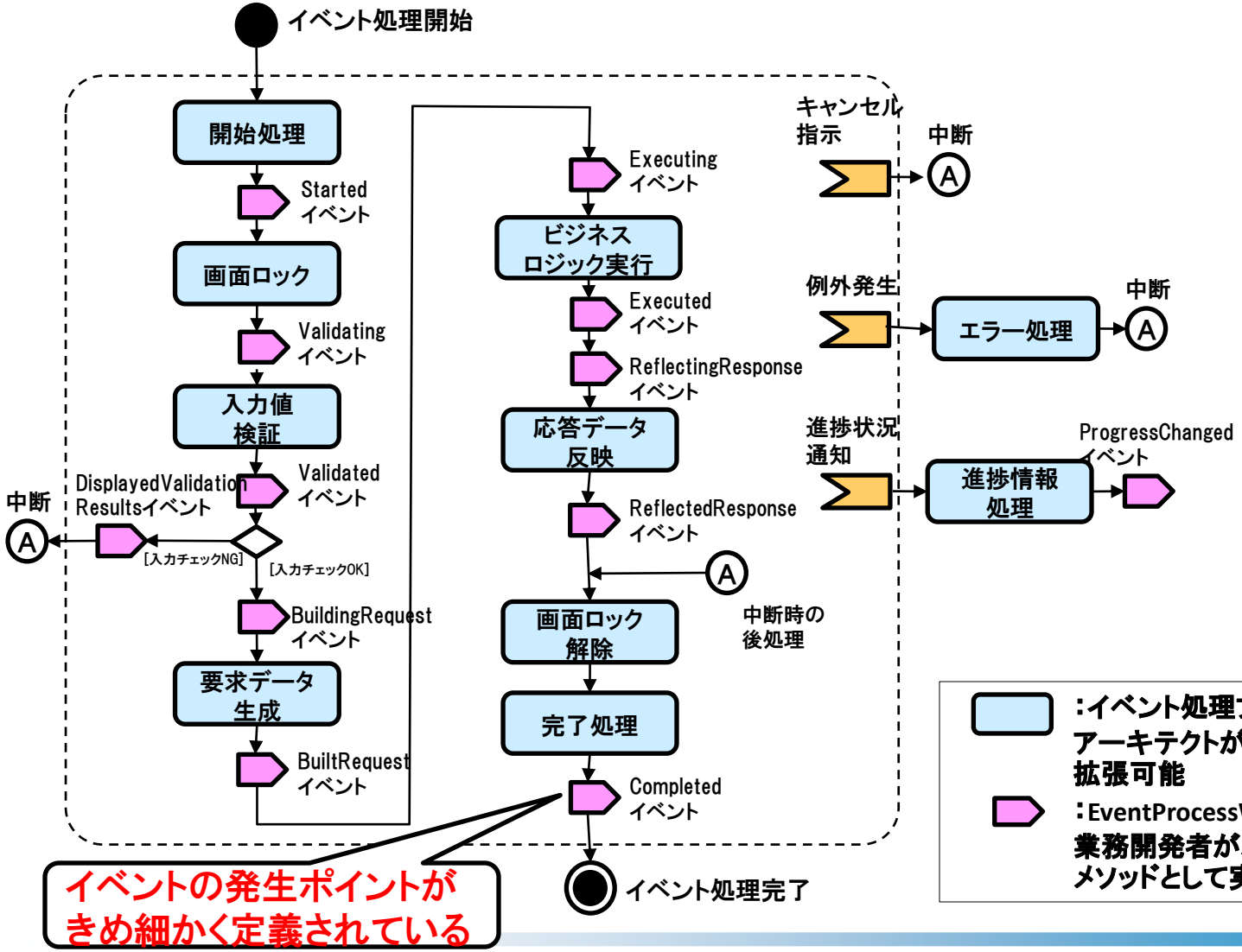
EventProcessWorkerの利用イメージ

イベントハンドラメソッドの実装

EventProcessWorkerのイベント



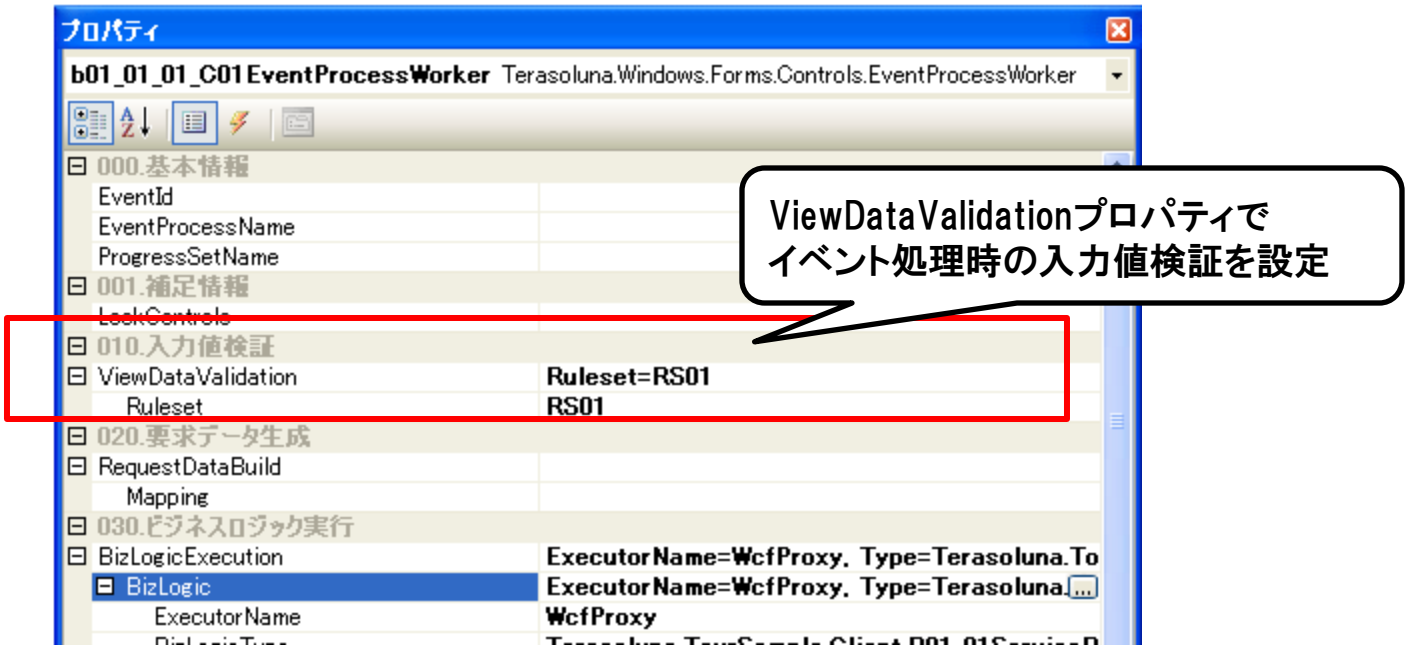
EventProcessWorkerによるイベント処理フロー



イベント処理実行機能の検討経緯（入力値検証）

- ビジネスロジック実行前に入力値検証を実施
  - 「画面データ」クラスの入力値を検証する
    - 画面データ機能を参照のこと

EventPrcessWorkerのプロパティ



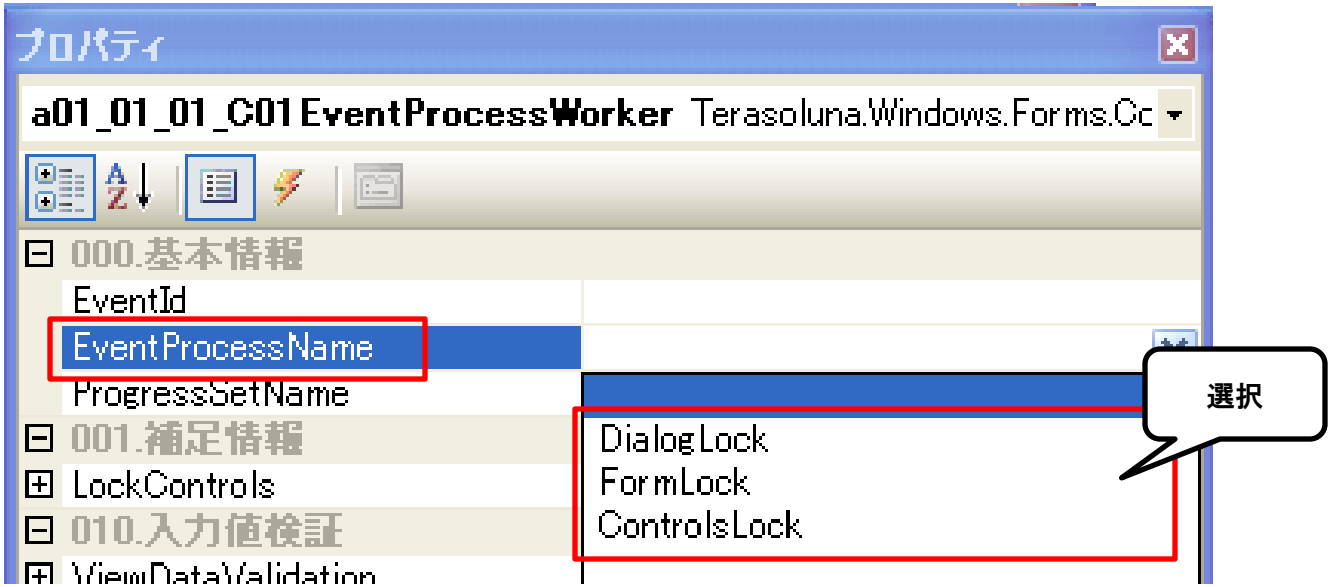
イベント処理実行機能の検討経緯（画面ロック）

- 画面ロック機能
  - ボタンのダブルクリックによるデータの二重登録など、ユーザの誤操作によるイベント処理の多重起動を防止
- 標準提供するイベント処理フロー
  - 画面ロックの方法ごとに3種類のイベント処理フローを標準提供

イベント処理フロー名	同期/非同期	動作説明	デフォルト
FormLock	同期/非同期実行可（同期向き）	イベント発生元画面全体を無効化する	
ControlsLock	同期/非同期実行可（非同期向き）	開発者が指定した、イベント発生元画面のUIコントロールを無効化する	○
DialogLock	非同期のみ実行可	モーダルダイアログを表示しイベント発生元画面を操作不可にする	

画面ロックの設定イメージ

- 画面ロック方式は、イベント処理フロー名を選択して決定する



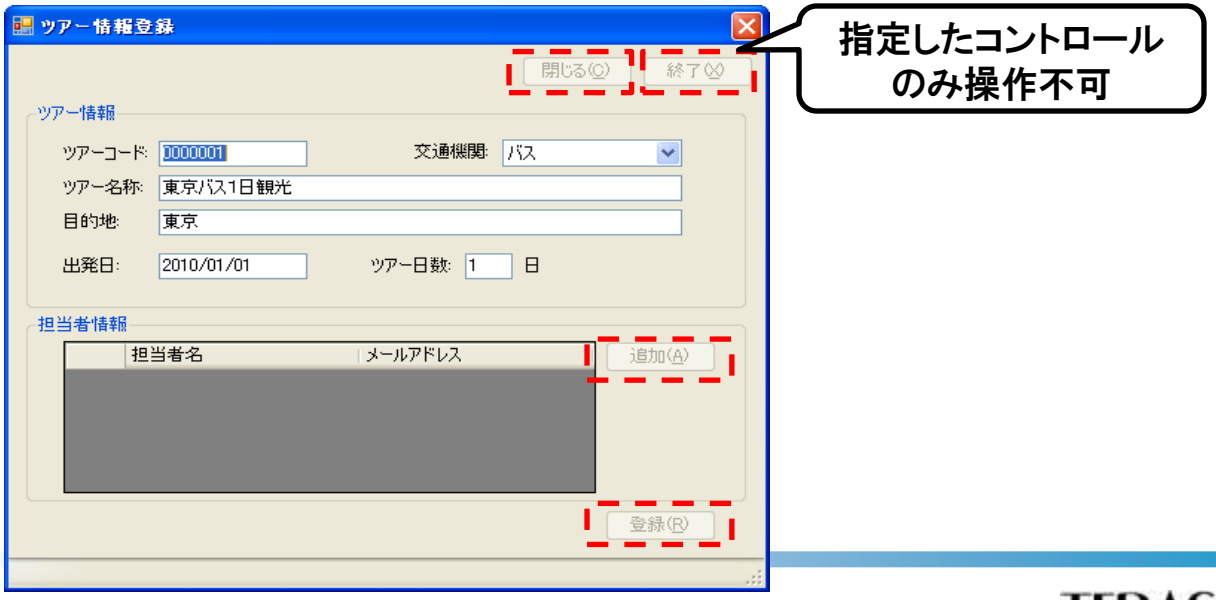
画面ロック(FormLock)

- イベント発生元画面全体を無効化(Enable=false)
  - 同期実行向けの画面ロック

イベント発生元
画面を無効化

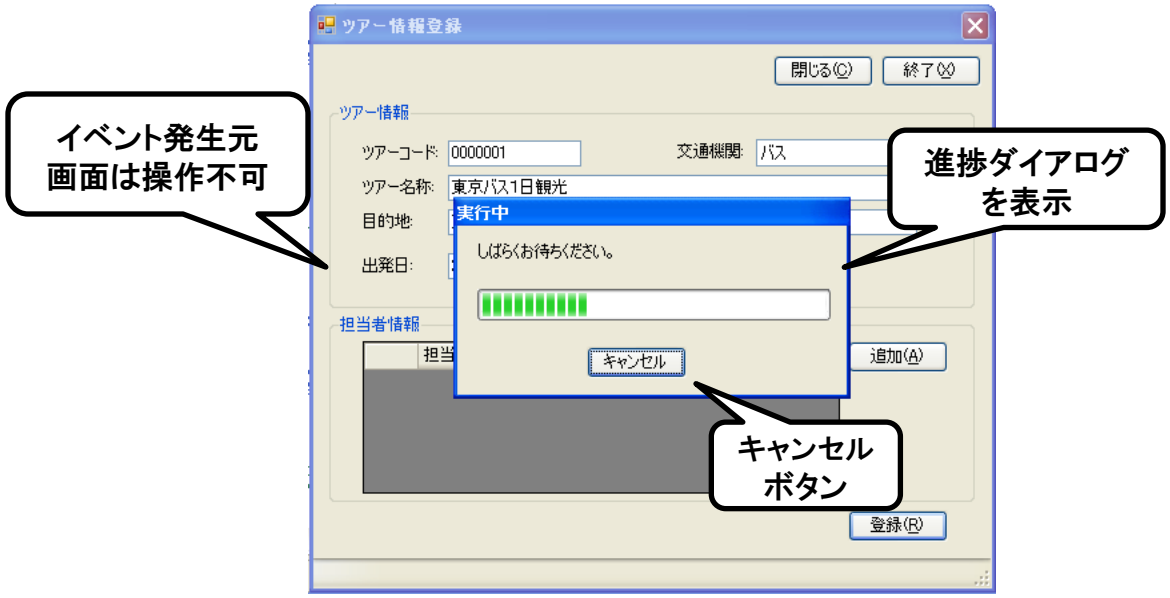
画面ロック(ControlsLock)

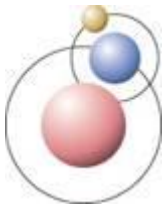
- 特定のUIコントロールを無効化(Enable=false)
  - 開発者は無効化対象のコントロールをLockControlsプロパティで列挙
  - キャンセルボタンなど特定のコントロールを有効化するといった細かい制御が可能
  - 非同期処理時にもイベント発生元画面を自由に移動可能にするといった操作性の高い画面の作成が可能
  - 非同期処理向けの画面ロック



画面ロック（DialogLock）

- モーダルダイアログを表示し、イベント発生元画面を操作不可にする
  - イベント処理の進捗状況を表示するプログレスバーを標準提供
  - キャンセルボタン押下によるキャンセル処理が可能
  - 非同期実行時のみ利用可能





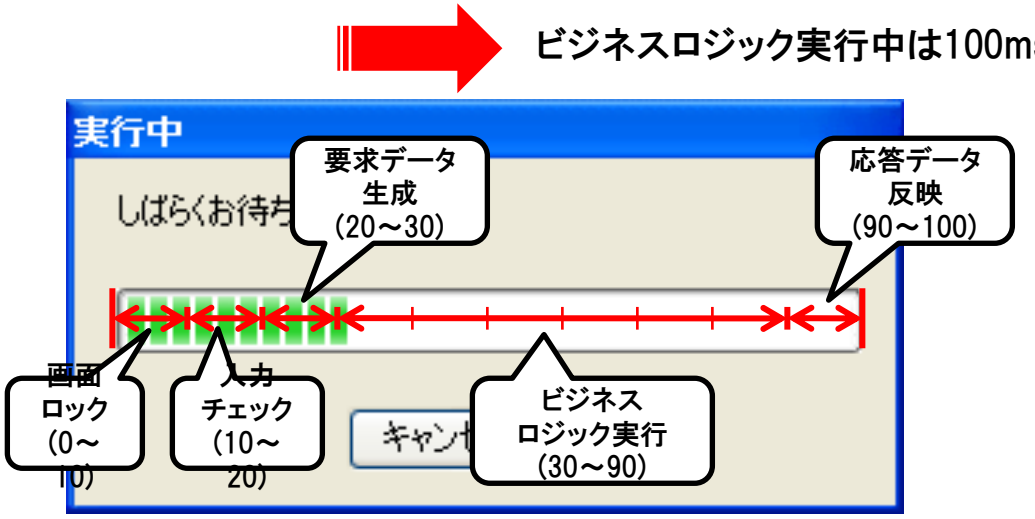
イベント処理実行機能の検討経緯(進捗通知)

- クライアントでの重い処理や、ファイルアップロード・ダウンロードといったサーバ通信に時間がかかる処理など、進捗状況の通知処理が必要なケースが多い
- 非同期処理であれば、プログレスバー表示
  - 前述の「DialogLock」を使用すれば、ダイアログ上のプログレスバーに表示
  - 開発者が作成した画面上のプログレスバーに表示するように実装することも可能

イベント処理実行機能の検討経緯(進捗通知)

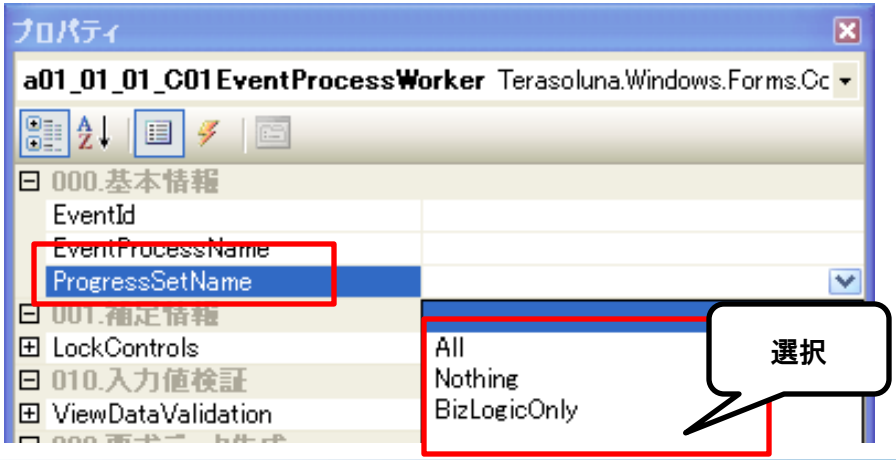
- 進捗率の自動計算機能
  - フェーズの経過や時間の経過から、大まかな進捗率をフレームワークが自動計算し、画面に通知(表示)

ProgressSetNameプロパティ="All"の場合の例



イベント処理の進捗通知の実装イメージ

- 進捗率の自動計算機能の設定
  - ProgressSetNameプロパティで選択可能
    - Nothing(デフォルト)
      - 自動計算機能をオフ(開発者が厳密な進捗率計算を実装する)
    - All
      - イベント開始～終了までの全体の進捗率を自動計算
    - BizLogicOnly
      - ビジネスロジック処理中の進捗率を自動計算



イベント処理の進捗通知の実装イメージ

- 開発者による厳密な進捗率計算の実装
  - サーバへのファイルアップロード時など、送信するファイルサイズに応じた進捗率の厳密な計算が可能な場合に、進捗率をイベント通知することが可能
  - IEventProcProgressManager.SetProgressメソッドで進捗率を設定

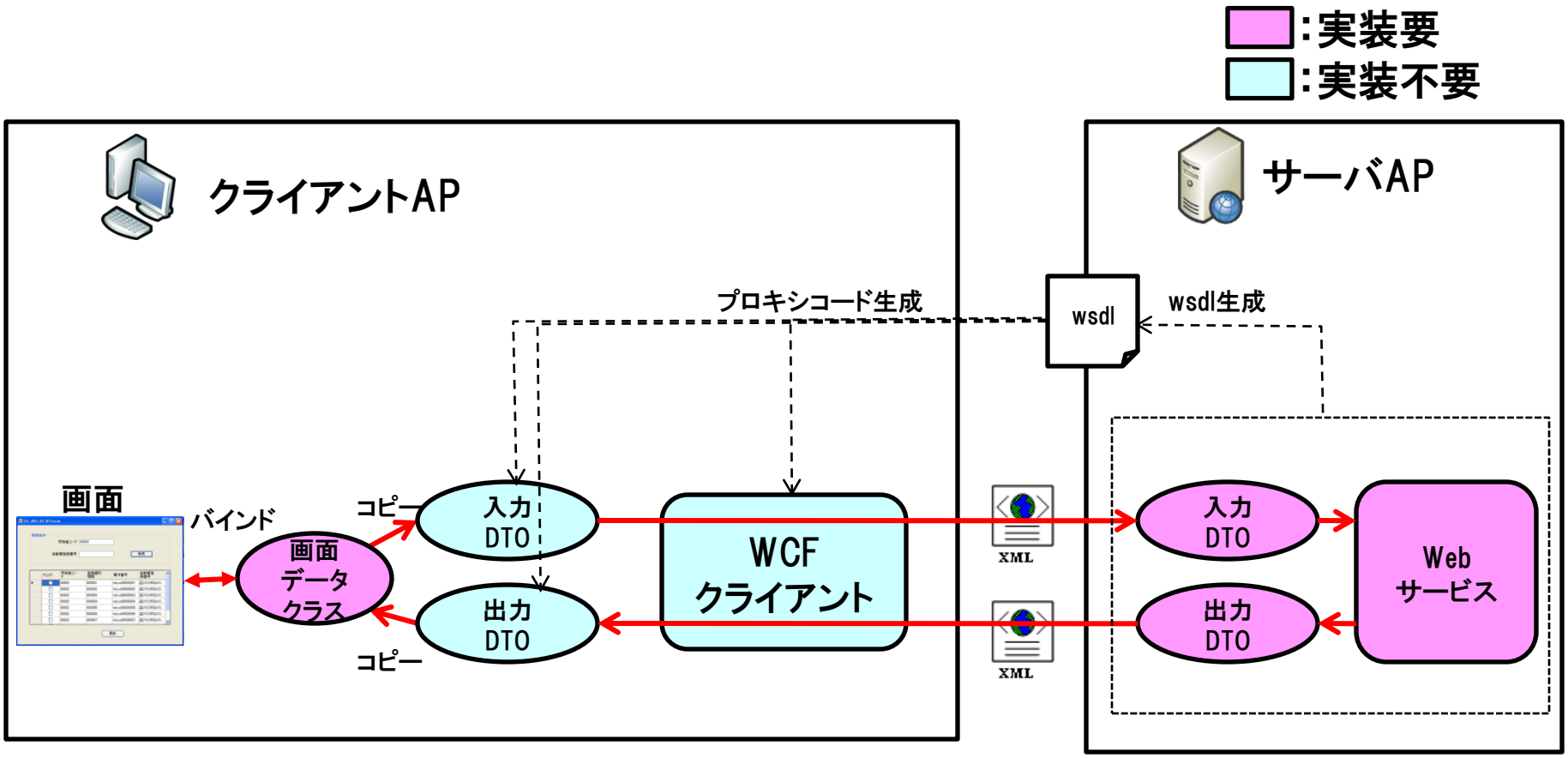
```
///イベント処理の進捗管理クラスを取得
IEventProcProgressManager progressManager =
        InvocationScope.Current.GetContext<EventProcessContext>().ProgressManager;
int result = fileStream.Read(buffer, offset, count);
int percentage = (int)(fileStream.Position * 100 / fileStreamLength);

///進捗率を通知する
progressManager.SetProgress(new ProgressInfo(percentage));
```

イベント処理実行機能の検討経緯(ビジネスロジック実行)

- ビジネスロジック実行の処理パターン
  - クライアント処理を伴わないサーバ通信処理
    - WCFクライアントを直接実行
  - クライアント処理を伴うサーバ通信処理
    - 開発者が実装したクライアントビジネスロジックを実行
    - クライアントビジネスロジック内でWCFクライアントを呼び出しサーバ通信処理を実行
  - サーバ通信を伴わないクライアント処理
    - 開発者が実装したクライアントビジネスロジックを実行

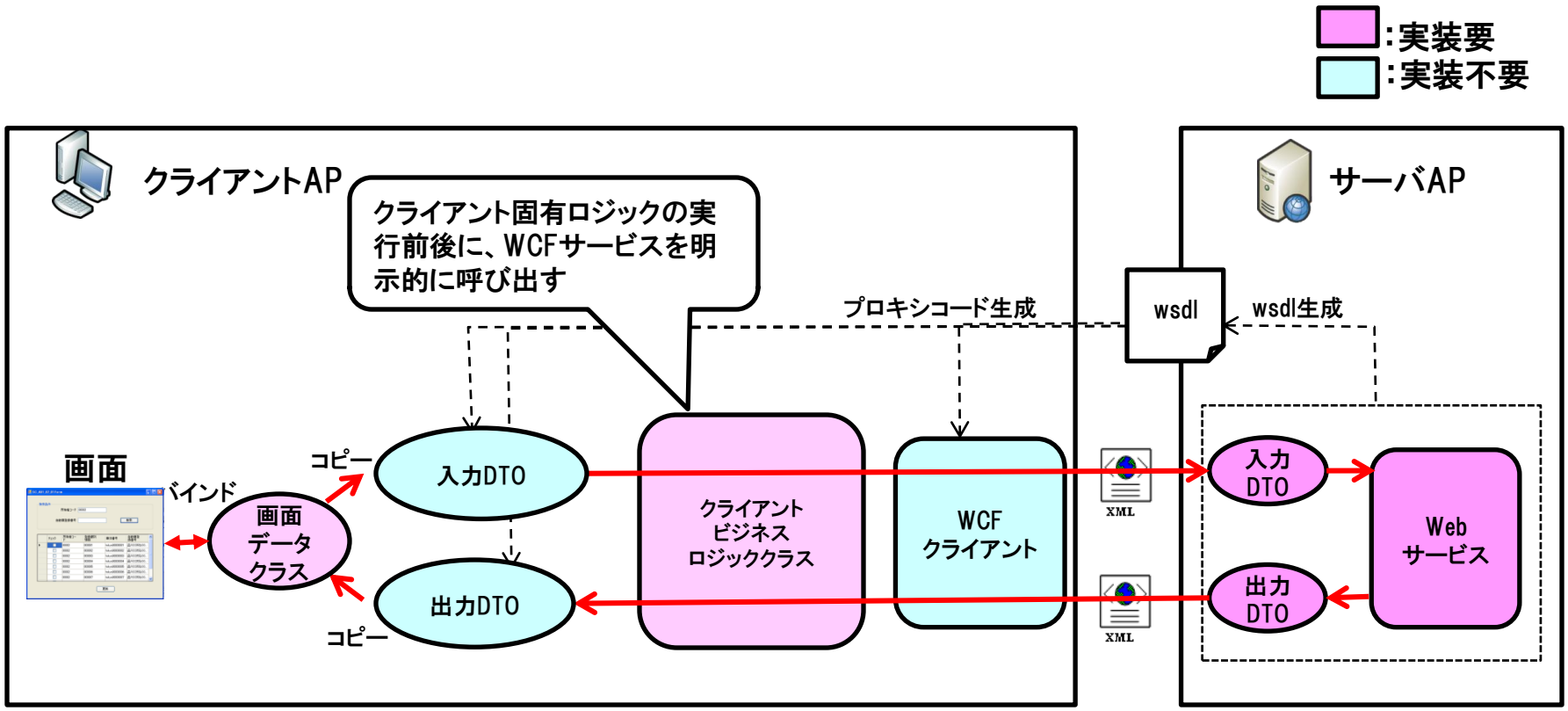
イベント処理実行機能の検討経緯（ビジネスロジック実行）

- クライアント処理を伴わないサーバ通信処理



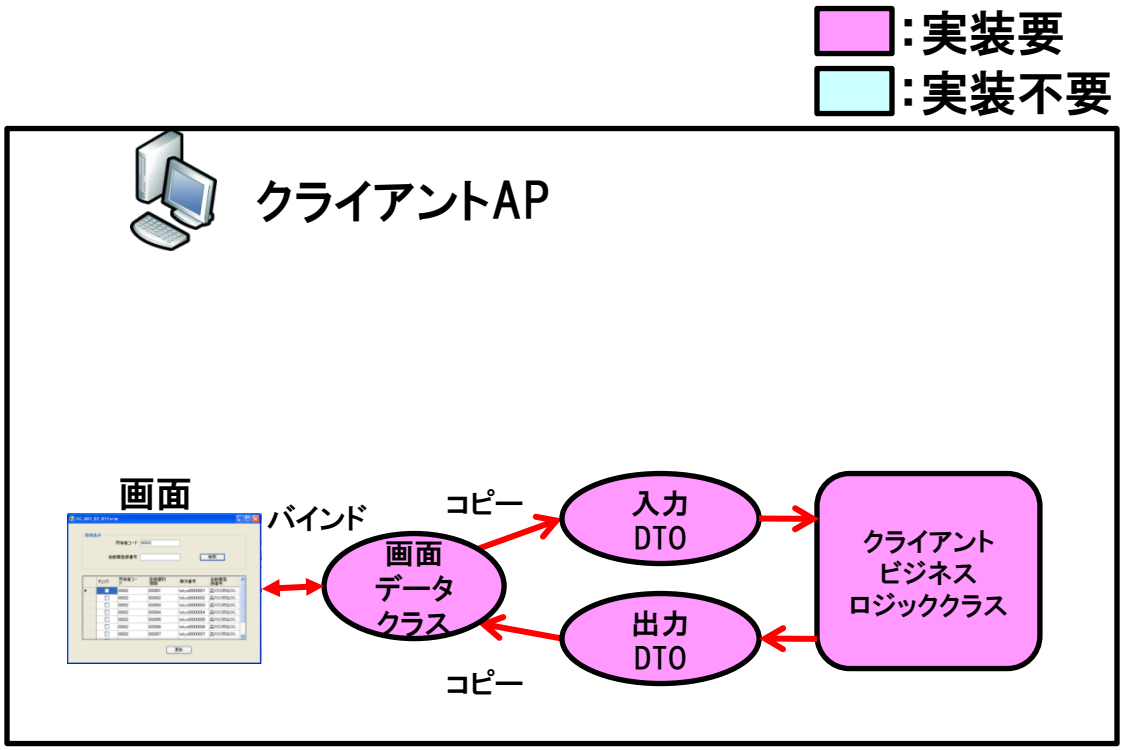
イベント処理実行機能の検討経緯(ビジネスロジック実行)

クライアント処理を伴うサーバ通信処理



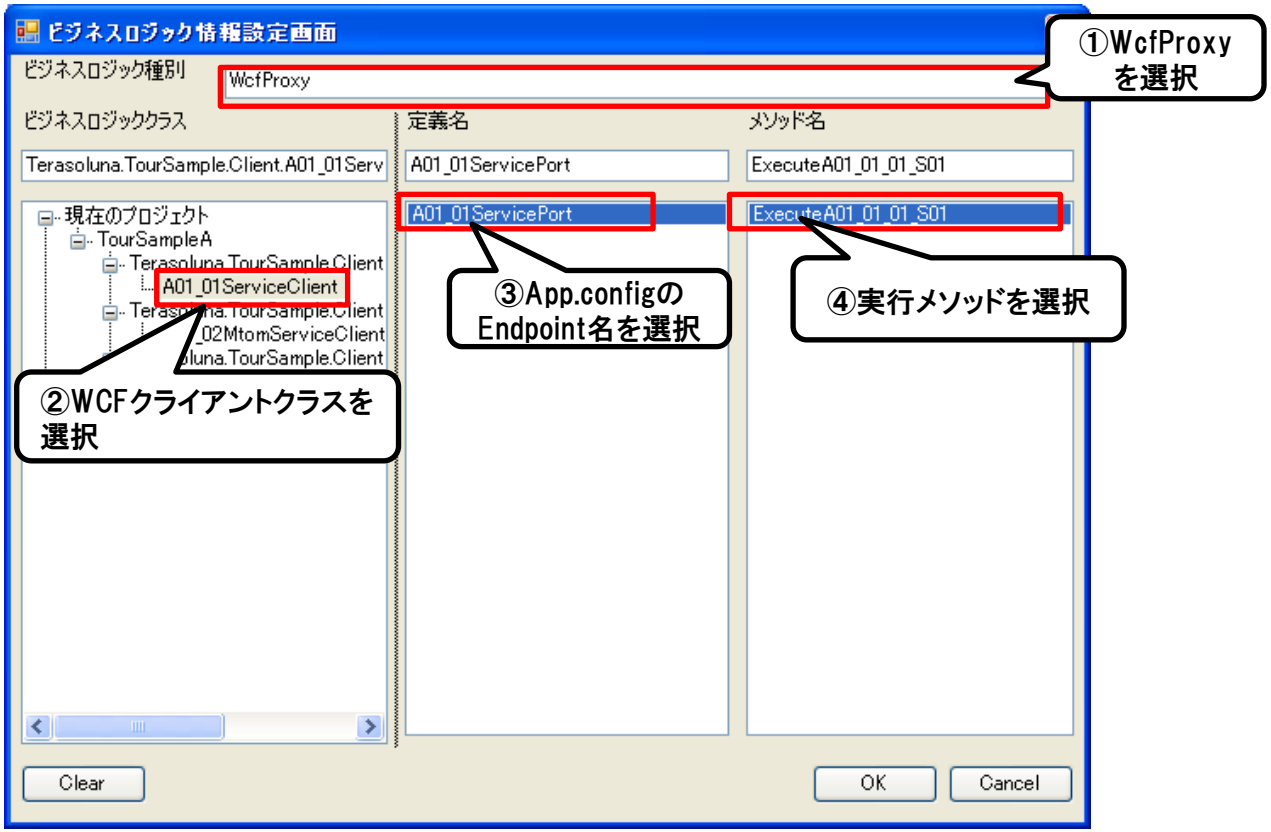
イベント処理実行機能の検討経緯(ビジネスロジック実行)

- サーバ通信を伴わないクライアント処理



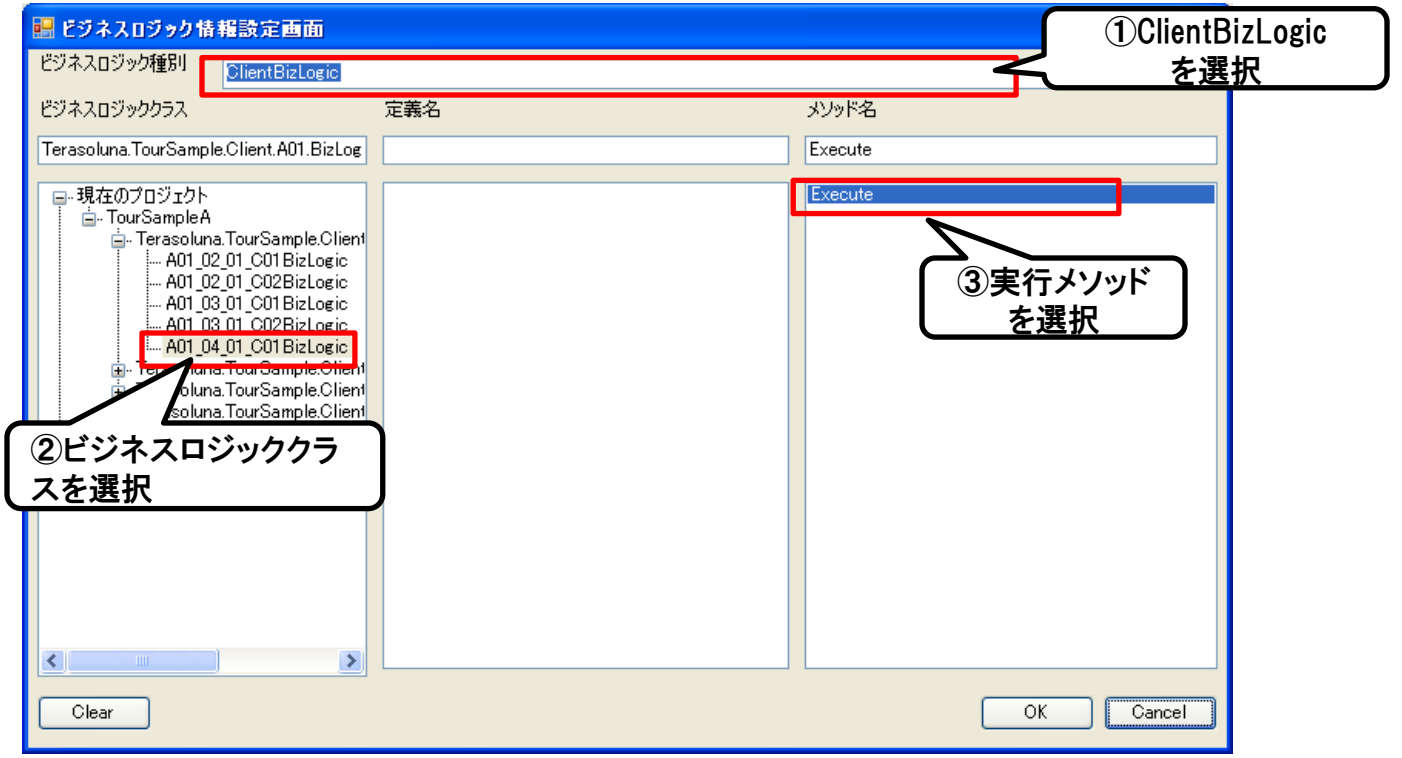
EventProcessWorkerの利用イメージ

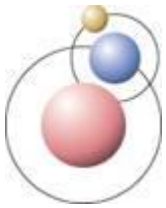
- クライアント処理を伴わないサーバ通信処理
  - 専用エディタで対象のWCFクライアントを選択



EventProcessWorkerの利用イメージ

- クライアント処理を伴うサーバ通信処理
- サーバ通信を伴わないクライアント処理
  - 専用エディタで対象のビジネスロジッククラスを選択



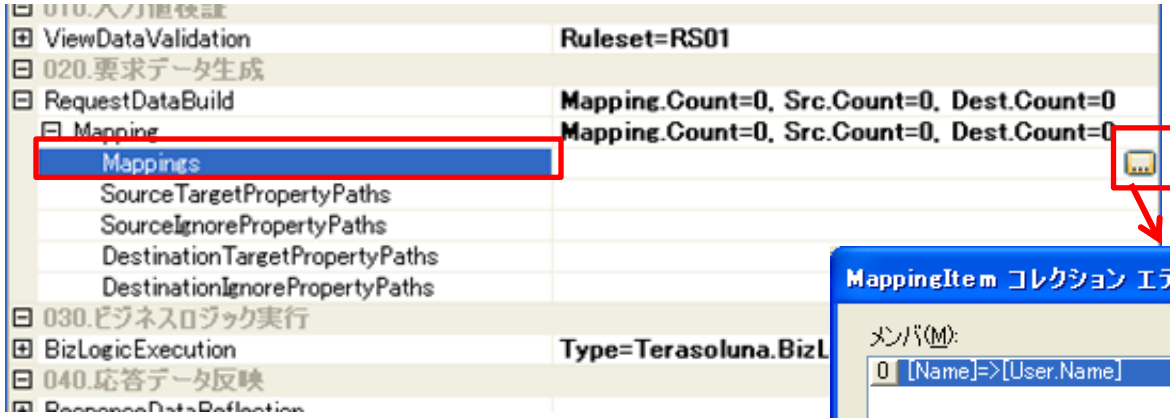


イベント処理実行機能の検討経緯(データコピー)

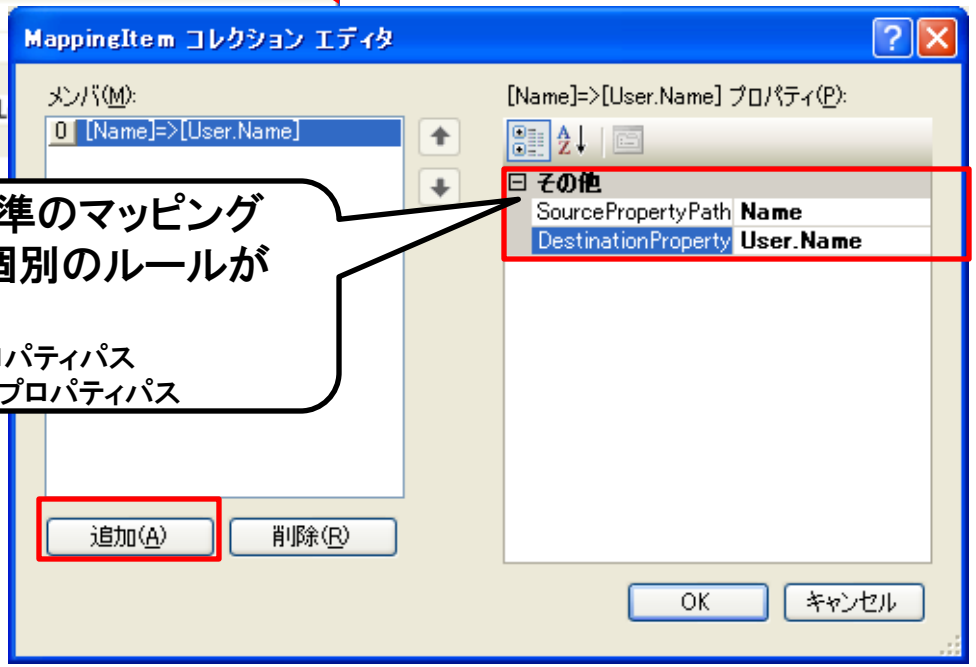
- 「画面データ」と入出力DTOのデータコピー処理
  - 前述の「画面データ機能」を参照
  - 「データコピー機能」により、階層構造とプロパティ名が一致すれば自動コピー
- Visual Studioのエディタ機能を拡張したプロパティ入力支援の提供
  - デフォルトのルールで自動コピーされないものは、個別のマッピングルールを設定可能
    - RequestDataBuildプロパティ
    - ResponseDataReflectionプロパティ
  - 専用エディタを提供しマッピング定義の入力を支援
    - 従来のXMLベースの設定ファイルは排除

データコピー処理のマッピング設定イメージ

EventProcessWorkerのプロパティエディタ(RequestDataBuildプロパティ)

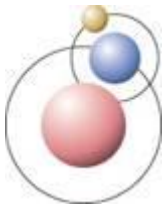


クリックすると
コレクションエディタが起動



プロパティ名について、標準のマッピングルールに当てはまらない個別のルールがあれば定義

SourcePropertyPath:コピー元のプロパティパス
DestinationPropertyPath:コピー先のプロパティパス

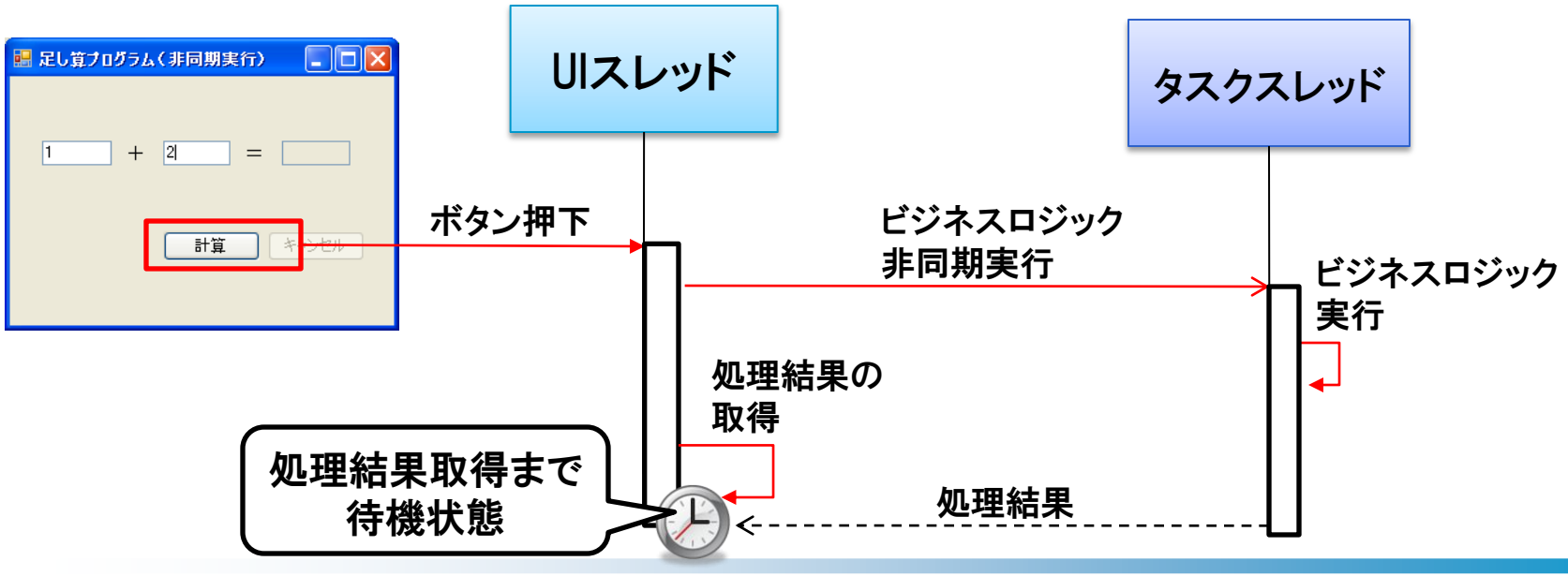


イベント処理実行機能の検討経緯（非同期実行）

- 同期実行時の問題
  - 業務処理に時間がかかる場合に画面がフリーズ
    - 画面を移動できない
    - プログレスバーの進捗率が進まない
  - ユーザビリティを考慮すると業務処理の非同期実行が必要
    - 同期処理では、メインスレッド（UIスレッド）がビジネスロジックを実行するため、イベント処理完了まで画面の描画処理を実行できない
    - タスクスレッド（非UIスレッド）にビジネスロジックを非同期実行させることで、すぐにUIスレッドに制御を戻すことができる

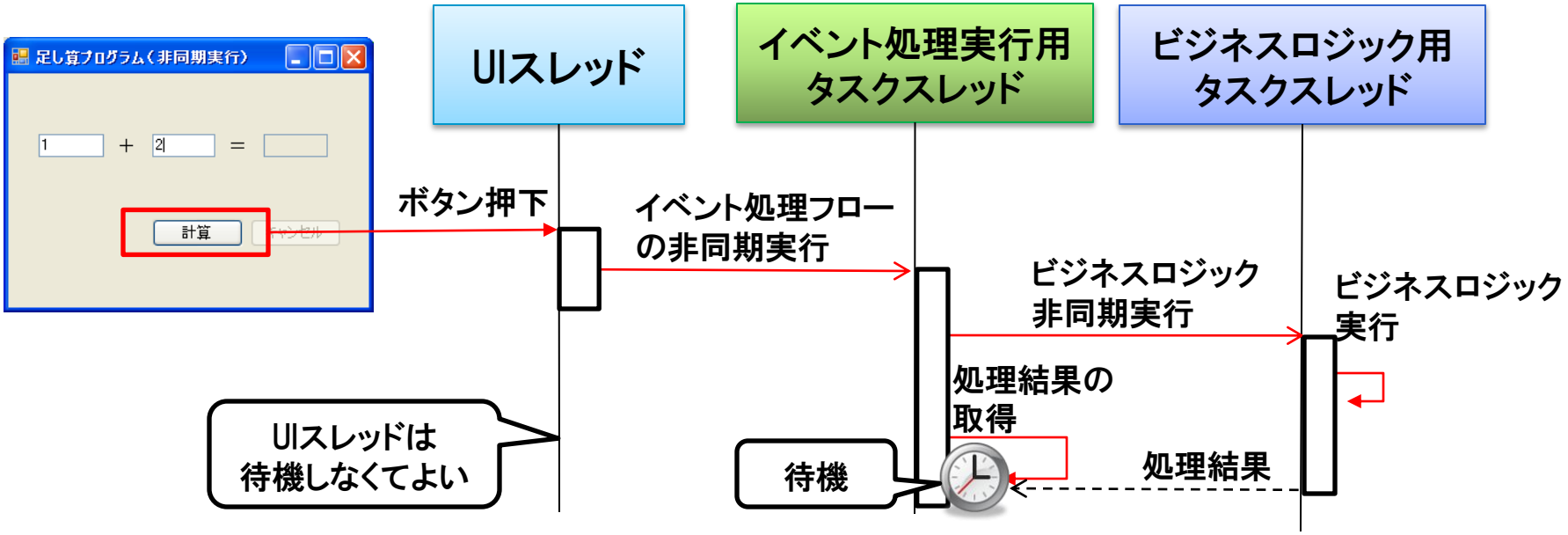
イベント処理実行機能の検討経緯（非同期実行）

- ビジネスロジック処理結果取得時の待機の問題
  - 処理結果を取得するには、タスクスレッドの非同期処理が終わるまで待機する必要がある
  - UIスレッドが直接処理結果を取得しようとすると、処理終了まで待機状態に入ってしまい、画面フリーズになる



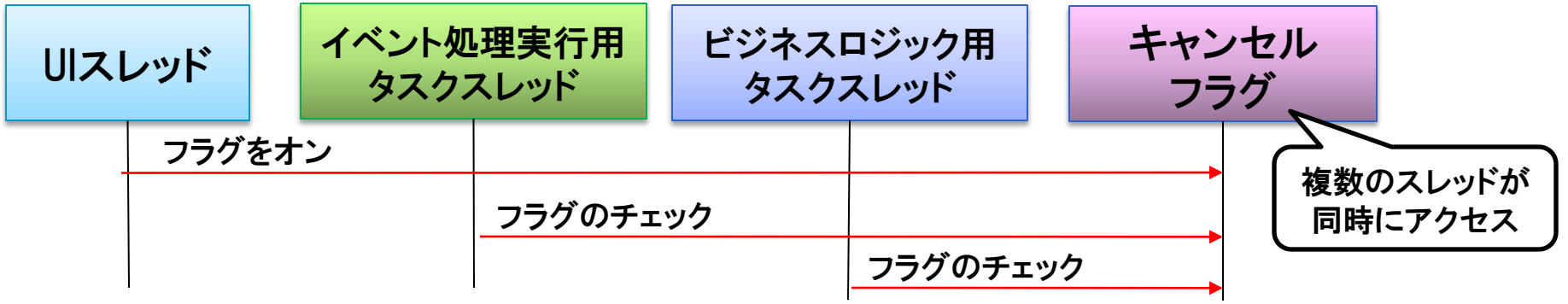
イベント処理実行機能の検討経緯（非同期実行）

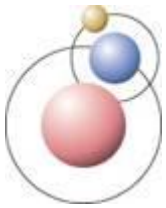
- 画面フリーズの防止策
  - 2つのタスクスレッドを使用
    - イベント処理実行用タスクスレッドが、ビジネスロジックの処理結果を取得するまで待機
    - UIスレッドは待機しなくて済むのでフリーズしない



イベント処理実行機能の検討経緯（非同期実行）

- キャンセル処理の実装の問題
  - 画面フリーズの問題
    - 画面がフリーズすると、ユーザはキャンセルボタンを押下できない
    - 前述の2つのタスクスレッドを使用する方式により解決
  - キャンセルフラグを使ったマルチスレッド制御
    - 通常、ロジック内でキャンセルフラグを定期的にチェックし、非同期処理を終了させるなどの個別実装が必要
    - キャンセルフラグは、複数のスレッドがアクセスするため排他制御の実装が必要



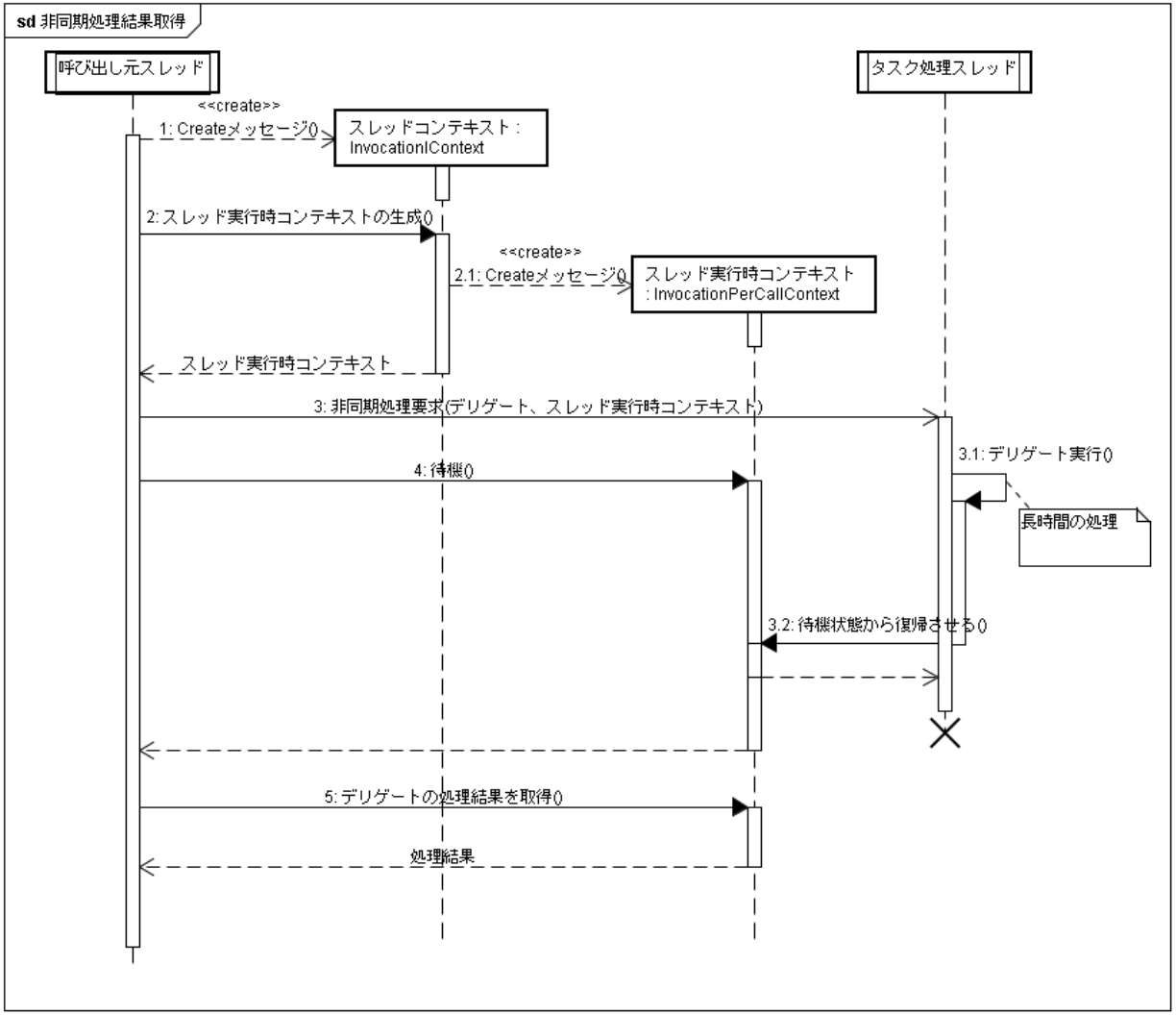


イベント処理実行機能の検討経緯(非同期実行)

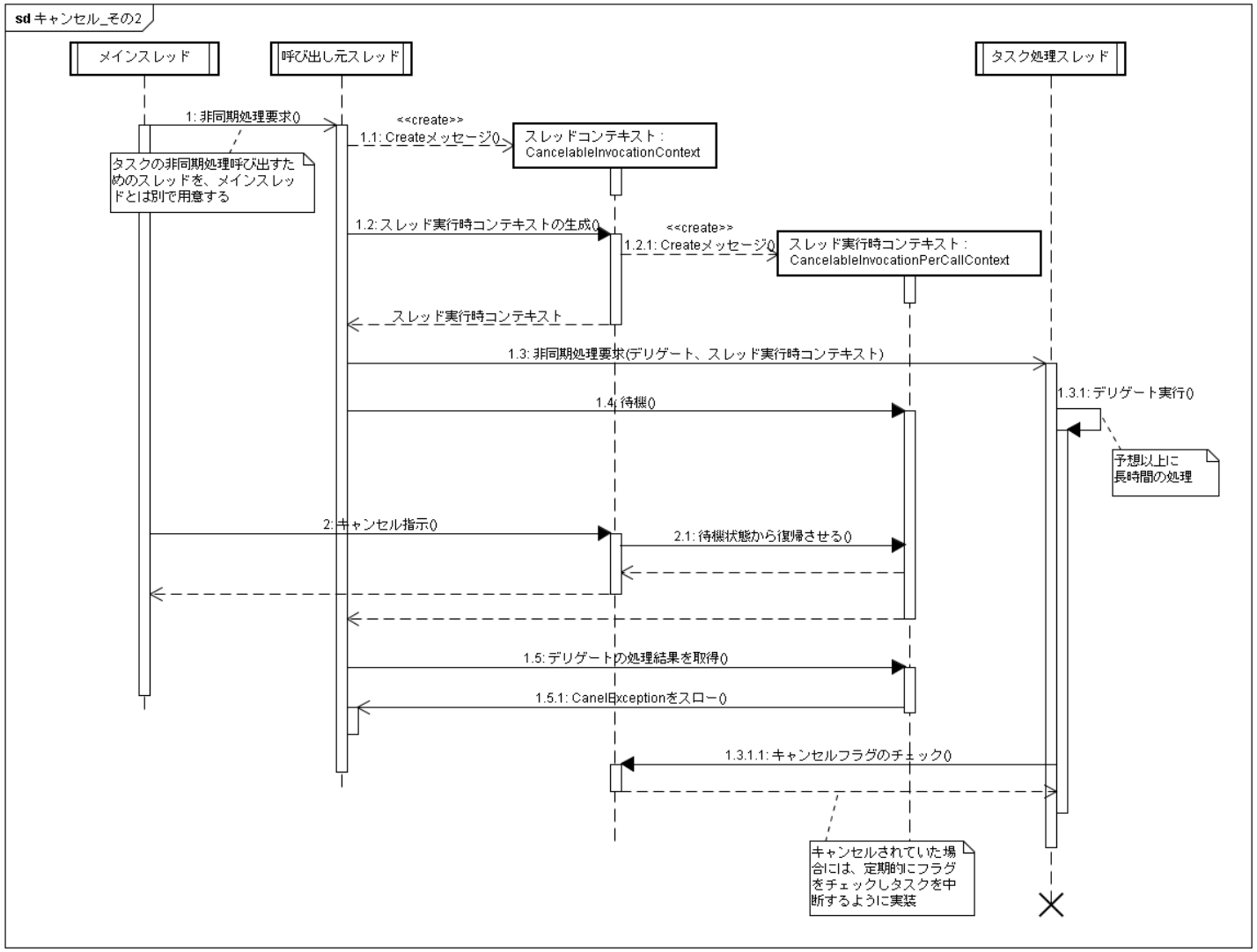
- 「スレッド制御機能」
  - イベント処理実行機能の基盤となる全体共通機能
  - 複雑なマルチスレッド制御を隠ぺいし、簡易に非同期処理を実装可能にするための基盤機能を提供
    - 処理結果取得時の待機やキャンセルの仕組みを提供
  - 「コンテキスト」の概念を導入
    - スレッドコンテキスト(InvocationContext)
      - スレッド実行のために必要な制御情報を格納
    - スレッド実行時コンテキスト(InvocationPerCallContext)
      - スレッドコンテキストをひな型として、スレッド実行ごとに作成
      - スレッド間でスレッド実行コンテキストを介してデータをやり取り
        - » 複雑なマルチスレッド制御を隠ぺい化
        - » 処理結果やキャンセルフラグなど実行時の情報へ安全にアクセス可

TERASOLUNA

コンテキストによる実行結果の取得制御

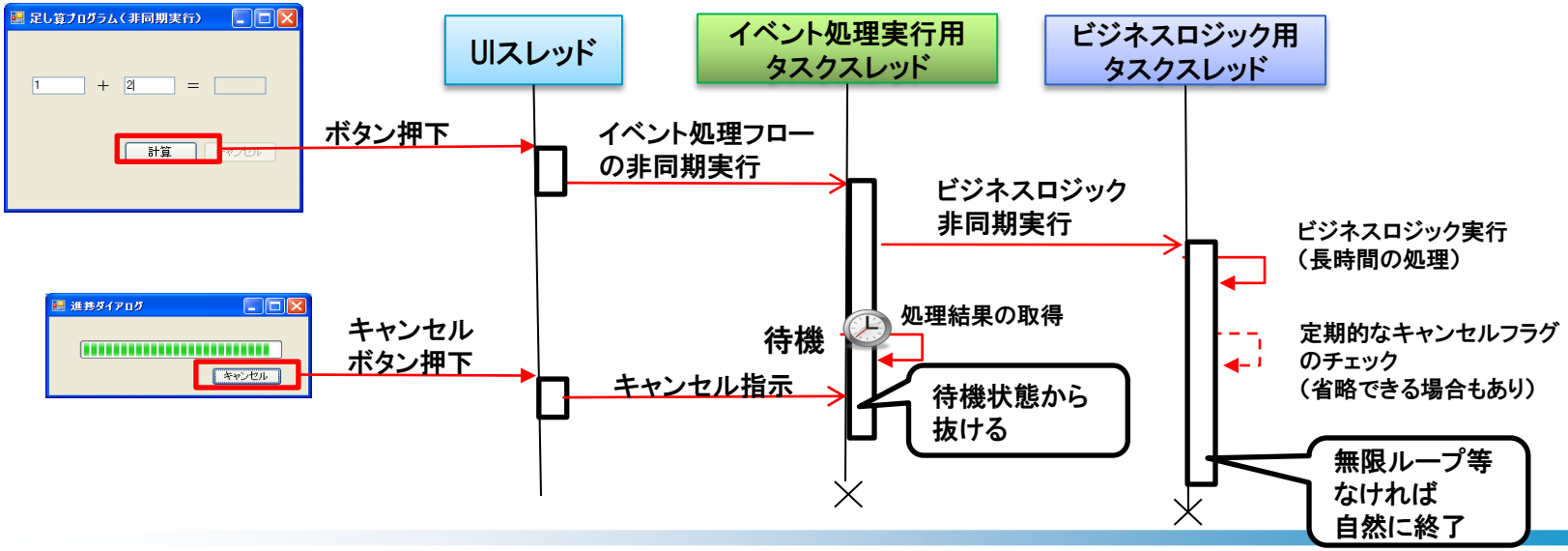


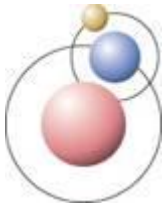
コンテキストによるキャンセルフラグ制御



イベント処理実行機能の検討経緯（非同期実行）

- ビジネスロジックを実行するスレッドの異常時でも、イベント処理の実行を安全に終了できる
  - 2つのタスクスレッドを使用することで、ビジネスロジック用タスクスレッドの実行状態に関わらずイベント処理実行用のタスクスレッドを終了できる
  - ビジネスロジック内でのスレッド停止処理は原則不要
    - もちろん、ロジック内で無限ループ処理がある場合や、マシン負荷が高い処理で即時の停止が必要な場合など、キャンセルフラグのチェックによるスレッド停止の実装が必要なケースもあるが、特別な実装は原則不要





ビジネスロジックでのキャンセルチェックの実装イメージ

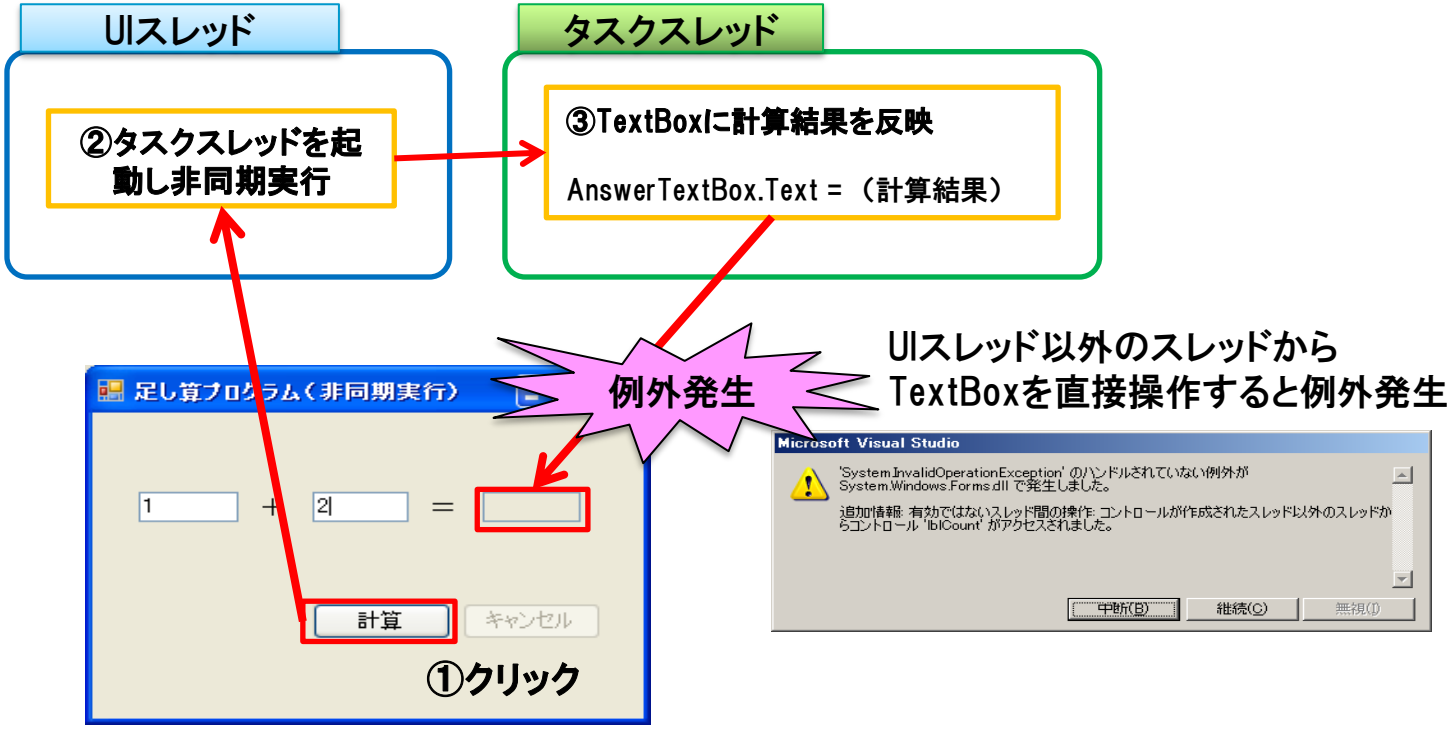
- 開発者によるキャンセルフラグのチェック
  - 定期的にビジネスロジック内でEventProcessContext.CancellaionPendingプロパティでキャンセル済みかチェック
  - 開発者がキャンセルフラグの排他制御を意識することはない

```
///キャンセル済かのチェック
if (InvocationScope.Current.GetContext<EventProcessContext>().CancellationPending)
{
   ///処理を中断し終了
  return;
}
```


TERASOLUNA

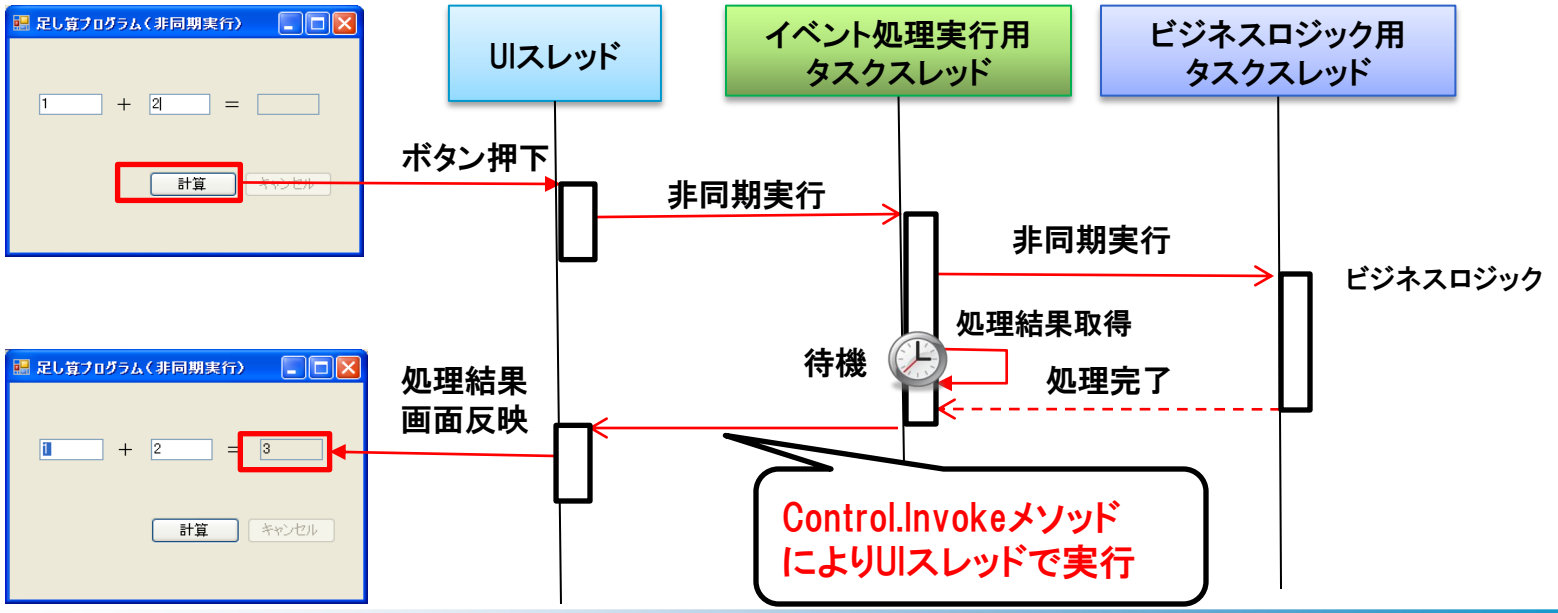
イベント処理実行機能の検討経緯（非同期実行）

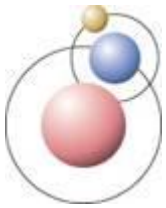
- タスクスレッド処理からの画面描画処理
  - Windowsアプリケーションでは、UIスレッド（メインスレッド）以外のスレッド（タスクスレッド）からUIコントロールを直接操作してはならない



イベント処理実行機能の検討経緯(非同期実行)

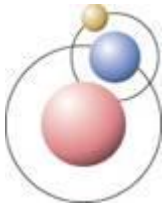
- タスクスレッド処理からの画面描画処理
  - 「スレッド制御機能」により実現
    - System.Windows.Controlクラスの Invokeメソッドを利用
    - フレームワークが、UIコントロールの操作を実装したメソッド(デリゲート)を自動的にUIスレッド上で実行





イベント処理実行機能の検討経緯（非同期実行）

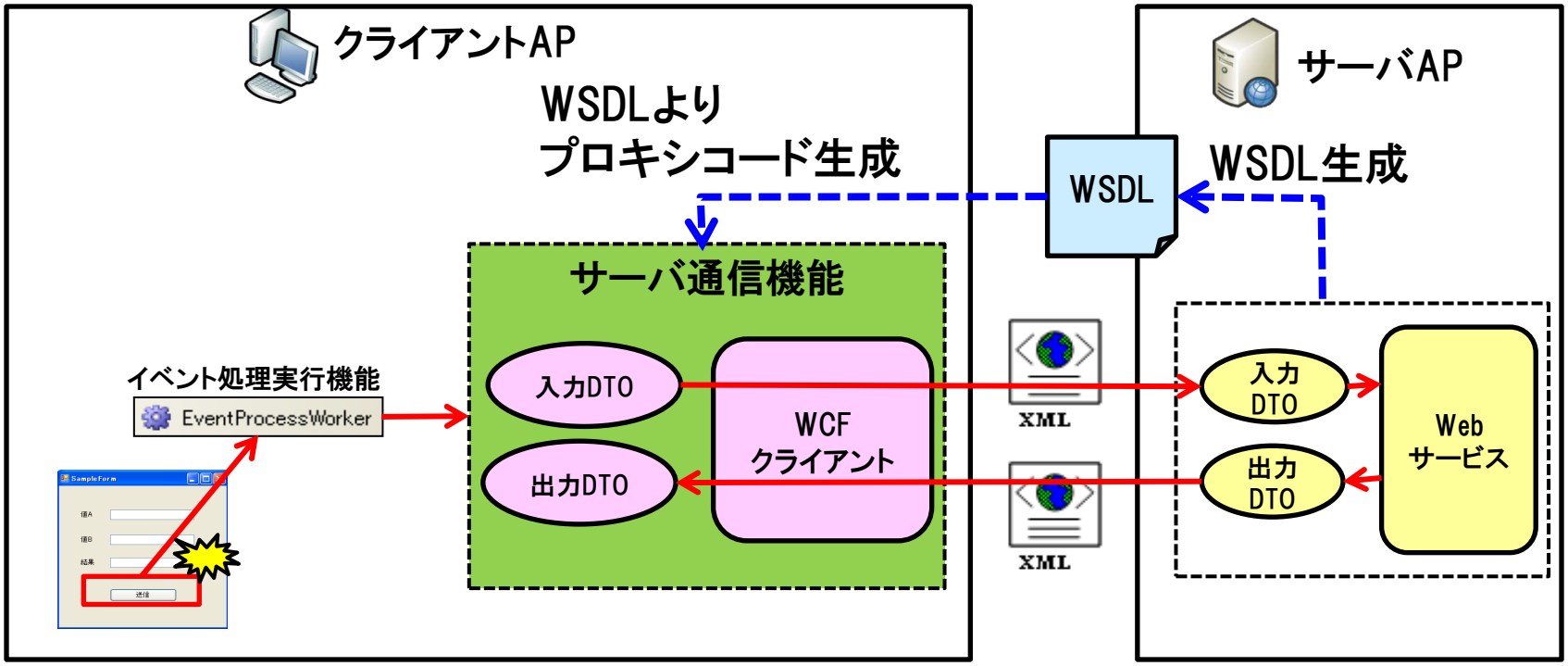
- タスクスレッド処理からの画面描画処理
  - 各スレッドが実行するコードがきれいに分離されるためコードの見通しがよく保守しやすい
    - UIスレッドが実行する処理
      - 画面のイベントハンドラメソッドで実装
    - タスクスレッドが実行する非同期処理
      - ビジネスロジッククラスで実装

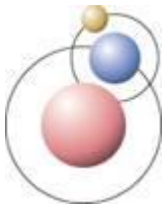


サーバ通信機能

サーバ通信機能の責務

- クライアントからのサーバ通信機能を提供
  - WSDLより自動生成したWCFクライアントを利用
    - System.ServiceModel.ClientBase継承クラス



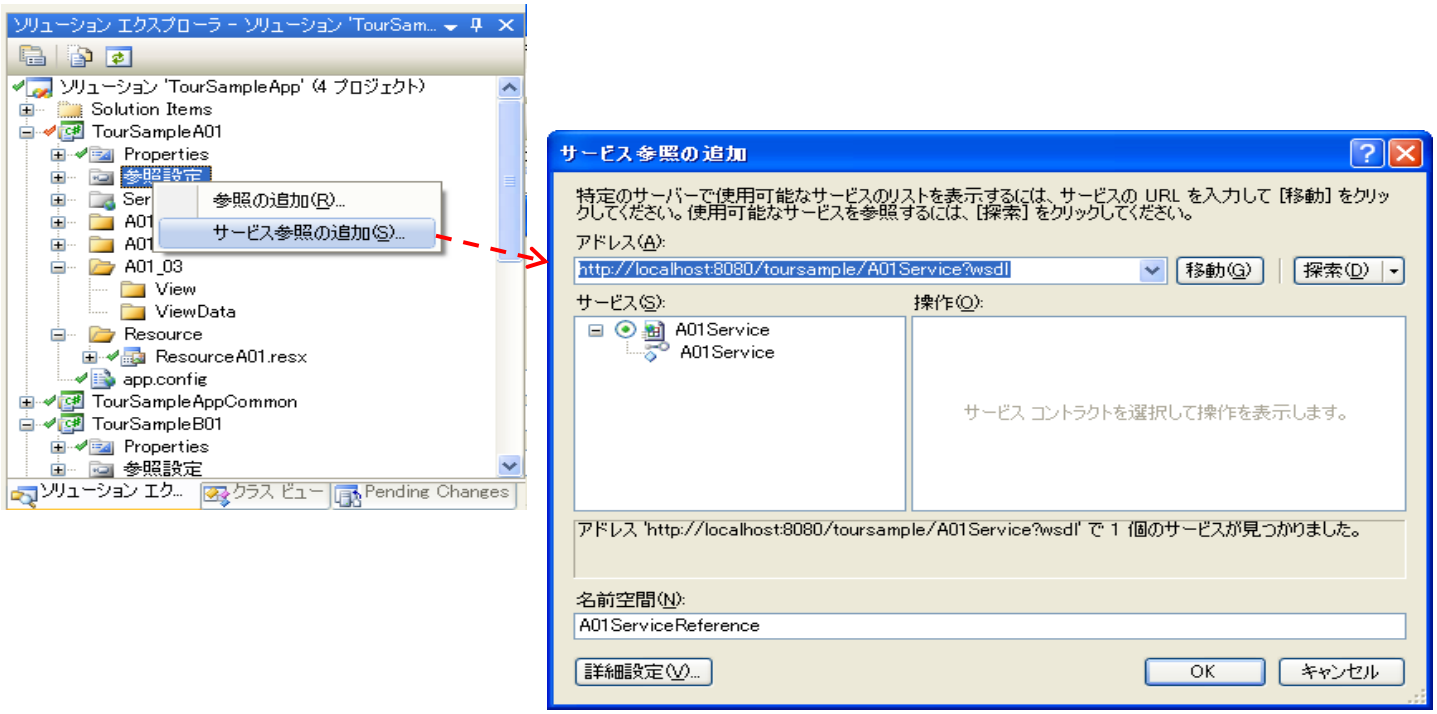


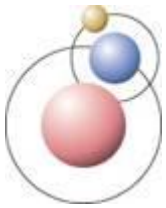
サーバ通信機能の検討経緯(WCFの採用)

- 相互運用性の考慮
  - TERASOLUNAフレームワーク独自の通信基盤からの脱却
    - 前バージョン(ver2.x)では、HTTPをベースとした独自の通信方式であったため、TERASOLUNAフレームワークを使ったAP間でしか接続できなかった
  - WCFの採用
    - 標準的なWCFサービスに対応し言語やプラットフォームに依存せず接続可能
    - システムの接続機会を増やし新たなビジネス機会を創造できる
    - システム開発における柔軟性が高く、少ないコストでシステムの統合が図れる
- 実装スタイルの統一
  - WCFにより通信プロトコルに依存しない、同一の実装スタイルになる
  - システムの非機能要件に合わせて通信プロトコルや通信方式を変更可能
    - 原則、アプリケーション構成ファイル(App.config)の設定のみで実現(例外あり)
    - 同一サービスに対して、複数のエンドポイントを用意することも可能
      - イントラの.NETアプリ向けにnetTcpBinding、外部システム向けにはbasicHttpBindingといった複数のチャネルを提供可能

WCFクライアントの自動生成

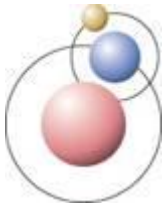
- Visual Studioの「サービス参照の追加」でプロキシコードを生成可能
  - WSDLを参照できればノンコーディングで作成できる





WCFクライアントの実行

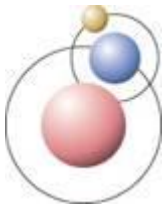
- 「イベント処理実行機能」により実行
  - 「ビジネスロジック実行」フェーズで実行可能
    - クライアント処理を伴わないサーバ通信処理
      - EventProcessWorkerで設定したWCFクライアントを呼び出す
      - 開発者は、プロパティの設定作業のみ
    - クライアント処理を伴うサーバ通信処理
      - EventProcessWorkerで設定したクライアントビジネスロジックからWCFクライアントを呼び出す
      - 開発者は、クライアントビジネスロジック内で、WCFクライアントを呼び出す処理を実装
  - 前述の「イベント処理実行機能」を参照



画面遷移機能

画面遷移機能の責務①

- 画面間の依存関係の疎結合化
  - 画面を一意に識別する文字列(画面ID)により遷移先画面を特定
  - 製造時は遷移先(遷移元)の画面クラスの参照が不要
    - 分散開発に対応
      - クラスやアセンブリの依存関係がないため、個々の開発者(クラス単位)や請負業者(アセンブリ単位)の作業を容易に
    - テスタビリティの向上
      - 製造時に、本番環境用の遷移先画面が完成していなくてもモック画面に切り替えることで試験可能

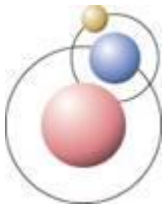


画面遷移機能の責務②

- 画面遷移方式のパターン化
  - 業務APIにおける典型的な遷移パターンを実装
    - 画面(Form)の切り替えによる遷移
      - FormForwarderクラスを提供し、遷移パターンに依存せず同じ実装スタイルで記述可能
        - ShowModeless
        - ShowAsChild
        - ShowWindowModal
        - ShowApplicationModal
        - ShowAndHide
      - 「スコープ」の概念を導入し、各画面の表示状態を管理
    - パネル(Control)の切り替えによる遷移
      - ContentPlaceHolderクラスを提供
      - ContentPlaceHolderが各パネルの表示状態を管理

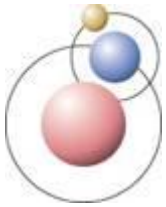
画面遷移機能の責務③

- 画面間のデータ引き継ぎ機能の提供
  - FormFowarderによる遷移
    - 「スコープ」内の共通データ領域
      - 一連の業務(ユースケース)等単位でデータを持ちまわる
      - 同一画面を複数起動しても別のワークスペースとして管理
      - スコープの消滅とともに自動的に破棄
    - 遷移元画面と遷移先画面間での「画面データ」のコピー
      - 遷移元画面、遷移先画面間でのみデータを持ちまわる
      - 「データコピー機能」によるコピー処理を自動実行



画面遷移機能の責務③

- 画面間のデータ引き継ぎ機能の提供
  - ContentPlaceHolderによる遷移
    - 各パネルが保持する画面データを管理
      - ContentPlaceHolderがパネルインスタンスを管理
        - » 遷移時に、パネルの破棄を実行することで画面データも自動的に破棄
        - » 管理中のパネルが保持する画面データは任意の場所で取得可能
      - FormFormwarderの「スコープ」と同等の機能が実現可能
    - 遷移元、遷移先パネル間での「画面データ」のコピー
      - 遷移元パネル、遷移先パネル間でのみデータを持ちまわる
      - 「データコピー機能」によるコピー処理を開発者が実装



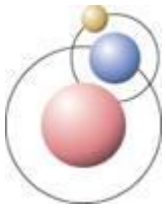
画面遷移機能の検討経緯(画面間の疎結合化)

- 画面間の依存関係を疎結合にする
  - 遷移先画面を画面IDで指定する
  - デフォルトでは画面IDはカスタム属性で設定
    - クラス命名規約などのCoCルールで、属性定義を省略することも可能

画面クラスに対するカスタム属性で定義

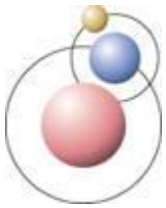
```
[ScreenId("SC_B01_01_01")]
public partial class SC_B01_01_01View：Form
{
…
}
```


TERASOLUNA



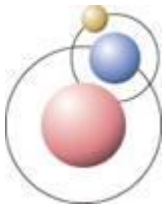
画面遷移機能の検討経緯(画面間の疎結合化)

- 画面IDをカスタム属性で定義する際の課題と対策
  - 全ての画面のScreenIdカスタム属性を取得するためには、必要なアセンブリを全てロードしておく必要がある
  - 大規模プロジェクトにおける分散開発の考慮
    - 業務を複数のソリューションやプロジェクトに分けて開発するためアプリケーションが必要とするDLLが複数に分割
    - 画面遷移時点で、まだロードされていないアセンブリ(DLL)内で属性定義されている画面IDを識別できない
  - アプリケーション起動時に全アセンブリをロードすることで、全画面のScreenId属性値を取得することで対処
    - パフォーマンスを考慮し、アプリケーション起動時に非同期でアセンブリをロード
    - 「アプリケーション起動・終了機能」で実施



画面遷移機能の検討経緯(画面遷移方式)

- 画面(Form)の切り替えによる遷移は、Formクラスのメソッドを組み合わせて実現
  - Form.Show() メソッド　(Control.Show()メソッド)
    - 親画面(owner)を持たない
    - モードレスで表示
  - Form.Show(IWin32Window)メソッド
    - 親画面(owner)を持つ
    - モードレスで表示
  - Form.ShowDialog(IWin32Window)メソッド
    - 親画面(owner)を持つ
    - モーダルダイアログで表示
  - Form.Hide()メソッド　(Control.Hide()メソッド)
    - 非表示(インスタンス破棄しない)
  - Form.Closeメソッド
    - 閉じる(インスタンスを破棄)



画面遷移機能の検討経緯(画面遷移方式)

- FormForwarderにより典型的な遷移パターンを実装

画面遷移方式	説明	実現方法
ShowModeless	モードレスで表示。遷移元画面との関係はなし。	・遷移先画面をForm.Show()
ShowAsChild	遷移元画面を親画面としてモードレスで表示。	・遷移先画面をForm.Show(IWin32Window)
ShowWindowModal	遷移元画面を親画面としてモーダルダイアログで表示。 親画面のみ操作不可。	・遷移先画面をForm.Show(IWin32Window) ・遷移元画面のEnableプロパティをfalse ・遷移先画面がForm.Close()またはForm.Hide()されたら、遷移元画面のEnableプロパティをtrue
ShowApplicationModal	モーダルダイアログで表示。画面表示中は、同じアプリケーション上の他の全ての画面が操作不可。	・遷移先画面をForm.ShowDialog(IWin32Window)
ShowAndHide	遷移先画面を表示(Show)し、現在の画面を非表示(Hide)。 現在の画面を非表示または閉じると遷移元画面を再表示。	・遷移先画面をForm.Show(IWin32Window) (ShowAsChildを使用) ・遷移元画面をForm.Hide() ・遷移先画面がForm.Close()またはForm.Hide()されたら、遷移元画面をForm.Show()

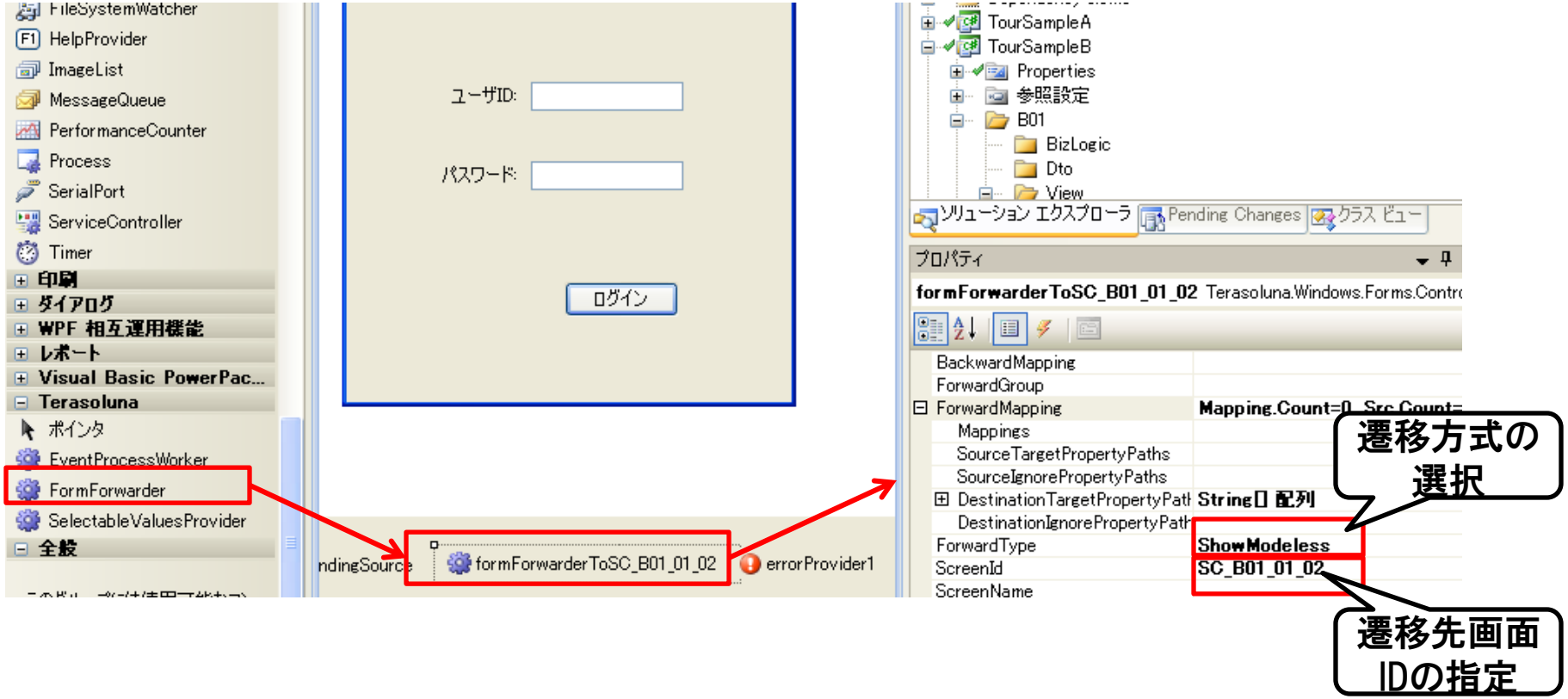
TERASOLUNA

FormForwarderの利用イメージ

- 画面コンポーネントによる開発スタイル
  - 業務開発者は画面部品として画面に貼り付けて開発
    - Componentクラスを継承した画面部品として提供
    - プロパティを設定することで定型的な動作を具体化
      - 遷移先画面ID、画面遷移方式、スコープ、遷移元画面と遷移先画面間の「画面データ」コピーの設定など
  - 業務開発者は、画面クラスのイベントハンドラメソッドで、細かい処理を実装可能
    - 遷移先画面のオープン時やクローズ時にイベント発生
- 画面遷移方式の種類に関わらず同じ実装スタイル
  - 遷移方式の種類によらず、同一メソッド(Forwardメソッド)により遷移処理を実行

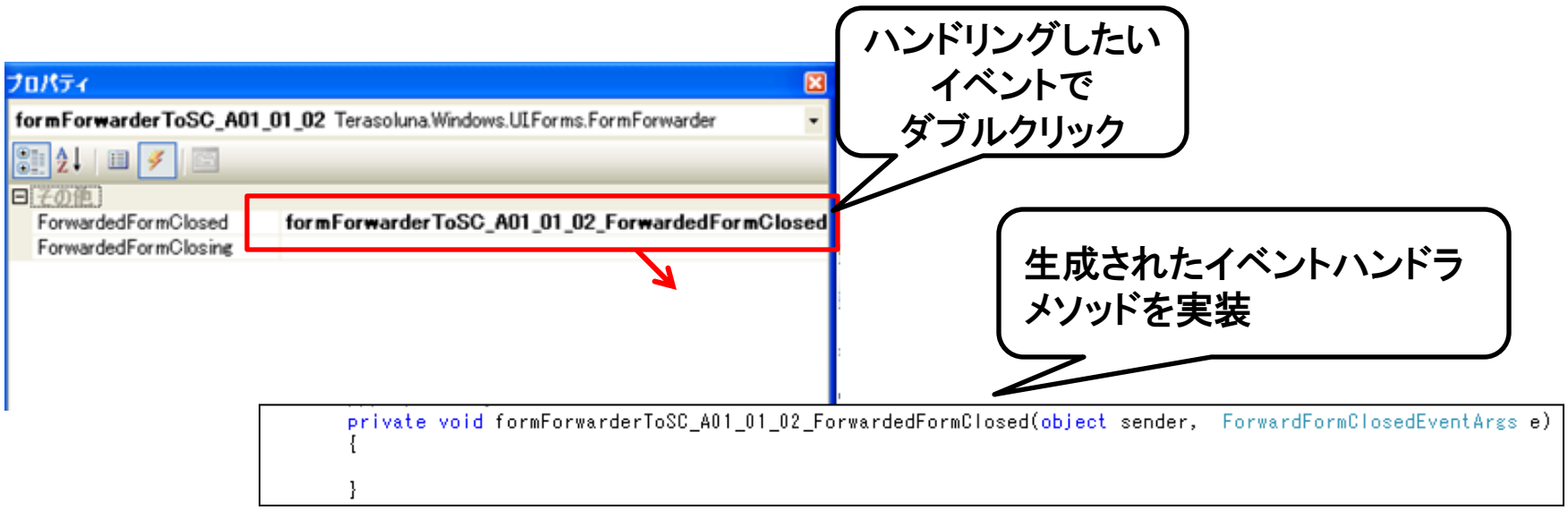
FormForwarderの利用イメージ

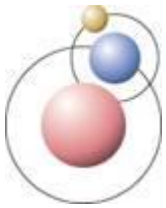
- FormForwarderを画面部品として貼り付け
  - プロパティを編集し、画面遷移処理を定義



FormForwarderの利用イメージ

- FormForwarderを画面部品として貼り付け
  - 画面遷移処理に伴うイベントを定義





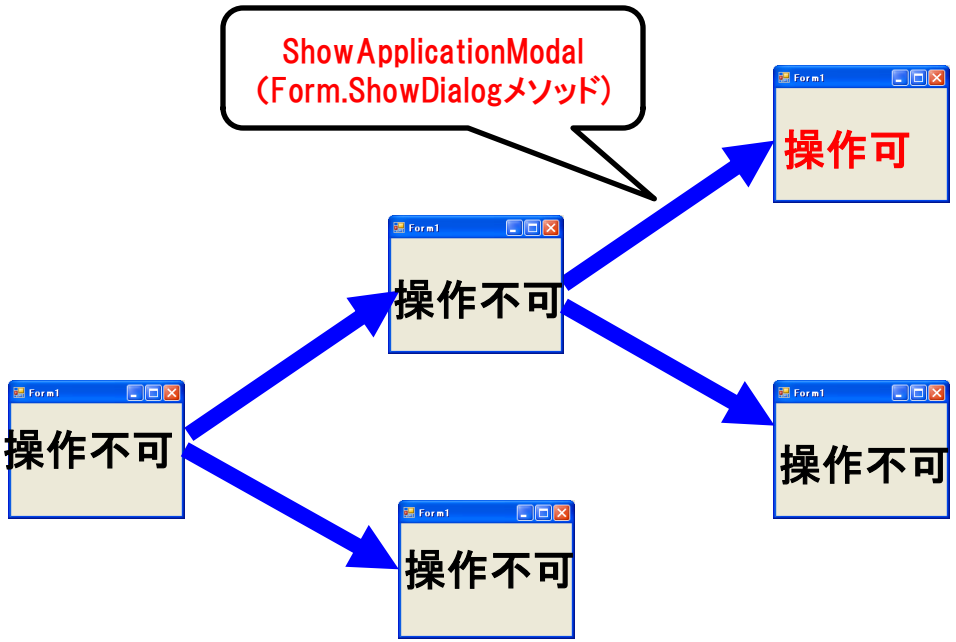
FormForwarderの利用イメージ

- FormForwarder.Forwardメソッドを実行
  - 画面遷移方式の種類に関わらず同じ実装スタイル

```
public partial class SC_B01_01_01View : Form
{
 ・・・
        private void loginButton_Click(object sender, EventArgs e)
        {
            formForwarderToSC_B01_01_02.Forward();
        }
}
```

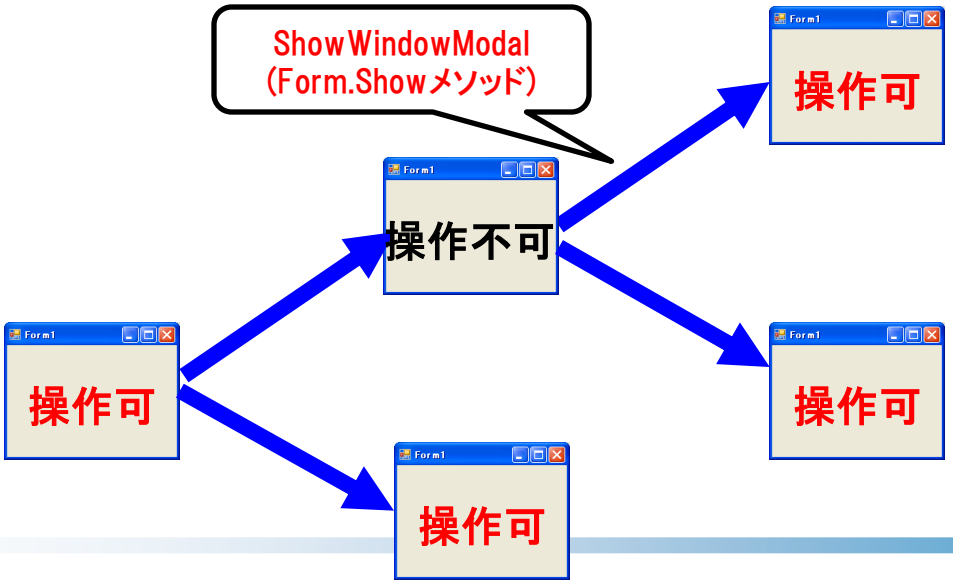
画面遷移機能の検討経緯（ShowWindowModal）

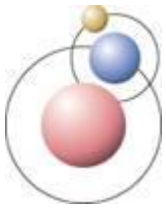
- .NETのデフォルトのモーダルダイアログはForm.ShowDialog()
  - TERASOLUNAフレームワークでは「ShowApplicationModal」として提供
  - モーダルダイアログ表示した画面以外はすべて操作不可
    - 遷移元画面の兄弟画面も親画面も操作できない



画面遷移機能の検討経緯(ShowWindowModal)

- ShowWindowModalの実現
  - 遷移元画面の兄弟画面は操作可能
    - モーダルダイアログを表示しつつ、別の業務画面を同時に操作することが可能
  - Show(IWin32Window)メソッドを使って、親画面に対するモーダルダイアログを擬似的に再現
    - モーダルダイアログ表示中は、親画面を無効状態(Enable=false)





画面遷移機能の検討経緯（ShowAndHide）

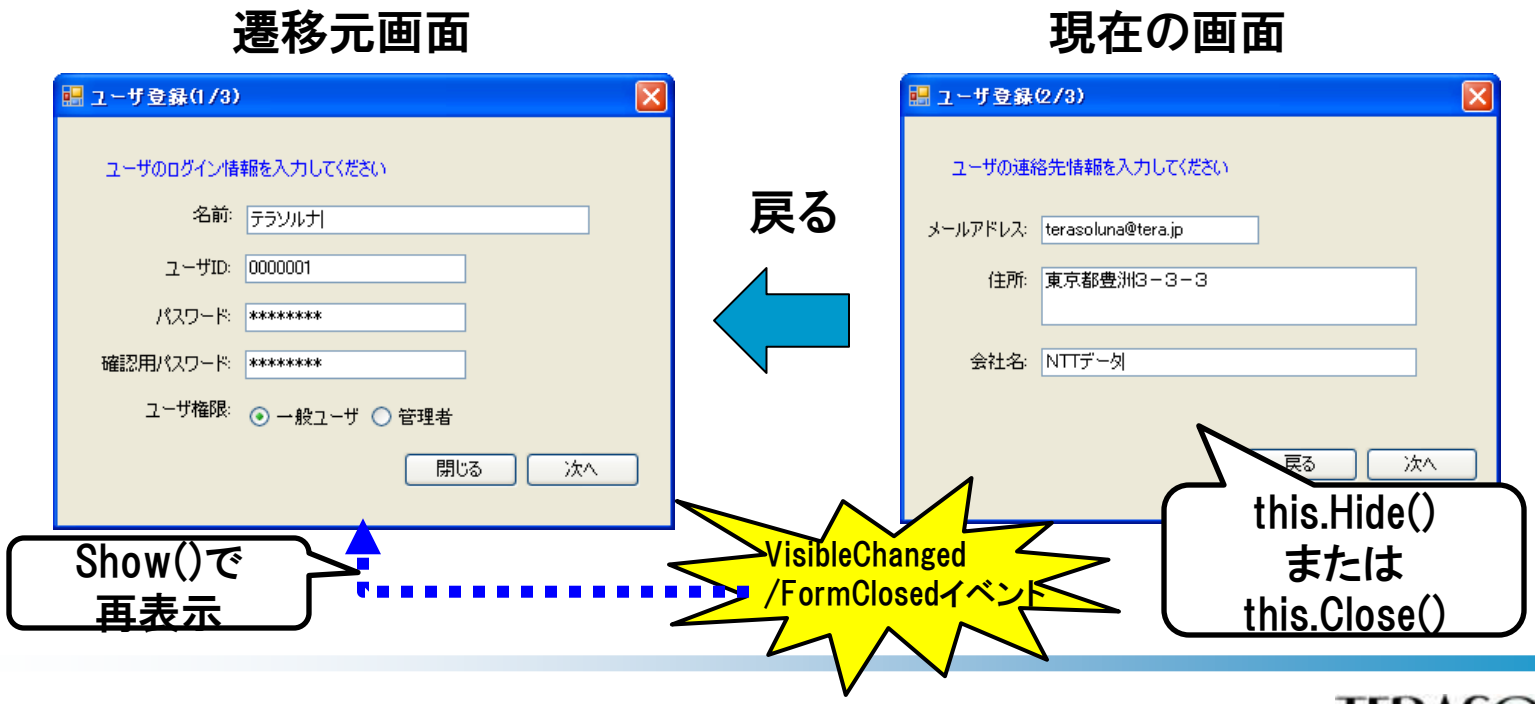
- ShowAndHideの実現
  - 業務の一連の流れに合わせて画面が切り替わる
    - 次の画面へ画面遷移すると、遷移元画面は消える(Form.Hide())
    - 遷移先画面が消えると遷移元画面が再表示される(Form.Show())
    - 一連の処理が終了すると、画面が自動的に破棄される
- ShowAndHideの利用例
  - 「メニュー画面」⇔「検索画面」⇔「更新画面」⇔「更新確認画面」⇔「メニュー画面」といった一連の業務（ユースケース）等単位ごとに画面遷移がパターン化されたAP
  - 「入力画面1」⇔「入力画面2」⇔・・・⇔「登録確認画面」といったウィザード形式の画面

画面遷移機能の検討経緯(ShowAndHide)

- 遷移先画面へ進む
  - Form.Show(IWin32Window)メソッドにより現在の画面を親として遷移先画面を表示
  - フレームワークが現在の画面を自動的にForm.Hide()

画面遷移機能の検討経緯（ShowAndHide）

- 遷移元画面へ戻る
  - 業務開発者は現在の画面をForm.Hide()またはForm.Close()するだけ
  - フレームワークにより、遷移元画面が自動的に再表示（Form.Show()）
    - 遷移先画面のVisibleChangedイベントやFormClosedイベントで、自動的に再表示（Form.Show()）するようフレームワークがイベントハンドラを登録



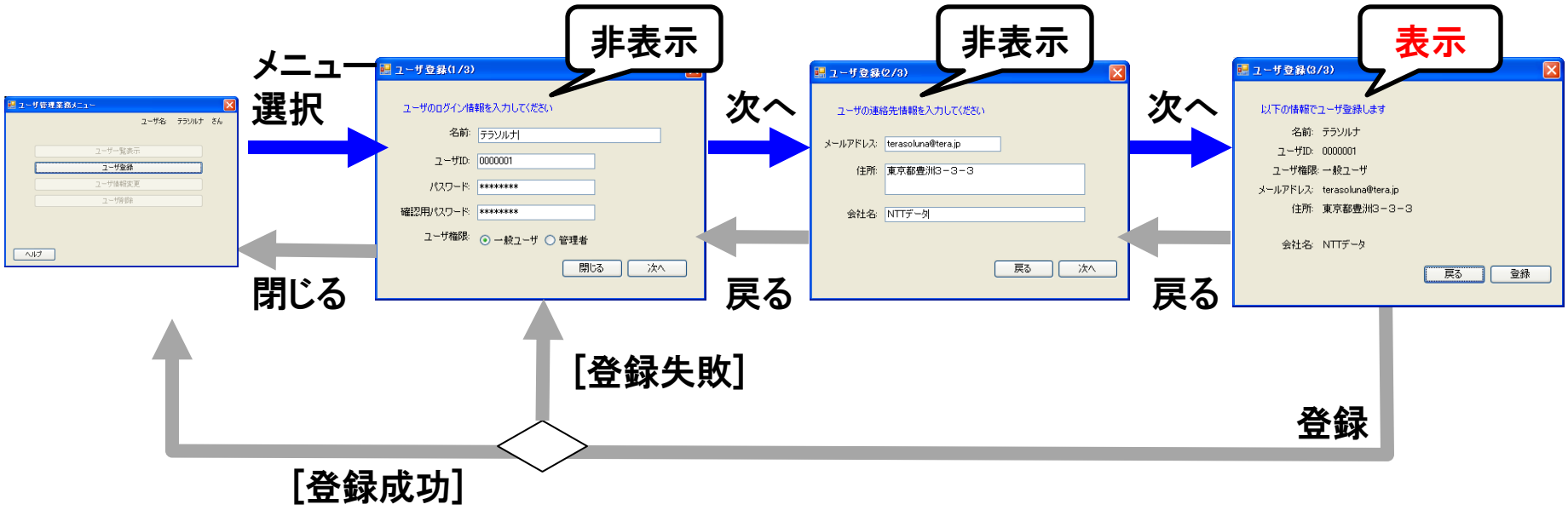
画面遷移機能の検討経緯（ShowAndHide）

- 再度、次の画面へ遷移
  - 遷移先画面をForm.Hide()して遷移元画面へ戻った場合、再度遷移先画面へ遷移すると、以前入力した内容が残っている
  - 遷移元画面にForm.Close()して戻った場合、再度遷移先画面へ遷移すると、画面のインスタンスは再作成されるので、未入力状態で表示される
- 前の画面へ戻るときにForm.Hide()するかForm.Close()するかは、開発者が業務要件によって使い分ける

Form.Hide()の場合は、Form.Show()のみ　Form.Close()の場合は、newしてForm.Show()

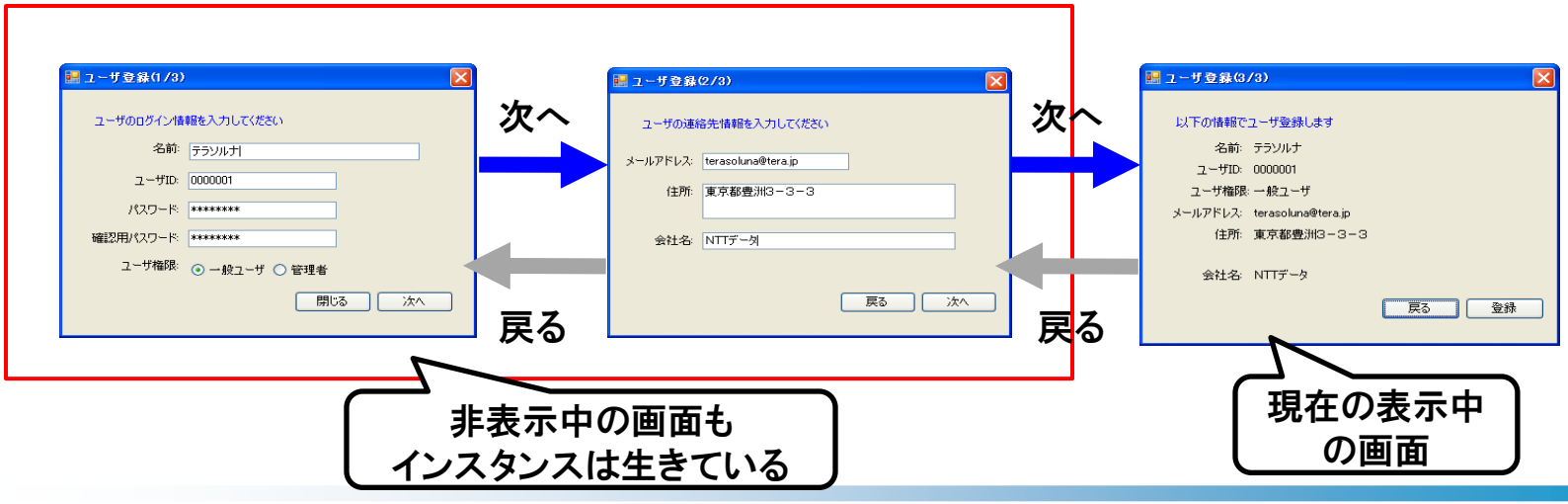
画面遷移機能の検討経緯(スコープ)

- 画面の表示状態の管理
  - ShowAndHide等の実現には一連の画面遷移に含まれる画面の表示状態を管理する必要がある



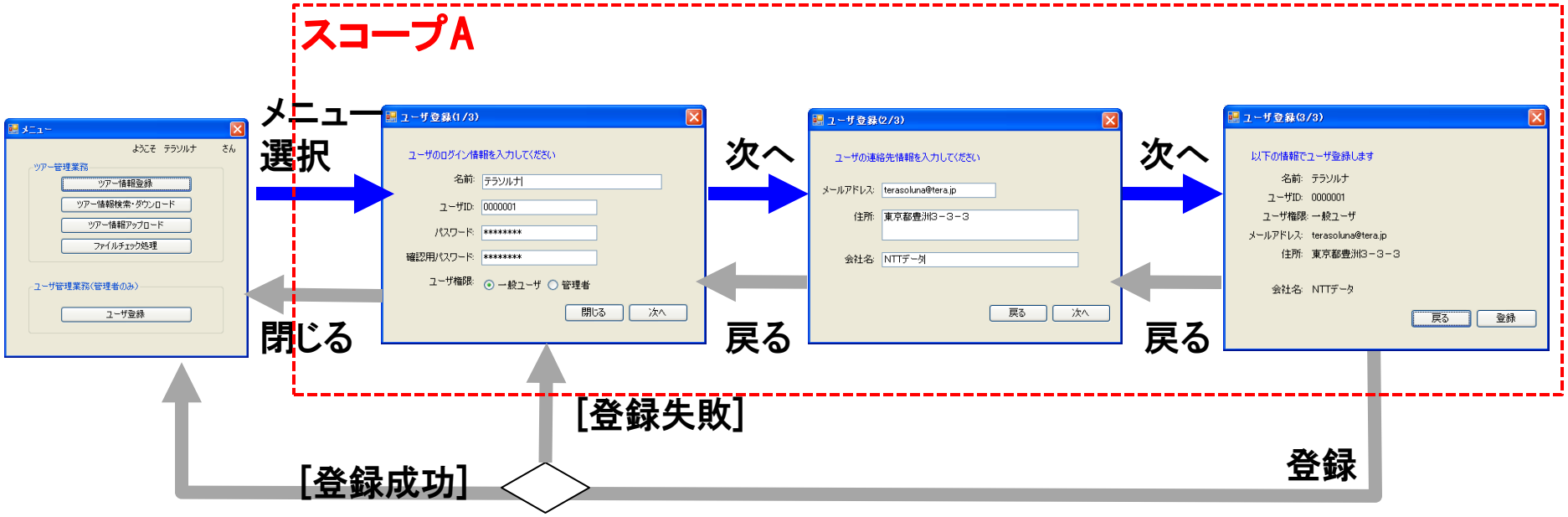
画面遷移機能の検討経緯(スコープ)

- 画面のインスタンス管理
  - 非表示(Form.Hide())を使った画面遷移では、現在の表示画面以外の画面も含めインスタンスは全て存在
  - 開発者は画面全てを明示的に破棄(Form.Close()/Form.Dispose())する必要がある
    - 開発者の実装漏れによるメモリリークの恐れあり



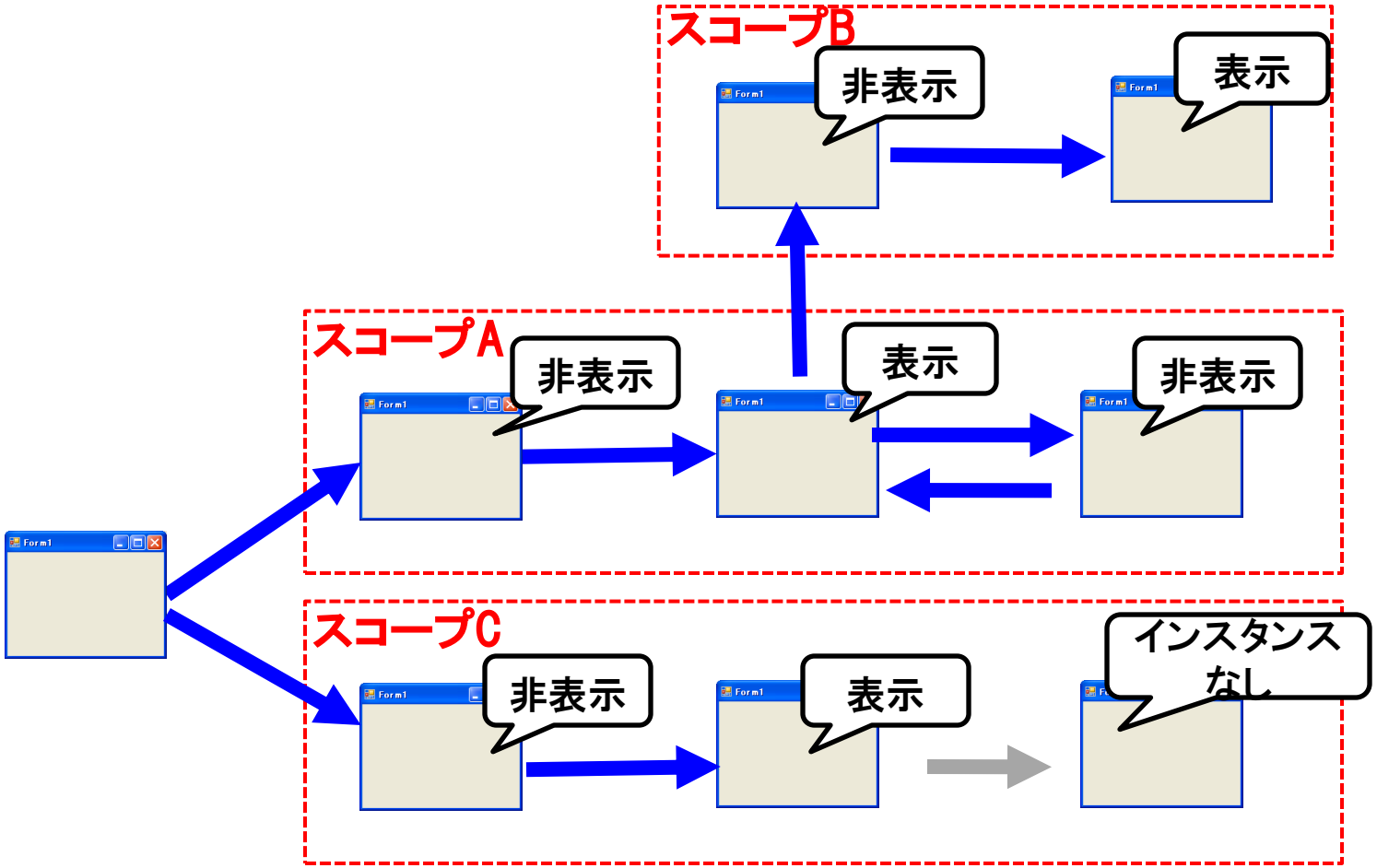
画面遷移機能の検討経緯(スコープ)

- 「スコープ」の導入
  - 業務的に意味のある一連の画面遷移の単位で、画面の表示状態や画面インスタンスの管理を実施
  - FormForwardGroupScopeクラスより実現



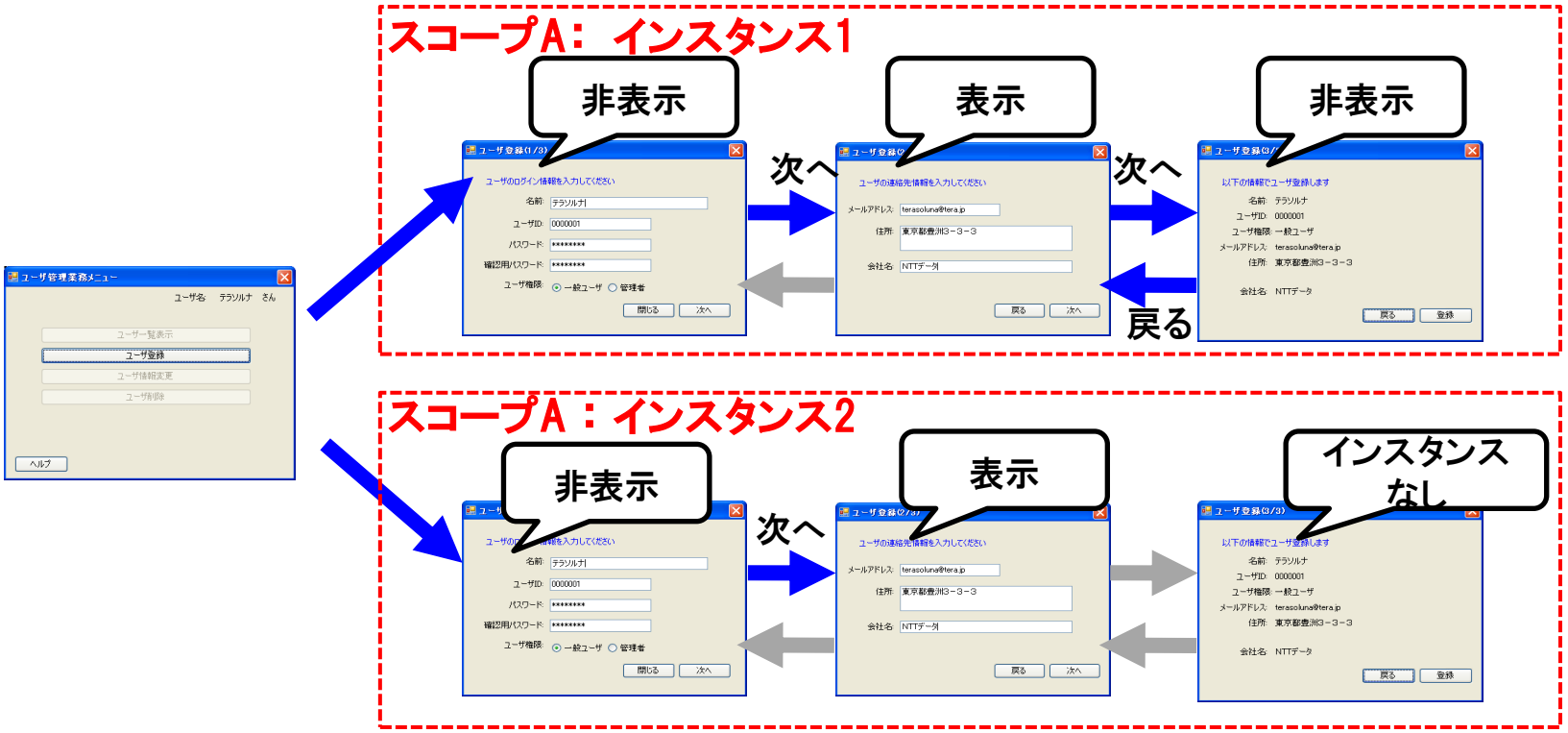
スコープの概念

- スコープごとに画面遷移状態を管理



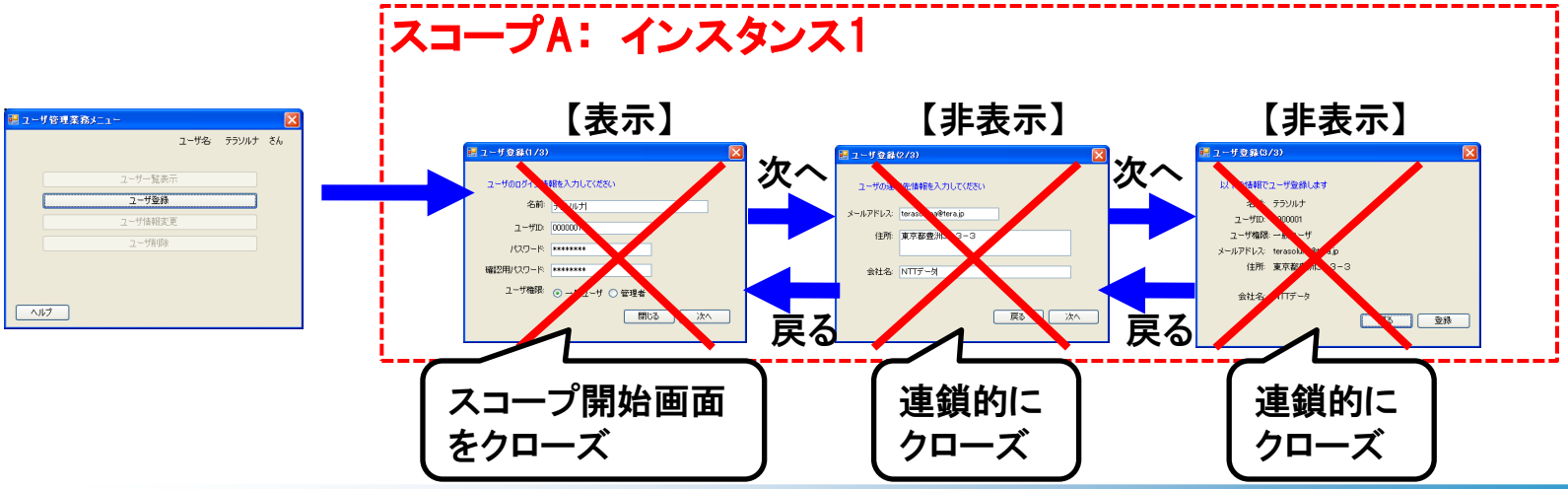
スコープの概念

- スコープ(FormForwardGroupScopeクラス)のインスタンスごとに遷移状態を管理
  - 同一の画面を複数起動しても、スコープごとに状態管理可能



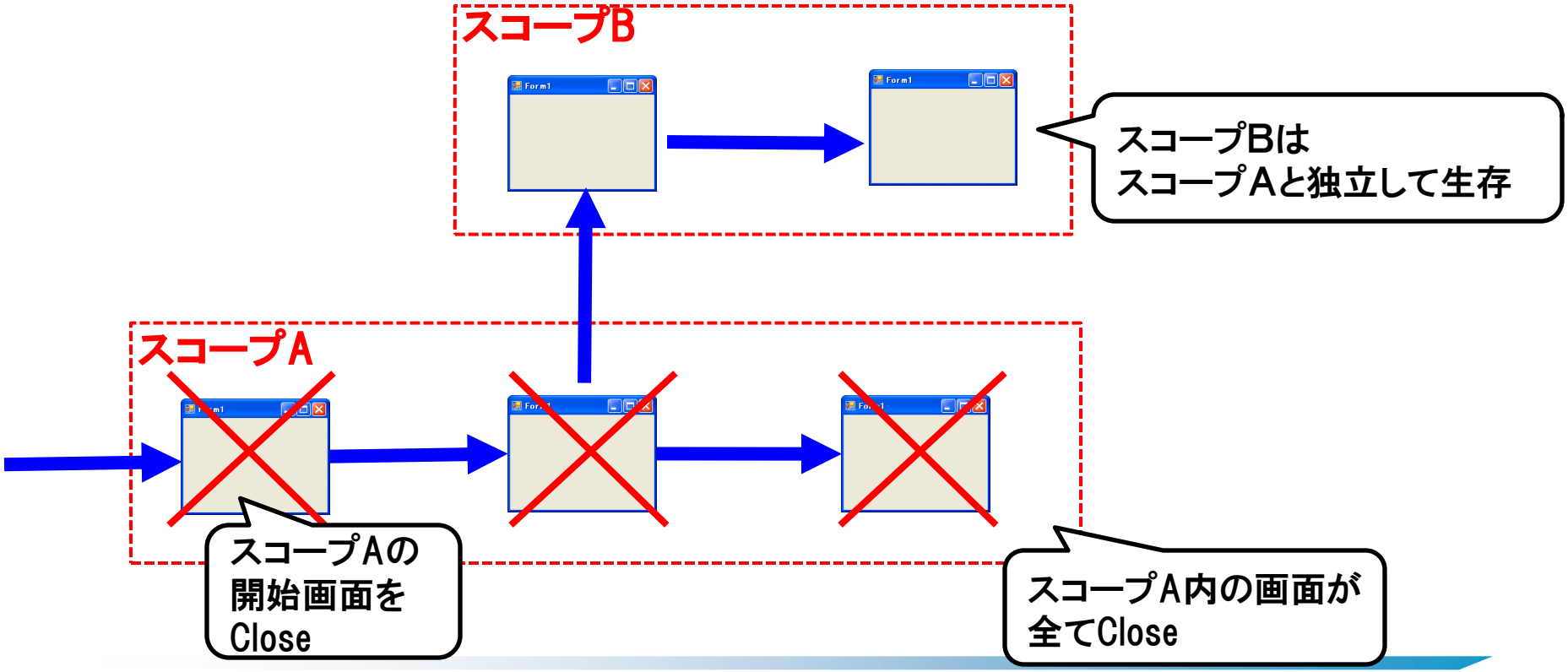
スコープの概念

- スコープを利用することで、スコープ消滅時にスコープ内の画面を全て自動的に破棄(Form.Close()/Form.Dispose())
  - 開発者の実装漏れによるメモリリークを防ぐ
- スコープ単位で画面を破棄できる
  - スコープを終了するか、スコープの開始画面をクローズすることで、スコープ内全画面が連鎖的にクローズ



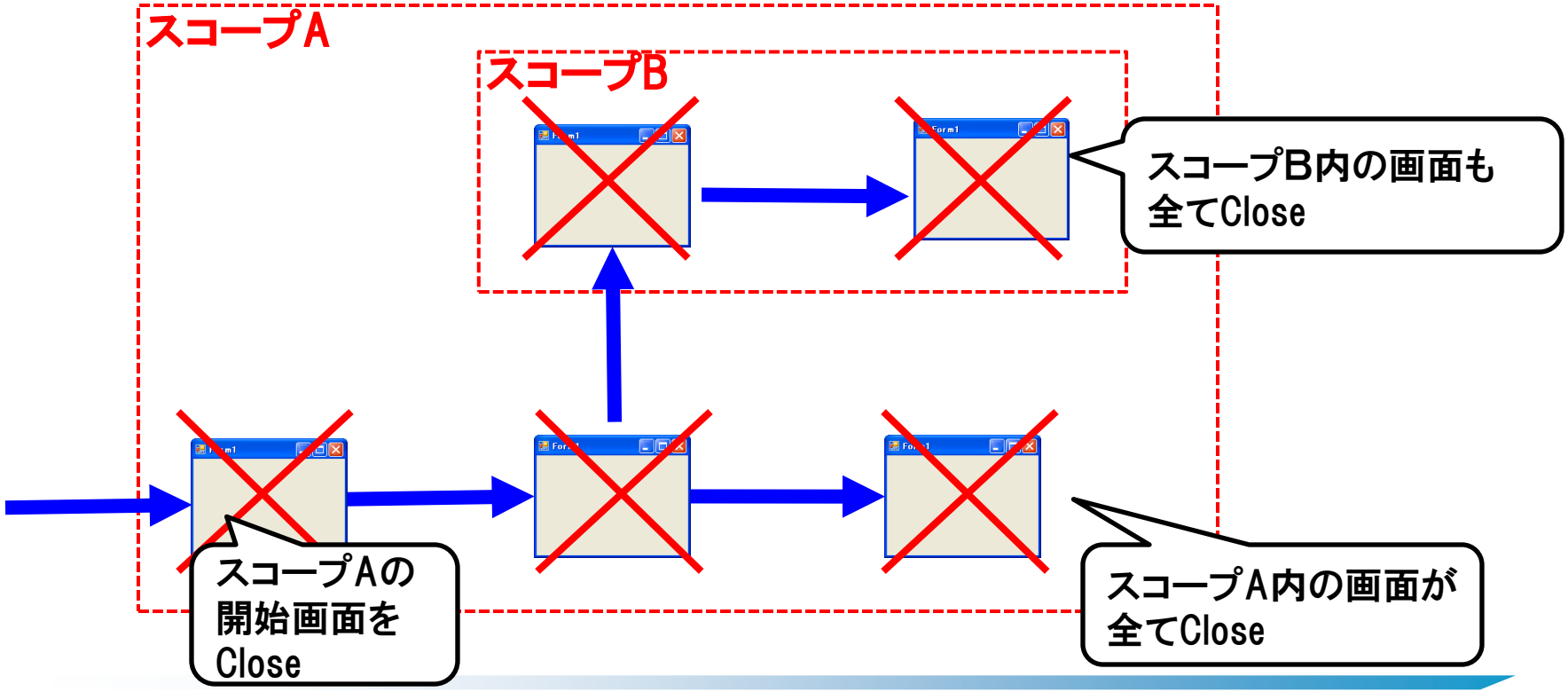
スコープの概念

- 異なるスコープは、独立して生存する
  - スコープAを全てCloseしても、スコープBに波及しない



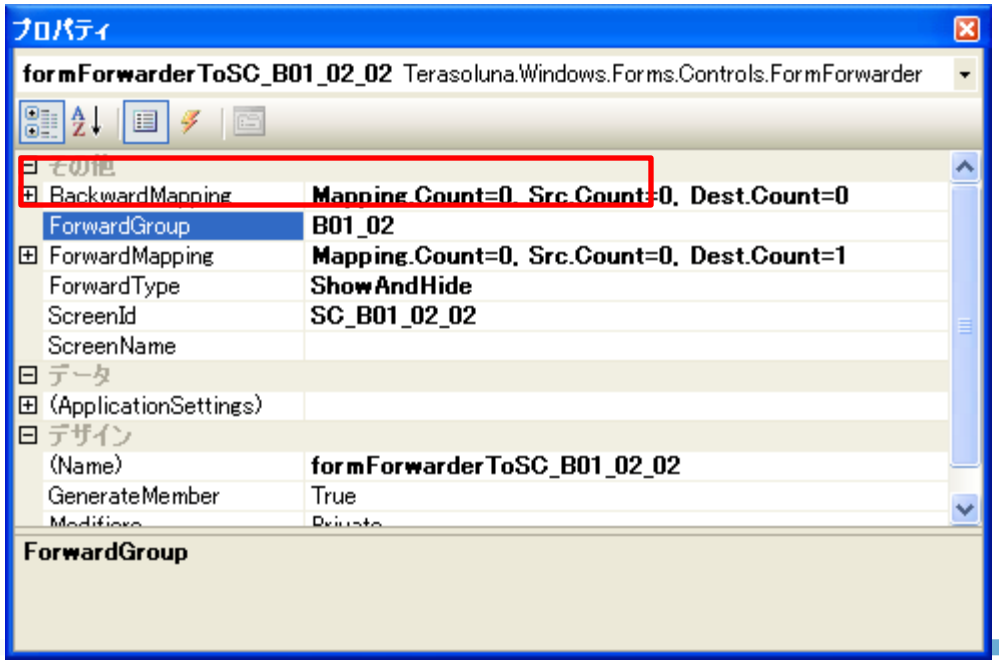
スコープの概念

- 入れ子の場合は、外側のスコープの影響を受ける
  - 外側のスコープ(スコープA)が終了すると、入れ子のスコープ(スコープB)も同時に(連鎖的に)終了する



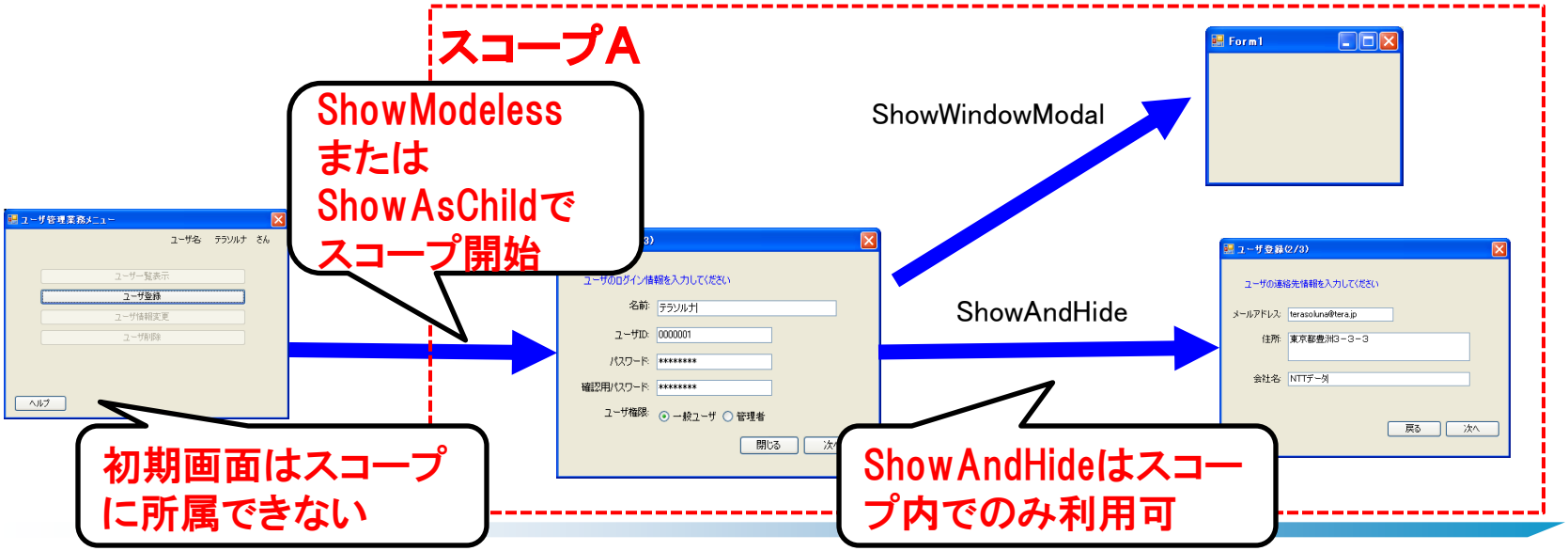
スコープの利用イメージ

- スコープの設定
  - FormForwarderのForwardGroupプロパティを設定
    - スコープを一意に特定する任意の文字列
    - スコープ内のFormForwarderに全て同じ値(業務IDやユースケースIDなど)を設定



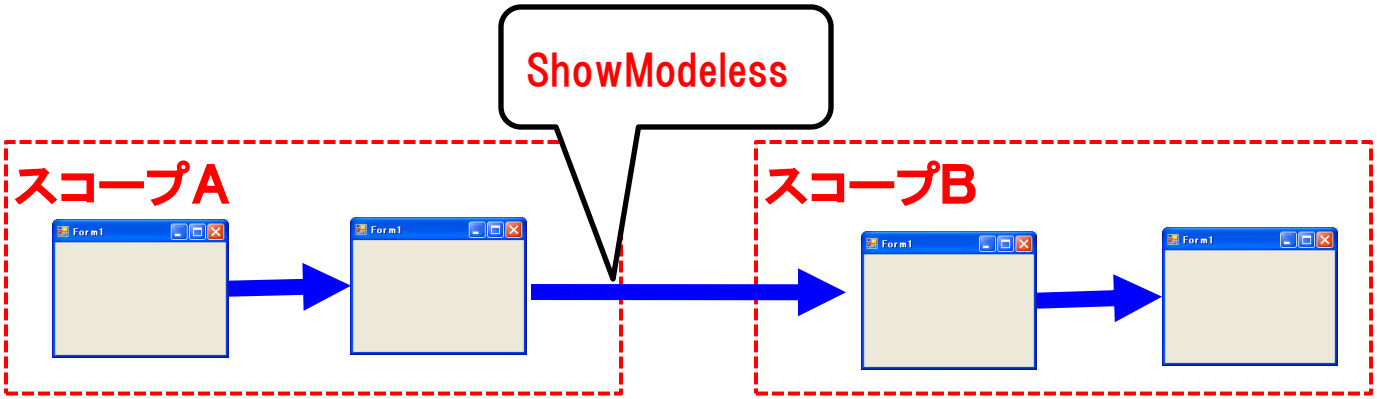
スコープの利用イメージ

- スコープの開始方法
  - 「ShowModeless」、「ShowAsChild」でのみ開始可能
  - FormForwarder.ForwardGroupプロパティに初めて値が設定された場合か、現在のスコープと異なる値が設定された場合に、新しいスコープが開始する
  - AP起動時の初期画面はスコープに所属できない
    - 初期画面は、FormForwarderによる画面表示をしないため



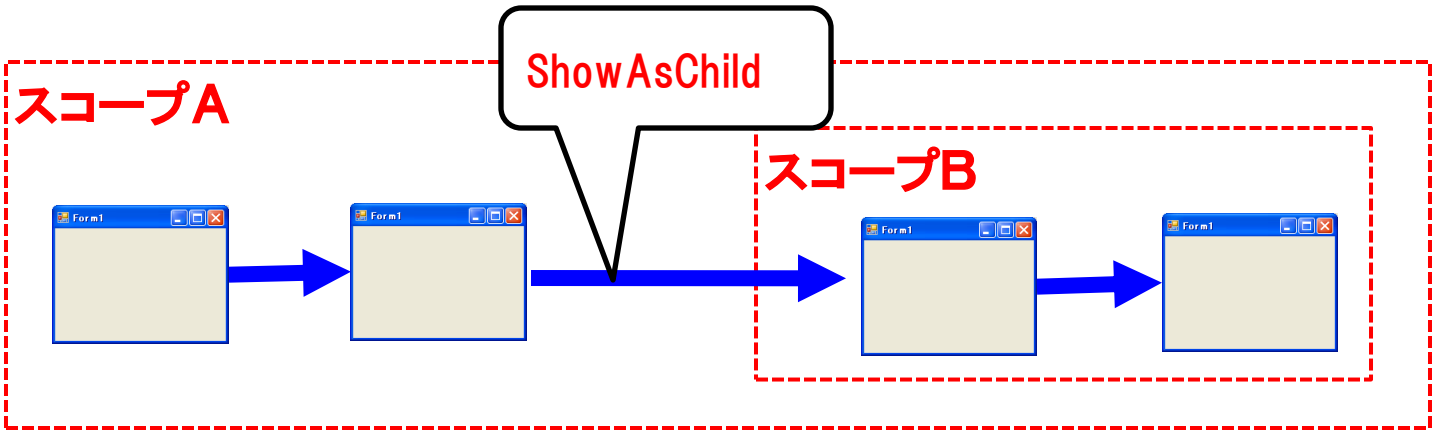
スコープの利用イメージ

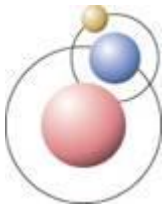
- 「ShowModeless」で新しいスコープを開始する場合
  - 遷移元があるスコープに所属している場合、スコープ同士は独立したスコープになる



スコープの利用イメージ

- 「ShowAsChild」で新しいスコープを開始する場合
  - 遷移元があるスコープに所属している場合、スコープ同士は、入れ子の関係になる
  - 入れ子の場合、外側のスコープが消滅すると内側のスコープも連鎖して消滅する
    - スコープAと連鎖してスコープBの全画面もクローズ



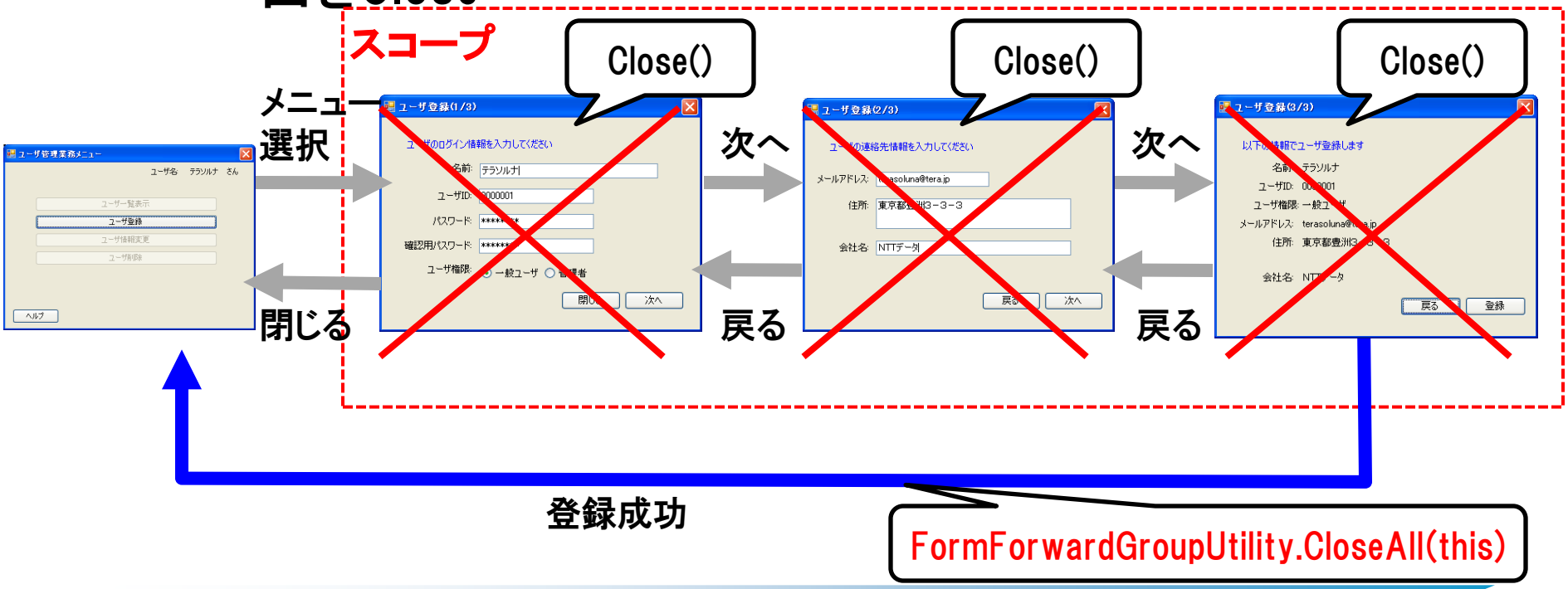


スコープの利用イメージ

- 現在のスコープを終了し、スコープ外の画面へ遷移する方法
  - （ネストではない）新たなスコープを開始して遷移
    - 前述のスコープの開始方法を参照
  - もしくは、ForwardGroupプロパティに値を指定せずにShowModelessで次画面へ遷移
- スコープ外に遷移しても、スコープ自体やスコープ内の画面群は破棄されない
  - スコープを破棄するためには、以下のいずれかの方法でスコープ内に生存する全画面をクローズ（破棄）する
    - スコープ開始画面をクローズ（Form.Close()）
    - FormForwardGroupUtilityクラスのスコープ破棄用メソッドを実行（後述）

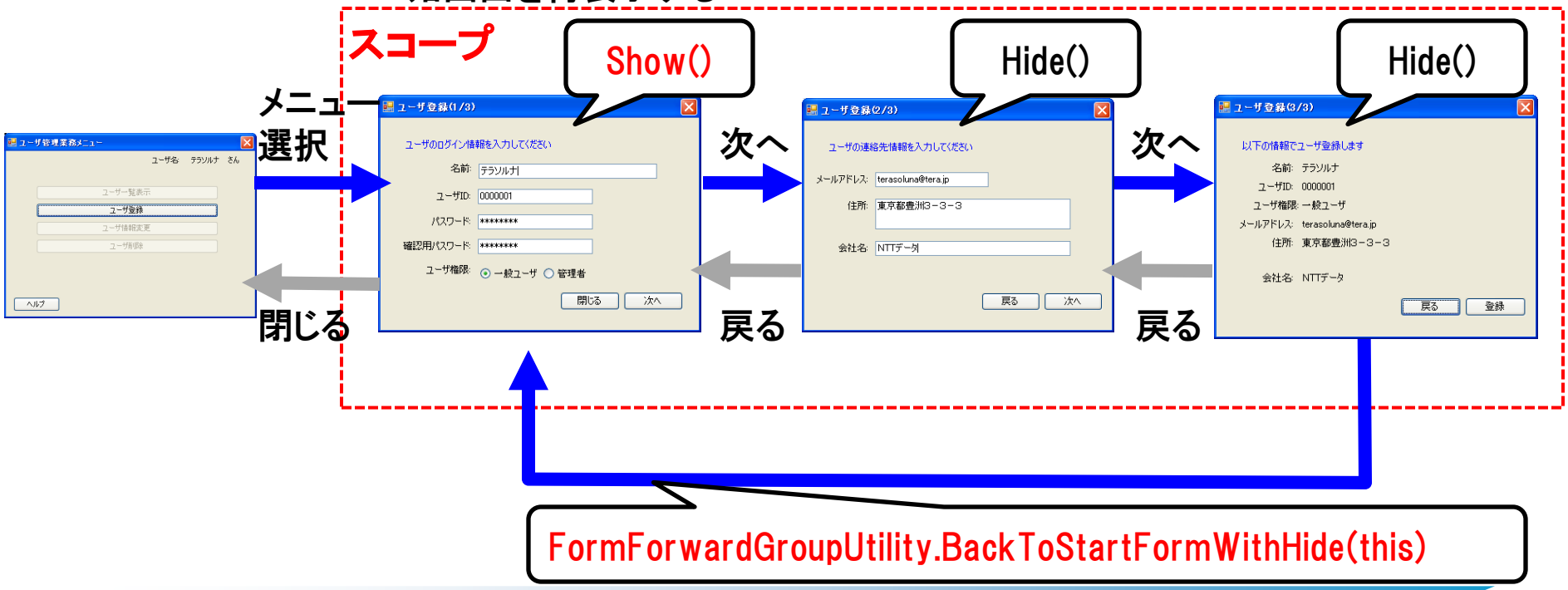
スコープの利用イメージ

- スコープの破棄(全画面クローズ)機能
  - FormForwardGroupUtility.CloseAll(Form)メソッド
  - 一連の業務処理が完了したときに、スコープ内の全画面をClose



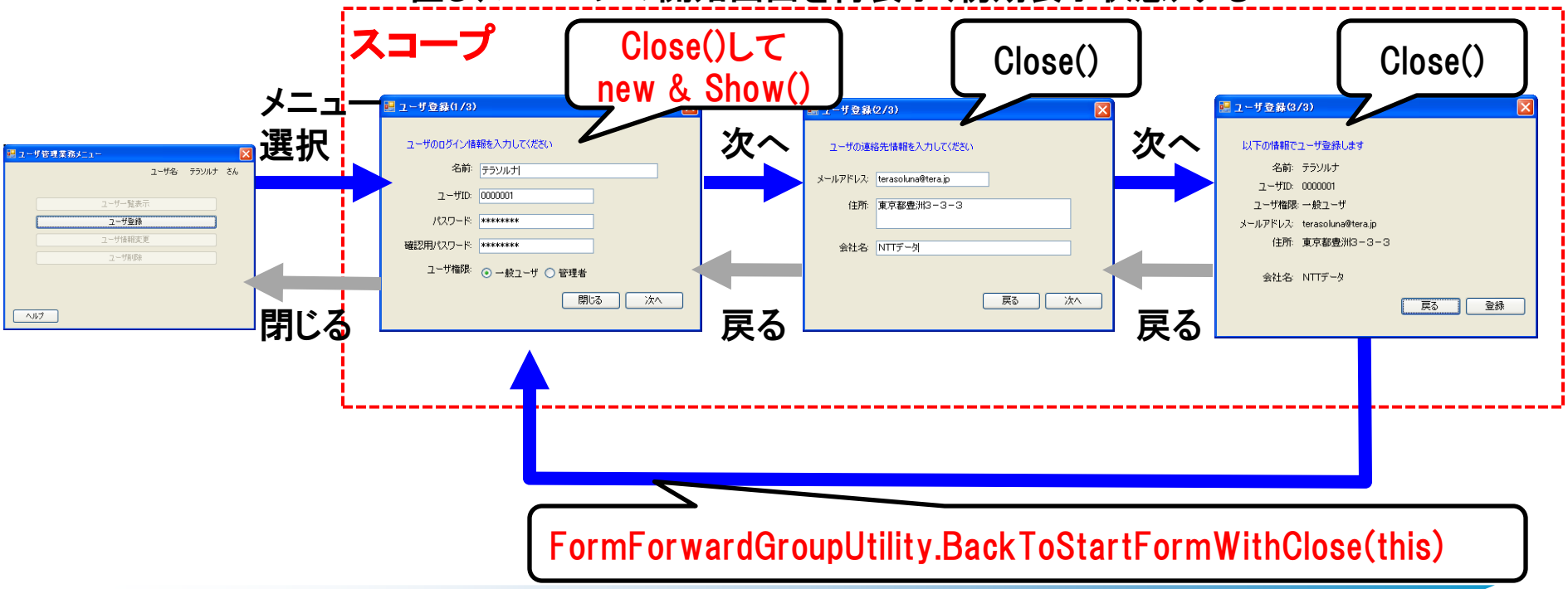
スコープの利用イメージ

- スコープ開始画面への遷移機能
  - スコープ内の開始画面へ戻るユーティリティメソッドを提供
    - FormForwardGroupUtility.BackToStartFormWithHide(Form)メソッド
      - 開始画面以外のスコープ上の全画面を非表示(Hide)とし、スコープの開始画面を再表示する



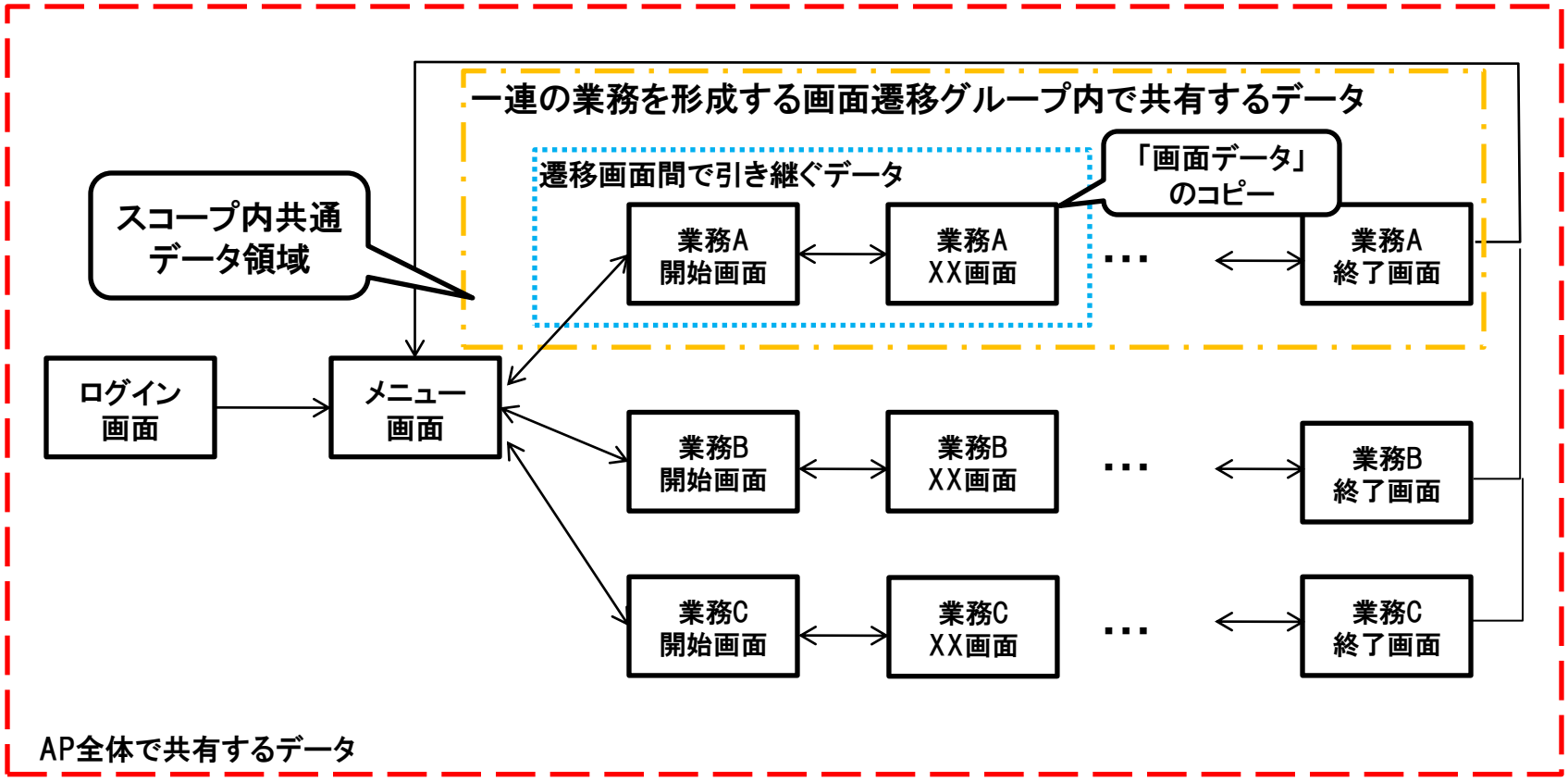
スコープの利用イメージ

- スコープ開始画面への遷移機能
  - スコープ内の開始画面へ戻るユーティリティメソッドを提供
    - FormForwardGroupUtility.BackToStartFormWithClose(Form)メソッド
      - 全画面をクローズしてスコープをいったん破棄したのち、スコープを開始し直し、スコープの開始画面を再表示(初期表示状態)する



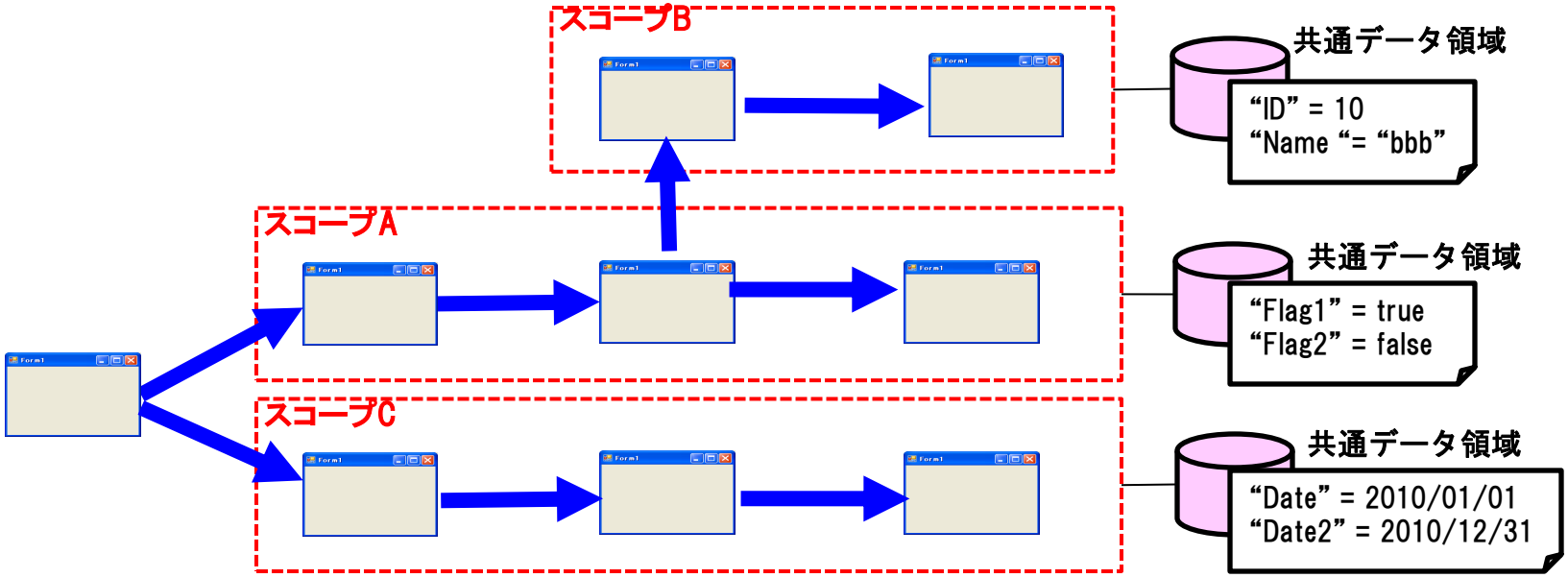
画面遷移機能の検討経緯（画面間のデータ引き継ぎ）

- FormForwarderを使ったデータの引き継ぎ



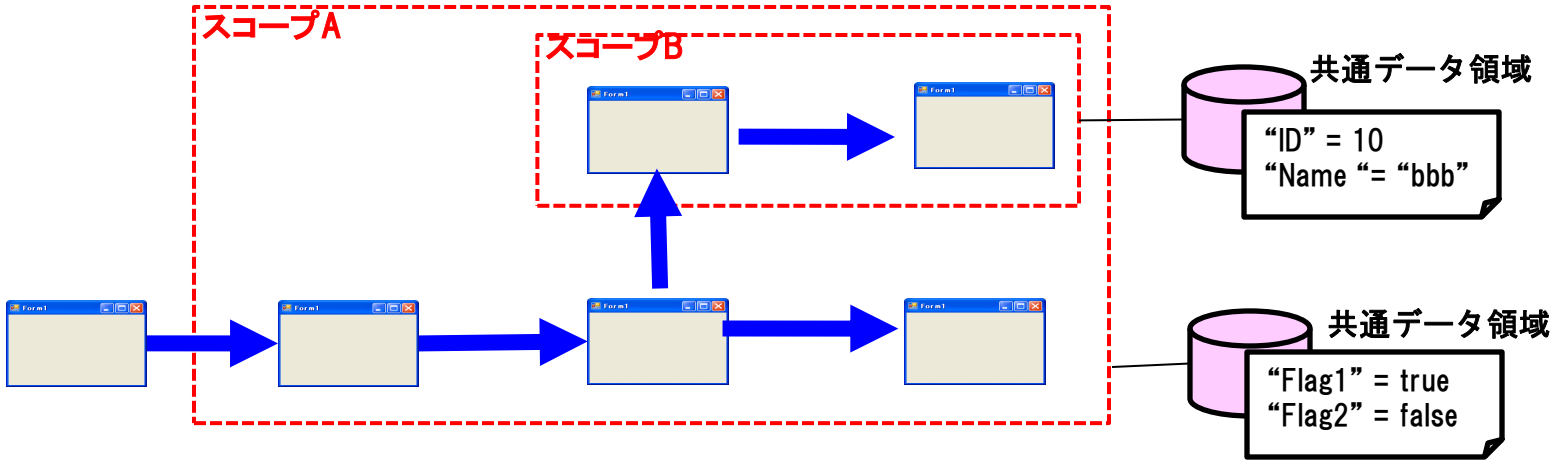
スコープ内共通データ領域

- 「スコープ内共通データ領域」の提供
  - 一連の業務(ユースケース)等単位内でデータを引き継ぐ仕組みが必要
- FormForwardGroupScope.Itemプロパティ(インデクサ)を利用
  - FormFowardManager.GetScope(現在の画面)["キー"] = "値"
    - FormForwardManager.GetScopeメソッドで現在のスコープを取得し、キーに対する値を格納したり、キーを指定して値を取得したりすることが可能



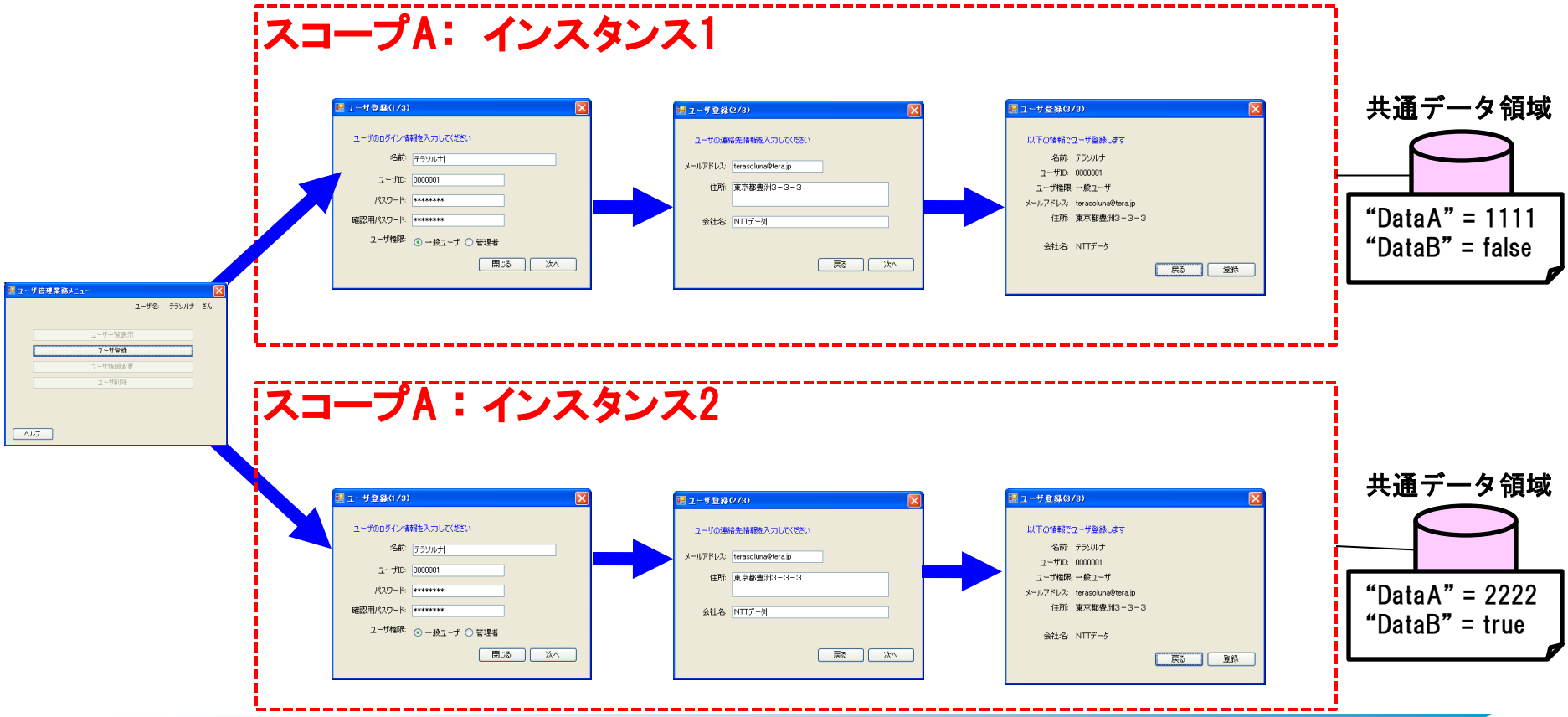
スコープ内共通データ領域

- スコープが入れ子関係の場合でも、各スコープの共通データ領域は独立して管理される



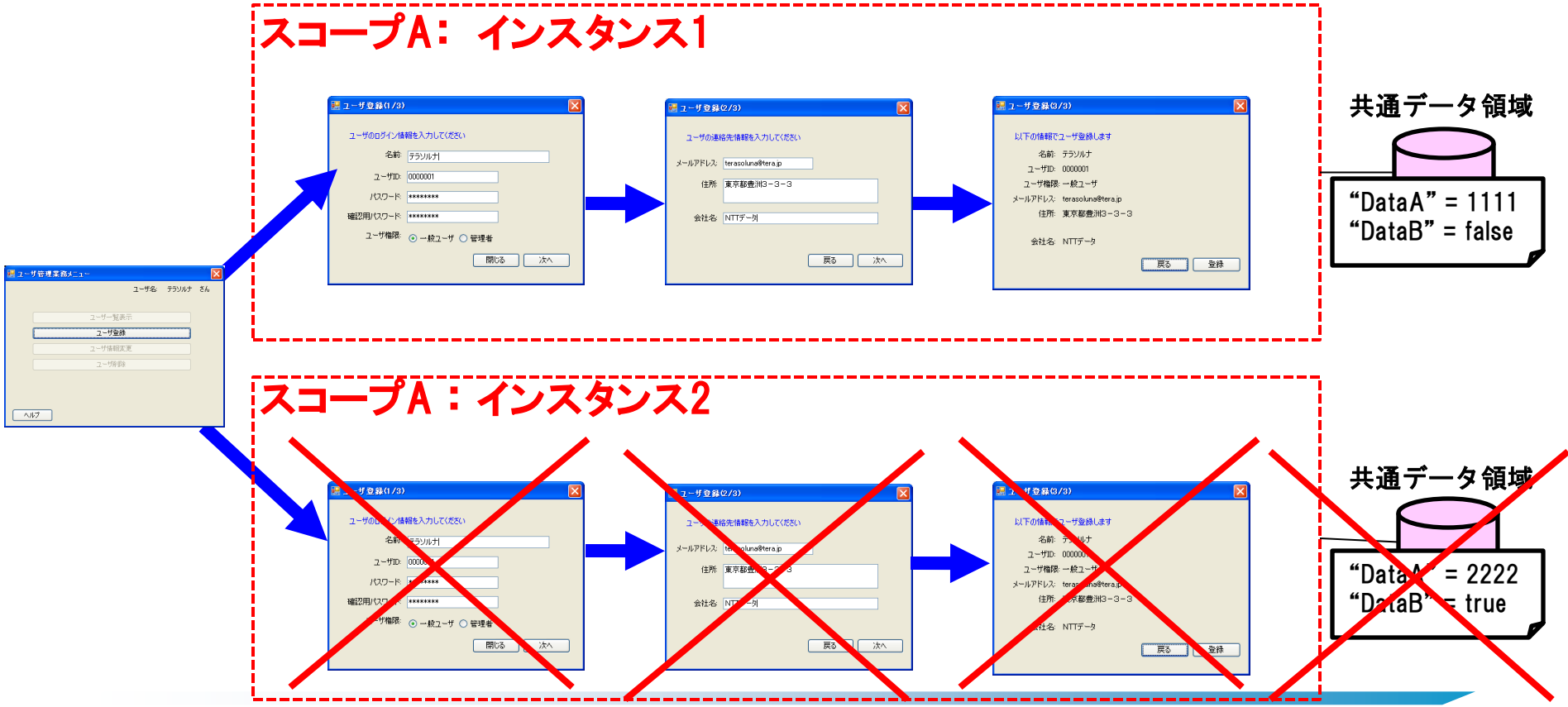
スコープ内共通データ領域

- スコープのインスタンスごとに共通データ領域を保持
  - 同じ業務でも複数のセッション情報を同時に複数を持つことが可能



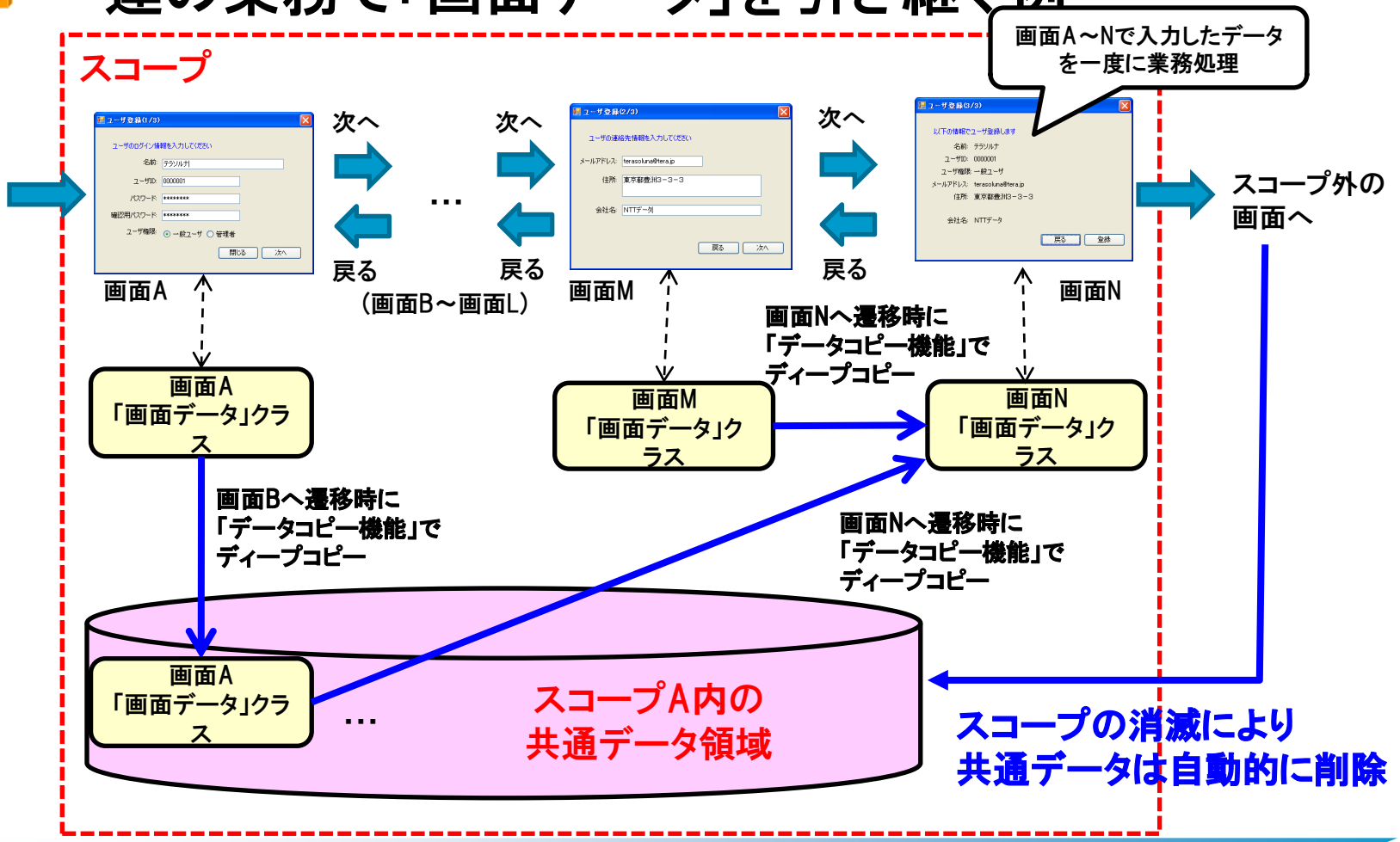
スコープ内共通データ領域

- スコープの消滅時に共通データ領域も自動的に消滅
  - セッション情報の消し忘れを気にする必要がない



スコープ内共通データ領域の利用イメージ

一連の業務で「画面データ」を引き継ぐ例

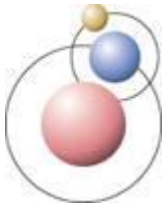


スコープ内共通データ領域の利用イメージ

- 次画面遷移時に、「画面データ」をスコープ内の共通データ領域に格納する
  - FormForwarder.ForwardFormLoadingイベントで実装

最終画面の１つ前までの画面遷移で、「画面データ」を引き継ぐための実装例（前ページの図の画面Ａ～Ｌに相当）

```
//次画面遷移時のイベントハンドラメソッド
private void formForwarderToSC_B01_02_02_ForwardFormLoading(object sender,
                ForwardFormLoadingEventArgs e)
{
    SC_B01_02_01ViewData sC_B01_02_01ViewData =
            ValidatableViewDataManager.CreateViewData<SC_B01_02_01ViewData>();
    // 「データコピー機能」で「画面データ」をディープコピー
    DataCopyManager.Copy(ViewData, sC_B01_02_01ViewData);
     //スコープ内共通データ領域に「画面データ」を保存
    FormForwardManager.GetScope(this)["SC_B01_02_01ViewData"] = sC_B01_02_01ViewData;
}
```



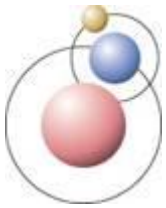
スコープ内共通データ領域の利用イメージ

- スコープの最終画面遷移時に、スコープ内共通データ領域に格納した「画面データ」を取得
  - FormForwarder.ForwardFormLoadingイベントで実装

スコープ最終画面（前々ページの図の画面Ｎに相当）へ遷移時の実装例
最終画面の１つ前の画面（画面Ｍに相当）で実装

```
//スコープ内の最終画面（画面Ｎ）へ遷移時のイベントハンドラメソッド
private void formForwarderToSC_B01_02_03_ForwardFormLoading(object sender,
                ForwardFormLoadingEventArgs e)
{
    //画面Ａの「画面データ」をスコープ内共通データ領域から取得
    object sC_B01_02_01ViewData = FormForwardManager.GetScope(this)["SC_B01_02_01ViewData"];
    if (sC_B01_02_01ViewData != null)
    {
      //「データコピー機能」で画面Ａの「画面データ」を画面Ｎの「画面データ」へディープコピー
      DataCopyManager.Copy(sC_B01_02_01ViewData, e.ViewData);
    }
       ・・・

    // 「データコピー機能」で現在（画面Ｍ）の「画面データ」を画面Ｎの「画面データ」へディープコピー
    DataCopyManager.Copy(ViewData, e.ViewData);

}
```


TERASOLUNA



スコープ内共通データ領域の利用イメージ

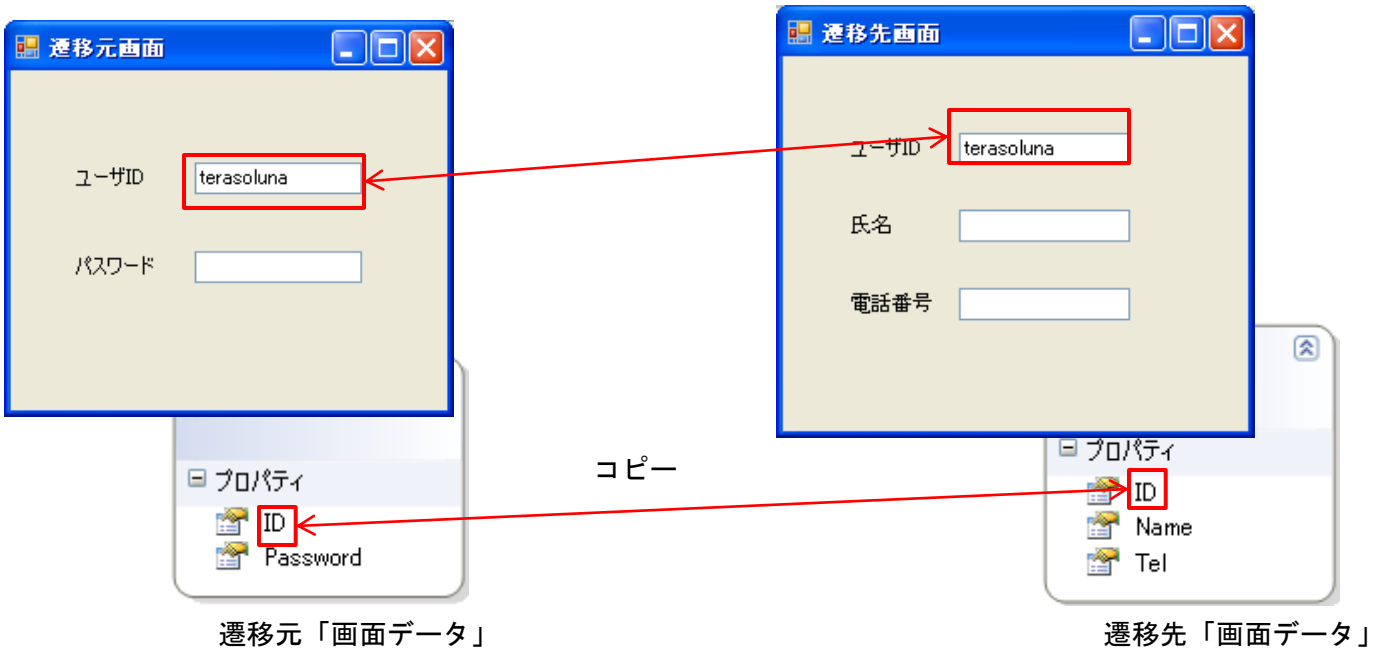
- 業務完了時、スコープ内の画面を全てクローズ（スコープ消滅）する
  - スコープとともに共通データ領域は自動的に消滅

スコープ内の最終画面（画面Ｎ）の実装例

```
/// 登録ボタン押下時のイベント処理完了時
private void b01_02_03_C01EventProcessWorker_Completed(object sender,
                EventProcCompletedEventArgs e)
{
    if (e.Result.IsSuccess)
    {
        //業務処理成功時にスコープ上の画面を全てクローズ
        FormForwardGroupUtility.CloseAll(this);
        //TODO: 必要に応じて次画面へ遷移処理を実装
    }
    else if (e.Result.IsAnyBizLogicError || e.Result.IsAnyValidationError)
    {
        //業務エラー時には、スコープ上の画面開始画面へ戻る例
        //全てHideして開始画面を再表示（Show）して戻る
        FormForwardGroupUtility.BackToStartFormWithHide(this);
    }
}
```

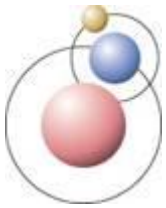
遷移元・遷移先画面間の画面データの自動コピー

- 遷移元画面と遷移先画面間でデータの引き継ぎ
  - 「データコピー機能」により、遷移元画面と遷移先画面の「画面データ」を自動的にコピー



画面データの自動コピー

- 遷移方向によってコピーの設定が2つある
  - 遷移元画面⇒遷移先画面の遷移時
    - FormForwarder.Forwardメソッドの呼び出しによる順方向の遷移
    - 遷移元画面⇒遷移先画面のコピーは、指定項目に関しては、無条件に実行
    - 「ForwardMapping」プロパティでマッピング設定を実施
  - 遷移先画面⇒遷移元画面の遷移時
    - 遷移先画面のクローズ(または非表示)による逆方向の遷移
    - 遷移先画面⇒遷移元画面のコピーは、遷移先画面をDialogResult.OKまたはDialogResult.Yesで閉じたときにのみ実行
      - モーダルダイアログから戻り遷移元画面へ結果を反映する場合など、遷移元画面へデータコピーしてよいかは遷移先画面での処理結果を見て判定する必要があるため
    - 「BackwardMapping」プロパティでマッピング設定を実施

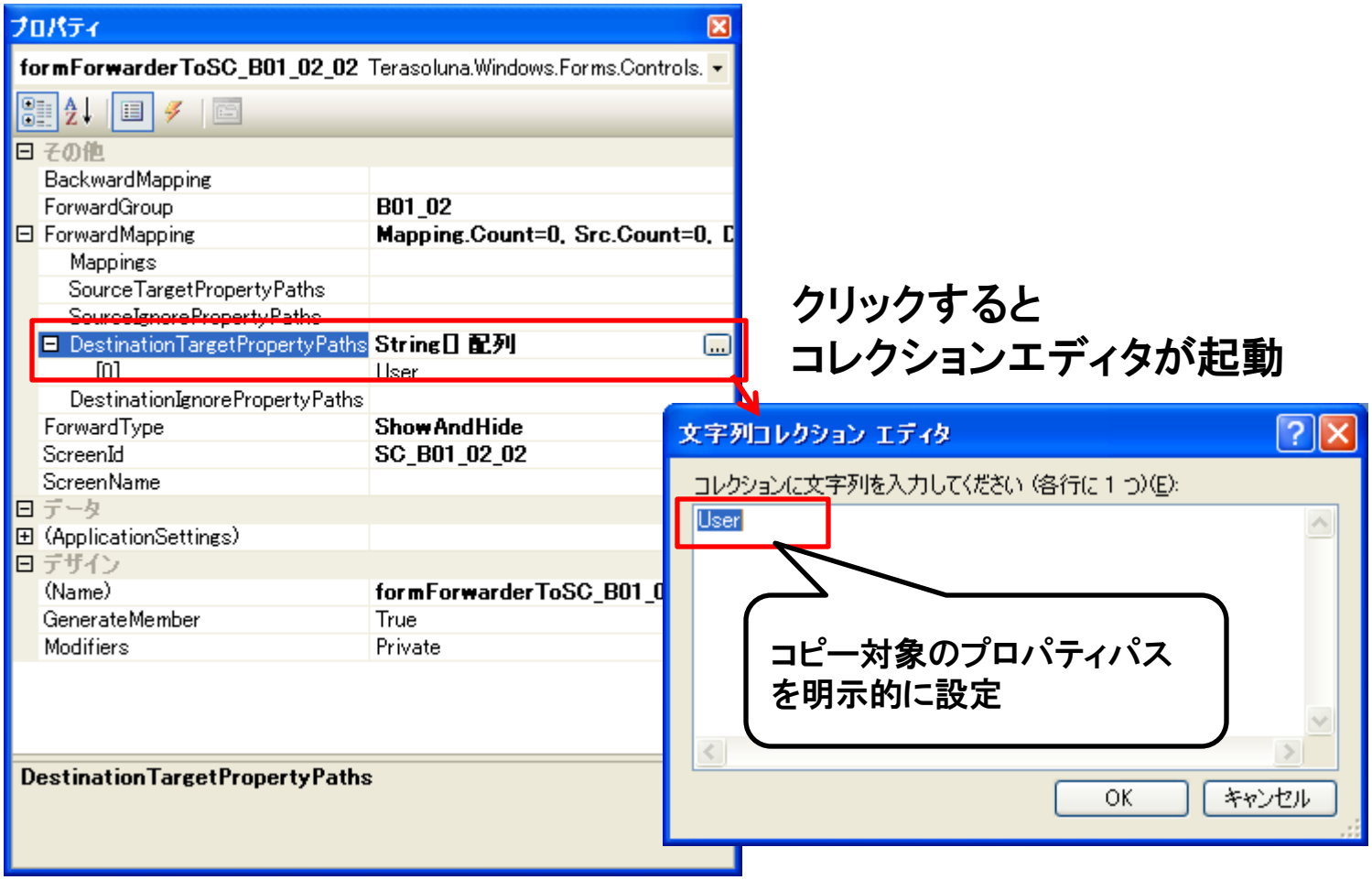


画面データの自動コピーの使用時の注意点

- データコピー対象のプロパティ(パス)を明示的に指定する必要がある
  - DestinationTargetPropertyPathsプロパティを設定
    - 「イベント処理実行機能」における「画面データ」⇔DTOの自動コピーと異なり、指定した項目以外のコピーは実施されない
- 異なる画面間では、「画面データ」のプロパティが同名でも異なる意味のデータが格納される可能性があり、無条件コピーによる誤動作を防ぐため、暗黙的な自動コピーは実行しない
  - たとえば、画面Aの「画面データ」ではNameプロパティに「氏名」を格納、画面Bでは「商品名」が格納される
  - 「画面データ」とDTOは同一画面のデータを扱うため、プロパティ名が一致した場合、同じ意味のデータである可能性が高い。そのため、イベント処理実行時に、暗黙的な自動コピーを実行する仕様とした

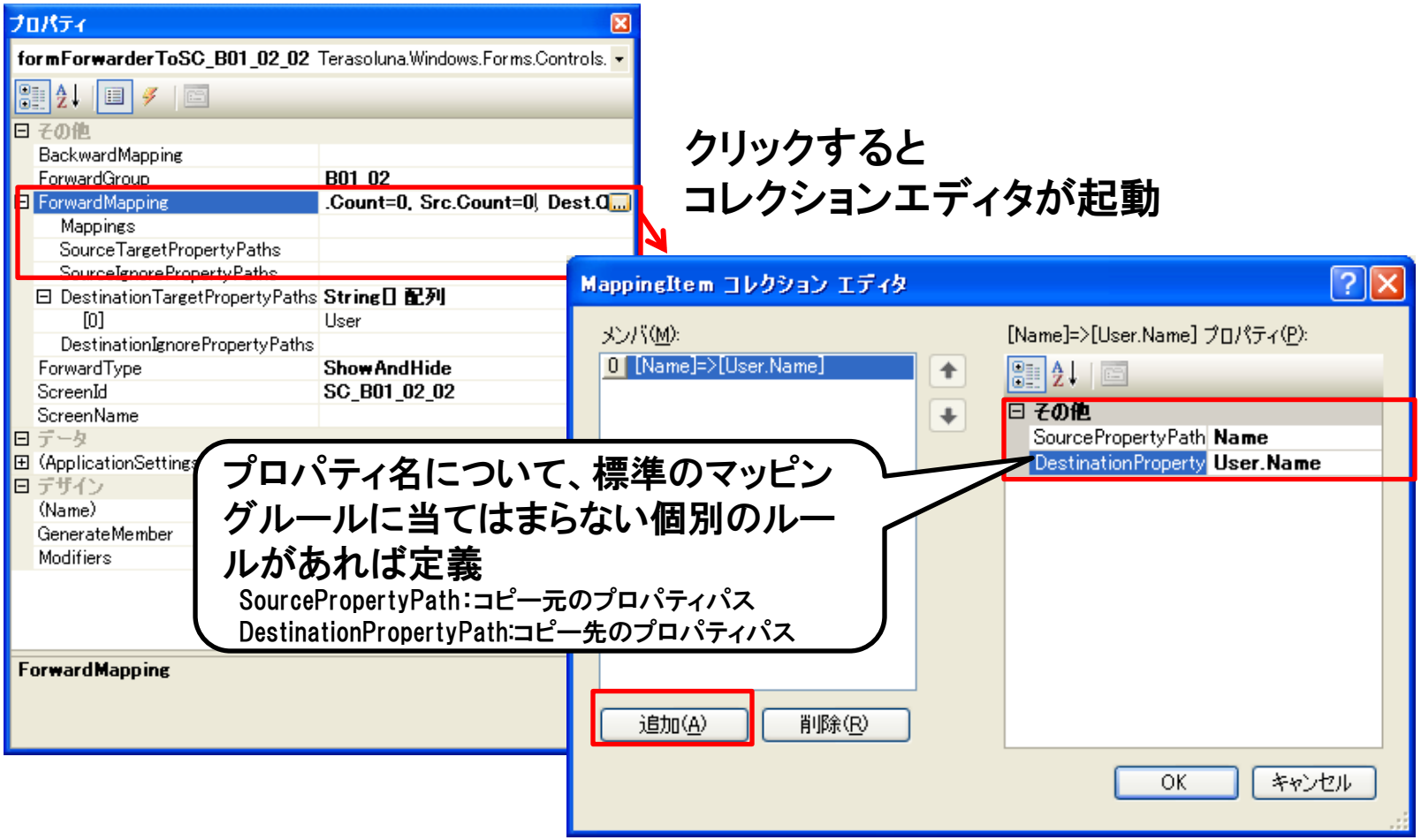
画面データのコピー処理のマッピング設定イメージ

FormForwarderのDestinationTargetPropertyPathsプロパティ



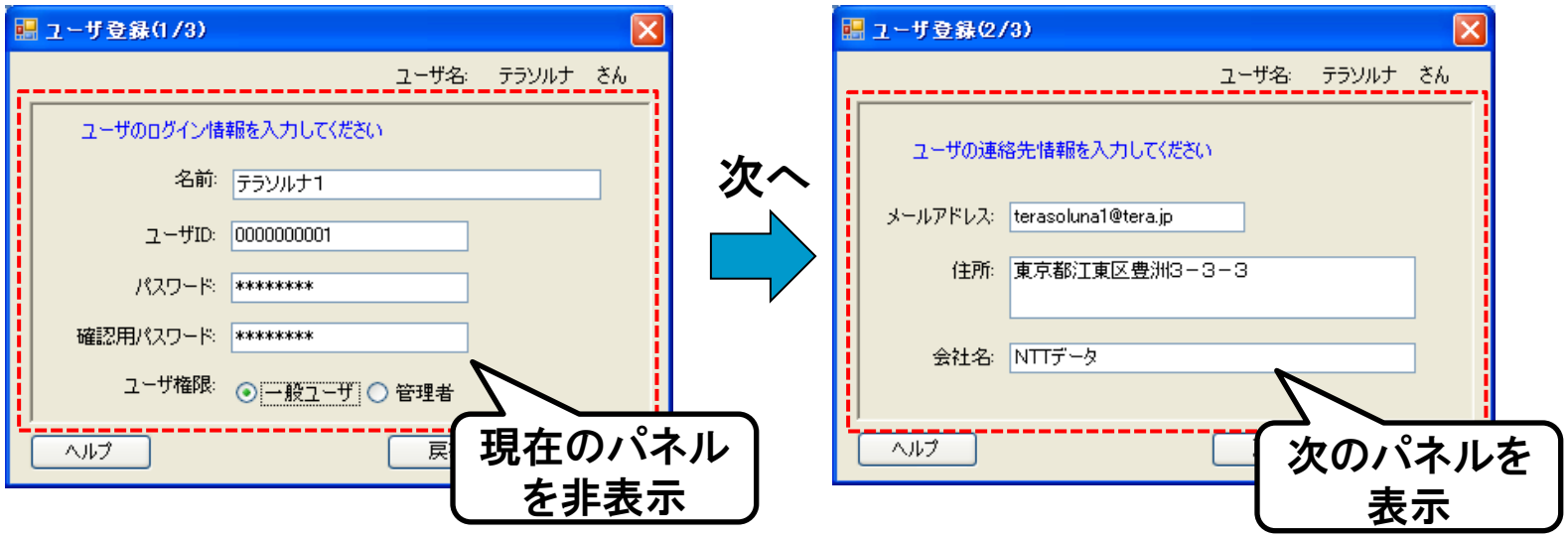
画面データのコピー処理のマッピング設定イメージ

FormForwarderのMappingsプロパティ



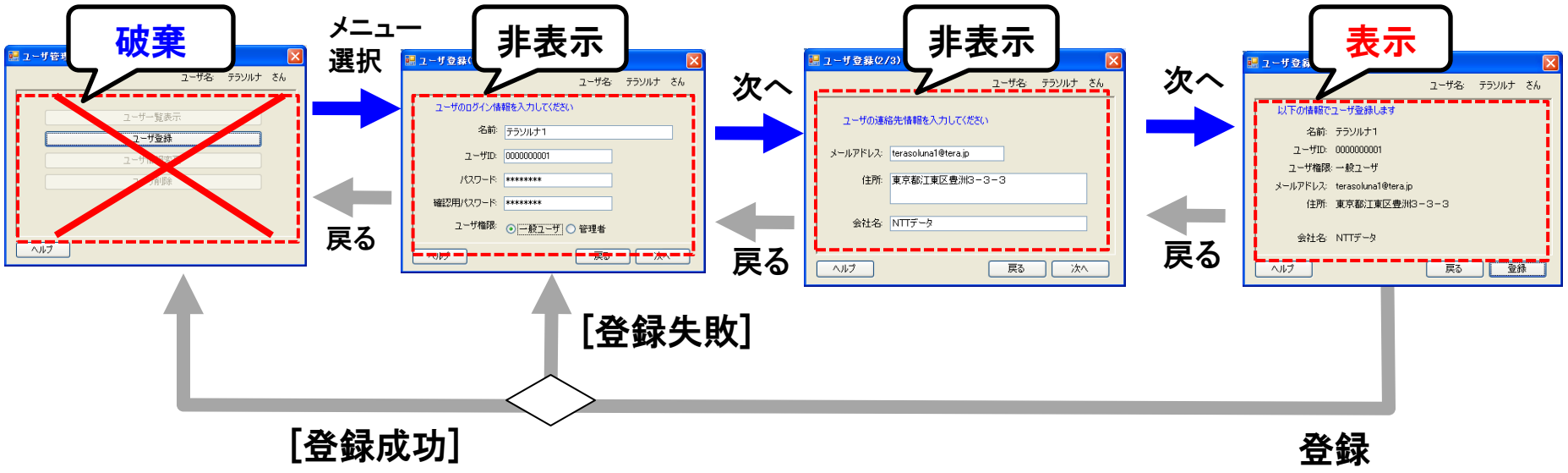
画面遷移機能の検討経緯(パネル切り替え)

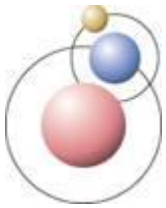
- パネルの切り替えによる画面遷移
  - 画面上に配置したパネル(Control)を切り替える遷移方式
    - 遷移先パネルを表示
      - コンテナとなるContentPlaceHolderに新しいパネルを追加し、表示
    - 遷移元パネルを非表示
      - コンテナとなるContentPlaceHolderから現在のパネルを削除し、非表示



画面遷移機能の検討経緯(パネル切り替え)

- 遷移したパネルのインスタンスや表示状態を管理
  - ContentPlaceHolderが非表示状態のパネルも含めてインスタンス管理
  - パネルを破棄したい場合は、ContentPlaceHolderにインスタンス管理の終了を指示



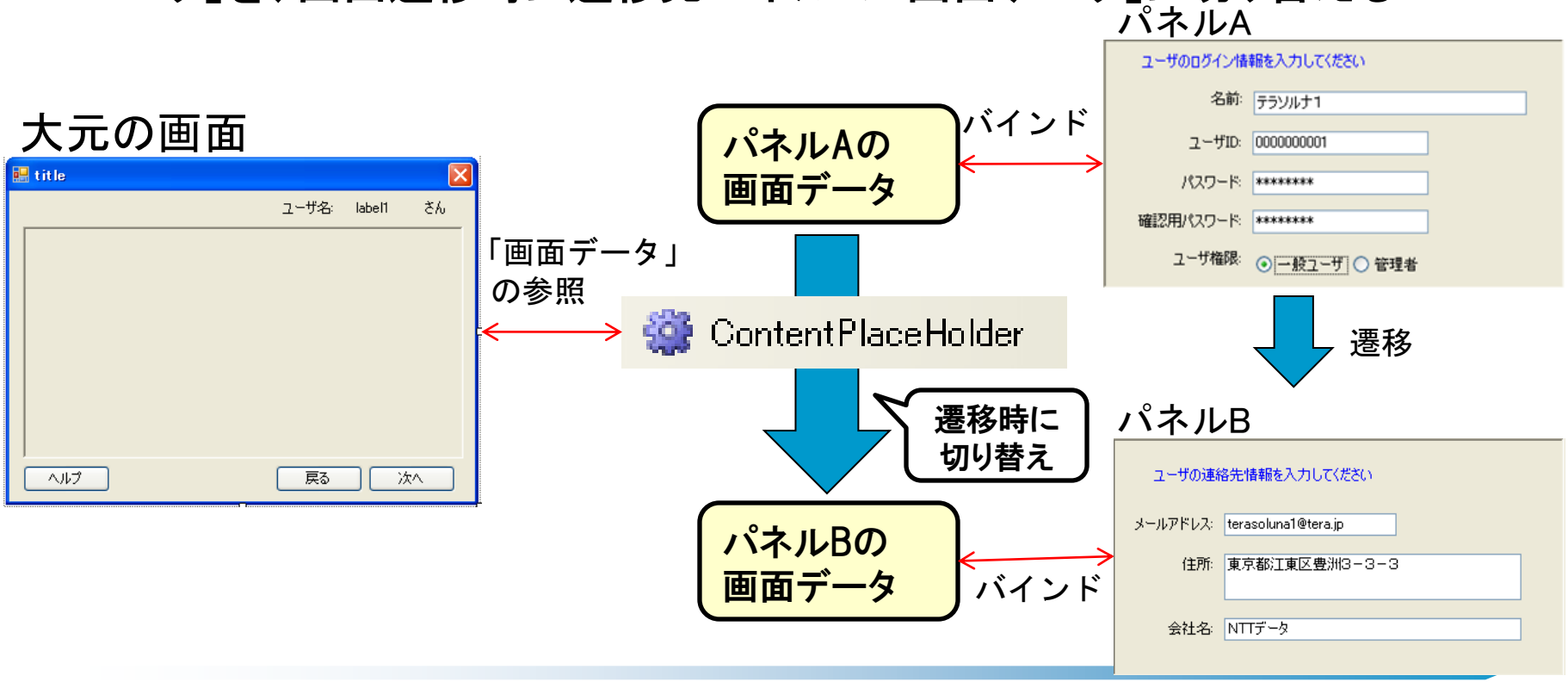


画面遷移機能の検討経緯(パネル切り替え)

- 「画面データ」の管理の問題
  - パネルが切り替わっても大元の画面は変化しない
    - TERASOLUNAフレームワークの設計思想は、画面(Form)と「画面データ(ルート)」が1対1
      - このルールを守らないと、EventProcessWorker等が動作しない
  - パネル切り替えの場合は、パネルと「画面データ(ルート)」が1対1になるように設計する
  - パネルが切り替わるタイミングで、大元の画面に定義された「画面データ(ルート)」を、パネルに対応した「画面データ(ルート)」に、動的に切り替える

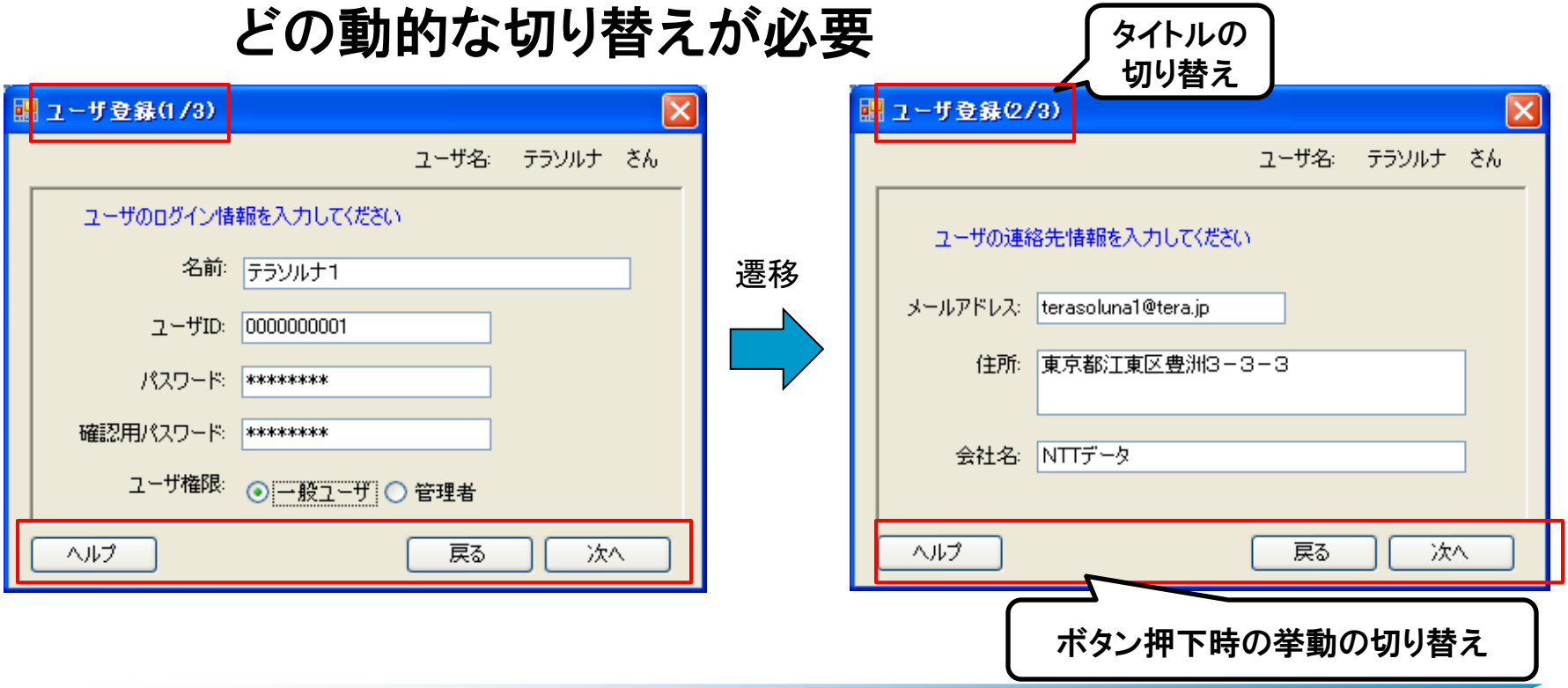
「画面データ」の切り替えイメージ

- 大元の画面はContentPlaceHolderクラス経由で、「現在」の「画面データ」を参照
- ContentPlaceHolderの機能により、大元の画面が参照する「画面データ」を、画面遷移時に遷移先パネルの「画面データ」に切り替える



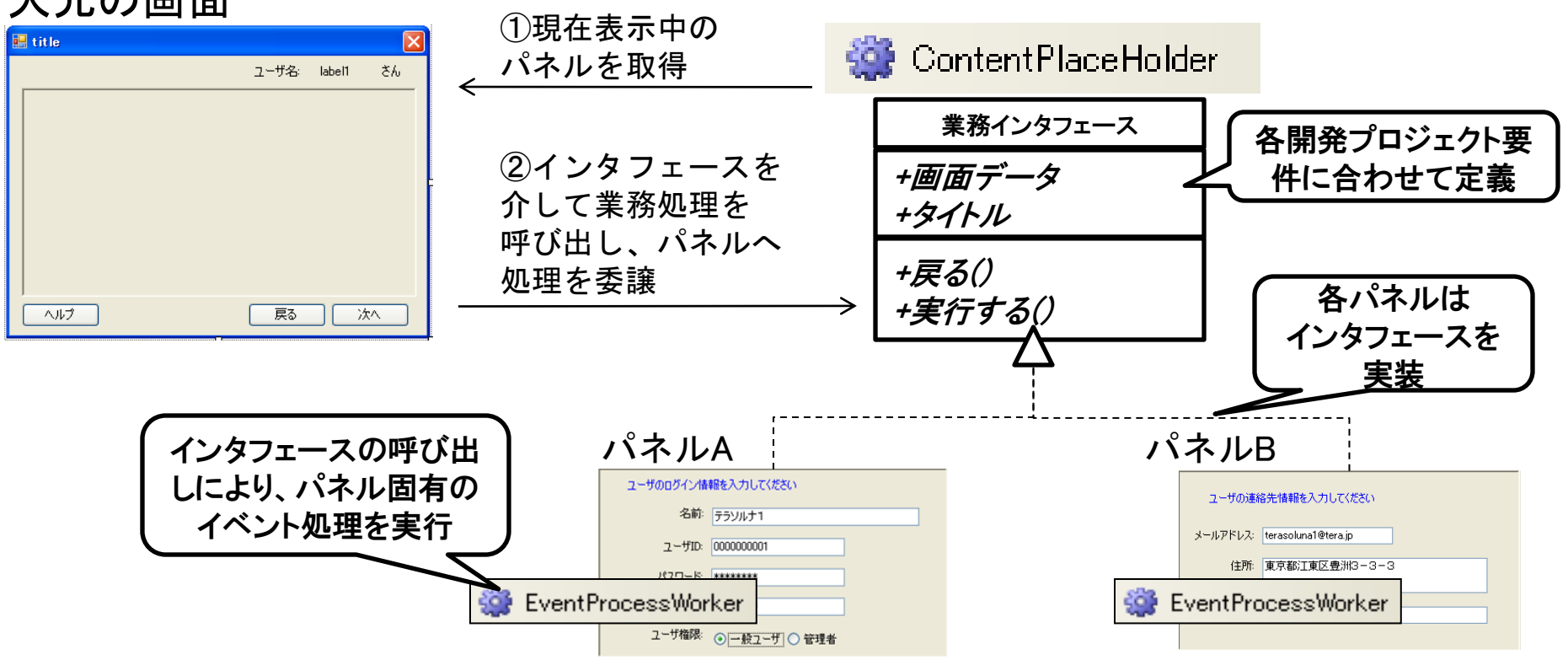
画面遷移機能の検討経緯(パネル切り替え)

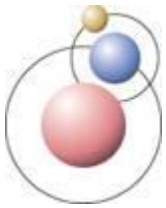
- 大元の画面に対する処理の切り替えの問題
  - 「画面データ」以外にも、表示されるパネルによって、大元の画面にあるボタン押下時の挙動やタイトル表示などの動的な切り替えが必要



大元の画面に対する処理の切り替えイメージ

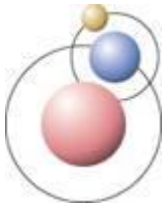
- 業務開発者は、パネル共通の業務インタフェースを実装
- 大元の画面は、ContentPlaceHolderクラス経由で表示中のパネルを取得し、インタフェースが定義するメソッドやプロパティを操作





ContentPlaceHolderの利用イメージ

- 画面コンポーネントによる開発スタイル
  - 業務開発者はContentPlaceHolderを画面に貼り付けて開発
- 業務開発者は、各パネルを実装
  - 画面データとのバインドやイベント処理の実装
  - IContentControlインタフェースの実装
    - ContentPlaceHolderが管理するパネル(Control)が共通で実装すべきインタフェース
  - パネル共通の業務インタフェースの実装
    - 大元画面に配置されたボタンイベントを契機とする処理をパネルごとに切り替える場合や、大元画面のタイトルの表示内容をパネルごとに切り替えたい場合などには、それらイベントなどに対応する業務インタフェースを定義し、パネルごとに固有の処理を実装することで振る舞いを切り替える。



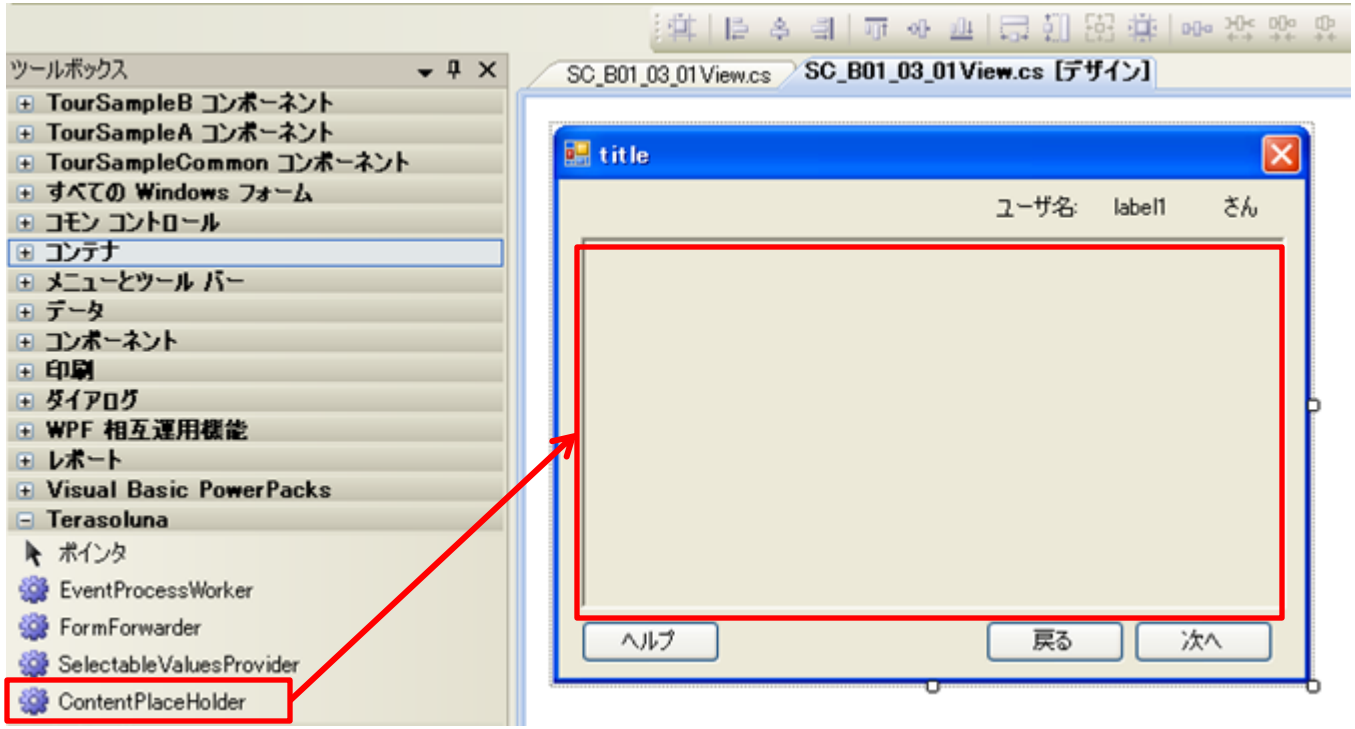
ContentPlaceHolderの利用イメージ

- ContentPlaceHolderの画面遷移メソッド
  - 「Show」、「Hide」したパネルは、インスタンス管理下
  - 「Close」したパネルはインスタンス管理外
  - FormForwarderにおける「ShowAndHide」や「スコープ」の開始・終了と同等

ContentPlaceHolderのメソッド	説明
ShowAndCloseContent(Type)	指定した型のパネルを表示(Show)し、現在のパネルがあれば破棄(Close)する。
ShowAndHideContent(Type)	指定した型のパネルを表示(Show)し、現在のパネルがあれば非表示(Hide)する。
ShowWithCloseAllContents(Type)	インスタンス管理中のパネルを全て破棄(Close)して、指定した型のパネルを表示(Show)する。
CloseAllContents()	インスタンス管理中のパネルを全て破棄(Close)する。

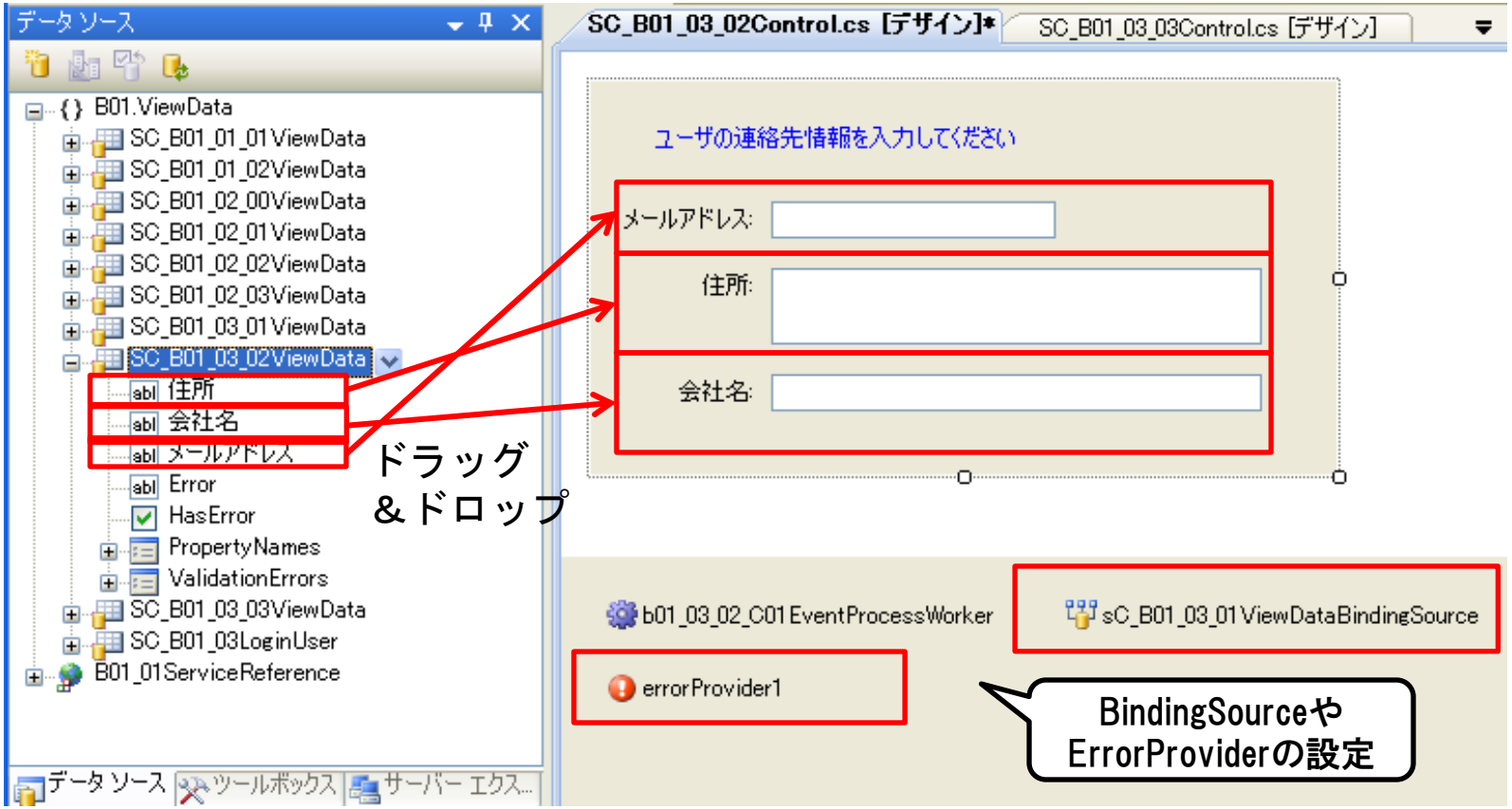
ContentPlaceHolderの利用イメージ

- ContentPlaceHolderを画面部品として貼り付け
  - その上に切り替え対象のパネルを張り付けておくことも可能



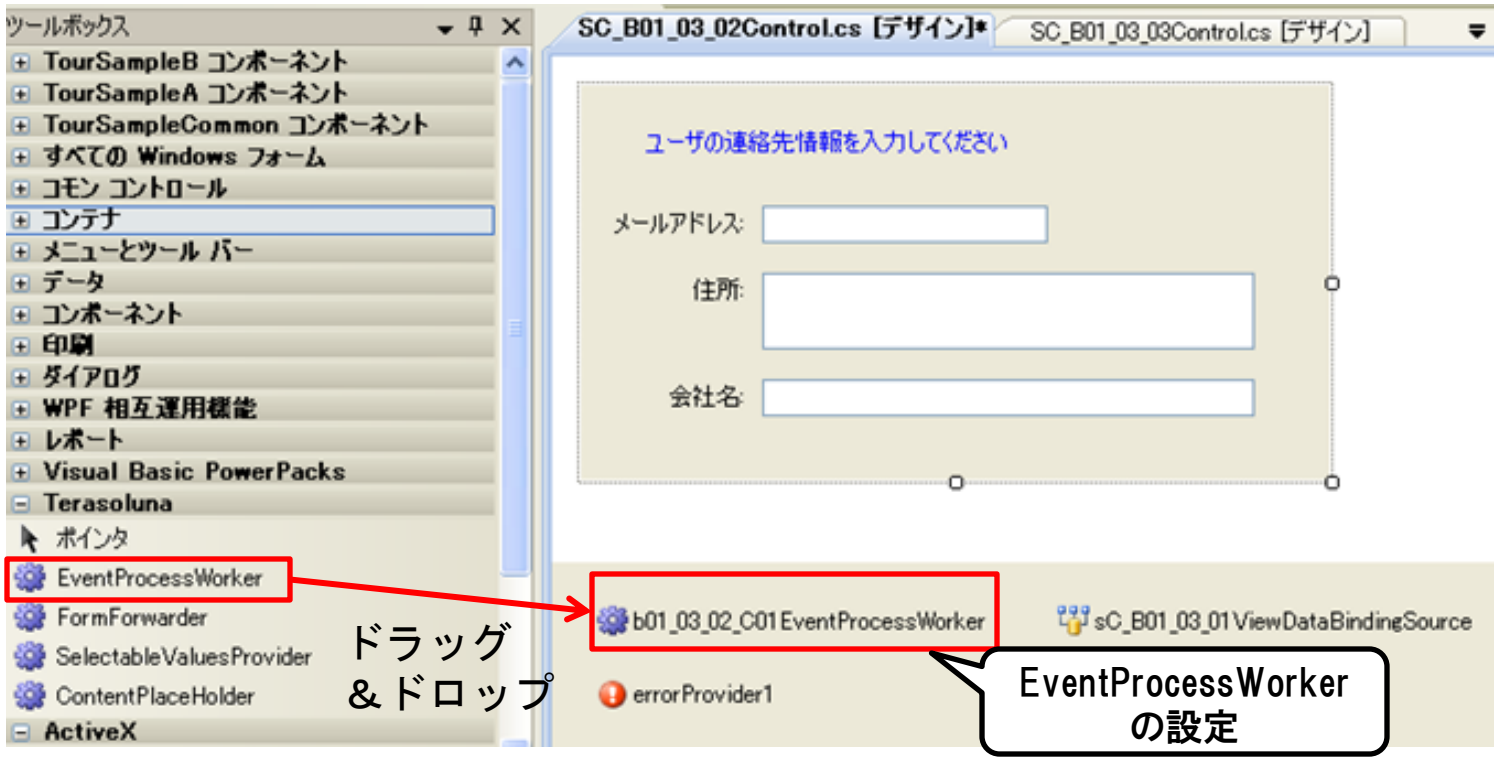
ContentPlaceHolderの利用イメージ

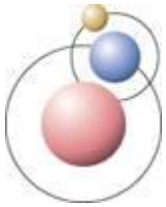
- 各パネルで、「画面データ」との双方向バインド設定



ContentPlaceHolderの利用イメージ

- 各パネルで、イベント処理を実装

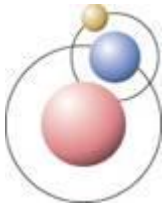




ContentPlaceHolderの利用イメージ

- パネル共通の業務インタフェースの定義
  - 業務開発者が、業務要件に応じてパネルに必要なインタフェースを定義

```
public interface ISC_B01_03_01MainContent
{
    /// 画面データ
    object ViewData { get; }
    /// 画面のタイトル
    string Title { get; }
    /// 実行ボタンを有効にするか
    bool ExecutionEnabled { get; }
    /// 実行ボタン押下による業務処理を実行する
    void Execute();
…
}
```



ContentPlaceHolderの利用イメージ

パネル共通の業務インタフェースの実装

```
/// 業務インタフェースおよびIContentControlインタフェースの実装
public partial class SC_B01_03_02Control : UserControl, ISC_B01_03_01MainContent, IContentControl
{
    /// 各コントロールの「画面データ（ルート）」を保持
    public SC_B01_03_02ViewData ViewData { ・・・ }

    /// 業務インタフェースの実装（タイトル）
    public string Title
    {
        get { return "ユーザ登録(2/3)"; }
    }

    /// 業務インタフェースの実装（業務処理の実行）
    public void Execute()
    {
        /// イベント処理の実行
        EventProcessResult result = b01_03_02_C01EventProcessWorker.RunWorker();
        if (result.IsSuccess)
        {
            /// ContentPlaceHolderの画面遷移処理メソッドの呼び出し
            ContentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_03Control));
        }
    }
 ・・・
}
```

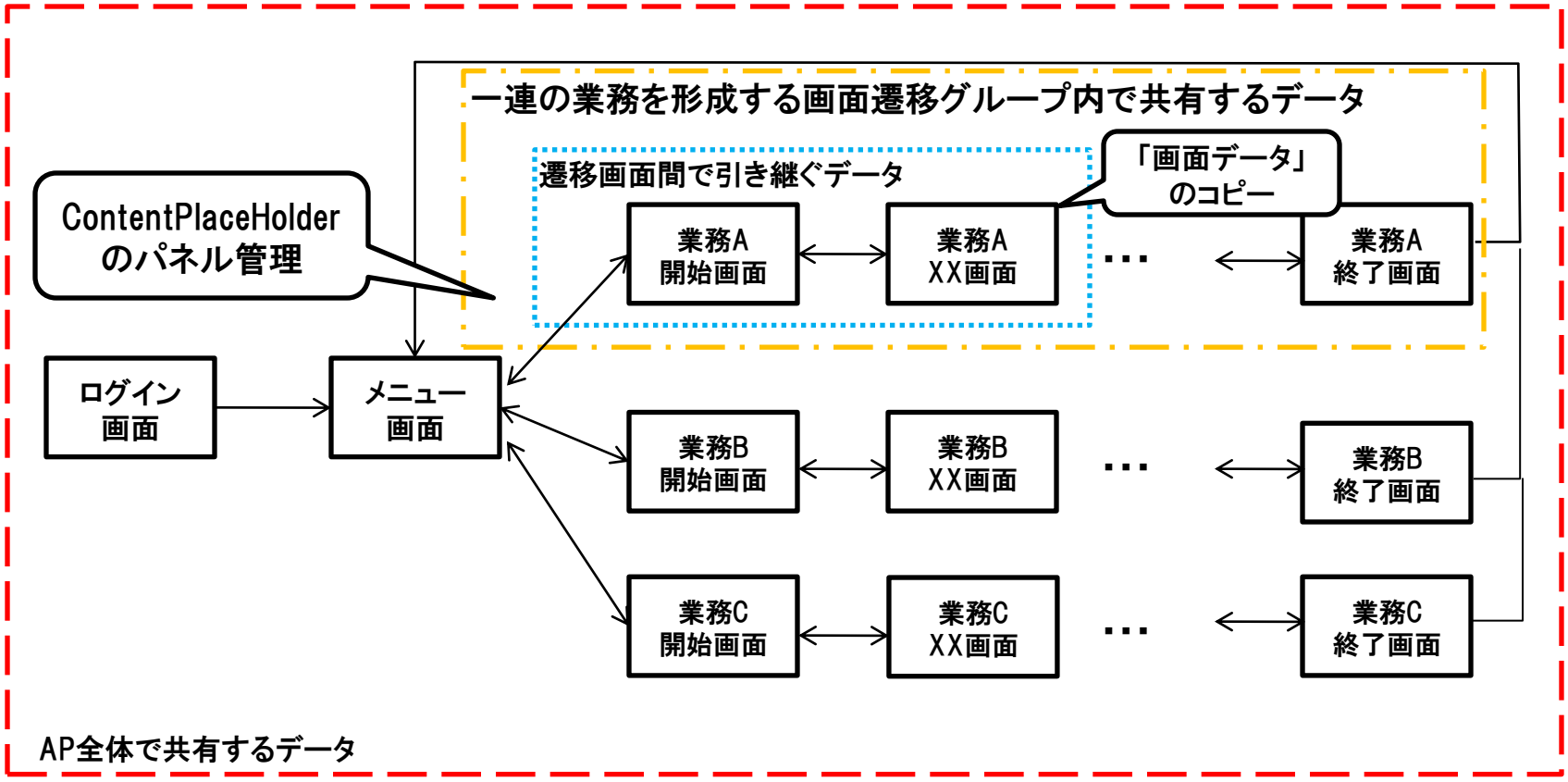
ContentPlaceHolderの利用イメージ

- 大元の画面は、ContentPlaceHolderを経由し、現在表示中のパネルに処理を委譲するように実装

```
[ScreenId("SC_B01_03_01")]
public partial class SC_B01_03_01View : Form
{
    public object ViewData
    {
        get
        {
            /// 表示中のパネルが保持する画面データを取得
            object viewData = null;
            ISC_B01_03_01MainContent mainContent =  contentPlaceHolder.CurrentContentControl as ISC_B01_03_01MainContent;
            if (mainContent != null)
            {
                viewData = mainContent.ViewData;
            }
            return viewData;
        }
    }
/// ボタン押下時
private void executeButton_Click(object sender, EventArgs e)
{
        /// 表示中のパネルに対する業務処理を実行
        ISC_B01_03_01MainContent mainContent = contentPlaceHolder.CurrentContentControl  as ISC_B01_03_01MainContent;
        if (mainContent != null)
        {
            mainContent.Execute();
        }
    }
・・・
}
```

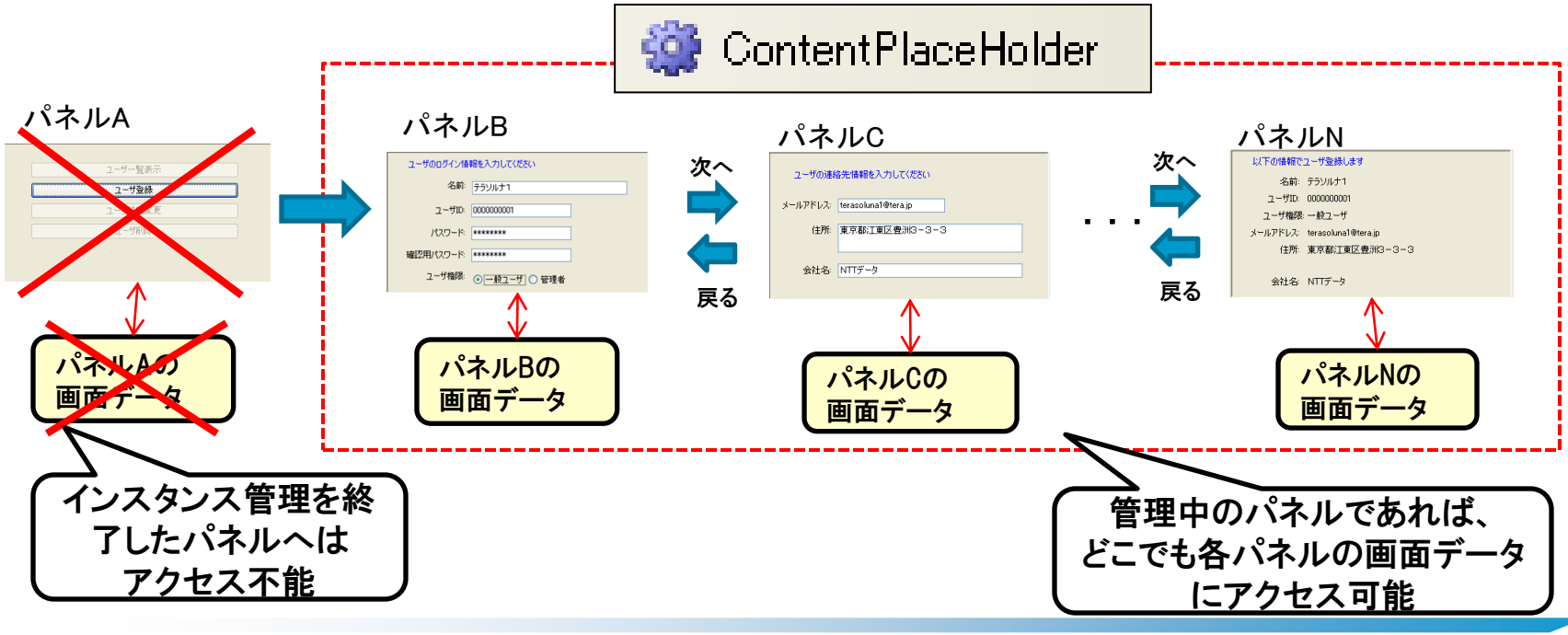
画面遷移機能の検討経緯（画面間のデータ引き継ぎ）

- ContentPlaceHodlerを使ったデータの引き継ぎ



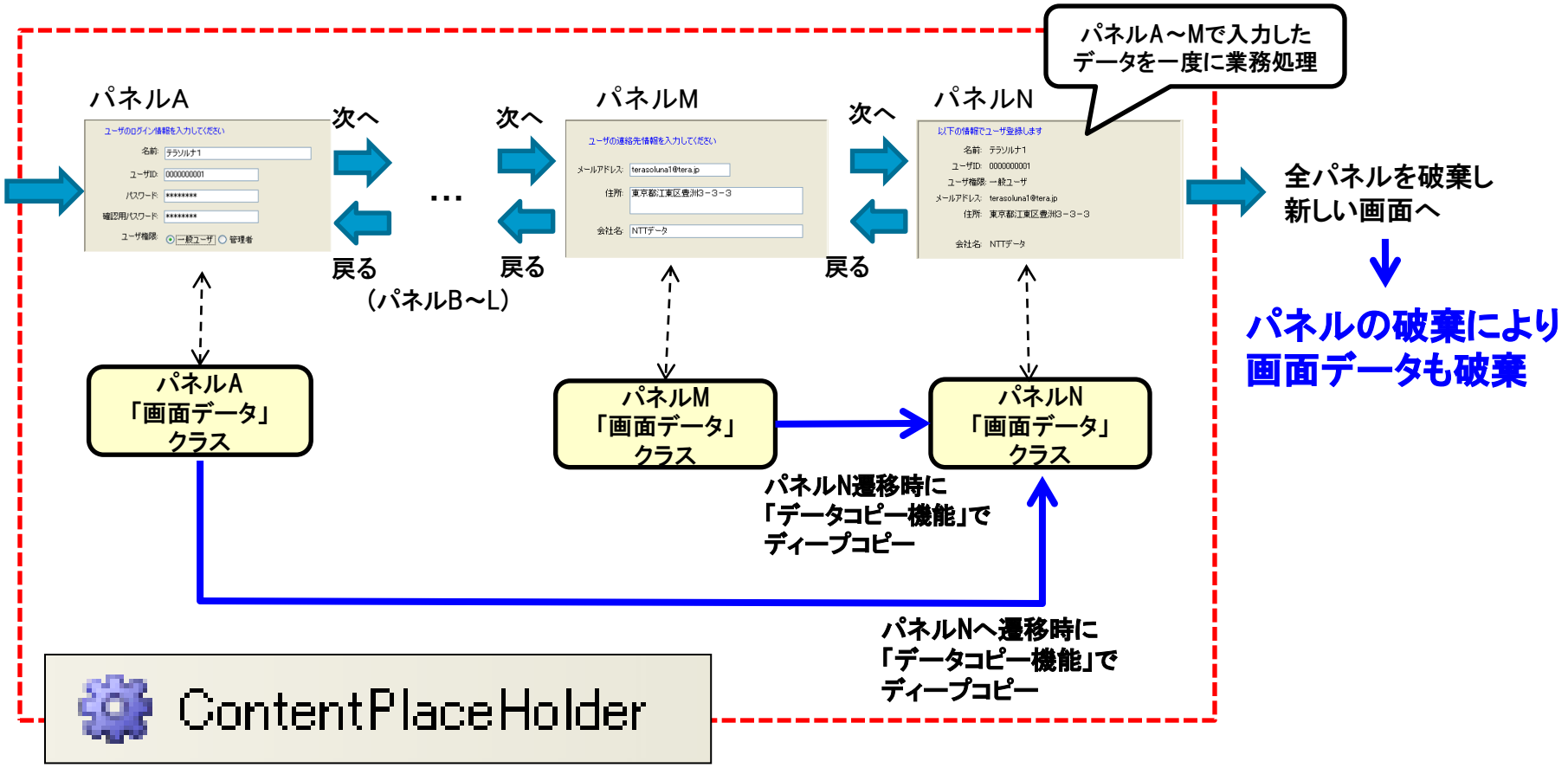
画面遷移機能の検討経緯（画面間のデータ引き継ぎ）

- ContentPlaceHolderによるインスタンス管理されたパネルの画面データ
  - インスタンス管理中のパネルであれば、任意の場所でパネルに対する「画面データ」を取得可能
  - FormForwarderの「スコープ」内共通領域に相当する機能
  - 「ContentPlaceHolder.GetContentControl<パネルの型>().ViewData」



ContentPlaceHolderの利用イメージ

一連の業務で「画面データ」を引き継ぐ例



ContentPlaceHolderの利用イメージ

- 最後のパネルで、各画面の「画面データ」を取得
  - パネルの表示時に実行されるメソッド（IContentControl.HandleShowingメソッド）で実装
  - DataCopyManager.Copyメソッドで「画面データ」をコピー

最後のパネル（パネルNに相当）での実装例

```
public partial class SC_B01_03_03Control : UserControl, ISC_B01_03_01MainContent, IContentControl
{
 ・・・
   public void HandleShowing(object sender, ControlEventArgs e)
   {
      //ContentPlaceHolderより、任意の場所でインスタンス管理中のパネルを取得可能
      SC_B01_03_01Control sc_B01_03_01Control =
                ContentPlaceHolder.GetContentControl<SC_B01_03_01Control>();
      SC_B01_03_02Control sc_B01_03_02Control =
                ContentPlaceHolder.GetContentControl<SC_B01_03_02Control>();
      //現在のパネルの画面へコピー
      DataCopyManager.Copy(sc_B01_03_01Control.ViewData, ViewData);
      DataCopyManager.Copy(sc_B01_03_02Control.ViewData, ViewData);
   }
}
```

ContentPlaceHolderの利用イメージ

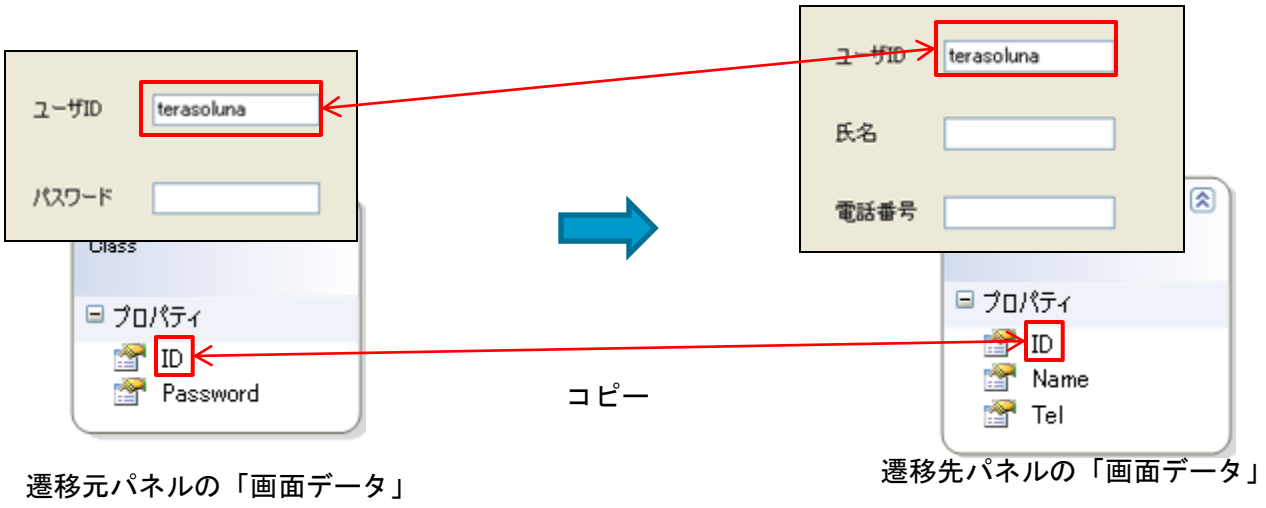
- 業務完了時、パネルを全てクローズする
  - パネルとともに「画面データ」も自動的に消滅
  - FormForwarderにおける「スコープ」の消滅と等価

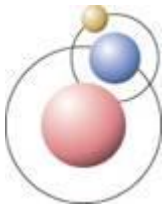
最後のパネル（パネルNに相当）での実装例

```
/// 登録ボタン押下時のイベント処理完了時
private void b01_03_03_C01EventProcessWorker_Completed(object sender,
                                EventProcCompletedEventArgs e)
{
    if (e.Result.IsSuccess)
    {
      //業務処理成功時には、パネル全て破棄し、新しいパネルへ遷移
      ContentPlaceHolder.ShowWithCloseAllContents(typeof(SC_B01_03_00Control));
    }
    else if (e.Result.IsAnyBizLogicError || e.Result.IsAnyValidationError)
    {
       //業務エラー時には、入力開始したパネルへ戻る
       ContentPlaceHolder.ShowAndHideContent(typeof(SC_B01_03_01Control));
    }
}
```

遷移元・遷移先パネル間の画面データのコピー

- 遷移元パネルと遷移先パネル間でデータの引き継ぎ
  - 遷移元・遷移先パネルの「画面データ」を「データコピー機能」を使ってコピー処理を実装
  - 遷移先パネルのIContentControl.HandleShowingメソッドで実装





ContentPlaceHolderの利用イメージ

- HandleShowingで、遷移元、遷移先パネル間の「画面データ」をコピー
  - 遷移元パネルの取得
    - ContentPlaceHolder.LastContentControlプロパティ
  - 遷移先パネルの取得
    - ContentPlaceHolder.CurrentContentControlプロパティ
  - DataCopyManager.Copyメソッドでコピー処理を実装

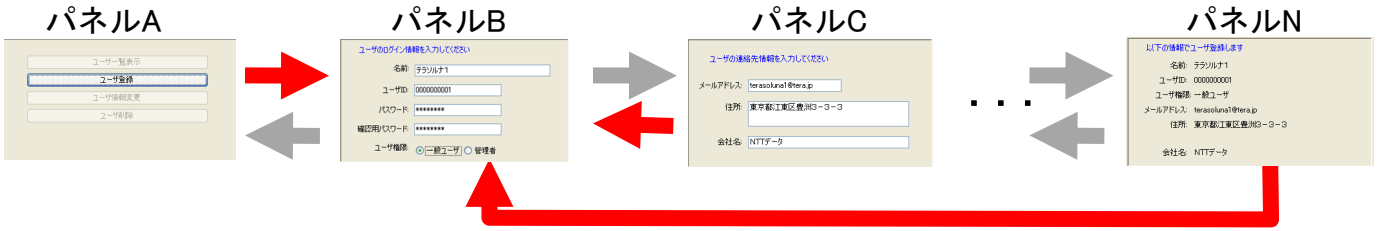
遷移先パネルでのHandleShowingメソッドの実装例

```
public partial class SC_B01_03_01Control : UserControl,  ISC_B01_03_01MainContent, IContentControl
{
 ・・・
   public void HandleShowing(object sender, ControlEventArgs e)
   {
      ///遷移元遷移先パネル間で画面データをコピーして引き継ぐ例
      ISC_B01_03_01MainContent lastContentControl =
         ContentPlaceHolder.LastContentControl as ISC_B01_03_01MainContent;
      ISC_B01_03_01MainContent currentContentControl =
         ContentPlaceHolder.CurrentContentControl as ISC_B01_03_01MainContent;
      DataCopyManager.Copy(lastContentControl.ViewData, currentContentControl.ViewData;
   }
}
```

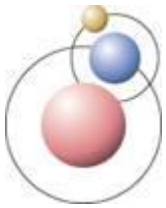

TERASOLUNA

画面データコピー時の注意点

- IContetntControl.HandleShowingメソッドは、パネルが表示されたときに無条件に呼び出されるため、遷移元パネルが複数存在し、遷移元によって処理を分岐させたい場合は、遷移元画面の判定処理の実装が必要



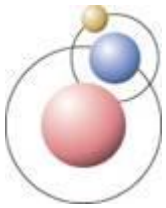

```
public void HandleShowing(object sender, ControlEventArgs e)
{
    ///遷移元パネルの取得
    ISC_B01_03_01MainContent lastContentControl = ContentPlaceHolder.LastContentControl as ISC_B01_03_01MainContent;
    if (lastContentControl is SC_B01_03_00Control)
    {
        //遷移元がパネルAの場合の処理
    }
    else if (lastContentControl is SC_B01_03_02Control)
    {
        //遷移元がパネルCの場合の処理
    }
    else if (lastContentControl is SC_B01_03_02Control)
    {
        //遷移元がパネルNの場合の処理
    }
}
```



FormFowarderとContentPlaceHolderの比較

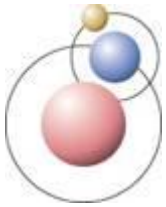
- FormForwarderとContentPlaceHolderは、ほぼ同等のデータ引き継ぎ機能を実現可能
  - ただし、ContentPlaceHolderには、「スコープ」の概念やデータコピーの自動実行機能はない

データの種別	FormFowarder	ContentPlaceHolder
AP全体で共有するデータ	Singleton等、開発者が実装する共通データクラス	Singleton等、開発者が実装する共通データクラス
一連の業務を形成する画面遷移グループ内で共有するデータ	スコープ共通データ領域	ContentPlaceHolderにインスタンス管理されたパネルの画面データ
遷移画面間の引き継ぐデータ	「データコピー機能」による画面データのコピー ※FormForwarderによるAP自動コピー	「データコピー機能」による画面データのコピー ※開発者による明示的な実装



（参考）アプリケーション全体で共有するデータ

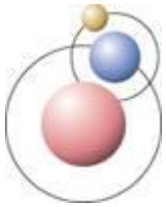
- アプリケーション全体で共有するデータについては、Singletonなどstaticな「共通データ」クラスを開発者が実装する
  - TERASOLUNAフレームワークは、機能提供しない
  - 「画面データ」から、「共通データ」にコピーする場合など、共有する項目が多い場合には、「データコピー機能」を利用する
  - FormForwarder、ContentPlaceHolderのどちらを使用しても、考え方は同じ



クライアントエラーハンドリング機能

クライアントエラーハンドリング機能の責務

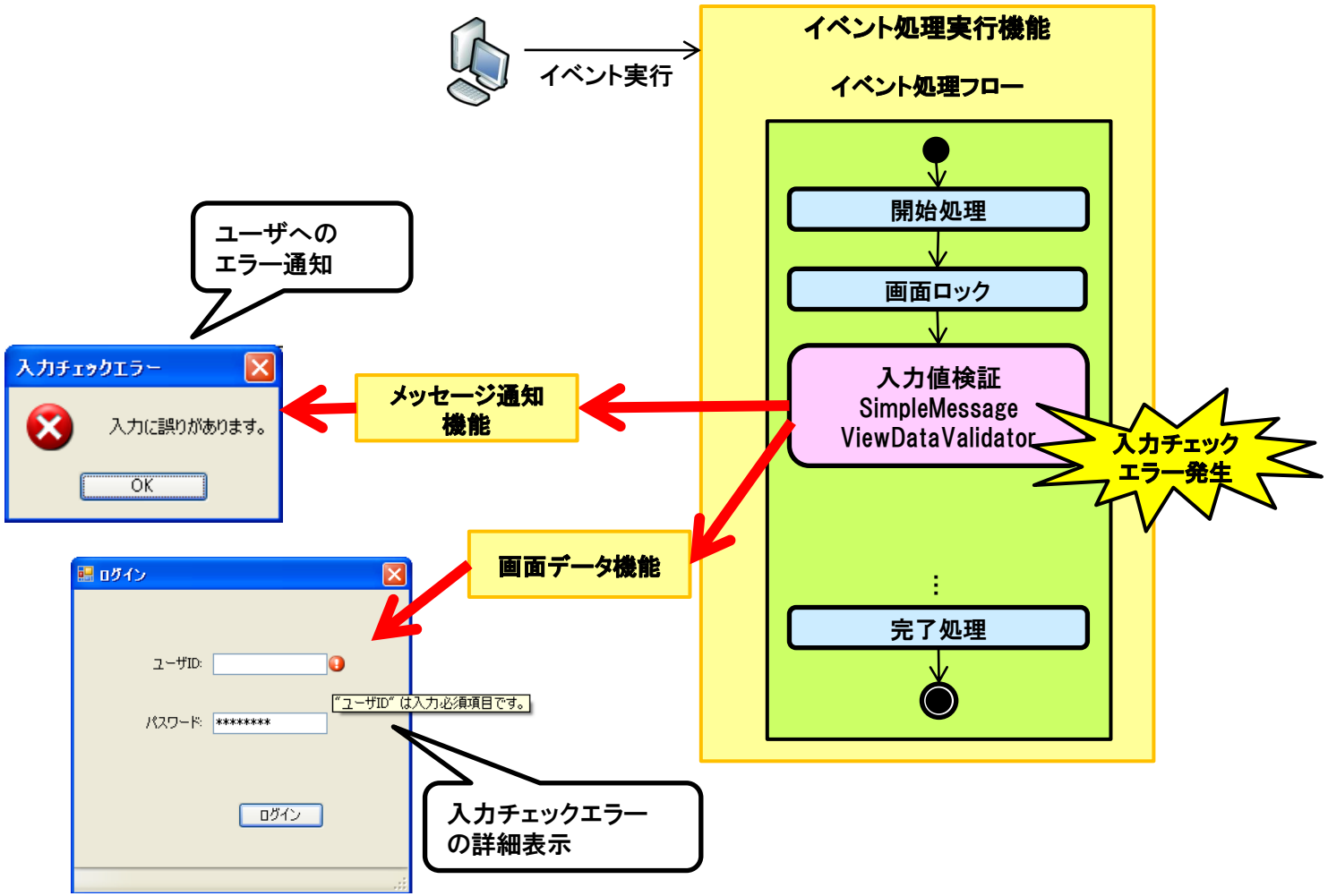
- システムで発生したエラーを集約例外処理
  - 入力値検証エラー
    - 必須項目や入力形式のチェックなど画面の入力値のみでチェックできるエラー
    - ユーザの再入力により復帰可能
    - 業務エラーの一種であるが処理方式が異なるので区別
  - 業務エラー
    - DBとの突き合わせや複雑なビジネスルールによるチェックなど、ビジネスロジック実行時に発生するエラー
    - ユーザの再入力により復帰可能
  - システムエラー
    - DBやネットワークのエラー、バグなどシステムやAPの不具合による復帰不可能なエラー

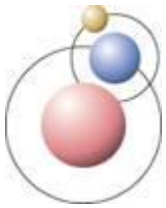


クライアントエラーハンドリング機能の検討経緯（入力値検証エラー）

- 入力値検証エラー
  - 「画面データ」に対して入力値検証を実施
    - 前述の「画面データ機能」を参照
  - エラー発生時にイベント処理フローの実行を中断（デフォルト）
  - Validation ABとIDataErrorInfoによるエラー詳細表示

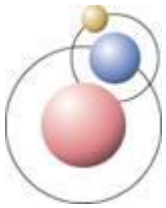
画面入力値検証エラーのハンドリング





クライアントエラーハンドリング機能の検討経緯（業務エラー）

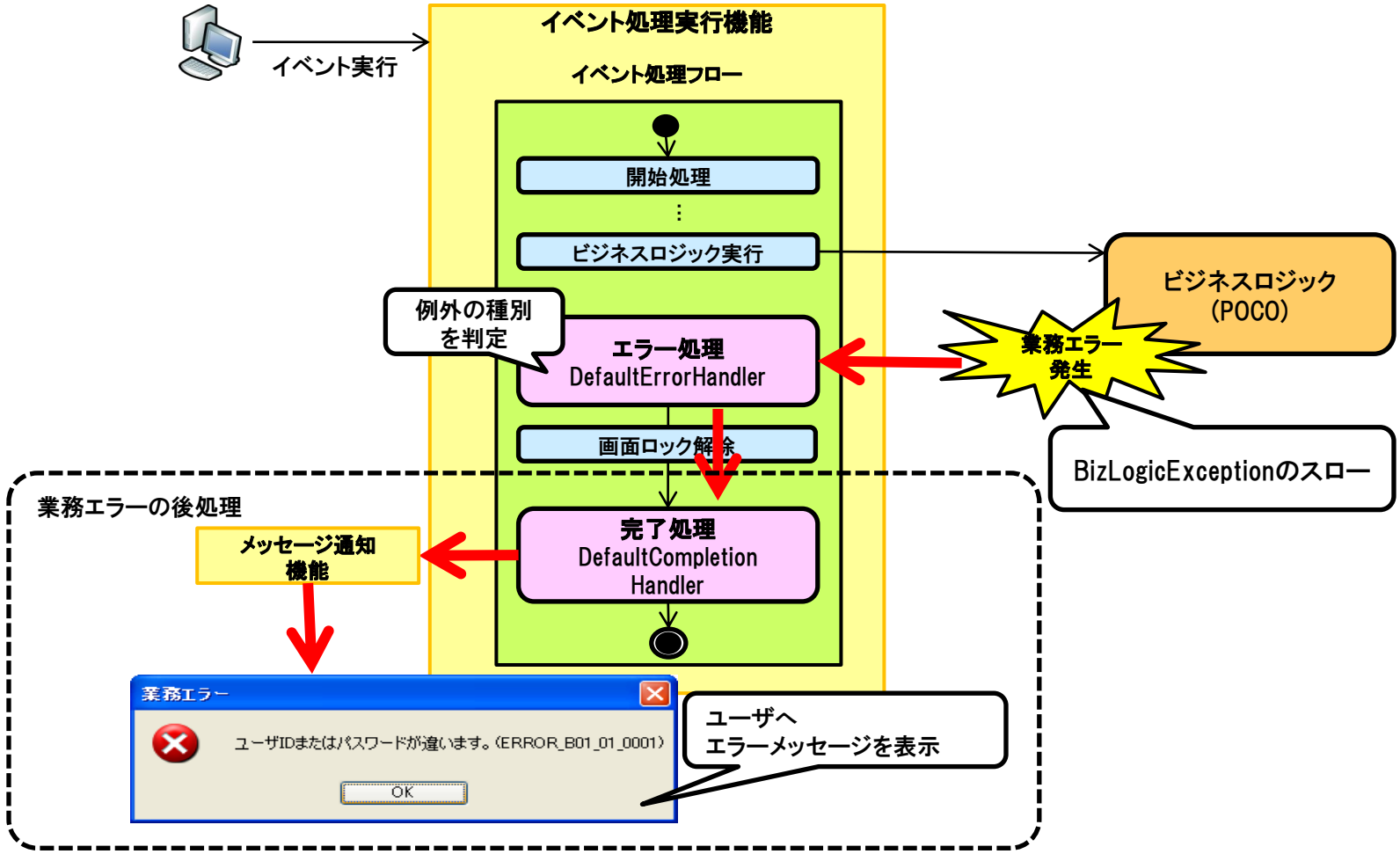
- 業務エラー
  - 通常、メソッドの「戻り値」で取り扱う場合が多い
    - .NETでは検査例外がないため
- TERASOLUNAフレームワークでは、ビジネスロジッククラスをPOCOで開発
  - メソッドの戻り値の形をフレームワークで縛ることができない
  - エラーを判定する共通的なデータを取得できなければ、集約的な例外処理ができない

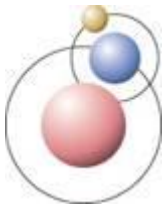


クライアントエラーハンドリング機能の検討経緯（業務エラー）

- 業務エラーは「例外」で取り扱う
  - 業務エラー専用の例外としてBizLogicExceptionクラスを提供
  - フレームワークは、イベント処理内で発生した例外の種類を判断し、システムエラー、業務エラー等に応じた例外処理を実施
    - BizLogicExceptionの場合、業務エラーとして処理しリスローしない
  - 捕捉したエラーを同じ仕組みで処理
    - デフォルトでは、発生したエラーのメッセージとエラーID（エラーコード）をダイアログに表示

業務エラーのハンドリング

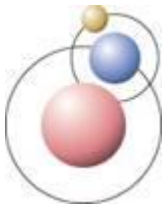




業務エラーの実装イメージ

- 業務開発者は、業務エラー発生時に、BizLogicExceptionをスローする

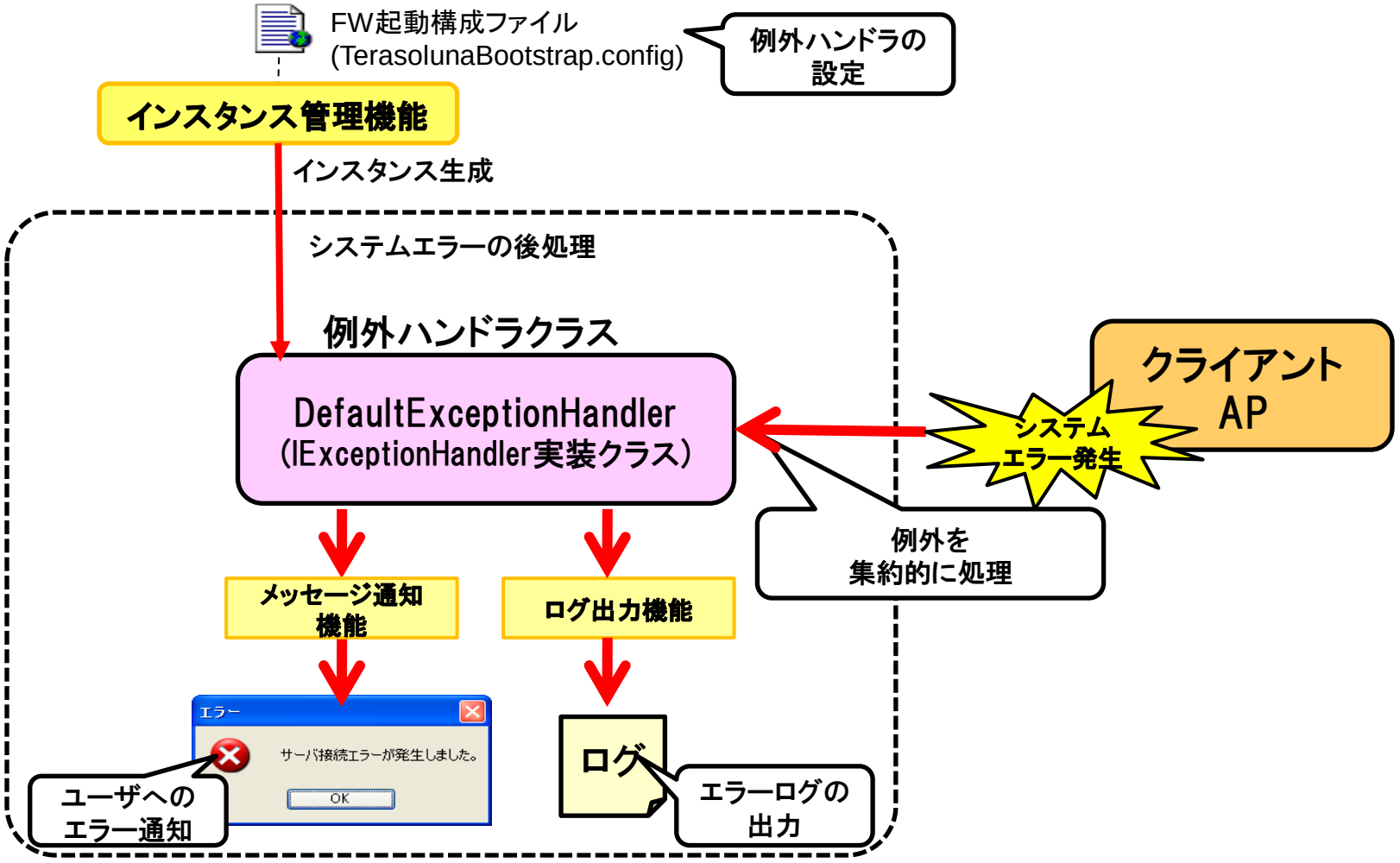
```
public class B01_01_01_C01BizLogic
{
    public void Execute(B01_01_01_C01InputDto inputDto)
    {
            . . .
                throw new BizLogicException(
                        BizLogicExceptionErrorType.BizLogicFailure,
                        ResourceCommon.APP_ERROR,
                        new List<ErrorInfo>()
                        {
                            new ErrorInfo("ERROR_B01_01_0001", null,
                                        ResourceB01.ERROR_B01_01_0001, null)
                         });
    }
}
```



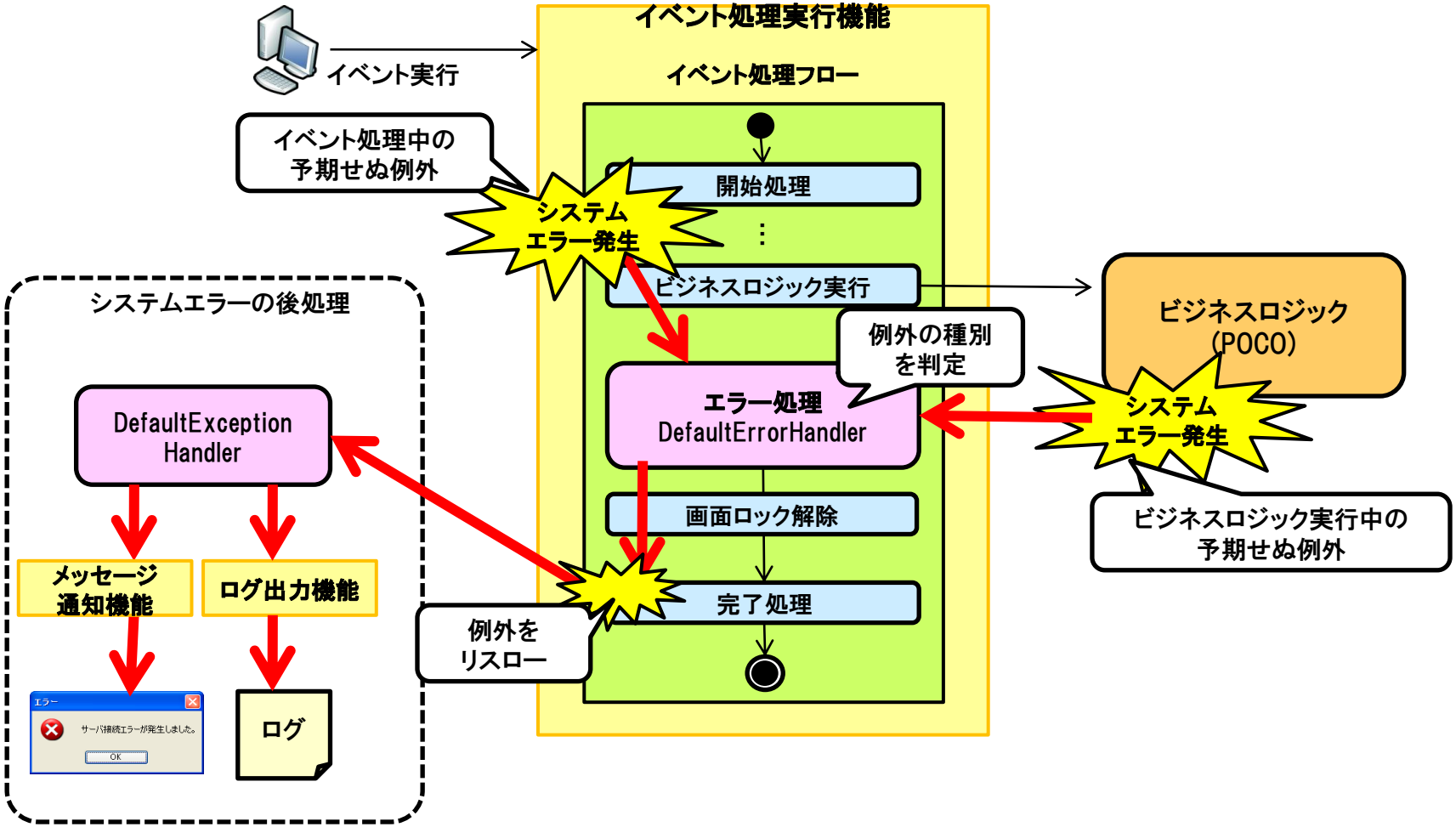
クライアントエラーハンドリング機能の検討経緯（システムエラー）

- システムエラー
  - 捕捉されなかった例外(System.Exception継承クラス)
- 捕捉した例外を同じ仕組みで処理
  - Terasoluna.ExceptionHandling.IErrorHanlderインタフェース
  - デフォルトでは、エラーログの出力とエラーダイアログの表示
  - 捕捉箇所
    - AppDomain.CurrentDomain.UnhandledExceptionイベント
    - Application.ThreadExceptionイベント
    - イベント処理実行機能の非同期処理で発生した例外
      - プールスレッド上で発生した例外は、AppDomain.UnhandledExceptionイベントでは捕捉できない
      - Form(Control)クラスのBeginInvokeメソッドでUIスレッド側へリスロー
    - フレームワーク初期化処理
    - フレームワーク終了処理

システムエラーのハンドリング

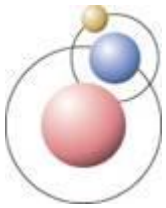


システムエラーのハンドリング（イベント処理）



エラーハンドリング機能の検討経緯(サーバエラー)

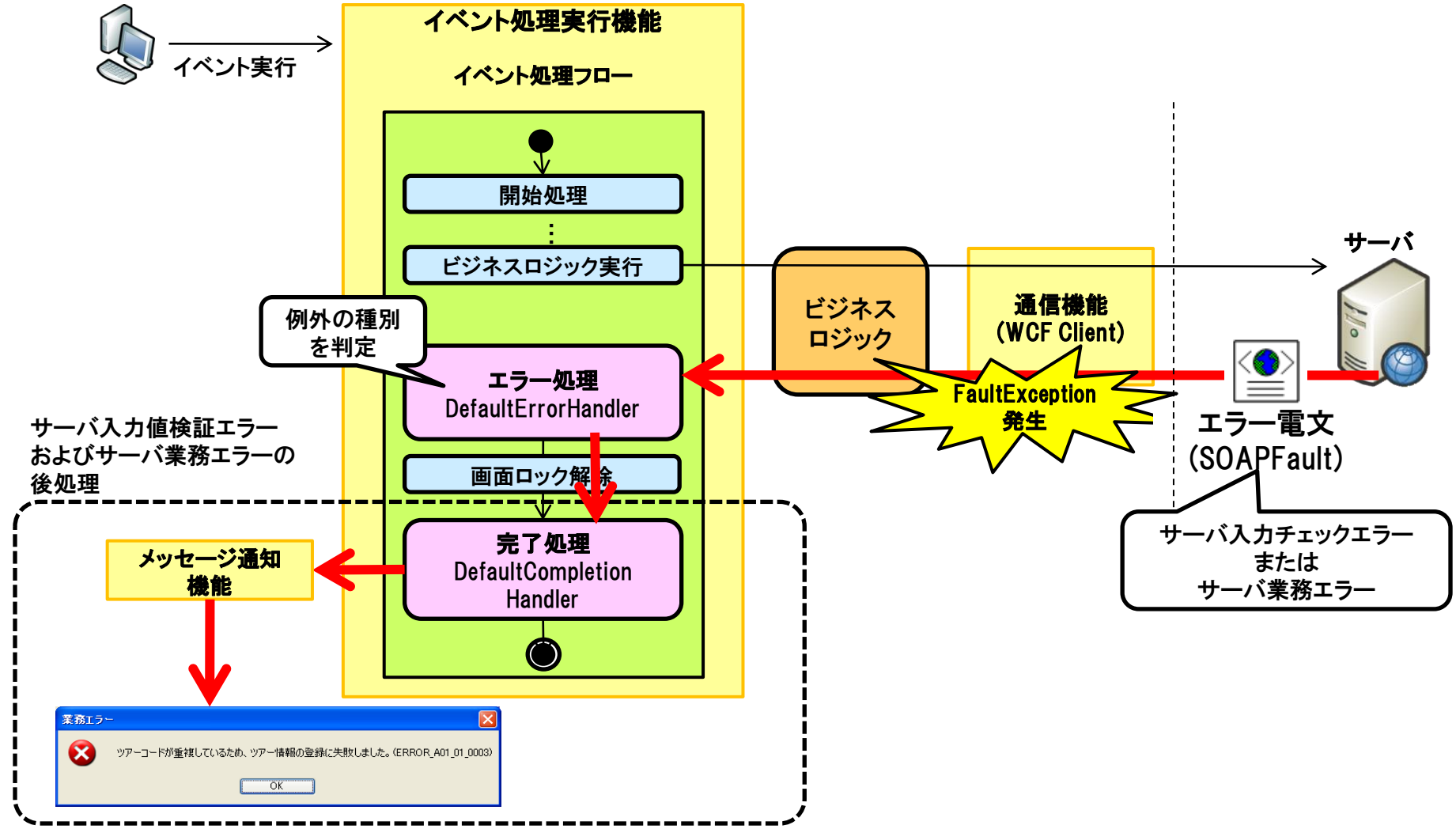
- SOAP Fault
  - サーバからのエラーレスポンス
  - 業務エラーは正常のレスポンスで扱うことも多い
- TERASOLUNAフレームワークでは、全てのサーバエラーを「SOAP Fault」で取り扱う
  - クライアント同様、サーバもPOCOベースの開発を想定
  - エラーの種類によらずSOAP Faultで返却することで、クライアントでの集約的な例外処理が可能
  - SOAP Faultの<detail>要素の内容に応じて、入力値検証エラー、業務エラー、システムエラーかを判定



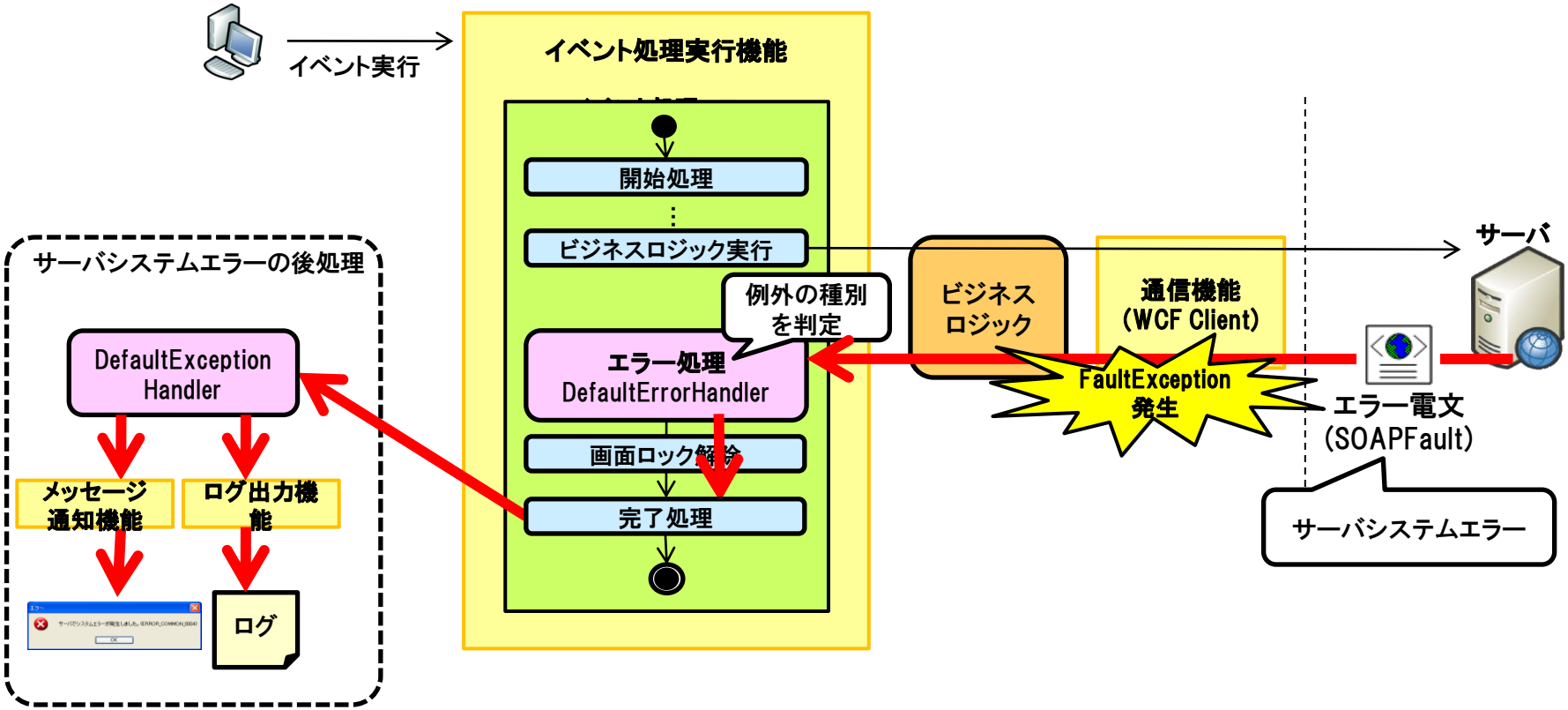
エラーハンドリング機能の検討経緯(サーバエラー)

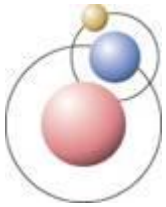
- FaultException(FaultException<T>)クラス
  - SOAP Faultが返却された場合にWCFクライアントで発生する例外
  - WCFサービスでFaultContract(JavaのJAX-WSならFaultBean)を定義していると、SOAP Faultの<detail>要素に詳細なエラー情報を格納し返却
    - クライアント側で<detail>要素の内容を解析可能
      - FaultException<T>.Detailプロパティ
- サーバから決まった形式をもつFaultContract(JAX-WSならFaultBean)を返却すれば、クライアントは、<detail>要素の中のエラー情報を確認し、集約的な例外処理が可能
  - エラー種類、エラーコード、エラーメッセージ
  - エラー種類をもとに適切な集約例外処理を実施
    - 入力値検証エラー、業務エラー、システムエラー
  - <detail>要素が解析できない場合、システムエラーとして扱う

サーバ入力チェックエラー・業務エラーのハンドリング

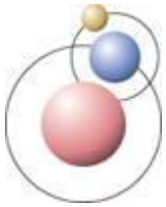


サーバシステムエラーのハンドリング



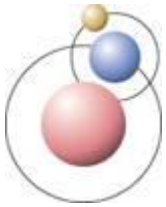


サーバフレームワーク機能概要



サーバフレームワーク機能概要

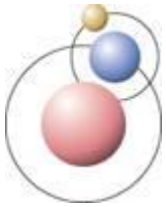
- ここでは、サーバフレームワークのアーキテクチャに大きく関わる主要な機能の検討経緯や動作概念を説明し、各機能の理解を深めることを目的とする



サーバフレームワーク機能概要 - 目次

- WCFサービス管理機能
- サーバ入力値検証機能
- サーバエラーハンドリング機能
- データアクセス機能

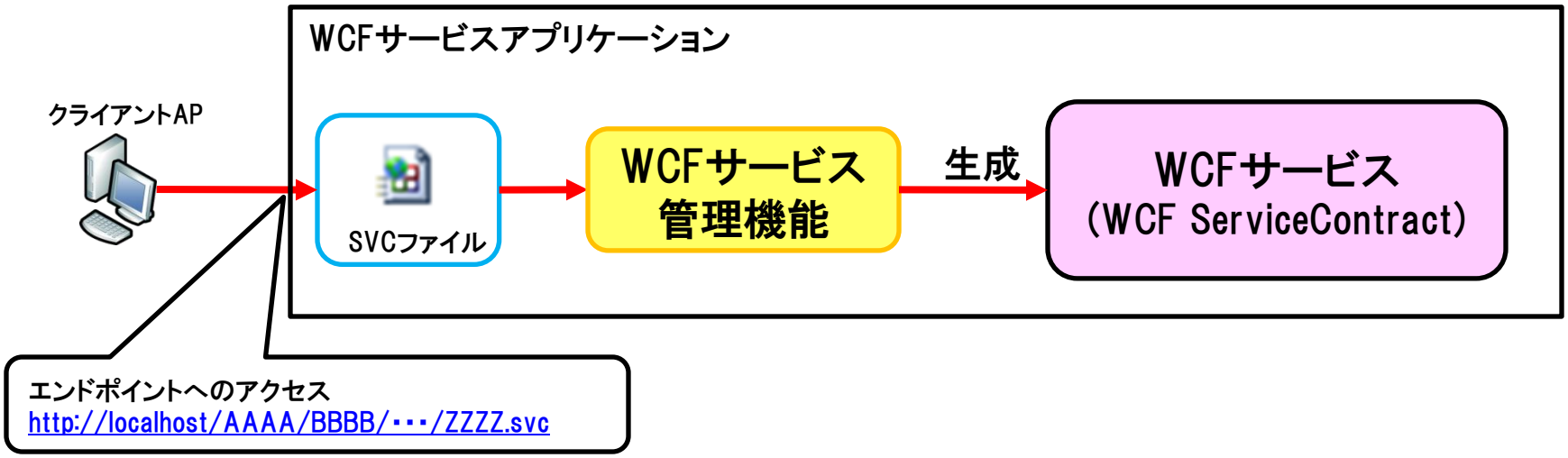
TERASOLUNA

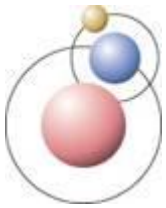


WCFサービス管理機能

WCFサービス管理機能の責務

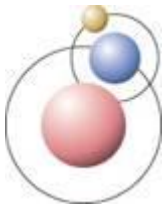
- WCFサービスのインスタンス管理
  - Webサービス等の実現にWCFサービスを使用
  - TERASOLUNAフレームワークにより機能追加されたWCFサービスを生成／起動





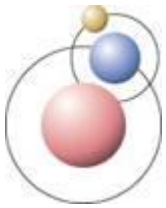
WCFサービス管理機能の責務

- バインディング設定の簡略化
  - 命名規約に基づき、URIとWCFサービスのマッピング
    - WCFサービスおよびエンドポイントの設定を省略可能
  - システムで既定のバインディング設定を提供
    - 個々のWCFサービスのバインディング設定を省略可能
    - 典型的な通信処理設定のパターンを用意しておくことで、エンドポイントの設定にかかる作業コストを削減



WCFサービス管理機能の検討経緯(WCFの採用)

- 相互運用性の考慮
  - TERASOLUNAフレームワーク独自の通信基盤からの脱却
    - 前バージョン(ver2.x)では、HTTPをベースとした独自の通信方式であったため、TERASOLUNAフレームワークを使ったAP間でしか接続できなかった
  - WCFの採用
    - 標準的なWCFサービスに対応し言語やプラットフォームに依存せず接続可能
    - システムの接続機会を増やし新たなビジネス機会を創造できる
    - システム開発における柔軟性も高く、少ないコストでシステムの統合が図れる
- 実装スタイルの統一
  - WCFにより通信プロトコルに依存せず同一の実装スタイルになる
  - システムの非機能要件に合わせて通信プロトコルを変更可能
    - Web構成ファイル(Web.config)の設定で実現
    - 複数のエンドポイントを用意することも可能
      - イントラの.NETアプリ向けにnetTcpBinding、外部システム向けにはbasicHttpBindingといった複数のチャネルを提供可能

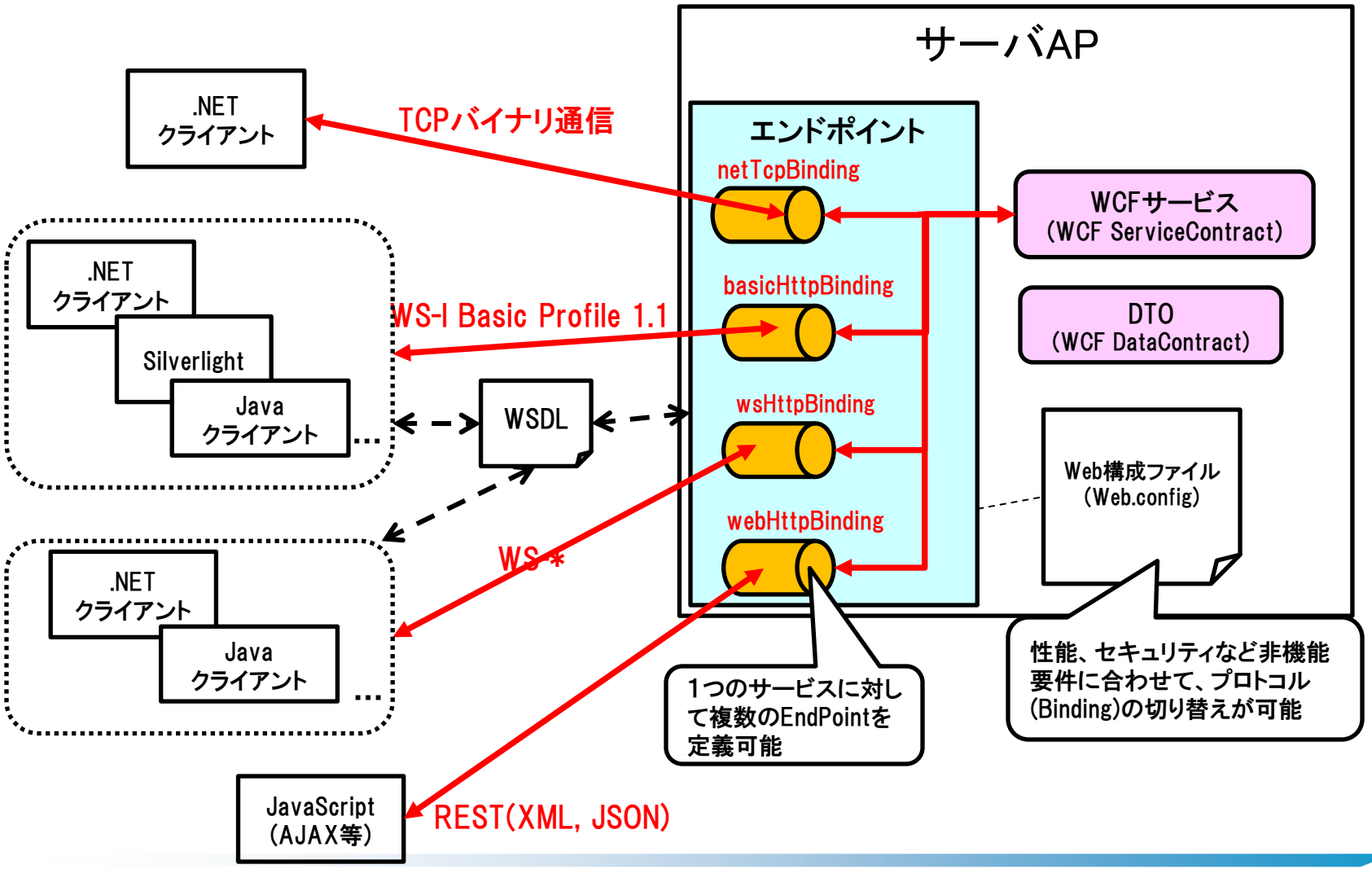


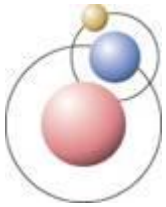
WCFサービスの実装イメージ

- ServiceContract属性やOperationContract属性を付与

```
[ServiceContract]
public class CalcService
{
    [OperationContract]
    [ValidationBehavior(RuleSet = "RS01")]
    public CalcOutputDto Add(CalcInputDto inputDto)
    {
        int result = inputDto.Num1 + inputDto.Num2;
        return new CalcOutputDto() { Answer = result };
    }
}
```

サーバAPから見たWCFによる通信





WCFサービス管理機能の検討経緯（Web.configの設定簡略化）

- エンドポイントの設定を省略した場合、命名規約により、URIからWCFサービスクラスを特定するようWCFを拡張
  - クライアントAPが指定するエンドポイントのURI
    - http://localhost/AAAA/BBBB/・・・/ZZZZ.svc
      - svcファイルのアドレスを指定
  - ベースとなる名前空間名
    - FW構成ファイルで「BaseNamespace」の値を設定
  - 対応するWCFサービスクラス名
    - (BaseNamespace).AAAA.BBBB.・・・.ZZZZ
  - 対応するWCFサービスコントラクト
    - WCFサービスクラス自身または実装するインタフェースの中で、[ServiceContract]属性を定義しているもの

WCFサービス管理機能の検討経緯（Web.configの設定簡略化）

- 典型的な通信処理パターンごとにServiceHostFactory継承クラスを提供
  - 開発者はパターンに応じたServiceHostFactoryを選択するだけでよい
    - ServiceHostFactoryは、対象のWCFサービスのSVCファイルに記述
  - バインディングやビヘイビアの設定を省略した場合、ServiceHostFactoryの種類ごとにデフォルト設定が適用されるようWCFを拡張
    - ServiceHostFacotryで対応するコンフィグレーションのname属性を定義
  - ServiceHostFactoryクラスとExtensionクラスを実装することで、デフォルト設定のバリエーションを追加可能

通信処理パターン	用途	使用するServiceHostFactory	binding	message Encoding	transfer Mode
デフォルト	標準的な通信	DefaultServiceHostFactory	wsHttpBinding	Text	-
MTOM	ファイルアップロード・ダウンロード(オンメモリ)	MtomServiceHostFactory	wsHttpBinding	Mtom	-
MTOM ストリミング転送モード	ファイルアップロード・ダウンロード(ストリーミング)	MtomStreamedServiceHostFactory	basicHttpBinding	Mtom	Streamed

SVCファイルの記述イメージ

- @ServiceHostディレクティブのFactory属性
  - 開発者は対象のServiceHostFactoryクラスを設定
- SVCファイルをカスタムテンプレート化
  - 標準提供のServiceHostFactoryを使用する場合は、開発者によるSVCファイルの記述は不要

デフォルトのWCFサービス用のSVCファイル

```
<%@ ServiceHost Language="C#" Debug="true"
     Factory="Terasoluna.Server.ServiceModel.DefaultServiceHostFactory" %>
```

MTOMを使用するWCFサービス用のSVCファイル

```
<%@ ServiceHost Language="C#" Debug="true"
       Factory="Terasoluna.Server.ServiceModel.MtomServiceHostFactory" %>
```

MTOM ストリーミング転送モードを使用するWCFサービス用のSVCファイル

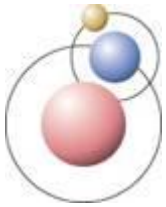
```
<%@ ServiceHost Language="C#" Debug="true “
Factory="Terasoluna.Server.ServiceModel.MtomStreamedServiceHostFactory" %>
```

Web.configの記述イメージ

- カスタムテンプレートによるプロジェクト作成時にServiceHostFactoryに対応する設定のひな形が生成済なので設定作業の負荷を軽減

```
<system.serviceModel>
  <bindings>
   <wsHttpBinding>
    <!-- DefaultServiceHostFactoryのBindingの設定（デフォルト） -->
    <binding name="DefaultBinding" />
    <!-- MtomServiceHostFactoryのBindingの設定(Mtom) -->
    <binding name="MtomBinding" messageEncoding="Mtom”  ・・・maxReceivedMessageSize=" 10485760 ">
     <readerQuotas ・・・   maxArrayLength=" 10485760 "   ・・・ />
    </binding>
   </wsHttpBinding>
   <basicHttpBinding>
    <!-- MtomStreamedServiceHostFactoryのBindingの設定（Mtomストリーミング転送モード） -->
    <binding name="MtomStreamedBinding" messageEncoding="Mtom“ transferMode="Streamed"   ・・・
          maxBufferSize="65536" maxReceivedMessageSize="10485760"/>
   </basicHttpBinding>
  </bindings>
  <behaviors>
   <serviceBehaviors>
    <!-- デフォルトのServiceBehaviorの設定 -->
    <behavior name=“DefaultServiceBehavior“>・・・</behavior>
   </serviceBehaviors>
   <!-- デフォルトのEndpointBehaviorの設定 -->
   <endpointBehaviors>
    <behavior name=“DefaultEndpointBehavior“>・・・</behavior>
   </endpointBehaviors>
  </behaviors>
</system.serviceModel>
```

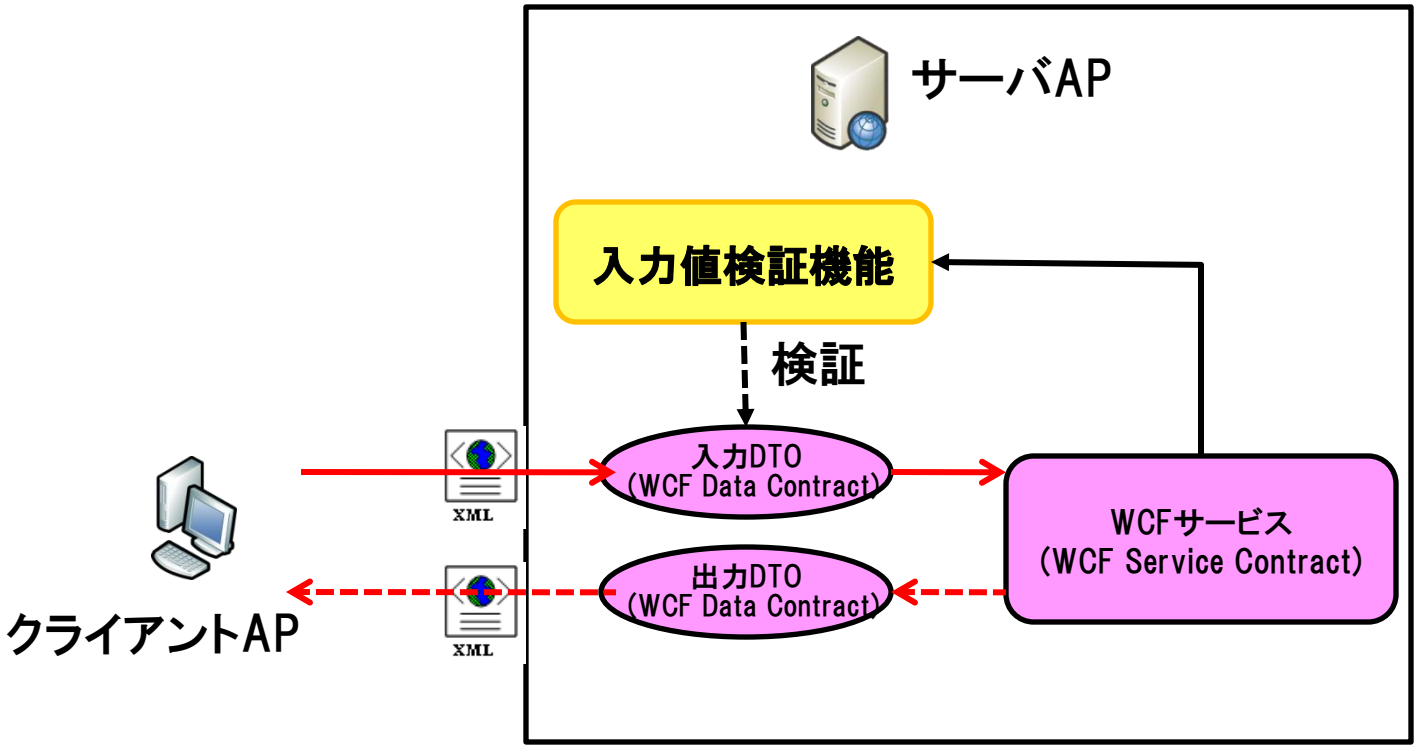
アーキテクトは、性能やセキュリティなど、非機能要件にもとづき、適切なパラメータ値に、設定値を変更する

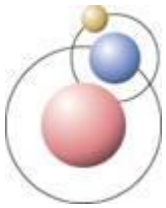


サーバ入力値検証機能

サーバ入力値検証機能の責務

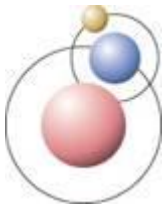
- サーバ入力DTOを検証
  - 検証エラー時にはクライアントへエラー電文を返却





サーバ入力値検証機能の検討経緯

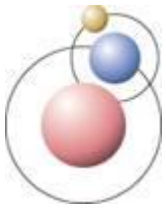
- クライアント(「画面データ」)と同じ実装スタイルにしたい
  - Validation ABを使用して、入力DTOに対してカスタム属性を付与
- Validation ABのWCF Integrationの利用
  - WCFサービス呼び出し時に、ビジネスロジック実行前に入力値検証を自動的に実施
  - 検証エラー時には、FaultException<ValidationFault>をスロー
    - 後述の「サーバエラーハンドリング機能」を参照
      - FaultExcetion<TerasolunaSoapFault>へ変換
      - <detail>要素にエラー情報を格納したSOAP Faultを返却
        - » クライアントで集約的にエラーハンドリング可能



サーバ入力値検証処理の実装イメージ

入力DTO(DataContract)にカスタム属性を付与

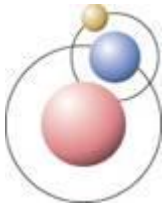
```
[DataContract]
[HasSelfValidation]
public class CalcInputDto
{
    [DataMember]
    [IntRangeValidator(Tag = "数値1", Ruleset = "RS01“, LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive)]
    public int Num1 { get; set; }

    [DataMember]
    [IntRangeValidator(Tag = "数値2", Ruleset = "RS01“, LowerBound = 0, LowerBoundType = RangeBoundaryType.Inclusive)]
    public int Num2 { get; set; }

    /// カスタム入力チェック
    [SelfValidation(Ruleset ="RS01")]
    private void CustomValidate01(ValidationResults results)
    {
        ///数値１と数値２が同じ数だとエラーとする
        if (Num1 == Num2)
        {
            ValidationResult resultForNum1 =
                new ValidationResult(CalcResource.ERROR_CALC_MSG001, this, string.Empty, string.Empty, null);
            results.AddResult(resultForNum1);
        }
    }
}
```



サーバ入力値検証処理の実装イメージ

- WCFサービスインタフェースのメソッドに、ValidationBehavior属性を付与
  - メソッド実行時に適用するルールセット名を指定

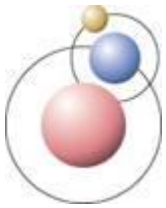
```
[ServiceContract]
public class CalcService
{
    [OperationContract]
    ///ValidatioinBehavior属性の付与
    [ValidationBehavior(RuleSet = "RS01")]
    public CalcOutputDto Add(CalcInputDto inputDto)
    {
        int result = inputDto.Num1 + inputDto.Num2;
        return new CalcOutputDto() { Answer = result };
    }
}
```



サーバエラーハンドリング機能

サーバエラーハンドリング機能の責務

- システムで発生したエラーを集約例外処理
  - 入力値検証エラー
    - 必須項目や入力形式のチェックなど要求電文の値のみでチェックできるエラー
    - クライアントからの再入力により復帰可能
    - 業務エラーの一種であるが処理方式が異なるので区別
  - 業務エラー
    - DBとの突き合わせや複雑なビジネスルールによるチェックなど、ビジネスロジック処理時に発生するエラー
    - クライアントからの再入力により復帰可能
  - システムエラー
    - DBやネットワークのエラー、バグなどシステムやAPの不具合による復帰不可能なエラー



サーバエラーハンドリング機能の検討経緯

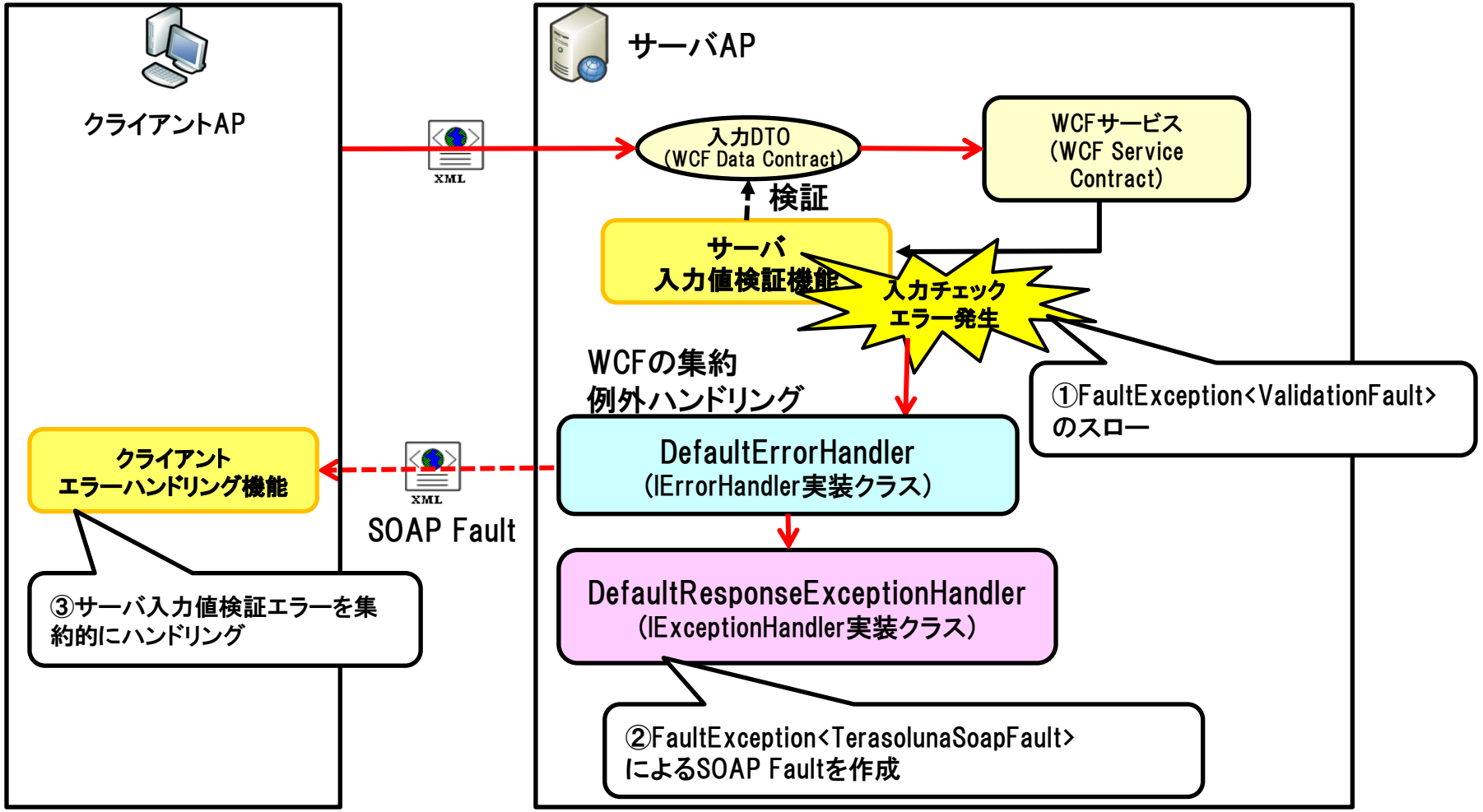
- TERASOLUNAフレームワークでは、全てのサーバエラーを「SOAP Fault」で取り扱う
  - エラー情報を保持するFaultContractとしてTerasolunaSoapFaultクラスを使用
    - <detail>要素にエラー情報を格納
    - FaultException<TerasolunaSoapFault>によるSOAP Faultを生成
- WCFの集約例外処理
  - System.ServiceModel.Dispatcher.IErrorHandlerインタフェース
    - SOAP Faultの作成や、ログ出力などの後処理を実施可能
  - TERASOLUNAフレームワークの集約例外処理クラスへ委譲
    - クライアント・サーバの集約例外処理を同一の実装スタイルにするため

サーバエラーハンドリング機能の検討経緯

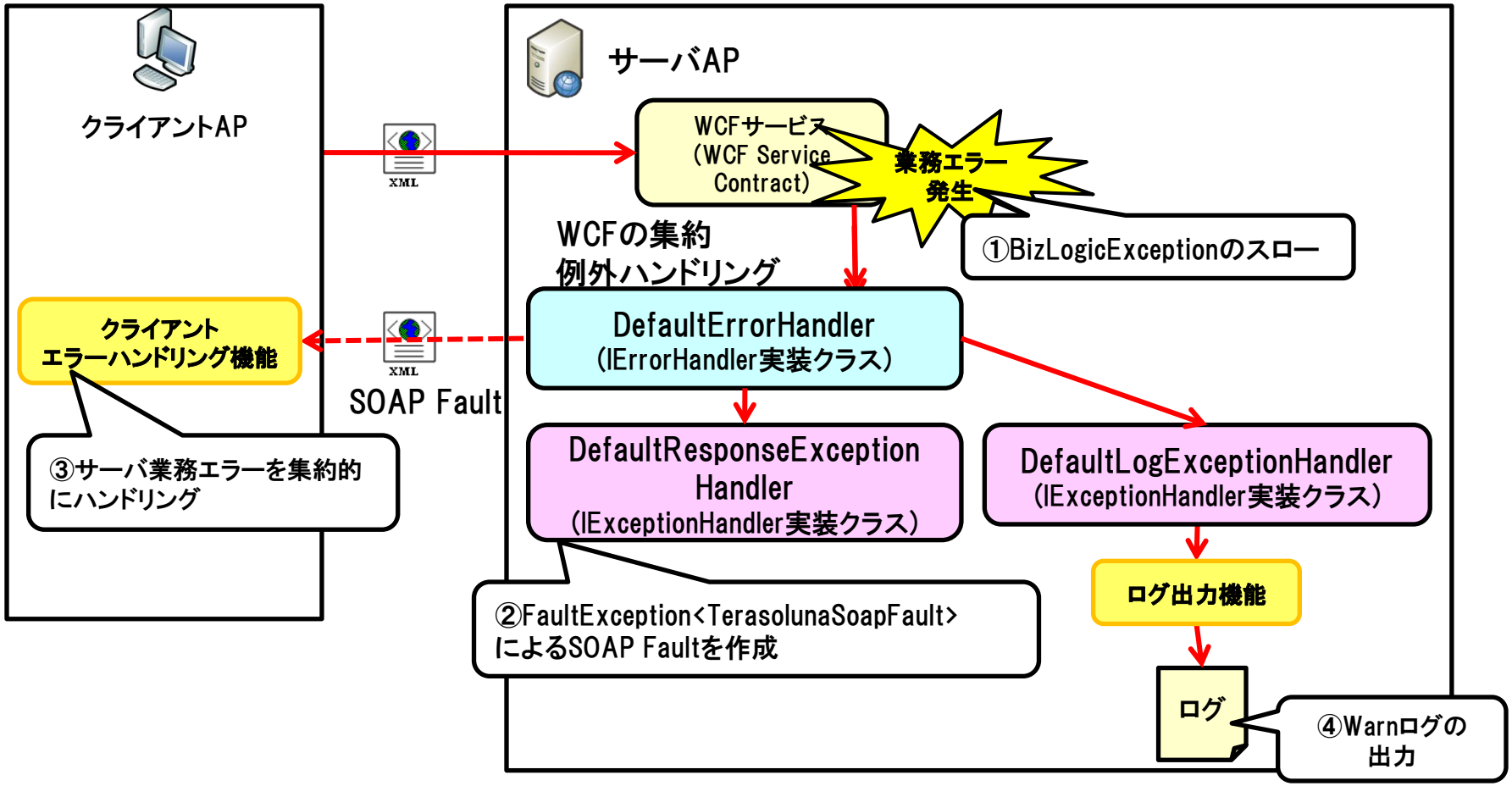
- TERASOLUNAフレームワークの集約例外処理
  - Terasoluna.ExceptionHandling.IErrorHanlderインタフェース
    - 「クライアントエラーハンドリング機能」が利用する集約例外処理と同一インタフェース
  - 例外の種類ごとにSOAP Faultの作成やログ出力処理を実施

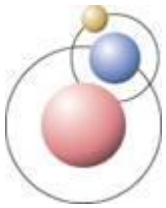
エラー種別	対象例外	処理内容
入力値検証エラー	FautlException<ValidationFault> (「サーバ入力値検証機能」によりスロー)	SOAP Faultの返却
業務エラー	BizLogicException	SOAP Faultの返却 Warnログの出力
システムエラー	その他の例外(System.Exception継承クラス)	SOAP Faultの返却 Errorログの出力

サーバ入力値検証エラーのハンドリング



サーバ業務エラーのハンドリング



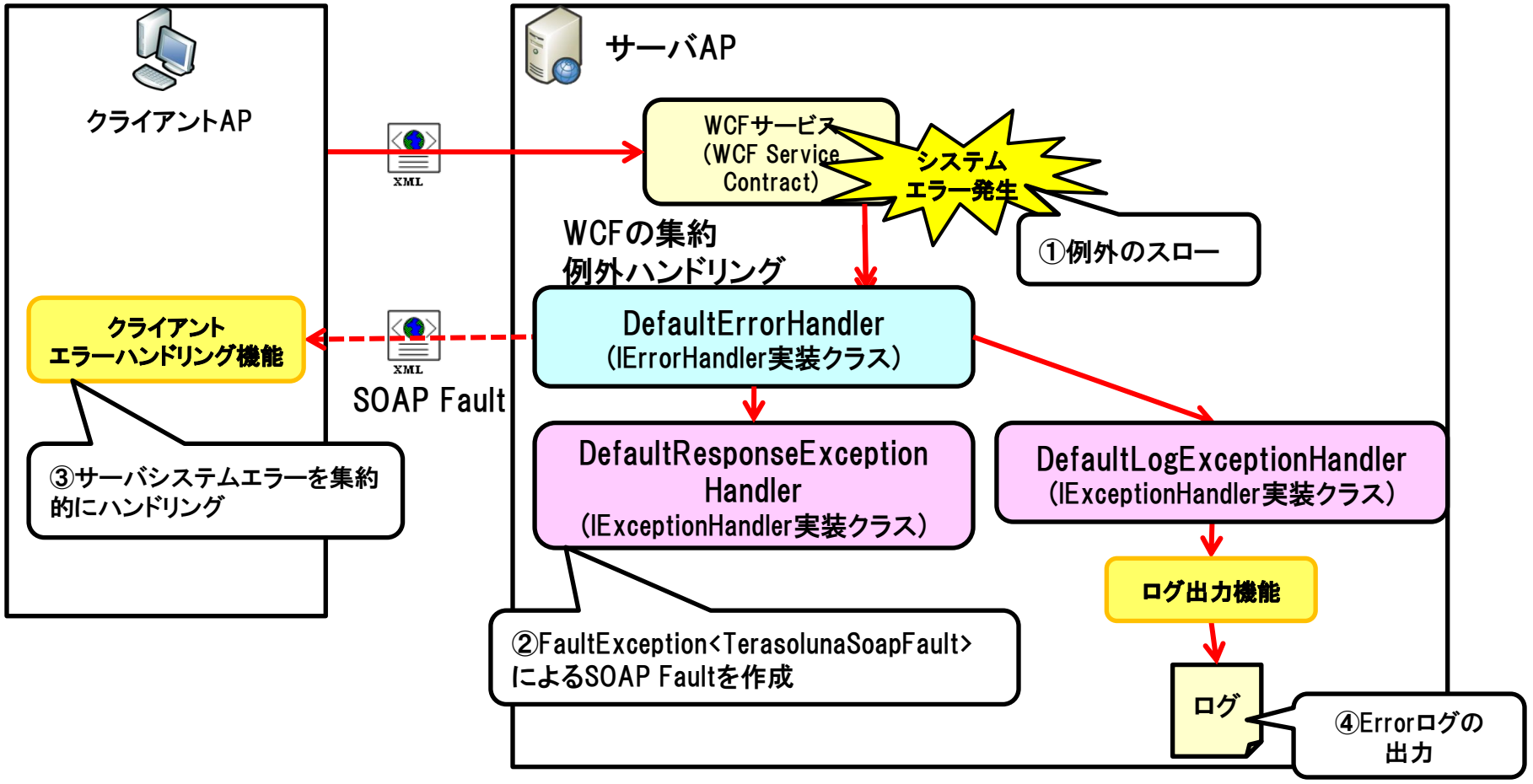


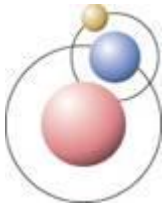
業務エラーの実装イメージ

- 業務開発者は、業務エラー発生時に、BizLogicExceptionをスローする
  - クライアントビジネスロジックと同じ実装スタイル

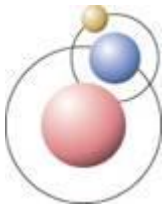
```
public class CalcService
{
    public void Login(LoginInputDto inputDto)
    {
・・・
        ///IDとパスワードが合致しなかった場合
        ///業務エラーとしてBizLogicExceptionをスロー
        throw new BizLogicException(
            BizLogicExceptionErrorType.BizLogicFailure,
            CalcResource.ERROR_CALC_MSG002,
            new List<ErrorInfo>()
            {
                new ErrorInfo("ERROR_CALC_MSG003", null,
                        CalcResource.ERROR_CALC_MSG003, null)
            });
    }
}
```

サーバシステムエラーのハンドリング





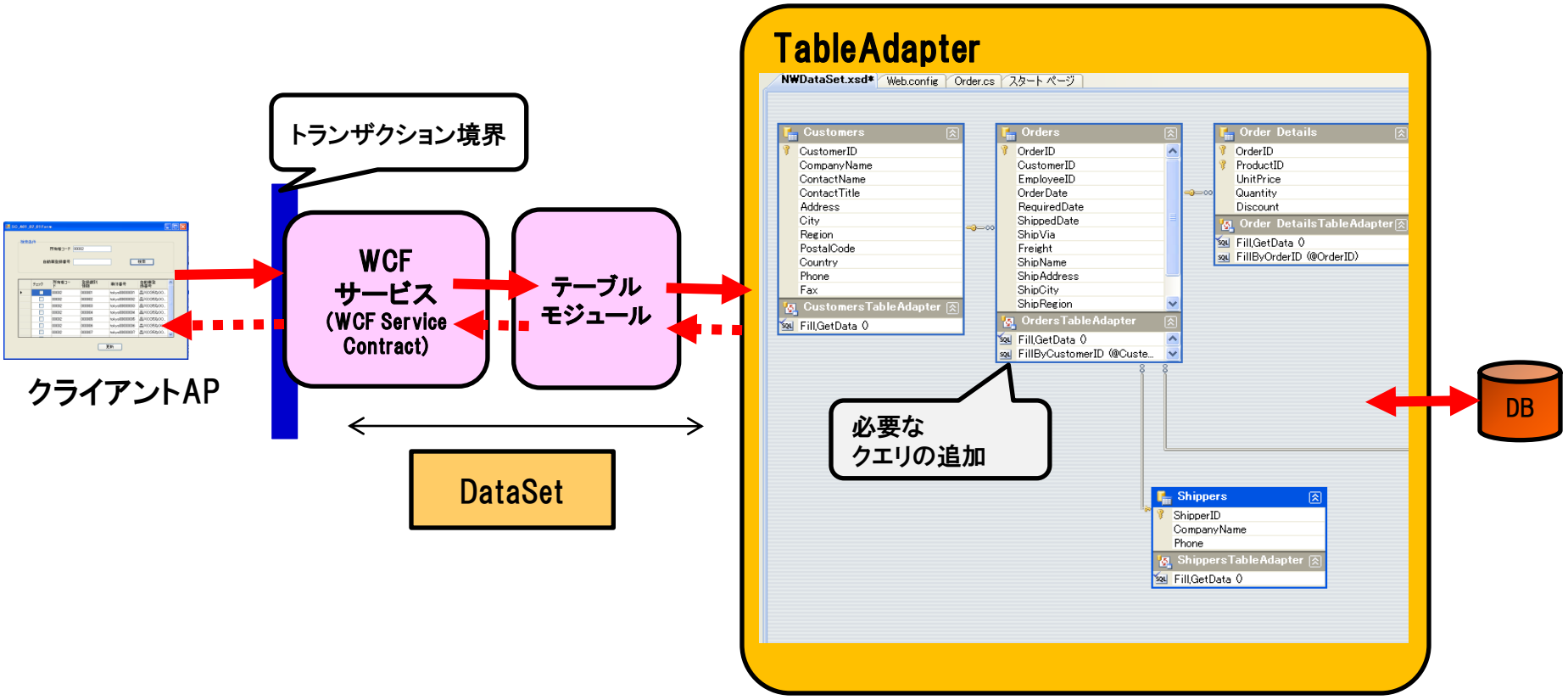
データアクセス機能



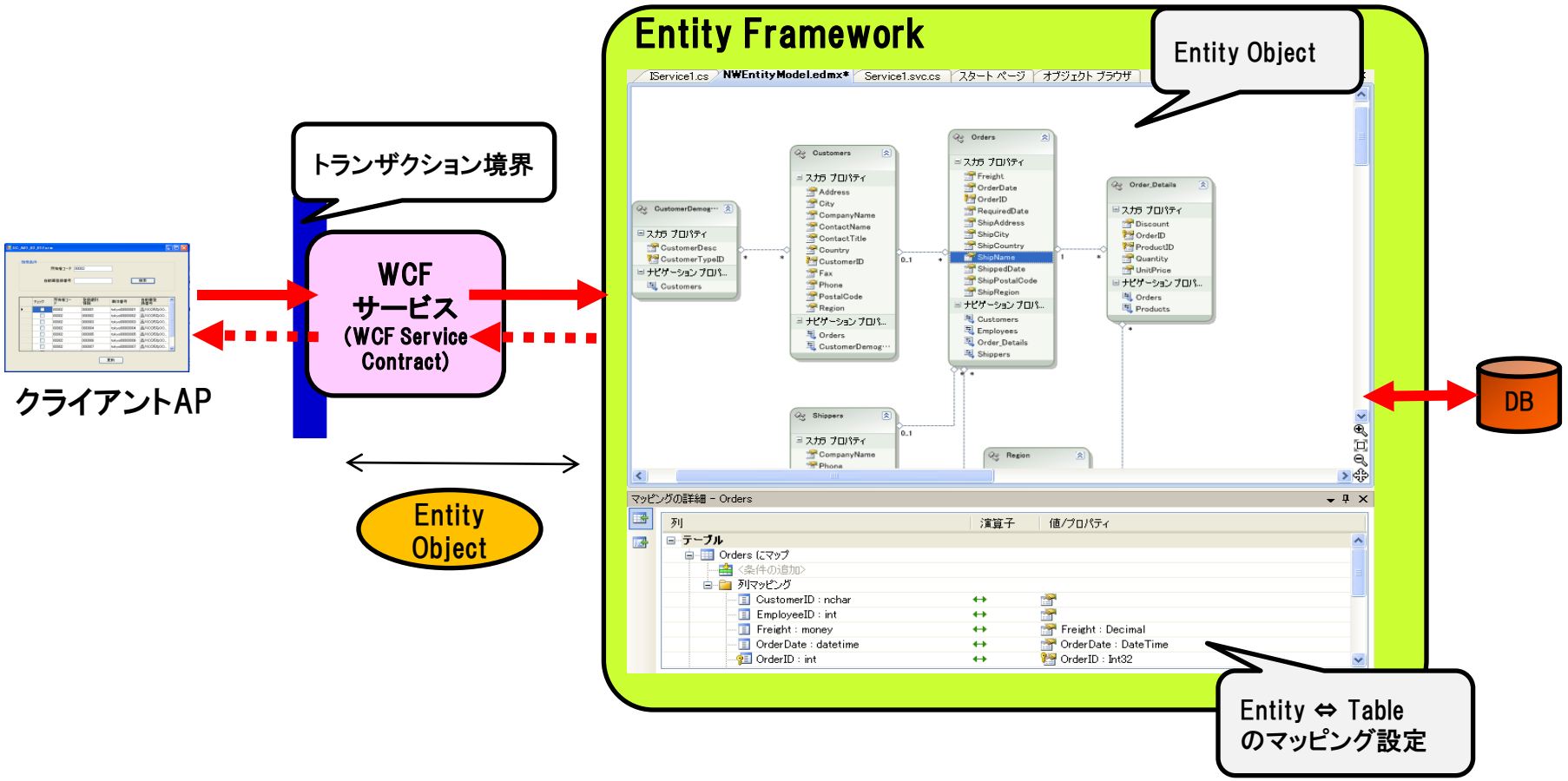
データアクセス機能の検討経緯(テクノロジーの選択)

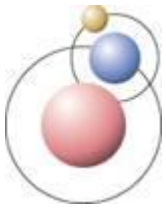
- ADO.NETをそのまま利用する
  - TERSOLUNA Frameworkで独自拡張をしてしまうと、Visual Studioのツールを使用できなくなってしまう
  - 特定のテクノロジーに依存しないようにする
    - プロジェクトの特色に合わせてテクノロジーを選択してもらう
      - TableAdapter&型付きDataSet
      - EntityFramework
      - DataReader
  - トランザクション制御
    - 基本的にはTransactionScopeによる自動トランザクション機能を使用

TableAdapter&型付きDataSetのイメージ



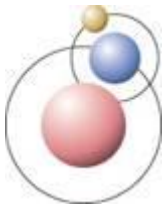
Entity Frameworkのイメージ





データアクセス機能の検討経緯(データコピー)

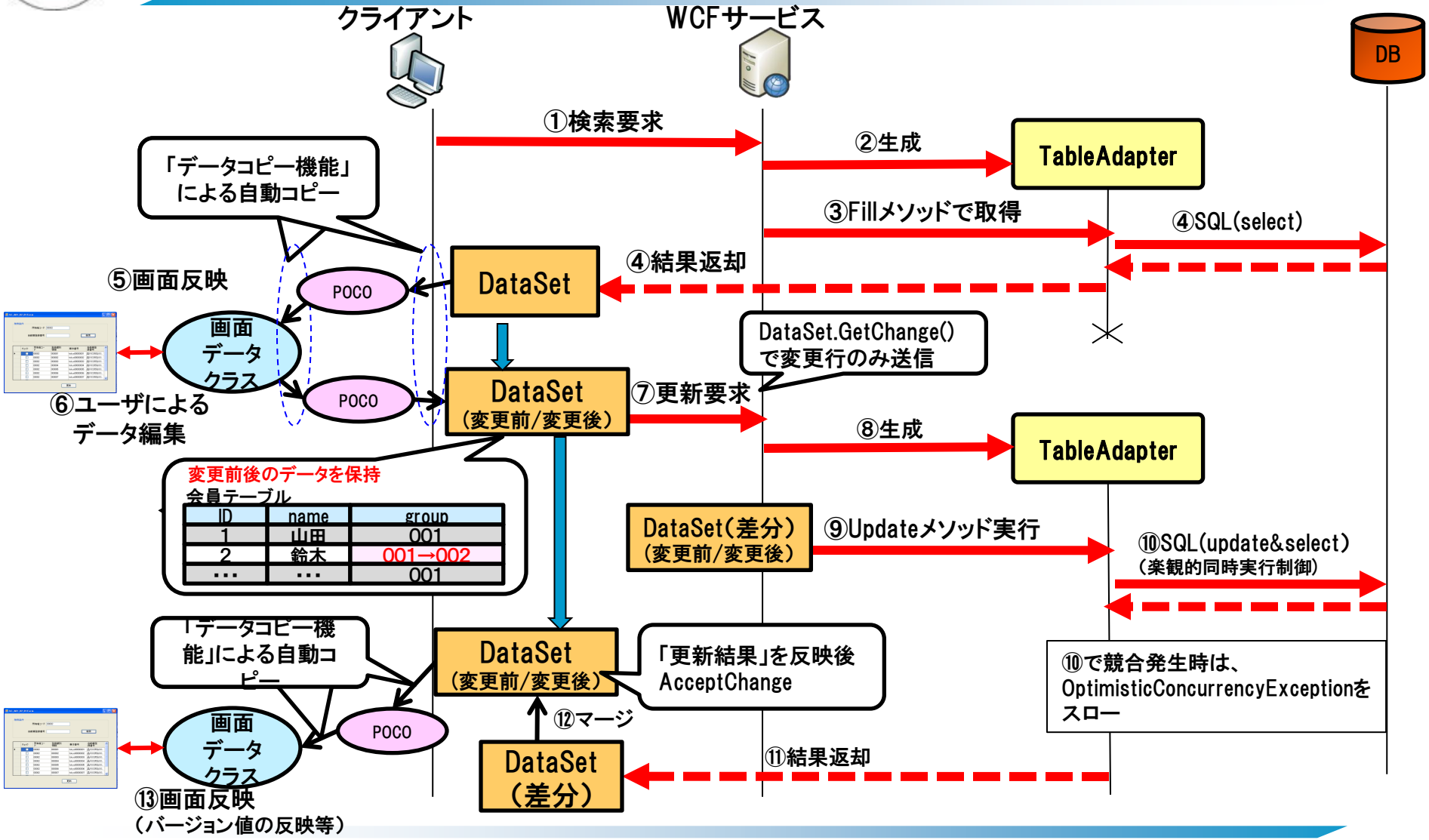
- DataSetまたはEntityObjectとDTOの間でコピー処理が発生する
  - 単調で大量の実装作業が発生
- 「データコピー機能」の充実化
  - DataSetやEntity Objectとのデータコピーにも対応
  - データアクセス処理の実装負担を削減

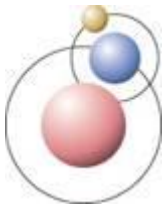


【参考】オフラインオプティミスティック(楽観的)ロックの実現検討

- 型付きDataSet&TableAdapter(DTOがDataSetの場合)
  - 従来からあるクライアント書き戻し方式で実施
  - 参照時には、TableAdapterからDataSetへFill
  - テーブルにバージョン列を用意
    - DataSetにテーブルから取得したバージョン値を格納するカラムを用意し、クライアントへ返却
    - SQLServerならTimestamp列
  - 変更前・変更後の値をサーバへ送信
    - 参照時に取得したDataSetに、「画面データ」の値をコピーすることでDataSetで変更状態を管理
    - DataSet.GetChangeメソッドで変更差分のみ送信
  - Visual Studioが自動作成する楽観的同時実行制御対応SQLを利用
    - TableAdapterを作成する際のウィザードで生成されるSQLをそのまま利用
    - 競合発生時は、OptimisticConcurrencyExceptionをスロー
  - 更新成功時には最新のバージョン値を含む値を返す
    - クライアントに最新のバージョン値を反映
  - .NET-.NET接続のみ利用可能
    - 相互運用性は確保できないが、リッチクライアントでの変更状態の管理が容易
    - TERASOLUNAフレームワークで、DTOとしてDataSetが利用するなら、クライアントビジネスロジックの中で、DataSetとPOCOの自動コピーが有効となるように実装する必要がある。

【参考】TableAdapterでの対話型トランザクション（DTOがDataSet）

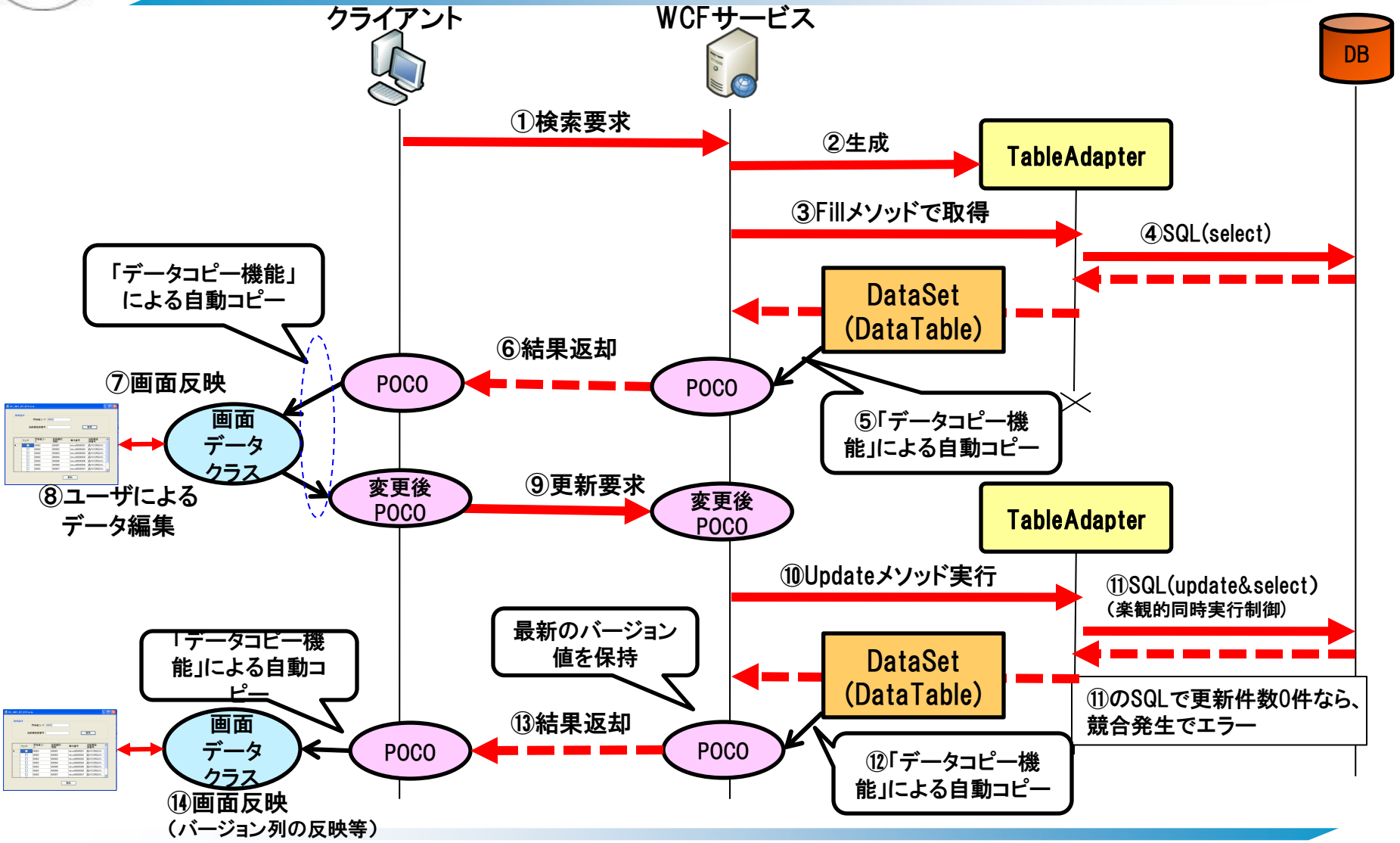


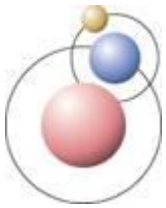


【参考】オフラインオプティミスティック(楽観的)ロックの実現検討

- 型付きDataSet&TableAdapter(DTOがPOCOの場合)
  - 参照時には、TableAdapterからDataSetへFillする
    - 「データコピー機能」でDataSetからDTOへコピー
  - テーブルにバージョン列、DTOにバージョンプロパティを用意
    - DTOにテーブルから取得したバージョン値を格納するプロパティを用意し、クライアントへ返却
    - SQLServerならTimestamp列
  - バージョン値を含めて更新内容をサーバへ送信
    - クライアントでの編集後、参照時に取得したバージョン値を含めてサーバへ更新要求
  - バージョン列をチェックする更新系SQLを実装
    - DataSetがクライアントへ送信されないためデータ変更前の状態を保持できないので、TableAdapter作成時に自動生成する楽観的同時実行制御対応SQLは使えない
    - Where句でバージョン列をチェックする更新系SQLを実装し、更新件数0件なら競合発生
    - 「クエリの追加」で作成。引数がDataTableにならないので、DataSetへの変換は行わず、そのままPOCOから取得した値を引数として渡す
  - 更新成功時には最新のバージョン値を含む値を返す
    - クライアントに最新のバージョン値を反映

【参考】TableAdapterでの対話型トランザクション（DTOがPOCO）

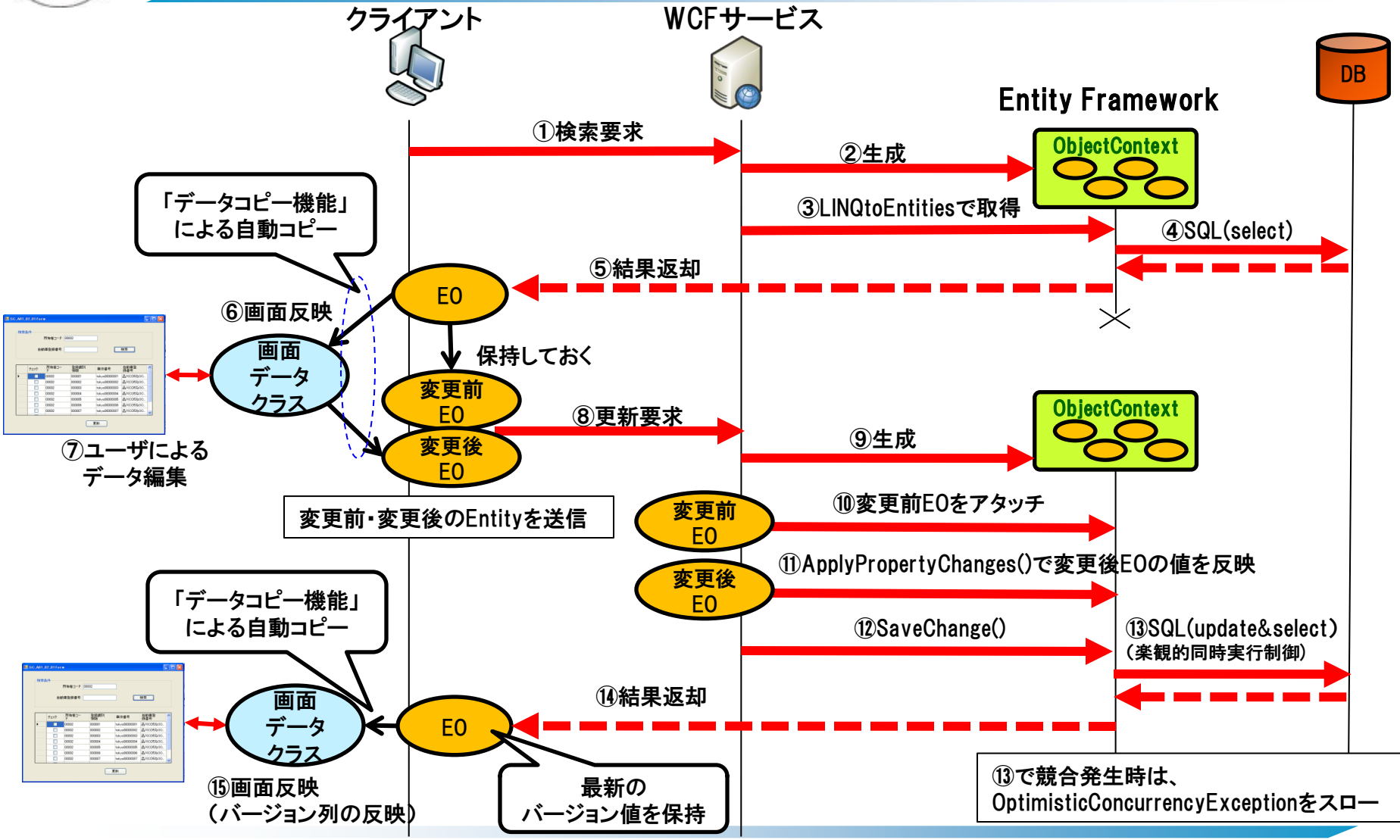


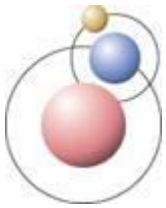


【参考】オフラインオプティミスティック(楽観的)ロックの実現検討

- Entity Framework(ver1)の場合(DTOがEntityObject)
  - テーブルにバージョン列、EOにバージョンプロパティを用意
    - Entity Editorで、EntityObjectのバージョンプロパティを「同時実行モード」プロパティを「None」から、「Fixed」に変更
      - 「Fixed」に変更した項目は、SQLのWHERE句の条件として追加される
    - SQLServerならTimestamp列
  - 参照時は、LINQ to EntitiesでEntityObjectを取得
    - LINQでは難しい場合には、Entity SQLやQueryBuilderも利用
  - ユーザによる編集後、変更前、変更後のEntityObjectをサーバへ送信
    - DataSetと違い、EntityObject自体は変更状態を管理できない
  - 変更前EOをアタッチ後、ApplyPropertyChangesメソッドで変更後EOの値で更新し、SaveChangeメソッドを実行
    - 変更前EOをアタッチ後にObjectContextの内部で記録されたEntityObjectの変更を更新系SQLとしてDBに反映
    - 競合発生時は、OptimisticConcurrencyExceptionをスロー
  - 更新成功時には最新のバージョン値を含む値を返す
    - クライアントに最新のバージョン値を反映

【参考】Entity Frameworkでの対話型トランザクション（DTOがEntityObject）

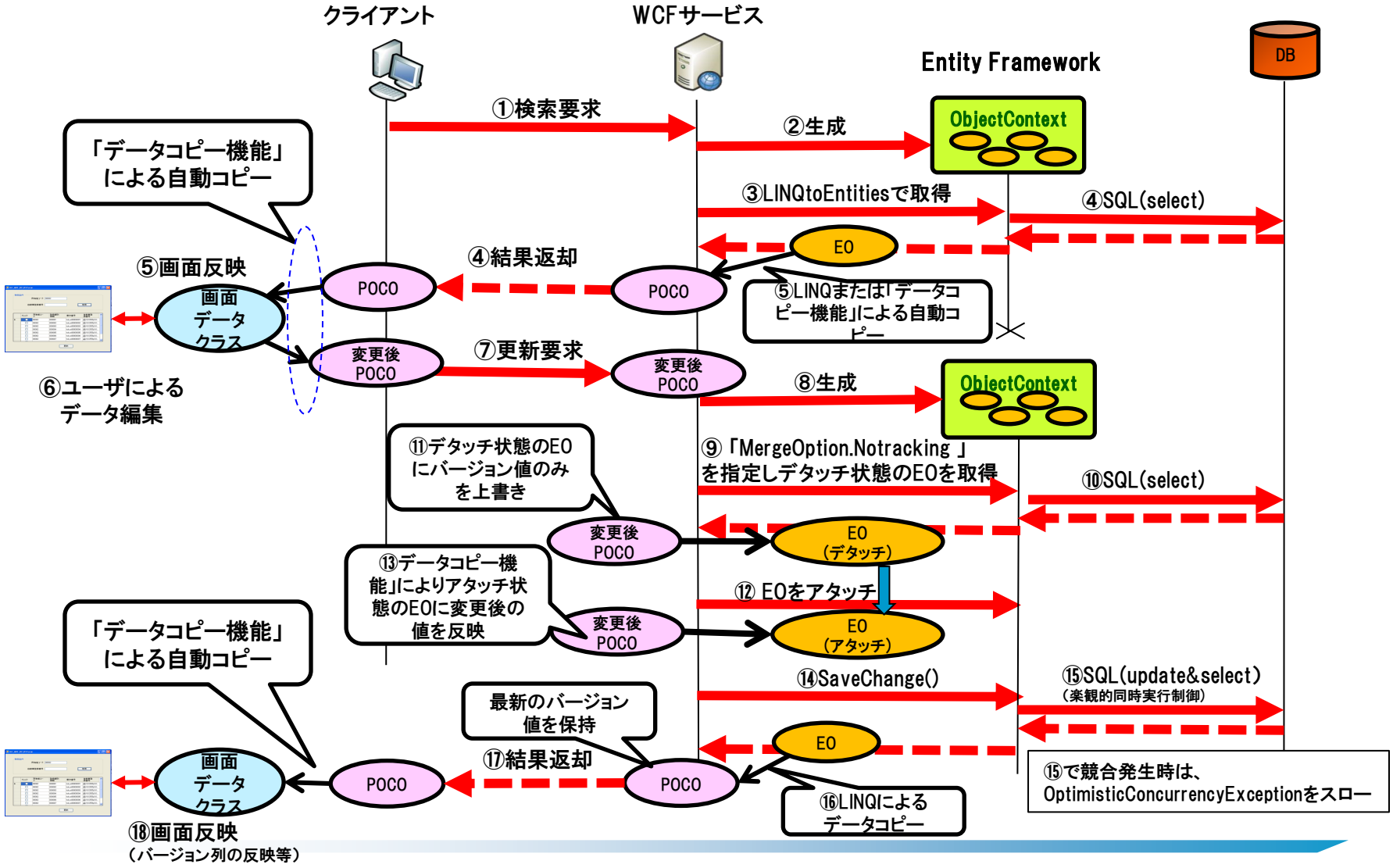


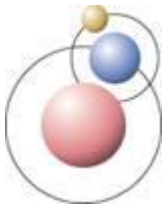


【参考】オフラインオプティミスティック(楽観的)ロックの実現検討

- Entity Framework(ver1)の場合(DTOがPOCO)
  - テーブルにバージョン列、EO、POCOにバージョンプロパティを用意
    - Entity Editorで、EntityObjectのバージョンプロパティを「同時実行モード」プロパティを「None」から、「Fixed」に変更
      - 「Fixed」に変更した項目は、SQLのWHERE句の条件として追加される
    - SQLServerならTimestamp列
  - 参照時は、LINQ to EntitiesでPOCOに変換して取得
    - LINQを使ってEntityObjectと取得する同時にそのままPOCOに変換
    - プロパティのコピーが大変な場合は、「データコピー機能」を利用
  - ユーザによる編集後、バージョン値を含めて更新内容をサーバへ送信
    - クライアントでの編集後、参照時に取得したバージョン値を含めてサーバへ更新要求
  - 対象のEntityObjectをデタッチ状態でDBより再取得
    - SaveChangeメソッドで更新系SQLを発行させるには、EnitityObjectに変更を記録する必要がある
    - ObjectQuery.Executeメソッドで、「MergeOption.Notracking」を指定して対象のEntityObjectを再取得
  - デタッチ状態でEOのバージョン値のみ上書きして、アタッチ
    - サーバ上に擬似的に、参照時と同じバージョン値のEntityObjectを再現
    - アタッチすると、ObjectContextがEntityObjectの変更記録を開始
    - 「データコピー機能」を利用しコピーして変更を反映
  - EOの値を更新した値で上書きし、SaveChangeメソッドを実行
    - ObjectContextの内部で記録されたEntityObjectの変更を更新系SQLとしてDBに反映
    - 競合発生時は、OptimisticConcurrencyExceptionをスロー
  - 更新成功時には最新のバージョン値を含む値を返す
    - クライアントに最新のバージョン値を反映

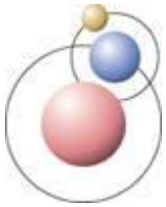
【参考】Entity Frameworkでの対話型トランザクション（DTOがPOCO）



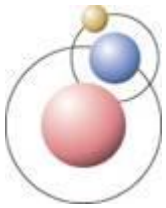


【参考】Entity Framework (v2) について

- .NET4.0(Visual Studio2010)でEntity Frameworkのver2がリリース予定
  - エンティティをPOCOをとして扱えるようになるため、上記のオフラインオプティミスロックの開発方法も追加または変更になる可能性はある

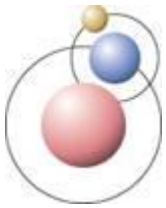


TERASOLUNAフレームワークの拡張ポイント



FW拡張の考え方

- インスタンス管理機能
  - Unity ABが提供するDI/AOPコンテナ(UnityContainer)を利用
- DI(Dependency Injection)によるFW実装の差し替え
  - フレームワーク機能を構成する各コンポーネントは全てインタフェースで定義されている
  - インタフェースに対する実装クラスをDIにより差し替えることでFWの機能拡張が可能
  - DIの設定は設定ファイルまたはコードでの記述が可能
    - 構成ファイル
    - UnityContainer.ResiterTypeメソッド



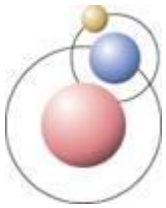
FW拡張の考え方

- Extensionクラスによるフレームワーク機能の統合
  - Microsoft.Practices.Unity.UnityContainerExtension
    - UnityContainerExtension継承クラスにより、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可能
  - TERASOLUNAフレームワークの機能を統合するための統一的な手段として利用
    - 各フレームワーク機能が標準実装のクラスをDIするExtensionクラスを提供
    - 各ExtensionクラスをUnityContainerに追加設定することで、フレームワークの機能が利用可能になる
  - FW拡張時は、機能拡張したクラスをDIするExtensionクラスを実装する

Extensionクラスの例

- 画面遷移機能のExtensionクラス(FormForwardExtension)
  - Initializeメソッドをオーバライドし、DI設定を記述

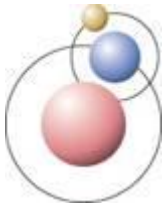
```
namespace Terasoluna.Windows.Forms.FormForward
{
    public class FormForwardExtension : UnityContainerExtension
    {
・・・
        protected override void Initialize()
        {
・・・
            ///フレームワークの機能を構成するインタフェースに対する実装クラスをDIする
            Container.RegisterType<FormForwardManager>(new ContainerControlledLifetimeManager());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



機能拡張用のExtensionクラスの作成

- FW拡張時は、インタフェースに対する実装クラスを拡張したクラスに変更したExtensionクラスを作成する

```
namespace Terasoluna.Sample.FormForward
{
   public class SampleFormForwardExtension : UnityContainerExtension
   {
 ・・・
      protected override void Initialize()
      {
 ・・・
         Container.RegisterType<FormForwardManager>(new ContainerControlledLifetimeManager());
         Container.RegisterType<IFormForwardFlowController, FormForwardFlowController>(
            new ContainerControlledLifetimeManager());
         ///インタフェースに対する実装クラスを機能拡張したクラスに差し替え
         Container.RegisterType<IFormFactory, SampleFormFactory>(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());
      }
   }
}
```



構成ファイルによるExtensionクラスの設定

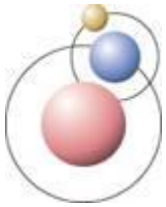
- FW機能を有効にするには、構成ファイル上でExtensionクラスを設定
  - /configuration/unity/containers/container/extensions/addタグ

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

機能拡張用のExtensionクラスの設定

- FW拡張時は、新たに作成したExntesionクラスに設定を変更する

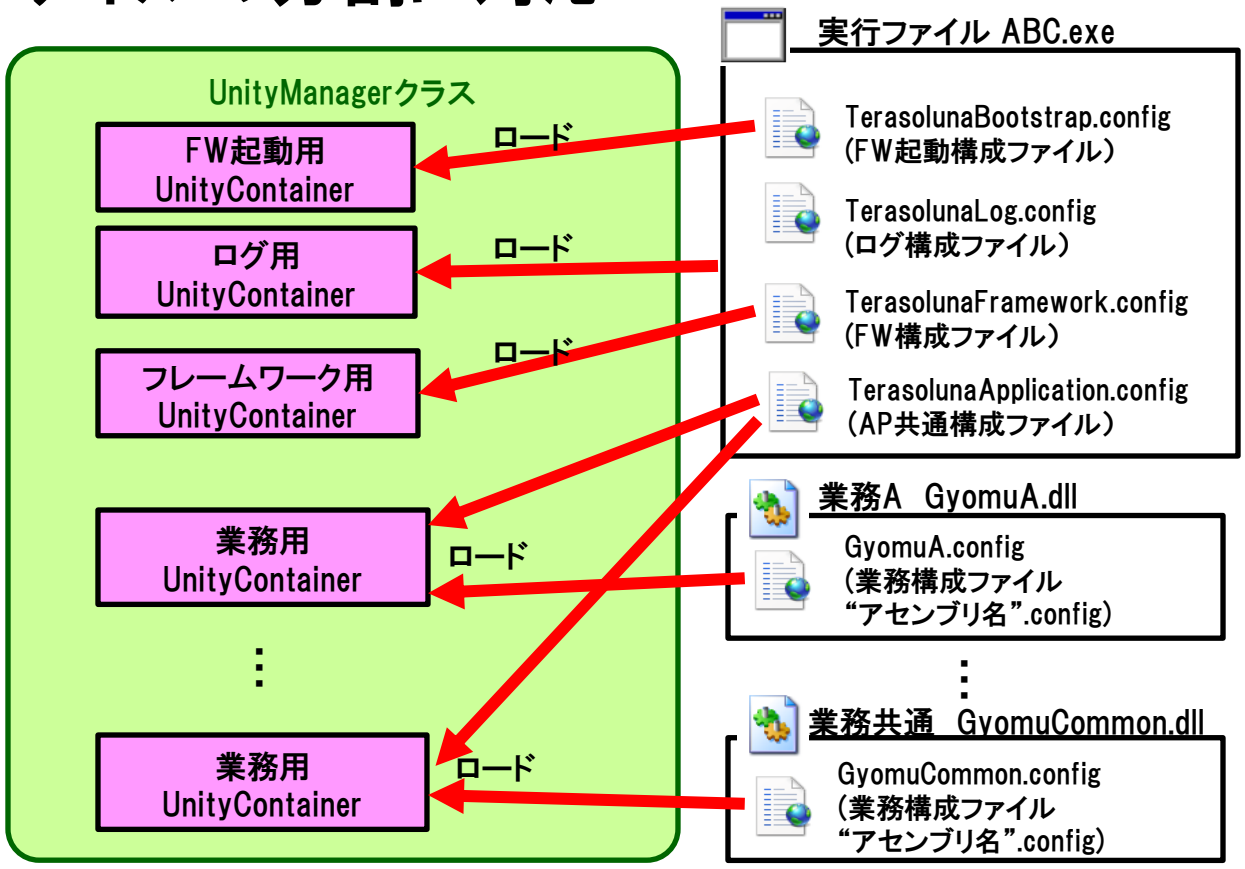
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
       <extensions>
         <!-- 画面遷移機能の機能拡張版に差し替え-->
         <add type="Terasoluna.Sample.FormForward.SampleFormForwardExtension,
                Terasoluna.Sample" />
       </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```



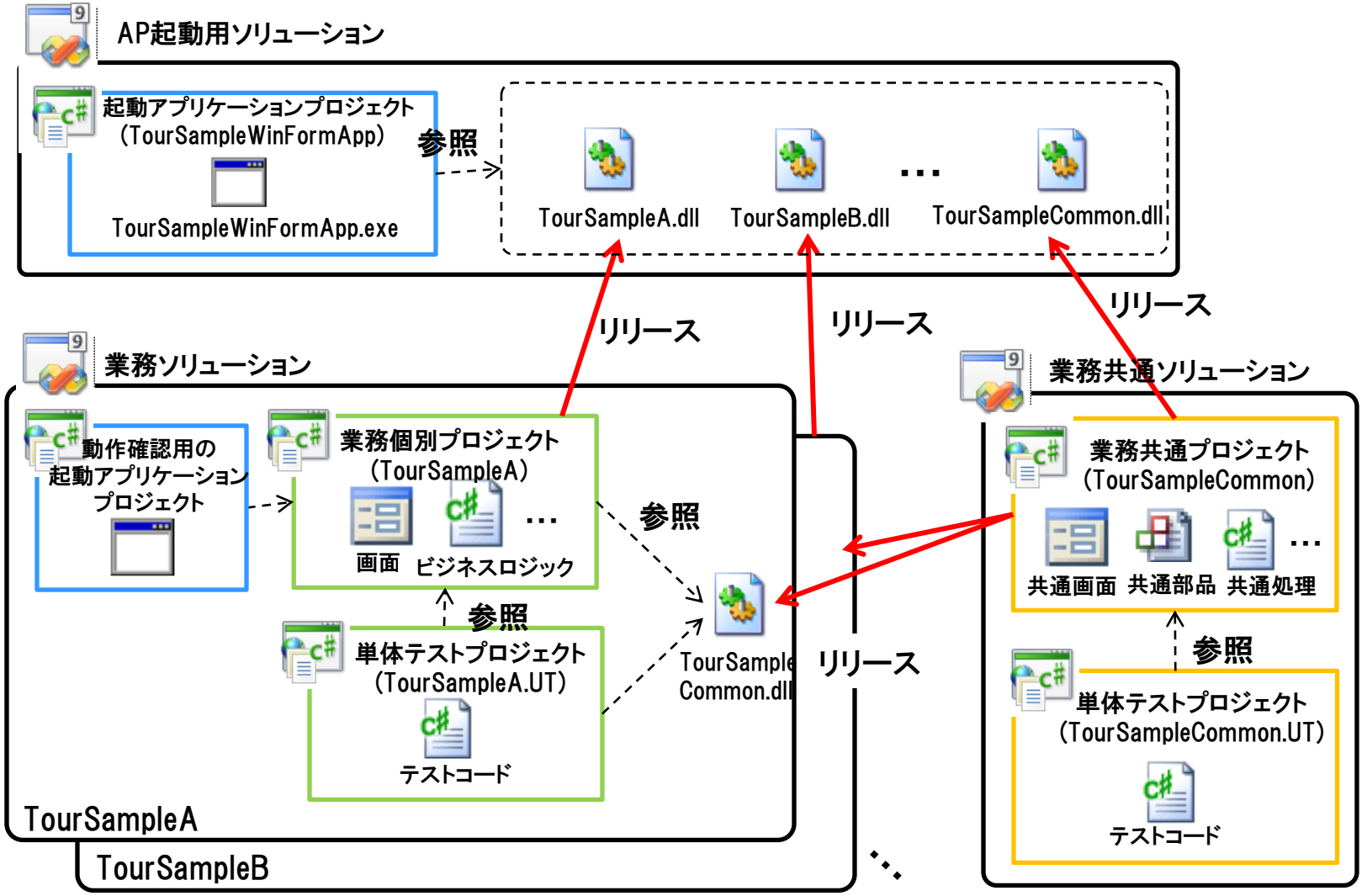
TERASOLUNAフレームワークによるアプリケーションのソリューション構成

大規模開発を考慮した構成ファイルの分割

- インスタンス管理機能はプロジェクトの分割を考慮し、構成ファイルの分割に対応



大規模プロジェクトを想定したクライアントAPのソリューション構成



クライアントAPにおける構成ファイルの配置

起動アプリケーションプロジェクト
(TourSampleWinFormApp)

TerasolunaBootstrap.config
(FW起動構成ファイル)

TerasolunaLog.config
(ログ構成ファイル)

TerasolunaFramework.config
(FW構成ファイル)

TerasolunaApplication.config
(AP共通構成ファイル)

業務プロジェクト(TourSampleA)

TourSampleA.config
(業務構成ファイル
“アセンブリ名”.config)

業務個別プロジェクト(TourSampleB)

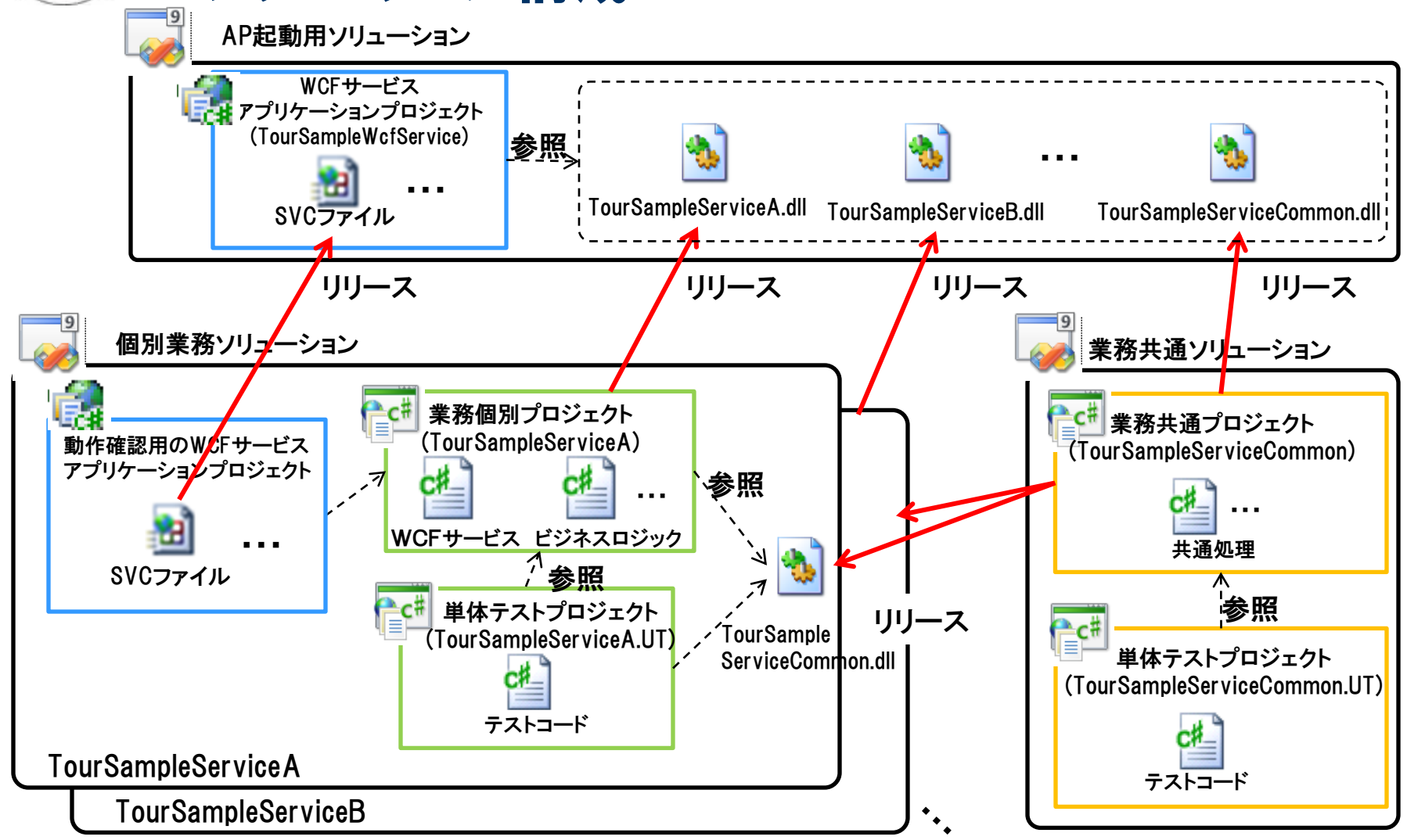
TourSampleB.config
(業務構成ファイル
“アセンブリ名”.config)

.
.
.

業務共通プロジェクト(TourSampleCommon)

TourSampleCommon.config
(業務構成ファイル
“アセンブリ名”.config)

大規模プロジェクトを想定したサーバAPの ソリューション構成



サーバAPにおける構成ファイルの配置

WCFサービス
アプリケーションプロジェクト
(TourSampleWcfService)

TerasolunaBootstrap.config
(FW起動構成ファイル)

TerasolunaLog.config
(ログ構成ファイル)

TerasolunaFramework.config
(FW構成ファイル)

TerasolunaApplication.config
(AP共通構成ファイル)

業務個別プロジェクト(TourSampleServiceA)

TourSampleServiceA.config
(業務構成ファイル
“アセンブリ名”.config)

業務個別プロジェクト(TourSampleServiceB)

TourSampleServiceB.config
(業務構成ファイル
“アセンブリ名”.config)

.
.
.

業務共通プロジェクト(TourSampleServiceCommon)

TourSampleServiceCommon.config
(業務構成ファイル
“アセンブリ名”.config)