ＰＳＬＸプラットフォーム計画

C#版

共通モジュール実装マニュアル

第 2 部

PPS ドキュメントサービス

＜レベル 2 実装＞

バージョン 1.0

2009 年 6 月

NPO 法人ものづくり APS 推進機構

改訂履歴

日付	内容	備考
2009/03/12	バージョン 1.0 ベータ版	
2009/06/08	バージョン 1.0	

もくじ

1. はじめに

◆ 目的

PPS 共通コンポーネントは、通信の細かなしくみ、および XML に関する専門知識の浅いアプリケーション・プログラマであっても、容易に PSLX プラットフォーム対応のシステム構築が可能となるように設計されています。また、OASIS PPS 技術委員会が定めた国際標準を準拠したうえで、今後、個々に開発されるさまざまなアプリケーション間の相互接続性を保証するための、共通の実装環境を提供します。

この仕様書は、実装マニュアル第 2 部「PPS ドキュメントサービス（レベル 2 実装)」です。この実装マニュアル第 1 部と第 2 部では、PSLX プラットフォーム対応ソフトウェア間で交換するメッセージの内容を生成または解釈するためのプログラミングのための方法やルールを解説しています。ここで解説する内容は、XML に関する基本的な概念さえ知っていれば、XML を扱うための具体的なプログラミング方法を知らなくてもプログラム開発ができるようになっています。

第 2 部では、レベル 2 の実装として、OASIS PPS 規約のすべての機能を前提としたプログラミングの内容を解説しています。これに対して、第 1 部では、OASIS PPS 規約でさだめたレベル 1 の実装を行うための内容を抜粋して解説されています。

◆ 対象とする読者

（1）資格

この仕様書は、PSLX プラットフォーム計画プロジェクトに参加している企業の従業員に対して、PSLX プラットフォーム対応ソフトウェアを開発するために公開している文章です。PSLX プラットフォーム計画プロジェクトのメンバー以外であっても、この仕様書を閲覧することは可能ですが、NPO 法人ものづくり APS 推進機構の許可なく複製や再配布を行うことは禁止されています。

（2）必要とする知識・技術

ソフトウェア開発の一般知識を有する人を対象にした文章です。特に、下記の項目についての知識が必要です。

— C#の言語仕様
— Visual Studio による開発方法
— オブジェクト指向モデリングの概要

◆ 稼動環境

本仕様書にふくまれる内容を実行するためには、以下のソフトウェア環境が必要となります。

区分	内容
オペレーションシステム	WindowsXP　ServicePack2、　Windows Vista
コンポーネント環境	.NET Framework 3.0 以降
ブラウザ	InternetExploler 6 以降
開発ツール	Visual Studio 2005 ServicePack1、または Visual Studio 2008

2. 複数型業務プロパティの定義

◆ 複数型の業務プロパティの設定

業務オブジェクトの各業務プロパティは、単数型、複数型の２種類に分かれています。複数型の業務プロパティは、文字どおり複数の値を保持することができます。ただし、正確に説明すると、複数型の業務プロパティは、業務オブジェクトに直接所属するのではなく、１つ以上の中間オブジェクトを介して所属しています。

つまり、同一種類の業務プロパティを複数設定するには、その業務プロパティが所属する中間オブジェクトを複数生成し、その上位にある業務オブジェクトに追加しなければなりません。各業務プロパティがどの中間オブジェクトに所属しているかは、あらかじめプロファイルで規定されています。中間オブジェクトは、ElementName（要素名）と Modifier（修飾子）によって識別されます。

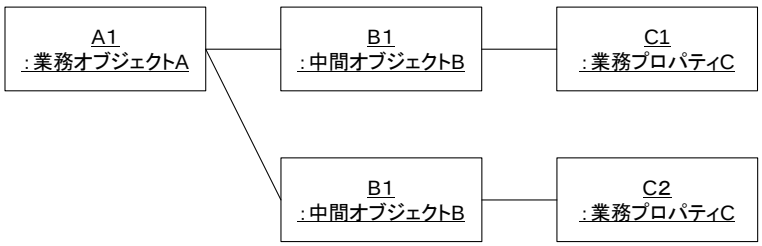


図２－１　複数型の業務プロパティ

たとえば、業務ドキュメント Product に設定される業務オブジェクト Product は、product-id および product-name を持ちますが、前者は単数型、後者は複数型です。したがって、product-name の値は、以下のように、複数設定することができます。

ここで、プロファイル情報を参照し、product-name が所属する中間オブジェクトは、ElementName が Spec、Modifier が pps:name であるため、以下のようなプログラムとなります。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("Product");
DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "P001";

DomainObject[] subObjs = obj.CreatePropertyObjects("Spec", "pps:name", 2);
```

```
    subObjs[0]["product-name"] = "製品ABC";
    subObjs[1]["product-name"] = "N123-GX456";
```

ここで、CreatePropertyObjects メソッドによって、中間オブジェクトを 2 つ生成しています。第一引数が ElementName、第二引数が Modifier、そして第三引数が生成する中間オブジェクト数です。メソッドの戻り値として、中間オブジェクトのリストが返されます。この中間オブジェクトのリストのそれぞれの要素に対して、単数型の場合と同様に、インデクサを利用して値を設定してください。

プログラムをみて分かるとおり、中間オブジェクトは、業務オブジェクトと同じ DomainObject クラスとなっています。したがって、プロファイルの定義において、さらに下位の中間オブジェクトの定義がある場合には、中間オブジェクトの下位に、さらに中間オブジェクトを同様の方法で設定することが可能です。

◆ 中間オブジェクトによって業務プロパティを構造化する

中間オブジェクトは、複数型の業務プロパティの設定のためのみでなく、以下のように、複数の異なる種類の業務プロパティを構造化するために利用することができます。たとえば、業務オブジェクト Product には、構成品目名称（child-item-name）と構成数量（child-quantity）という業務プロパティを持ちます。これらは、ともに複数型ですが、部品表（BOM）などを表現する際に、複数の構成品目名称のそれぞれについて構成数量を設定したい場合、各組の関係を維持しながら複数の値を定義する必要があります。

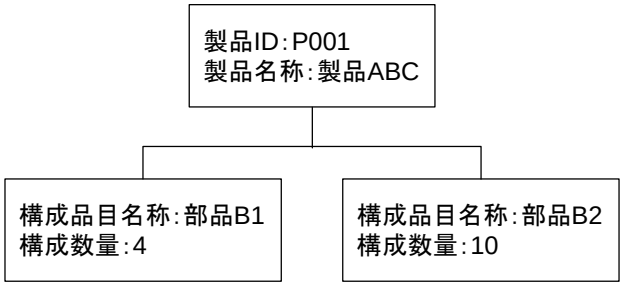


図２－２　対象とする製品の構造

上図のような対象製品の情報を表現する場合には、構成部品名称：部品 B1 と構成数量：４とが組みで、構成部品名称：部品 B2 と構成数量：１０が組みとなるように以下のようにプログラムを作成します。


```
DomainObject[] subObjs = obj.CreatePropertyObjects("Compose", "pps:child", 2);

subObjs[0]["child-name"] = "部品B1";
subObjs[0]["child-quantity"] = 4;

subObjs[1]["child-name"] = "部品B2";
subObjs[1]["child-quantity"] = 10;
```

これは、構成部品名称（child-item-name）と構成数量（child-quantity）とがともに同一の中間オブジェクト（ElementName=”Compose”, Modifier=”pps:child”）に所属していることで可能となります。言い換えれば、プロファイルにおける業務プロパティの定義時点で、このように関係付けられる業務プロパティのグループは、あらかじめ決められています。詳細は、プロファイル定義を参照してください。

上記の結果、生成された XML メッセージは以下のとおりです。部品 B1 の構成数量が 4、部品 B2 の構成数量 10 となっていることが確認できます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="87" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" confirm="Always">
        <Document name="Product" id="185" action="Add">
            <Item id="P001">
                <Spec type="pps:name">
                    <Char value="製品ABC" />
                </Spec>
                <Compose type="pps:child" name="部品B1">
                    <Qty type="pps:general" value="4" />
                </Compose>
                <Compose type="pps:child" name="部品B2">
                    <Qty type="pps:general" value="10" />
                </Compose>
            </Item>
        </Document>
    </Transaction>
</Message>
```


◆ 複数型の業務プロパティの簡易な設定方法

中間オブジェクトは、複数の異なる種類の業務プロパティをグループ化し構造化するために活用すると説明しましたが、逆に、グループ化されていない複数型の業務プロパティにとっては、この中間オブジェクトは、単に業務プロパティを物理的に複数にして保持するためのみに利用されることになります。そこで、このような場合には、以下のような、簡易的な方法によって、複数型の業務プロパティの値を設定することが可能です。

たとえば、業務オブジェクト Product において、構成部品名称は、構成数量とグループですが、単なる部品名称（product-name）は、他のどの業務プロパティともグループ化されていません。このような複数型の業務プロパティは、以下のように、CreatePropertyObjects メソッドの引数として、ElementName と Modifier の代わりに、単に業務プロパティ名“product-name”を指定することで、同様の処理が可能となります。つまり、グループ化されていない単独の複数型業務プロパティについては、中間オブジェクトの定義情報をあえて調べる必要はありません。

```
DomainObject[] subObjs = obj.CreatePropertyObjects("product-name", 2);
subObjs[0]["product-name"] = "製品ABC";
subObjs[1]["product-name"] = "N123-GX456";
```

実際のプログラミング上では、上記の方法を、グループ化された複数型の業務プロパティに関して実行することも可能です。しかし、プログラムの可読性を高める上で、先に説明したように、中間オブジェクトを意識した ElementName と Modifier の情報を利用した生成を行うようにしてください。

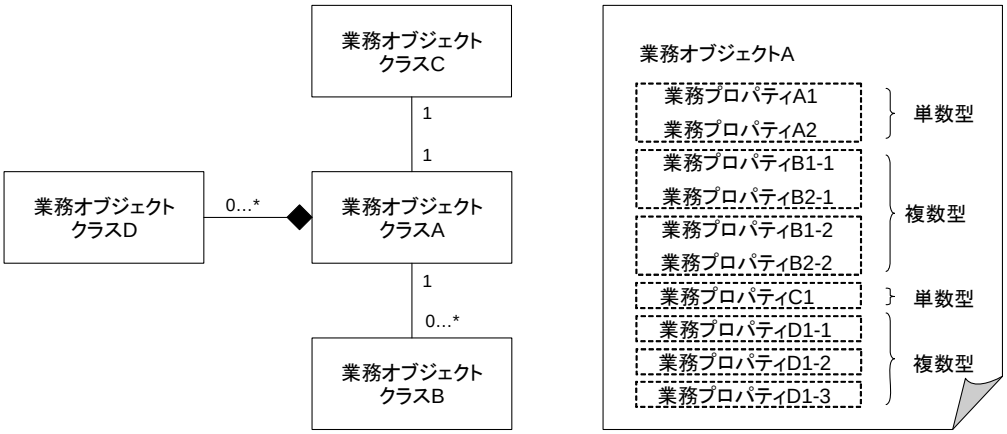
なお、複数型の業務プロパティを単数型として利用する場合には、上記のいずれの場合であっても、中間オブジェクトを意識することなく、インデクサによる値の設定および取得を行うことができます。

3. 複数型業務プロパティの変更依頼

◆ 複数型業務プロパティの追加方法

XML メッセージ作成の応用形において、複数型の業務プロパティの生成方法についてすでに説明しました。PPS ドキュメントサービスコンポーネントでは、プロファイルで定義された業務オブジェクトの構造として、その下位にある中間オブジェクトを明示的に定義することが可能です。複数型の業務プロパティは、その中間オブジェクトが業務オブジェクトに対して複数存在するという関係になっています。

ただし、これは XML メッセージにおける表現の話であり、サーバ側では、ある業務オブジェクトに対して、関係する独立した他の業務オブジェクトが 1 対多の関係にあるという状況に対応しています。



上図のように、たとえば業務オブジェクトクラス B に属する業務オブジェクト（インスタンス）は、業務オブジェクト A に関する XML メッセージ上では、中間オブジェクトとして扱われますが、サーバ側では、独立した業務オブジェクト（または RDB テーブルのレコード）である場合があります。

ここで、業務オブジェクトクラス B あるいは D に属する業務オブジェクトを追加したい場合で、該当する業務オブジェクトクラス B あるいは D がプロファイル上で業務オブジェクトとして定義されていない場合には、クライアントは、業務オブジェクト B あるいは D の追加依頼ドキュメントではなく、業務オブジェクト A の修正依頼ドキュメントとして XML メッセージを送信します。

たとえば、製品 ABC の在庫情報を、サーバ側で時系列でもっており、クライアントから新しい在庫情報を追加したいような場合には、製品 ABC に対する修正依頼ドキュメントを生成して送信します。また、部品表（BOM）の一部の部品についてその構成数を変更したり、その部品の利用を中止し構成リストから削除するような場合にも、修正依頼ドキュメントの生成が必要となります。

◆ 複数型プロパティの挿入（INSERT）

たとえば、複数型業務プロパティを追加するための修正依頼ドキュメントの生成のためには、以下のようなプログラムが必要となります。業務プロパティを複数型として追加するためには、以下のプログラムのように、セレクションのタイプを Insert とし、中間オブジェクトの単位で業務プロパティをまとめ、追加する数だけ繰り返しセレクションオブジェクトを生成します。

```
TransactionProcess process = manager.CreateProcess();
process.Confirm = TransactionProcess.ConfirmTypes.Always;
Document doc = process.CreateDocument("MaterialInventory");
doc.Action = Document.ActionTypes.Change;
doc.CreateCondition("材料Z01");

Selection sel = doc.CreateSelection(Selection.SelectionTypes.Insert);
sel["stock-value"] = 130;
sel["stock-time"] = new DateTime(2009, 1, 10, 18, 0, 0);

sel = doc.CreateSelection(Selection.SelectionTypes.Insert);
sel["stock-value"] = 100;
sel["stock-time"] = new DateTime(2009, 1, 11, 18, 0, 0);
```

◆ 複数型プロパティの修正（UPDATE）

また、複数型業務プロパティを修正するための修正依頼ドキュメントの生成のためには、以下のようなプログラムが必要となります。この例は、製品 ABC の部品表（ID は P001）の部品表（BOM）において、構成部品である“部品 B1”の構成数量が 4 から 8 へ変更になった状況に対応するものです。まず、セレクションオブジェクトのタイプを Update にします。そして、対象となる中間オブジェクトが部品 B1 に対応するものであるように、業務プロパティ用のコンディション情報を設定します。そして、その上で、修正後の値を指定します。なお、ここで新たに単位として“個”を設定しています。これは中間オブジェクト

単位でみた場合には追加ではなく、その内容の修正となりますので、ここで指定することが可能です。

```
TransactionProcess process = manager.CreateProcess();
process.Confirm = TransactionProcess.ConfirmTypes.Always;
Document doc = process.CreateDocument("Product");
doc.Action = Document.ActionTypes.Change;
doc.CreateCondition("P001");

Selection sel = doc.CreateSelection(Selection.SelectionTypes.Update);
sel.CreateCondition()["child-name"] = "部品B1";
sel["child-quantity"] = 8;
sel["child-quantity-unit"] = "個";
```

◆ 複数型プロパティの削除（DELETE）

最後に、複数型業務プロパティを削除するための修正依頼ドキュメントの生成のためのプログラムを示します。この例は、生産手順マスタの中で、製品 ABC（ID が P001）の生産に関するものすべてについて、その消費材料である P999 を中止し、削除するというものです。これによって、部品 P999 の消費に関する中間オブジェクトが削除され、それによって、該当する中間オブジェクトに属していた業務プロパティの値は、消費品目 ID を表す業務プロパティ consume-item-id 以外にも、消費品目名称 consume-item-name や、消費数量 consume-quantity 情報なども消去されます。

```
TransactionProcess process = manager.CreateProcess();
process.Confirm = TransactionProcess.ConfirmTypes.Always;
Document doc = process.CreateDocument("RoutingRecord");
doc.Action = Document.ActionTypes.Change;
doc.CreateCondition()["process-item-id"] = "P001";

Selection sel = doc.CreateSelection(Selection.SelectionTypes.Delete);
sel.CreateCondition()["consume-item-id"] = "P999";
```


4. 複数型業務プロパティの処理（サーバ側）

◆ 複数型プロパティをもつ業務ドキュメントの追加

複数型業務プロパティを含む業務ドキュメントを追加する場合には、対象とするデータベースのテーブルあるいはクラスとは別に、関係先のテーブルまたはクラスを用意する必要があります。複数型の業務プロパティは、それぞれ中間オブジェクトに対応するこれらの関連テーブルあるいは関連クラスに追加していきます。プログラムのサンプルを以下に示します。

以下のプログラムは、通常の業務オブジェクトの追加に加えて、複数型の業務プロパティの処理の部分だけを抽出しています。ここでは、GetRelationTable メソッドによって、関連テーブルあるいは関連クラスを取得しています。この結果、RelationTable に関連する業務プロパティが追加されます。

```
// 対象オブジェクトの複数型プロパティを設定します。(LEVEL2)
DomainObjectProfile[] subObjectProfileList = obj.GetGetPropertyObjectProfiles();
foreach (DomainObjectProfile profile in subObjectProfileList)
{
    DomainObject[] list = obj.GetPropertyObjects(profile);
    DataTable RelationTable = GetRelationTable(doc, profile);

    // 中間オブジェクトの数だけ関連テーブルのレコードを生成します。
    foreach (DomainObject subObj in list)
    {
        DataRow myRelationObj = RelationTable.NewRow();

        // 業務オブジェクトのIDを関係テーブルに設定します。
        string idName = doc.GetDomainObjectProfile().PrimaryKey.Name;
        if (idName != null && obj[idName] != null)
        {
            string idLocalName = GetLocalName(RelationTable.TableName,
                doc.DocumentName, idName);
            SetValue(RelationTable, myRelationObj, idLocalName, obj[idName]);
        }
        // 中間オブジェクトの業務プロパティを設定します。
        Property[] subProperties = subObj.GetAllProperties();
        foreach (Property prop in subProperties)
        {
            // 生成した中間オブジェクトに値を設定します。
            string localName = GetLocalName(RelationTable.TableName,
                doc.DocumentName, prop.Name);
```

```
                SetValue(RelationTable, myRelationObj, localName, prop.Value);
            }
            RelationTable.Rows.Add(myRelationObj);
        }
    }
```

◆ 複数型業務プロパティの変更方法

複数型のプロパティの修正依頼ドキュメントを受け取ったサーバは、ただちに、その依頼内容にしたがって、サーバがもつ該当する業務ドキュメントの内容を修正する必要があります。ただし、ここでの修正作業は、実質的には、中間オブジェクトに対応する別の業務オブジェクトに対する追加、修正、削除に相当している場合があります。以下に、具体的なプログラムを用いて説明します。

修正依頼ドキュメントの中で、セレクションオブジェクトに SelectionType プロパティがあり、そこに Insert、Update、Delete のいずれかが設定されている場合には、複数型の業務プロパティに対する処理となります。それぞれのセレクションオブジェクトは、複数型の業務プロパティが属する中間オブジェクトの単位となっています。したがって、先頭にある単数型の業務プロパティ用のセレクションオブジェクトと、構成する中間オブジェクトの数分のセレクションオブジェクトが存在しています。
以下のプログラムにおいて、セレクション情報によって指定された業務プロパティが所属する中間オブジェクトに対応してサーバ上のテーブルを RelationTable として定義しています。中間オブジェクトがサーバ上のどの業務オブジェクトあるいはテーブルに相当するかは、各アプリケーションの環境に依存します。

```
private void ServiceChange(Document doc)
{
    // 業務ドキュメントに対応したテーブルを取得します。
    DataTable MyTable = GetMyTable(doc);
    // 修正対象となる業務オブジェクトのIDのリストを作成します。
    List<DataRow> targetList = new List<DataRow>();
    foreach (Condition cond in doc.Conditions)
    {
        List<DataRow> list = FilterDatabase(MyTable, GetAllObjects(MyTable),
            cond, doc);
        foreach (DataRow id in list) if (targetList.IndexOf(id) < 0) targetList.Add(id);
    }
    // 対象となる業務オブジェクトを修正します。
    foreach (DataRow myObj in targetList)
    {
```



```
        object idValue = GetKeyValue(MyTable, myObj, doc);
        string keyLocalName = GetKeyName(MyTable, doc);

        bool modified = false;
        string errorMessage = null;
        //DataRow myObj = MyTable.Rows.Contains(id);
        foreach (Selection sel in doc.Selections)
        {
            if (sel.SelectionType == Selection.SelectionTypes.Insert)
            {
                // 複数型プロパティの挿入処理を記述します。
            }
            else if (sel.SelectionType == Selection.SelectionTypes.Update)
            {
                //  複数型プロパティの修正処理を記述します。
            }
            else if (sel.SelectionType == Selection.SelectionTypes.Delete)
            {
                //  複数型プロパティの削除処理を記述します。
            }
            else
            {
                // 複数型でない業務プロパティの処理を記述します。
            }
        }
        // 実際に修正が行われた場合は返信オブジェクトに記録します。
        string idString = (idValue == null) ? "IDが設定されてません。" :
            idValue.ToString();
        if (modified) doc.Reference.CreateDomainObject().Id = idString;
        if (errorMessage != null) CreateWarning(doc, idString, "011", errorMessage);
    }
}
```

以下では、上記のプログラムリストにおいて、複数型プロパティの挿入、修正、削除処理としてコメントとなっている部分の具体的な内容を説明します。

◆ 複数型プロパティ挿入（INSERT）の実施

まず、業務プロパティの追加のための処理としては、以下のようなプログラムが考えられます。ここでは、中間オブジェクトに対応する RelationTable にオブジェクトを１つ生成し、そこに要求された業務プロパティを設定しています。

```
    if (sel.SelectionType == Selection.SelectionTypes.Insert) // LEVEL2機能
    {
```



```
        // セレクション情報をもとに関連するテーブルを決定します。
        DomainObjectProfile subObjClass = sel.GetDomainObjectProfile();
        DataTable RelationTable = GetRelationTable(doc, subObjClass);
        if (RelationTable == null) break;

        // 業務プロパティを追加する中間オブジェクトを生成します。
        DataRow myRelationObj = RelationTable.NewRow();

        // 業務オブジェクトのIDを関係テーブルに設定します。
        string idLocalName = GetKeyName(RelationTable, doc);
        if (idLocalName != null && RelationTable.Columns.Contains(idLocalName))
        {
            if (idValue != null)
            {
                myRelationObj[idLocalName] = idValue;
                modified = true;
            }
        }
        // 中間オブジェクトに関する情報を設定します。
        foreach (Property prop in sel.Properties)
        {
            // 生成した中間オブジェクトに値を設定します。
            string localName = GetLocalName(RelationTable.TableName,
                doc.DocumentName, prop.Name);
            if (SetValue(RelationTable, myRelationObj, localName, prop.Value))
                modified = true;
        }
        RelationTable.Rows.Add(myRelationObj);
    }
```

◆ 複数型プロパティ修正（UPDATE）の実施

業務プロパティの修正のための処理は、以下のようになります。ここでは、セレクションオブジェクトの内部にもつコンディション情報によって、中間オブジェクト内での対象の限定が行われています。中間オブジェクト内でのコンディション情報の利用方法は、業務オブジェクト上での利用方法と同じです。これによって、選択された中間オブジェクトの業務プロパティが修正されます。

```
    else if (sel.SelectionType == Selection.SelectionTypes.Update) // LEVEL2機能
    {
        // セレクション情報をもとに関連するテーブルを決定します。
        DomainObjectProfile subObjClass = sel.GetDomainObjectProfile();
        DataTable RelationTable = GetRelationTable(doc, subObjClass);
        if (RelationTable == null) break;
```



```
        // 関連テーブルの該当レコードを対象レコードの条件で限定します。
        List<DataRow> subListBase = new List<DataRow>();
        foreach (Condition cond in doc.Conditions)
        {
            List<DataRow> list = FilterDatabase(RelationTable,
                GetAllObjects(RelationTable), cond, doc);
            foreach (DataRow subId in list)
                if (subListBase.IndexOf(subId) < 0) subListBase.Add(subId);
        }
        if (subListBase.Count == 0) continue;

        // 関連テーブルのレコードをさらに関連テーブルの条件で限定します。
        List<DataRow> subList = new List<DataRow>();
        foreach (Condition cond in sel.Conditions)
        {
            List<DataRow> list = FilterDatabase(RelationTable, subListBase, cond, doc);
            foreach (DataRow subId in list)
                if (subList.IndexOf(subId) < 0) subList.Add(subId);
        }
        if (subList.Count == 0) continue;

        // 条件にマッチした中間オブジェクトに値を設定します。
        foreach (DataRow myRelationObj in subList)
        {
            //DataRow myRelationObj = RelationTable.Rows.Contains(subId);
            foreach (Property prop in sel.Properties)
            {
                string localName = GetLocalName(RelationTable.TableName,
                    doc.DocumentName, prop.Name);
                if (SetValue(RelationTable, myRelationObj, localName, prop.Value))
                    modified = true;
            }
        }
    }
```

◆ 複数型プロパティ削除（DELETE）の実施

最後に、業務プロパティの削除の場合には、以下のプログラムのように、セレクションオブジェクト内部のコンディション情報によって、中間オブジェクトを限定した後に、そこで選択された中間オブジェクトを削除しています。

```
    else if (sel.SelectionType == Selection.SelectionTypes.Delete) // LEVEL2機能
    {
        // セレクション情報をもとに関連するテーブルを決定します。
```



```
        DomainObjectProfile subObjClass = sel.GetDomainObjectProfile();
        DataTable RelationTable = GetRelationTable(doc, subObjClass);
        if (RelationTable == null) break;

        // 関連テーブルの該当レコードを対象レコードの条件で限定します。
        List<DataRow> subListBase = new List<DataRow>();
        foreach (Condition cond in doc.Conditions)
        {
            List<DataRow> list = FilterDatabase(RelationTable,
                GetAllObjects(RelationTable), cond, doc);
            foreach (DataRow subId in list)
                if (subListBase.IndexOf(subId) < 0) subListBase.Add(subId);
        }
        if (subListBase.Count == 0) continue;

        // 関連テーブルのレコードをさらに関連テーブルの条件で限定します。
        List<DataRow> subList = new List<DataRow>();
        foreach (Condition cond in sel.Conditions)
        {
            List<DataRow> list = FilterDatabase(RelationTable, subListBase, cond, doc);
            foreach (DataRow subId in list)
                if (subList.IndexOf(subId) < 0) subList.Add(subId);
        }
        if (subList.Count == 0) continue;

        // 選択された業務オブジェクト（中間オブジェクト）を削除します。
        foreach (DataRow myRelationObj in subList)
        {
            RelationTable.Rows.Remove(myRelationObj);
            modified = true;
        }
    }
```

なお、複数型の業務プロパティに対する追加、修正、削除は、ひとつの修正依頼ドキュメントの中で、組み合わせて設定することができます。たとえば、ある条件にマッチする中間オブジェクトをいったん削除した後で、新たに中間オブジェクトを追加するといったことが可能です。これらの連続した処理は、クライアントが、セレクションオブジェクトを生成した順にしたがって、サーバ側で処理が行われます。

5. 複数型プロパティの照会および回答方法

◆ 業務オブジェクトの複数型業務プロパティの範囲を指定する方法

複数型の業務プロパティを利用してサーバ上の関連オブジェクトの内容を照会したい場合には、照会ドキュメントにおいて、セレクション情報のセレクションオブジェクトにコンディションオブジェクトを設定します。コンディションオブジェクトは、業務ドキュメントの下位に設定される場合と、セレクションオブジェクトの下位に設定される場合の２種類の使われ方があります。セレクションオブジェクトの下位に設定されたコンディション情報は、そのセレクションオブジェクトが示す中間オブジェクトの選択のための制約を示します。

セレクション情報の中で最初に設定されたオブジェクトは、単数型あるいは複数型を単数型として照会する場合のものであり、２つめ以降のオブジェクトにおいて、明示的に複数型の業務プロパティの照会内容を依頼します。ここでコンディション情報を付加しない場合には、関係する中間オブジェクトがもつすべてのデータが回答ドキュメントに設定されることになります。したがって、必要なオブジェクトの範囲を限定したい場合には、コンディション情報を指定する必要があります。

以下のプログラムは、資材倉庫にある材料 Z01 の在庫について、1 月 10 日以降のデータを要求するためのものです。ここでは、最初のセレクションオブジェクトにおいて、業務プロパティの中から資材の品目名称を返信することを要求し、２つ目のセレクションオブジェクトにおいて、複数型の業務プロパティである在庫日時と在庫量を指定しています。そして同時に、２つ目のセレクションオブジェクトでは、在庫量と在庫日時について、最早値の制約を指定しています。ここで、在庫量と在庫日時は、プロファイルの定義上では要素名“Capacity”、修飾名“pps:actual”によって識別される中間オブジェクトに所属しています。このように、２つ目以降のセレクションオブジェクトでは、同一の中間オブジェクトに含まれる業務プロパティ以外を設定してはいけません。また、中間オブジェクトに属していない複数型の業務プロパティを複数型として照会する場合には、それぞれ別々のセレクションオブジェクトに指定しなければなりません。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("MaterialInventory");
doc.Action = Document.ActionTypes.Get;
doc.CreateCondition()["material-name"] = "材料Z01";
```



```
    Selection sel;
    // 単数型としての業務プロパティの指定
    sel = doc.CreateSelection();
    sel.CreateProperty("material-name");

    // 複数型の業務プロパティの指定と制約の定義
    sel = doc.CreateSelection();
    sel.CreateProperty("stock-time");
    sel.CreateProperty("stock-value");
    sel.CreateCondition().SetConstraint("stock-time", new DateTime(2009, 1, 10),
        Pps.Documents.Constraint.ConstraintTypes.GE);
```

上記のプログラムを実行すると、以下のような照会依頼ドキュメントが生成されます。ここで、Condition 要素が、業務オブジェクトを選択するためのものと、セレクション情報の内部として中間オブジェクトを選択するものとが生成されていることが分かります。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="88" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="MaterialInventory" id="186" action="Get">
            <Condition>
                <Property name="material-name" type="Condition">
                    <Char value="材料Z01" condition="EQ" />
                </Property>
            </Condition>
            <Selection>
                <Property name="material-name" />
            </Selection>
            <Selection>
                <Condition>
                    <Property name="stock-time" type="Condition">
                        <Time value="2009-01-10T00:00:00" condition="GE" />
                    </Property>
                </Condition>
                <Property name="stock-time" />
                <Property name="stock-value" />
            </Selection>
        </Document>
    </Transaction>
</Message>
```


◆ 複数型の業務プロパティの照会に対する回答方法（サーバ側）

サーバは、複数型の業務プロパティの照会に対して、回答ドキュメントを生成し、サーバ上に存在する DB の内容にもとづき、その内容を回答します。以下のように、複数型の業務プロパティは、２つ目以降のセレクションオブジェクトの対応として返信ドキュメントに設定されて行きます。

まず、値を保持している関連テーブルを決定します。これは、GetRelationTable メソッドによってあらかじめ定義されたマッピング情報をもとに処理されます。そこで得られた関連テーブルのうち、必要に応じてコンディション情報を適用し、対象を絞り込みます。対象となるレコードが確定したら、その数だけあらかじめ中間オブジェクトを生成し、その後、各中間オブジェクトに対して関連テーブルにある値を設定していきます。ここで、関連テーブル上のフィールド名と、業務プロパティ名とのマッピングを GetLocalName メソッドによって行っています。

```
// 業務オブジェクトを選択し、その内容を新規に返信ドキュメントに追加します。
for (int i = dataOffset; i < dataOffset + dataCount && i < targetList.Count; i++)
{
    DataRow myObj = targetList[i];
    DomainObject obj = doc.Reference.CreateDomainObject();
    foreach (string key in selections)
    {
        // 単数型の業務プロパティの値を設定します。
        string localName = GetLocalName(MyTable.TableName, doc.DocumentName, key);
        object value = GetValue(MyTable, myObj, localName);
        if(value!=null && !(value is DBNull)) obj[key] = value;
    }
    // 複数型の業務プロパティの値を設定します。(LEVEL2)
    for (int k = 1; k < doc.Selections.Count; k++) // カウンタは１から開始
    {
        // 値を保持している関連テーブルを決定します。
        Selection sel = doc.Selections[k];
        DomainObjectProfile subObjClass = sel.GetDomainObjectProfile();
        DataTable RelationTable = GetRelationTable(doc, subObjClass);
        if (RelationTable == null) continue;

        // 関連テーブルの該当レコードを対象レコードの条件で限定します。
        List<DataRow> subListBase = new List<DataRow>();
        foreach (Condition cond in doc.Conditions)
        {
            List<DataRow> list = FilterDatabase(RelationTable,
                GetAllObjects(RelationTable), cond, doc);
            foreach (DataRow subId in list)
```



```
                    if (subListBase.IndexOf(subId) < 0) subListBase.Add(subId);
            }
            if (subListBase.Count == 0) continue;
            // 関連テーブルのレコードをさらに関連テーブルの条件で限定します。
            List<DataRow> subList = new List<DataRow>();
            foreach (Condition cond in sel.Conditions)
            {
                List<DataRow> list = FilterDatabase(RelationTable,
                    subListBase, cond, doc);
                foreach (DataRow subId in list)
                    if (subList.IndexOf(subId) < 0) subList.Add(subId);
            }
            if (subList.Count == 0) continue;

            // 返信用のオブジェクトに中間オブジェクトを複数生成します。
            DomainObject[] subObjList = obj.CreatePropertyObjects(
                subObjClass.ElementName, subObjClass.Modifier, subList.Count);
            for (int j = 0; j < subList.Count; j++)
            {
                DataRow mySubObj = subList[j];
                foreach (Property prop in sel.Properties)
                {
                    // 生成した中間オブジェクトにプロパティを設定します。
                    string localName = GetLocalName(RelationTable.TableName,
                        doc.DocumentName, prop.Name);
                    object value = GetValue(RelationTable, mySubObj, localName);
                    if (value != null && !(value is DBNull))
                        subObjList[j][prop.Name] = value;
                }
            }
        }
    }
```


6. より高度な照会機能

◆ データ数制限とオフセット照会機能（クライアント側）

サーバにある業務オブジェクトの照会を行う際に、該当するデータ数が膨大な数の場合に送受信する XML メッセージのサイズが大きくなり、レスポンスが低下します。これを避けるために、データ数制限の指定とオフセット指定を行うことができます。サーバ上の業務オブジェクトが、あらかじめ何らかの順で並べられている場合に、先頭からのオフセット値を変えることで、対象データを分割して受け取ることが可能となります。

たとえば、対象となる業務オブジェクトのデータ数が 14,000 件あったとすると、データ数制約を 5,000 とし、第一回目のオフセット値を 0、第二回目の照会でオフセット値を 5,000、そして第三回目の照会でオフセット値を 10,000 とすることで、すべての情報を取得できます。以下のプログラムは、このような複数回に分けた照会を行った場合の三回目の例です。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("RoutingRecord");
doc.Action = Document.ActionTypes.Get;

Selection sel = doc.CreateSelection(Selection.SelectionTypes.All);
sel.DataCount = 5000;
sel.DataOffset = 10000;
```

◆ データ数制限とオフセット照会機能（サーバ側）

クライアントからの要求の中には、カウンタによる最大データ数の制約やオフセット制約などが付加される場合があります。また、特定の業務プロパティによるソートや、業務プロパティの値の集計などの要求が含まれる場合があります。これらの追加的な要求はセレクション情報に付加されています。

以下に、まず、カウンタによる最大データ数の制約とオフセット制約に対応するプログラムの一般形を示します。これは、ServiceGet メソッドの一部を変更することで実現します。まず、セレクションオブジェクトが複数ある場合には、先頭のものからカウンタおよびオフセットを取得します。そして、targetList に設定された対象業務オブジェクトについて、オフセット値からはじまるカウンタによって業務オブジェクトを取得しています。


```
int dataCount = 0;
int dataOffset = 0;
if (doc.Selections.Count > 0)
{
    dataOffset = doc.Selections[0].DataOffset;
    dataCount = doc.Selections[0].DataCount;
}
if (dataCount == 0) dataCount = targetList.Count;
if (dataOffset > 0) dataCount += dataOffset;

// 業務オブジェクトを選択し、その内容を新規に返信ドキュメントに追加します。
for (int i = dataOffset; i < dataOffset + dataCount && i < targetList.Count; i++)
{
    DataRow myObj = targetList[i];
    // 業務オブジェクトに対する処理を記述します。
}
Header header = doc.Reference.CreateHeader();
// 該当する業務オブジェクト数を設定します。
header.TotalObjectCount = targetList.Count;
// ヘッダにオフセット値を設定します（LEVEL2）
header.Offset = dataOffset;
```

◆ ソートと集計機能（クライアント側）

照会ドキュメントでは、サーバがもつ DB にある業務オブジェクトの内容を照会すると同時に、その回答ドキュメントの中で、業務オブジェクトのリストを特定の業務プロパティの値の大小関係でソートさせることができます。ソートの基準となる業務プロパティがもつ SortType プロパティの値を、Asc または Desc に指定してください。ソートの基準となる業務プロパティは複数設定することができますが、それらのすべてが反映されるかどうかは、サーバの機能に依存します。最低、先頭にある業務プロパティの値で結果がソートされます。以下の例は、受注オーダを納期順にソートするものです。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("SalesOrder");
doc.Action = Document.ActionTypes.Get;

Selection sel = doc.CreateSelection();
sel.CreateProperty("sales-id");
sel.CreateProperty("sales-party-id");
sel.CreateProperty("sales-item-name");
```



```
    sel.CreateProperty("quantity-plan");
    sel.CreateProperty("due-time-schedule").SortType = Property.SortTypes.Asc;
```

また、特定の業務プロパティについて、集計するように指定することができます。このためには、集計する業務プロパティの CalculationType プロパティの値に演算の種類を指定します。指定できる演算子は合計（Sum）、平均（Ave）、最大（Max）、最小（Min）そしてデータ数（Count）です。以下の例は、合計金額（price-amount）の合計値、および納期（delivery-time-schedule）の最早のものを計算するように指定しています。計算された結果は、回答ドキュメントのヘッダ情報の中に設定されてサーバより返信されます。

```
    Selection sel = doc.CreateSelection();
    sel.CreateProperty("sales-id");
    sel.CreateProperty("price-amount").CalclationType = Property.CalclationTypes.Sum;
    sel.CreateProperty("quantity-plan").CalclationType =
        Property.CalclationTypes.Max;
```

◆ ソートと集計機能（サーバ側）

次に、ソートおよび集計要求への対応例についてプログラムを示します。以下の例では、ServiceGet メソッドの中で、ソート情報と集計情報を収集し、その結果をいったんリストとして保持しています。そしてその後、作成したリストを用いて、SortTargetList メソッド、および CalcTargetList メソッドの中で、それぞれの処理が行われています。

```
    // ソート情報と集計情報に対する処理を行います。(LEVEL2)
    List<Property> sortList = new List<Property>();
    List<Property> calcList = new List<Property>();
    foreach (Selection sel in doc.Selections)
    {
        foreach (Property prop in sel.Properties)
        {
            if (prop.SortType != Property.SortTypes.Undefined) sortList.Add(prop);
            if (prop.CalclationType != Property.CalclationTypes.Undefined)
                calcList.Add(prop);
        }
    }
    SortTargetList(sortList, targetList, MyTable, doc);
    CalcTargetList(calcList, targetList, MyTable, doc);
```


ここでは、集計処理である CalcTargetList メソッドについてのみ、その例をあげて手順を解説します。まず、以下のように CalcTargetList メソッドの中では、集計すべき業務プロパティ個々について、その集計方法に応じたメソッド、たとえば合計（Sum）の場合には GetCalculateSum メソッドを実行し、その戻り値として集計値を得ます。そこで得られた集計結果は、セレクションオブジェクトがもつ元の業務プロパティの Value プロパティの値にいったん設定されます。

```
private void CalcTargetList(List<Property> list, List<DataRow> targetList, DataTable
table, Document doc)
{
    foreach (Property prop in list)
    {
        string localName = GetLocalName(table.TableName, doc.DocumentName, prop.Name);
        if (localName == null) continue;
        switch (prop.CalclationType)
        {
            case Property.CalclationTypes.Sum:
                prop.Value = GetCalculateSum(table, localName,
                    targetList, prop.DataType);
                break;
            case Property.CalclationTypes.Ave:
                prop.Value = GetCalculateAve(table, localName,
                    targetList, prop.DataType);
                break;
            case Property.CalclationTypes.Max:
                prop.Value = GetCalculateMax(table, localName,
                    targetList, prop.DataType);
                break;
            case Property.CalclationTypes.Min:
                prop.Value = GetCalculateMin(table, localName,
                    targetList, prop.DataType);
                break;
            case Property.CalclationTypes.Count:
                prop.Value = GetCalculateCount(table,localName,
                    targetList, prop.DataType);
                break;
        }
    }
}
```

それぞれの集計用メソッドの中で、合計（Sum）を行う GetCalculateSum メソッドについての例を示します。以下のように、対象とする業務オブジェクトのリスト targetList の各データ内容を取り出し、sum 変数に加算していきます。対象となるデータは、業務オブジェクト myObj において、業務プロパティ名が localName のものです。下記のプログラムで

は、この値を数値に変換したのちに集計を行っています。

```
private object GetCalculateSum(DataTable table, string localName, List<DataRow>
targetList, Property.DataTypes type)
{
    if (type == Property.DataTypes.Qty)
    {
        decimal sum = 0;
        foreach (DataRow myObj in targetList)
        {
            object value = GetValue(table, myObj, localName);
            if (value != null && !(value is DBNull)) sum += CastToDecimal(value);
        }
        return sum;
    }
    throw new PpsDocumentsException("合計値は計算できません。" + localName);
}
```


7. ヘッダ情報の高度な利用

◆ ヘッダ用業務オブジェクトの利用

通常、ヘッダ用業務オブジェクトのクラスは、業務ドキュメントで指定された業務オブジェクトと同じものがデフォルトで設定されます。したがって、業務ドキュメント本体の業務プロパティとヘッダのそれとは同じ候補の中から選択されることになります。しかし、もし、ヘッダの生成時に、独自の業務オブジェクトのクラスを指定した場合には、本体である業務オブジェクトと異なる内容をヘッダに設定可能となります。

たとえば、ある仕入先がもつ製品リストを作成する場合、そのヘッダ情報として、該当する仕入先の情報を記載したい場合あります。以下の図は、ヘッダ情報として、タイトルおよび仕入先名とカテゴリの業務プロパティが設定されています。ここでは、本体となる業務オブジェクトはProductクラスであるのに対して、ヘッダ用業務オブジェクトはSupplierクラスとなります。

商社別部品リスト		
仕入先名	XYZ商事	
カテゴリ	部品商社	
ID	名称	価格
K001	ネジ	50
K002	シャフト	250

図３－３　ヘッダ用業務オブジェクトの利用例

以下に、上記の事例に対応するプログラムおよびXMLメッセージを示します。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("Product");

Header header = doc.CreateHeader("Supplier");
header.Title = "商社別部品リスト";

header["supplier-name"] = "XYZ商事";
header["supplier-category"] = "部品商社";

DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "K002";
obj["product-name"] = "ネジ";
obj["standard-price"] = 50;
```

```
    obj = doc.CreateDomainObject();
    obj["product-id"] = "K003";
    obj["product-name"] = "シャフト";
    obj["standard-price"] = 50;
```

上記のプログラムにおいて、CreateHeader メソッドを実行する際に、引数として“Supplier”を指定している点に注目してください。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="89" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="Product" id="187" action="Notify">
            <Header class="Supplier" title="商社別部品リスト">
                <Property name="supplier-name" type="Target" value="XYZ商事" />
                <Property name="supplier-category" type="Target" value="部品商社" />
            </Header>
            <Item id="K002">
                <Spec type="pps:name">
                    <Char value="ネジ" />
                </Spec>
                <Price type="pps:standard">
                    <Qty type="pps:general" value="50" />
                </Price>
            </Item>
            <Item id="K003">
                <Spec type="pps:name">
                    <Char value="シャフト" />
                </Spec>
                <Price type="pps:standard">
                    <Qty type="pps:general" value="50" />
                </Price>
            </Item>
        </Document>
    </Transaction>
</Message>
```

◆ ヘッダによる業務オブジェクトの照会

照会において、ヘッダ情報に業務オブジェクトのクラスおよびインスタンスの ID を付加してサーバに送信することで、業務ドキュメント本体における照会とは別に、ヘッダ独自の照会を行うことができます。これは、業務ドキュメント本体に設定される業務オブジェク

トの種類と、ヘッダの業務オブジェクトの種類が異なる場合に便利です。

たとえば、前の節にように、ヘッダ情報に仕入先オブジェクトが設定され、業務オブジェクト本体には発注オーダの明細が設定されているような帳票を得たい場合には、ヘッダ情報に該当する仕入先の ID を設定することで、ヘッダ情報にその仕入先の詳細な情報（仕入先名や住所、担当者など）を一回の照会によって得ることが可能となります。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("Product");

Header header = doc.CreateHeader("Supplier");
header.ObjectId = "K00123";
header.CreateProperty("supplier-name");
header.CreateProperty("supplier-category");

Selection sel = doc.CreateSelection(Selection.SelectionTypes.None);
sel.CreateProperty("product-id");
sel.CreateProperty("product-name");
sel.CreateProperty("standard-price");
```


8. イベント通知機能

◆ イベント通知機能の概要

イベント依頼およびイベント通知機能は、レベル１実装においても必要となりますが、この章でまとめて説明しています。イベント通知機能とは、サーバが管理する特定のイベントが発生した場合に、そのサーバに対して通知を依頼した複数のクライアントに同時に通知を行うものです。イベントの発生は、サーバが管理する変数（業務プロパティの値）の変化によって判定可能とします。

まず、サーバは、自分の管理する提供可能なイベント通知サービスについて、その数だけ業務プロファイルに定義し公開します。クライアントは、サーバが公開している業務プロファイルを任意のタイミングで照会することができます。

もし、クライアントの一つがサーバの提供するイベント通知を要求する場合には、同期（Sync）メッセージをサーバに送信します。サーバは、クライアントの ID とトランザクション ID を保持し、イベントの発生を一定の間隔で監視します。もし、イベントが発生した場合には、そのイベントの通知を要求するクライアントのリストに従い、イベント内容を通知ドキュメントによって送信します。以上の全体の流れを以下の図に示します。

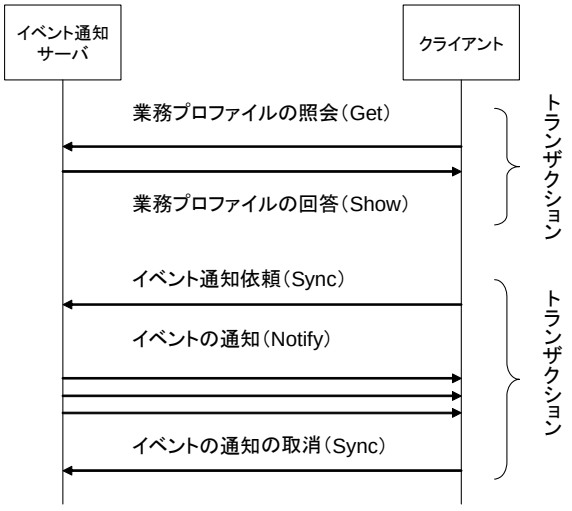


図　イベント通知処理の概要

図に示すように、イベント通知処理は、クライアントからのトランザクションの取消のメッセージによって終了することができます。まだ、同様に、サーバ側からもイベント通知

処理を強制的に終了させることができます。

◆ イベント通知機能の定義

イベント通知機能は、各業務アプリケーションがあらかじめ実装プロファイルによって宣言します。クライアントは、この宣言を見て、通知の依頼をすることになります。以下に、イベント機能の定義の例を示します。ここでイベント１は、“価格が変更になる”というイベントであり、イベント２は“製品データが新たに追加された”というイベントであり、イベント３は“価格が 1000 円を超えた”というイベントとなります。したがって、これらのイベントの通知を希望するクライアントは、このサーバに対して同期依頼ドキュメントを送信することになります。

イベントの種類は、以下の表のように５種類定義されています。対象となる値や常態のチェックは、サーバが定義した実行間隔で実施され、その間にどのような変化があっても次のイベント発生チェックのサイクルまでは影響しません。True や False の場合、たとえば常に True であった場合には、この実行間隔でイベント通知が送信されることになります。また、Change の場合、前回の実行時点と今回との値の比較の結果によってイベント発生を判定するため、その間に値が変更になったとしても無視されます。

表　イベント発生の判定区分

識別記号	説明	備考
True	条件が真である場合に常に通知します。	
False	条件が偽である場合に常に通知します。	
Enter	条件が偽から真に変更になった場合に通知します。	
Leave	条件が真から偽に変更になった場合に通知します。	
Add	オブジェクトまたはプロパティが追加になったら通知します。	
Remove	オブジェクトまたはプロパティが削除されたら通知します。	
Change	プロパティの値が変化したら通知します。	
Always	監視サイクルごとに常に通知します。	

```
// 業務ドキュメントの登録
ImplementDocument doc = manager.Implement.AddDocument("Product",
    "Subscription-01");
```



```
    // アクション種別の登録
    doc.AddAction(Document.ActionTypes.Sync, ImplementAction.RoleTypes.Server, 1);

    // 業務プロパティの登録
    doc.AddProperty("product-id", "ID", true, true, false, "製品ID");
    doc.AddProperty("product-name", "名称", true, true, true, "製品名");
    doc.AddProperty("standard-price", "価格", false, true, false, "標準価格");

    // イベントの登録１
    doc.AddEvent("PriceChangeEvent001","standard-price",
        ImplementEvent.EventTypes.Change,
        TimeSpan.FromDays(10), "価格が変更になったら通知します。");

    // イベントの登録２
    doc.AddEvent("PriceChangeEvent002", null, ImplementEvent.EventTypes.Add,
        TimeSpan.FromDays(1), "製品が追加されたら通知します。");

    // イベントの登録３
    ImplementEvent implementEvent = doc.AddEvent("ProceChangeEvent003",
        "standard-price", ImplementEvent.EventTypes.Enter, TimeSpan.FromDays(1),
        "価格が円を超えたら通知します。");
    implementEvent.Start = new DateTime(2009, 06, 24);
    implementEvent.Expire = new DateTime(2009, 06, 26);
    implementEvent.SetConstraint(1000, Property.DataTypes.Qty,
        Pps.Documents.Constraint.ConstraintTypes.GT);
```

上記のプログラムを実行した結果、コンポーネントの内部に実装プロファイルの定義情報が生成されます。この内容は、外部から実装プロファイルの照会があった時点で、以下のような XML ファイルとして返信されます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="84" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <ImplementProfile id="PSLX001" action="Show">
        <ImplementDocument name="Product" option="Subscription-01"
            profile="pslx-platform-1.0">
            <ImplementAction action="Sync" role="Server" />
            <ImplementProperty name="product-id" title="ID" use="Required"
                type="Typical" description="製品ID" />
            <ImplementProperty name="product-name" title="名称" multiple="Unbounded"
                use="Required" type="Typical" description="製品名" />
            <ImplementProperty name="standard-price" title="価格" type="Typical"
                description="標準価格" />
            <ImplementEvent name="ProceChangeEvent001" property="standard-price"
                type="Change" cycle="P10DT0H0M0S" expire="9999-12-31T23:59:59"
                description="価格が変更になったら通知します。" />
            <ImplementEvent name="ProceChangeEvent002" type="Add"
```



```
                cycle="P1DT0H0M0S" expire="9999-12-31T23:59:59"
                description="製品が追加されたら通知します。" />
            <ImplementEvent name="ProceChangeEvent003" property="standard-price"
                type="Enter" cycle="P1DT0H0M0S"
                start="2009-06-24T00:00:00" expire="2009-06-26T00:00:00"
                description="価格が1000円を超えたら通知します。">
                <Property type="Condition">
                    <Qty value="1000" condition="GT" />
                </Property>
            </ImplementEvent>
        </ImplementDocument>
    </ImplementProfile>
</Message>
```

参考のために、以下に MES（製造実行システム）における実装プロファイルの一部を示します。ここでは、装置のいくつかのパラメータを監視するための定義が示されています。この場合は、複数型のプロパティとなるために、Selection 要素によって、どのプロパティを監視対象とするかについて定義する必要があります。

```
    // 業務ドキュメントの登録
    ImplementDocument doc = manager.Implement.AddDocument("EquipmentState",
        "Monitoring-01");

    // アクション種別の登録
    doc.AddAction(Document.ActionTypes.Sync, ImplementAction.RoleTypes.Server, 1);

    // 業務プロパティの登録
    doc.AddProperty("equipment-id", "装置ID", true, true, false, "装置ID");
    doc.AddProperty("equipment-name", "装置名", true, true, false, "装置名称");
    doc.AddProperty("monitoring-id", "項目ID", true, true, true, "モニタリング項目");
    doc.AddProperty("monitoring-value", "値", true, true, true, "モニタリング結果");

    // イベントの登録１
    ImplementEvent implementEvent = doc.AddEvent("Monitoring001", "monitoring-value",
        ImplementEvent.EventTypes.Change, TimeSpan.FromMilliseconds(100),
        "装置Xの制御項目Aが変更になったら通知します。");
    implementEvent.CreateCondition("装置X");
    implementEvent.CreateSelection("monitoring-id", "制御項目A");

    // イベントの登録２
    implementEvent = doc.AddEvent("Monitoring002", "monitoring-value",
        ImplementEvent.EventTypes.Enter, TimeSpan.FromMilliseconds(100),
        "装置Xの制御項目Bが100～130の範囲になったら通知します。");
    implementEvent.CreateCondition("装置X");
    implementEvent.CreateSelection("monitoring-id", "制御項目B");
    implementEvent.SetConstraint(100, Property.DataTypes.Qty,
```



```
        Pps.Documents.Constraint.ConstraintTypes.GE);
    implementEvent.SetConstraint(130, Property.DataTypes.Qty,
        Pps.Documents.Constraint.ConstraintTypes.LE);
```

◆ クライアントからの依頼方法

クライアントがサーバに対してイベント通知を要求する場合には、サーバの実装プロファイルに記述されたイベント名を同期依頼ドキュメントに指定します。以下の例では、先に説明したイベント１“価格が変更になる”が発生した場合の通知を依頼しています。ここで、サーバからのイベント通知の識別や、通知依頼の取消のときに必要となるトランザクション ID を保管しておきます。

```
    TransactionProcess process = manager.CreateProcess();
    process.Confirm = TransactionProcess.ConfirmTypes.Always;

    Document doc = process.CreateDocument("Product", "Subscription-01");
    doc.Action = Document.ActionTypes.Sync;
    doc.EventName = "PriceChangeEvent001";

    // トランザクションIDを記録します。
    myEventId = process.Id;
```

上記のプログラムの実行によって、以下のような XML が生成され、サーバに送られます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="90" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" confirm="Always">
        <Document name="Product" id="188" action="Sync" option="Subscription-01"
            event="PriceChangeEvent001" />
    </Transaction>
</Message>
```

クライアントは必要に応じて、イベントの通知依頼を取り消すことができます。サーバはこのメッセージを受け取ると、そのクライアントに対するイベント通知は終了します。

```
    TransactionProcess process = manager.ResumeProcess(myEventId);
    process.TransactionCancel();
```



```
    return process.CreateMessage();
```

イベント通知の取消のための XML は、以下のようになります。ここで、id 属性にある値はトランザクション ID であり、イベント通知依頼として最初にクライアントが生成してサーバに依頼した ID です。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="91" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" type="Cancel" confirm="Always" />
</Message>
```

◆ サーバにおける依頼受付処理

サーバは、クライアントかあのイベント通知依頼を受け付けると、その情報をイベント通知のための配信リストに登録します。クライアントに対して、実際のイベント通知が行われるのは、それより後のイベント発生確認サイクルの中で行われます。以下のプログラムは、サーバのメイン処理から呼ばれるものであり、業務ドキュメントのアクション種別が Sync となっている場合の処理です。GetMonitor メソッドによって、イベント監視を行っているものの中から、該当するイベント監視を最初に取り出しています。この処理の最後に Subscriber クラスのインスタンスを生成し、そこにクライアントを設定し登録しています。実際に監視する業務オブジェクトは、同期依頼ドキュメントのコンディション要素の設定によって複数となる可能性があります。そのような場合には、Subscriber オブジェクトをその分だけ生成して登録しています。

```
private void ServiceSync(Document doc, TransactionProcess process)
{
    // 業務ドキュメントに対応したテーブルを取得します。
    DataTable MyTable = GetMyTable(doc);
    // 業務ドキュメントに対応したイベント定義情報を取得します。
    Monitor monitor = GetMonitor(doc);
    // 照会の対象となる業務オブジェクトのIDのリストを作成します。
    List<DataRow> targetList = new List<DataRow>();
    foreach (Condition cond in doc.Conditions)
    {
        List<DataRow> list = FilterDatabase(MyTable, GetAllObjects(MyTable),
            cond, doc);
        foreach (DataRow id in list) if (targetList.IndexOf(id) < 0) targetList.Add(id);
```



```
    }
    if (doc.Conditions.Count == 0) targetList = GetAllObjects(MyTable);

    // 業務オブジェクトを選択し、その内容を新規に返信ドキュメントに追加します。
    for (int i = 0; i < targetList.Count; i++)
    {
        DataRow myObj = targetList[i];
        object id = GetKeyValue(MyTable, myObj, doc);
        string idString = (id == null) ? "IDが設定されてません。" : id.ToString();

        // 返信用業務ドキュメントにIDを設定します。
        doc.Reference.CreateDomainObject().Id = idString;

        // イベント配信のための情報を設定します。
        Subscriber subscriber = new Subscriber();
        subscriber.PartyId = process.InitiatorName;
        subscriber.TransactionId = process.Id;
        subscriber.CreateDate = DateTime.Now;
        subscriber.Reference = myObj;
        monitor.subscribers.Add(subscriber);
    }
    doc.Reference.EventName = monitor.eventId;
}
```

サーバ上では、クライアントからのキャンセルの受付は上記とは異なる方法で受け取ります。これは、トランザクション処理の取消と同等の処理となりますので、以下のようなプログラムとなります。以下のメソッドは、取消イベントが共通コンポーネントの内部で発生した場合のコールバック関数です。ここで、該当する配信リストから除外しています。

```
private void manager_TransactionCancel(TransactionProcess transaction)
{
    // トランザクションの取り消し処理をここに追加する。
    foreach (Document doc in transaction.DocumentsReceived)
    {
        if(doc.Action == Document.ActionTypes.Sync && doc.EventName!=null)
        {
            Monitor monitor = GetMonitor(doc);
            if(monitor!=null)
            {
                monitor.CancelSubscriber(transaction.InitiatorName, transaction.Id);
                return;
            }
        }
    }
}
```


◆ イベントの監視と通知ドキュメントの生成

クライアントからの依頼により監視が開始されたら、あらかじめ定義されたサイクルで定期的にイベント発生をチェックします。もし、イベント発生が認められたら、配信リストにあるクライアントに対して、業務ドキュメントを生成し送信します。以下のプログラムは、イベントは発生した場合の通知ドキュメントの生成プログラムです。

```
private void event01(Monitor monitor, Subscriber subscriber)
{
    // イベント通知処理（サーバ側）

    // イベントが発生した場合に、サーバが登録された相手に通知メッセージを送信します。
    TransactionProcess process = manager.ResumeProcess(subscriber.TransactionId);
    Document doc = process.CreateDocument(monitor.documentName);
    doc.EventName = monitor.eventId;
    doc.Action = Document.ActionTypes.Notify;

    DomainObject obj = doc.CreateDomainObject();
    obj[monitor.definition.Property] = monitor.currentValue;

    TransactionMessage message01 = process.CreateMessage();
    // このメッセージの送信手順を記述します。
}
```

サーバは、自分の都合でイベントの通知を取消すことができます。その場合には、イベント通知取消をクライアントに伝える必要があります。イベント通知の取り消しは、業務ドキュメントを用いずに行います。イベント通知のキャンセルは、ResumeProcess メソッドを利用します。

```
private void event02(Monitor monitor, Subscriber subscriber)
{
    // イベントが発生した場合に、サーバが登録された相手に通知メッセージを送信します。
    TransactionProcess process = manager.ResumeProcess(subscriber.TransactionId);
    process.TransactionCancel();

    TransactionMessage message01 = process.CreateMessage();
    // このメッセージの送信手順を記述します。
}
```


付録　サンプル実装プログラム

本仕様書と合わせて、以下のサンプルプログラムが利用可能です。これらは、VisualStudio2005 のプロジェクトとして開発されたものです。本仕様書の内容は、これらのサンプルプログラムからの抜粋であり、本仕様書の内容を実際に起動することで理解を深めることができます。また、これらのサンプルプログラムをベースとして、実際のアプリケーションプログラムを開発することが可能です。

プロジェクト名	概要
PSLX_ClientServer_Level1	本仕様書第 1 部（レベル 1 実装）に対応したサンプルプログラムです。PSLX プロファイルに対応した業務ドキュメントの生成やサーバ側での処理、そして実装プロファイルの作成などのサンプルが含まれています。
PSLX_ClientServer_Level2	本仕様書第 2 部（レベル 2 実装）に対応したサンプルプログラムです。業務ドキュメントが複数型のプロパティを含む場合や、より高機能な照会方法、そしてイベント処理などのサンプルが含まれています。

