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１部

PPS ドキュメントサービス

＜レベル１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 1 部「PPS ドキュメントサービス（レベル 1 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 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。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。

◆ 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. XMLメッセージの作成方法

◆ XMLメッセージ作成準備

XML メッセージの作成および解釈のためのコンポーネントとして、Pps.Documents.dll が提供されています。まず、プロジェクトの参照設定において、“参照の追加”を選択し、“参照”タブの中で、このコンポーネントを選択します。また、以下のように、using の設定をファイルの先頭で指定します。

```
using Pps.Documents;
```

XML メッセージの作成および解釈のためには PPS ドキュメントマネージャ（DocumentManager）クラスのオブジェクトを生成します。以下のように、生成後、Initialize メソッドによって、利用するプロファイルのファイル名（フルパス）を指定する必要があります。

```
// PPSドキュメントマネージャ
DocumentManager manager;
manager = new DocumentManager("PSLX001");
manager.Initialize(Application.StartupPath + @"¥profile-pslx.xml");
```

PPS トランザクションマネージャは、内部で以下の XML スキーマを利用しています。XML スキーマの位置は、デフォルトで実行プログラムが存在するフォルダとなっていますが、もしこの位置を変更する場合には、以下のように、Initialize メソッドを呼ぶ前に各 XML スキーマの位置を指定する必要があります。

```
// PPSドキュメントマネージャ
DocumentManager manager;
manager = new DocumentManager();
manager.FnPSLXSchema = @"C:¥Schema¥pps-schema-1.0.xsd";
manager.Initialize(Application.StartupPath + @"¥profile-pslx.xml");
```


◆ 業務メッセージ、業務ドキュメントおよび業務トランザクションID

業務メッセージ、業務ドキュメントおよび業務トランザクションには、アプリケーションについてユニークな ID が、コンポーネントによって自動設定されます。ただし、アプリケーションプログラマは、ID カウンタおよび ID フォーマットプロパティを指定することで、自動設定の方法を変更することができます。

プロパティ種類	ID カウンタ	ID フォーマット
業務メッセージ	MessageCounter	MessageIdFormat
業務トランザクション	TransactionIdCounter	TransactionIdFormat
業務ドキュメント	DocumentIdCounter	DocumentIdFormat

たとえば、以下のように業務ドキュメントの ID カウンタと ID フォーマットを設定すると、結果として業務ドキュメントの ID には“PPS001234”という値が設定され、以後、業務トランザクションを生成する都度、数値の 6 桁の部分が 1 ずつカウントアップします。ここで、ID フォーマットに指定する文字列は、.Net Framework 標準の数値書式指定文字列またはカスタム数値書式指定文字列の形式に従って指定してください。

```
manager.TransactionIdCounter = 1234;
manager.TransactionIdFormat = "'PPS'000000";
```

なお、業務トランザクション用 ID カウンタ、および業務ドキュメント用 ID カウンタは、アプリケーションが終了後、その値が設定ファイルに保存されます。次回に起動したときに、その値を再度ファイルから読み込みます。したがって、ID カウンタの値を、この設定ファイルを書き換えることで行うことも可能です。

◆ トランザクションサービスの終了

トランザクションマネージャは、最終的に処理を終了するために、以下のように Close メソッドによって処理を終了させなければなりません。これによって、トランザクションマネージャは必要な情報をファイル等に設定保存します。

```
manager.Close();
```


◆ XMLメッセージ生成の基本形

XML メッセージを生成するには、以下のように、(1)業務トランザクションの生成、(2)業務ドキュメントの生成、(3)業務オブジェクトの生成、そして(4)業務プロパティの設定の順で行います。

業務ドキュメントの生成には、業務ドキュメント名の指定が必要です。以下の例では“Product”という業務ドキュメント名が指定されています。指定可能な業務ドキュメント名は、あらかじめプロファイルで定義されていますので、プロファイル定義書を参照してください。
業務オブジェクトの生成には、業務オブジェクト名の指定は不要です。これは、業務ドキュメントの種類によって、業務オブジェクトが一意に決定されるからです。業務ドキュメントと業務オブジェクトの対応関係は、プロファイル定義書を参照してください。

業務プロパティの定義は、以下のように、インデクサを用いて、引数として業務プロパティ名を指定します。指定可能な業務プロファイル名は、業務オブジェクトごとにあらかじめ定義されています。また、それぞれの業務プロパティは、数値型、文字型、日時型のいずれかがあらかじめ定義されています。プロファイル定義書の該当する業務オブジェクトの内容を参照してください。

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

DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "P001";
obj["product-name"] = "製品ABC";
```

これで、XML メッセージは生成されました。しかし、生成されたメッセージは、コンポーネントの内部形式となっているため、これを文字列の形式に変換するには、以下のようにXmlString メソッドを実行します。

```
TransactionMessage message = process.CreateMessage();
string xml = message.XmlString();
```


文字列として取り出す以外に、Write メソッドを利用して、以下のように、テキストファイルに出力する方法、および出力用ストリームまたはテキストライタに出力する方法があります。以下の例は、テキストファイルに出力する場合です。

```
message.Write(Application.StartupPath + @"¥request.xml");
```

この結果、文字列 xml および、出力ファイル request.xml には、以下のような XML メッセージが設定されます。ここで、TransactionProcess 要素が、業務トランザクションを、Product 要素が業務ドキュメントを、Item 要素が、業務オブジェクトを表しています。これらは、それぞれ、CreateProcess メソッド、CreateDocument メソッド、そして CreateDomainObject メソッドを複数回実行することで、複数要素を並列させることができます。

また、業務プロパティは、product-id、puroduct-name それぞれが、業務オブジェクト内の独自の位置にその値が記述されています。それぞれの業務プロパティが、XML 要素内のどの位置に記述されるかは、プロファイル定義の中で規定されています。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="82" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="Product" id="184" action="Notify">
            <Item id="P001">
                <Spec type="pps:name">
                    <Char value="製品ABC" />
                </Spec>
            </Item>
        </Document>
    </Transaction>
</Message>
```

◆ XMLメッセージ解釈の基本形

テキストファイルは受信メッセージなどの形で、すでに存在する XML メッセージを解釈するためには、まずいったん、コンポーネントの内部形式に取り込みます。以下の例は、input.xml という入力ファイルから、Read メソッドを利用して、XML メッセージを読み込

む場合を示しています。この例のように、テキストファイルから入力する以外に、入力用ストリームまたはテキストリーダより読み込むことができます。なお、以下の例では、ファイルから XML メッセージを読み込んでいます。

```
TransactionProcess process =
    manager.Read(Application.StartupPath + @"¥request.xml");
```

すでに XML テキストとなった XML メッセージについては、Parse メソッドによって内部形式に変換することができます。

```
TransactionMessage message =  manager.Parse(xml);
```

内部形式に変換された XML メッセージは、業務ドキュメント、業務オブジェクト、そして業務プロパティの順に階層化され、リスト形式で情報が保持されています。したがって、これらの内容に対して、リストを操作することで値を取り出すことができます。

```
foreach (Document doc in message.DocumentsReceived)
{
    foreach (DomainObject obj in doc.DomainObjects)
    {
        Property[] list = obj.GetAllProperties();
        foreach (Property prop in list)
        {
            Debug.Print("{0} : {1}", prop.Name, prop.Value);
        }
    }
}
```

ここで、業務プロパティのリストだけが、GetAllProperties メソッドによって明示的に取得するようになっています。これは、業務オブジェクトは、階層構造をもっているためです。最終的な値をもっている業務プロパティの大半は、中間的な業務オブジェクト（中間オブジェクト）に所属しており、それらがプロファイルの定義に従って、階層化されています。GetAllProperties は、それらの階層構造をいったん崩してリスト形式として提供するメソッドです。なお、階層構造を保持したままで各業務プロパティの値にアクセスする方法は、「複数型業務プロパティの扱い」の章にて説明します。

もし、業務プロパティ名があらかじめ分かっている場合には、以下のように、インデクサを利用してダイレクトに値を取得することができます。この場合、もし指定した業務プロパティに値が設定されていない場合には、null が返されます。

```
foreach (Document doc in process.DocumentsReceived)
{
    foreach (DomainObject obj in doc.DomainObjects)
    {
        if (doc.ObjectName == "Product")
        {
            Debug.Print("product-id : {0}", obj["product-id"]);
            Debug.Print("product-name : {0}", obj["product-name"]);
            Debug.Print("product-category : {0}", obj["product-category"]);
        }
    }
}
```

◆ ヘッダ情報の設定

通知メッセージおよび回答メッセージでは、業務ドキュメントには、業務オブジェクトの他に、ヘッダ情報を付加することができます。ここでは、通知メッセージに対するヘッダ情報の付加方法について説明します。

ヘッダの生成は、以下のように、業務ドキュメントに対して、CreateHeader メソッドを実行します。このメソッドの戻り値として、ヘッダオブジェクトが返されます。ひとつの業務ドキュメントに対して、設定できるヘッダは１つまでです。このヘッダオブジェクトは、業務ドキュメントの MainHeader プロパティとして保持されています。

```
Header header = doc.CreateHeader();
header.Title = "部品リスト";
```

ヘッダには、Title プロパティによって、そのタイトル文字列を設定することができます。また、ヘッダには、ヘッダ用プロパティとして、業務オブジェクトと同様に業務プロパティとその値を設定することができます。ただし、ヘッダ用プロパティは複数型であっても、値は１つしか設定できません。以下に、ヘッダ用の業務プロパティを設定する方法を示し

ます。

```
header.CreateProperty("product-id").Display = "ID";
header.CreateProperty("product-name").Display = "名称";
header.CreateProperty("standard-price").Display = "価格";
```

一般的なヘッダの利用として、テーブル形式の帳票におけるカラムの表題があります。このような用途のために、ヘッダに業務オブジェクトがもつ業務プロパティの表題を指定することができます。このためには、ヘッダ用業務プロパティがもつ Display プロパティを利用して、以下のようなプログラムを作成してください。

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

Header header = doc.CreateHeader();
header.Title = "部品リスト";

header.CreateProperty("product-id").Display = "ID";
header.CreateProperty("product-name").Display = "名称";
header.CreateProperty("standard-price").Display = "価格";
```

このプログラムの実行結果として、以下のような XML メッセージが生成されます。ここでは、ヘッダ用業務プロパティにおいて、value 属性ではなく display 属性に値が設定されている点に注目してください。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="96" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="Product" id="200" action="Notify">
            <Header class="Product" title="部品リスト">
                <Property name="product-id" display="ID" type="Target" />
                <Property name="product-name" display="名称" type="Target" />
                <Property name="standard-price" display="価格" type="Target" />
            </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>
        </Document>
    </Transaction>
</Message>
```


3. クライアント処理の基本形

◆ 業務オブジェクトの追加依頼

クライアントからサーバに対して、特定の業務オブジェクトの追加を依頼する場合には、クライアントは追加依頼ドキュメントを生成し、サーバに送信します。送信の方法は、別冊となる PPS メッセージサービス実装マニュアルに説明があるので、ここでは、送信すべき追加メッセージの生成方法を説明します。

追加依頼ドキュメントには、追加すべき 1 つ以上の業務オブジェクトをもち、アクション区分を表す業務ドキュメントの Action プロパティの値に Add が設定されています。追加依頼ドキュメントをもつ業務トランザクションには、追加依頼ドキュメントの他に、必要に応じて、削除依頼ドキュメントや修正依頼ドキュメントをあわせて含むことができます。ここでは、追加依頼ドキュメントを単独でもつ業務トランザクションを例にあげて説明します。

たとえば、作業指示を追加するためのプログラムは、以下のようになります。ここでは、作業指示として、ID が“L09F0001”と“L09F0002”である業務オブジェクトをサーバが管理する DB に追加するよう依頼しています。業務ドキュメントの Action プロパティにアクション区分が設定されている点に注目してください。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("OperationSchedule");
doc.Action = Document.ActionTypes.Add;
DomainObject obj;

obj = doc.CreateDomainObject();
obj["operation-id"] = "L09F0001";
obj["start-time-schedule"] = new DateTime(2009, 1, 10, 10, 0, 0);
obj["assign-resource-id"] = "R001";

obj = doc.CreateDomainObject();
obj["operation-id"] = "L09F0002";
obj["start-time-schedule"] = new DateTime(2009, 1, 10, 10, 30, 0);
obj["assign-resource-id"] = "R005";
```

上記のプログラムを実行した結果、以下のような XML メッセージが得られます。


```
<?xml version="1.0" encoding="utf-8"?>
<Message id="97" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="OperationSchedule" id="201" action="Add">
            <Operation id="L09F0001">
                <Start type="pps:production">
                    <Time type="pps:schedule" value="2009-01-10T10:00:00" />
                </Start>
                <Assign type="pps:equipment" resource="R001" />
            </Operation>
            <Operation id="L09F0002">
                <Start type="pps:production">
                    <Time type="pps:schedule" value="2009-01-10T10:30:00" />
                </Start>
                <Assign type="pps:equipment" resource="R005" />
            </Operation>
        </Document>
    </Transaction>
</Message>
```

サーバが実際に指定した作業指示を DB に追加したかどうかを確認したい場合があります。そのような場合には、以下のように、業務トランザクションがもつ Confirm プロパティに Always を指定してください。

```
    TransactionProcess process = manager.CreateProcess();
    process.Confirm = TransactionProcess.ConfirmTypes.Always;
    Document doc = process.CreateDocument("OperationSchedule");
    doc.Action = Document.ActionTypes.Add;
```

これにより、サーバ上で業務オブジェクトの追加処理が行われた後に、その結果が、確認ドキュメントによって返信されます。この確認ドキュメントには、正常に追加されたすべての業務オブジェクトが、ID の値のみ設定されています。ここで、ID とは、あらかじめプロファイルにて定義された主キーとなる業務プロパティのことであり、各業務オブジェクトにひとつ定義されています。例題である作業指示（OperationSchedule）オブジェクトの場合は、“operation-id”が文字通り ID として定義されています。

上記のプログラムによって生成された XML メッセージをサーバに送信し、サーバ上で 2 つの業務オブジェクトが正常に追加された場合に、以下のような XML メッセージが返信され

ます。クライアントは、これにより、依頼した追加処理がサーバ上で実施され正常に終了したことを知ることができます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="101" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001236">
        <Document name="OperationSchedule" id="207" action="Confirm" ref="205">
            <Operation id="L09F0001" />
            <Operation id="L09F0002" />
        </Document>
    </Transaction>
</Message>
```

追加ドキュメントでは、コンディション情報を効果的に設定することで、XML メッセージ全体としての合計バイト数を削減し、効率的な送受信につなげることが可能です。具体的には、コンディション情報の一部として、そこに設定された業務プロパティ情報は、その業務ドキュメントがもつすべての業務オブジェクトが共通してもつ内容であると規定しているからです。つまり、特に一業務ドキュメント内に多数の業務オブジェクトが存在する場合に、それらの共通部分をコンディション情報として書き出すことで、個々の業務オブジェクト内には記述する必要がなくなります。

たとえば、指定する作業指示を実行する装置がすべて等しい場合には、その情報を以下のようにコンディション情報として定義します。以下の例では、２つの作業指示がともに R001 の装置を利用するために、コンディション情報で指定し、各業務オブジェクトの内容としては指定していません。サーバは、特に指定がない場合には、コンディション情報の値が設定されたものとして DB に登録します。

```
    Condition cond = doc.CreateCondition();
    cond["assign-resource-id"] = "R001";

    DomainObject obj;
    obj = doc.CreateDomainObject();
    obj["operation-id"] = "L09F0001";
    obj["start-time-schedule"] = new DateTime(2009, 1, 10, 10, 0, 0);

    obj = doc.CreateDomainObject();
    obj["operation-id"] = "L09F0002";
    obj["start-time-schedule"] = new DateTime(2009, 1, 10, 10, 30, 0);
```


◆ 業務オブジェクトの修正依頼

サーバが管理する DB の特定の業務オブジェクトについて、その業務プロパティの値を変更するためには、修正依頼ドキュメントを生成しサーバに送信します。修正依頼ドキュメントは、コンディション情報として修正する業務オブジェクトを特定するための情報を指定し、セレクション情報として、修正内容を指定します。また、業務ドキュメントのアクション区分として、Action プロパティの値を Change とします。修正依頼ドキュメントも、削除依頼ドキュメントと同様に、業務ドキュメントを含めることができません。

修正可能な業務プロパティは、ID として定義されているもの以外のすべてです。ただし、対象とする業務プロパティが複数型であり、実際に複数の値が設定されている場合には、多少プログラムが複雑となりますので、後の章において説明することとし、ここでは、単数型か、あるいは複数型を単数型として扱う場合について説明します。

たとえば、以下の例は、以前に設定した L09F0001 の作業指示の開始時刻を 5 分早めて、10 時 5 分とし、進捗内容を“遅延”と修正するよう依頼しています。

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

doc.CreateCondition("L09F0001");
Selection sel = doc.CreateSelection();
sel["start-time-schedule"] = new DateTime(2009, 1, 10, 10, 5, 0);
sel["progress-status"] = "遅延";
```

上記のプログラムの実行結果として生成される XML メッセージは以下のとおりです。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="103" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" confirm="Always">
        <Document name="OperationSchedule" id="209" action="Change">
            <Condition id="L09F0001" />
            <Selection>
                <Property name="start-time-schedule">
                    <Time value="2009-01-10T10:05:00" />
                </Property>
                <Property name="progress-status" value="遅延" />
```



```
            </Selection>
        </Document>
    </Transaction>
</Message>
```

一つの修正依頼ドキュメントは、同時に複数の業務オブジェクトの内容を修正することが可能です。このためには、コンディション情報を用いて、選択対象となる業務オブジェクトを複数とするために条件によって対象を限定するか、あるいはコンディションオブジェクトを複数指定します。ただし、それらの複数の業務オブジェクトに対して、修正する内容、つまり修正対象となる業務プロパティとその修正後の値は、選択されたすべての業務プロパティについて同一となります。

◆ 業務オブジェクの削除依頼

サーバの DB から特定の業務オブジェクトを削除するためには、削除依頼ドキュメントを作成しサーバに送信します。削除依頼ドキュメントは、業務ドキュメントにコンディション情報を付加し、アクション区分を Remove とすることで作成できます。削除依頼ドキュメントには、業務オブジェクトを指定することはできません。ここで、コンディション情報は、サーバ上の DB において削除する業務オブジェクトを限定するために利用し、ID を用いて直接的に該当する業務オブジェクトを指定する方法や、フィルタ条件を指定して対象業務オブジェクトを限定するために利用します。

以下の例は、削除する業務オブジェクトを ID によって直接指定する方法です。これによって、以前に追加された２つの作業指示をサーバの DB から削除するよう依頼します。実際に削除するかどうかは、サーバ側の判断ですが、下記の例では、確認ドキュメントを依頼していますので、削除されたかどうかは返信によって確認することが可能です。

```
TransactionProcess process = manager.CreateProcess();
process.Confirm = TransactionProcess.ConfirmTypes.Always;
Document doc = process.CreateDocument("OperationSchedule");
doc.Action = Document.ActionTypes.Remove;

doc.CreateCondition("L09F0001");
doc.CreateCondition("L09F0002");
```

上記のプログラムを実行した結果、以下のような XML メッセージが生成されます。


```
<?xml version="1.0" encoding="utf-8"?>
<Message id="102" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" confirm="Always">
        <Document name="OperationSchedule" id="208" action="Remove">
            <Condition id="L09F0001" />
            <Condition id="L09F0002" />
        </Document>
    </Transaction>
</Message>
```

続いて、対象業務オブジェクトのフィルタ条件を指定して削除依頼する方法を示します。業務ドキュメントが持つコンディション情報は、ひとつのコンディションオブジェクト（CreateCondition によって生成される単位）内で AND の関係で条件を追加することができます。たとえば、以下の例では、割付資源が R001 であり、かつ予定数量が 20 以下である作業指示を削除するように依頼する業務ドキュメントを生成します。

```
    Condition cond = doc.CreateCondition();
    cond.SetConstraint("assign-resource-id", "R001",
        Pps.Documents.Constraint.ConstraintTypes.EQ);
    cond.SetConstraint("quantity-plan", 20,
        Pps.Documents.Constraint.ConstraintTypes.LE);
```

対象とする業務オブジェクトを、ワイルドカードを利用して設定することも可能です。ワイルドカードの利用は、以下のように、コンディションオブジェクトの Wildcard プロパティに対象となる業務プロパティ名を指定し、Value プロパティにワイルドカード文字列を指定します。以下の例は、作業指示の ID が L09F で始まる業務オブジェクトをすべて削除するよう依頼しています。

```
    Condition cond = doc.CreateCondition();
    cond.Wildcard = "operation-id";
    cond.Value = "L09F*";
```


4. アプリケーション個別処理への対応

◆ 独自の業務プロパティの設定

業務アプリケーション独自の業務プロパティを設定する場合には、以下のように、あらかじめ業務ドキュメントの定義に独自の業務プロパティを登録しておく必要があります。以下の例では、新たに“group-name”というプロパティ名を追加しています。この例のように、ユーザ独自の業務プロパティには必ず先頭に“user:”という接頭語をつけなければなりません。定義のためには、業務ドキュメント定義（DocumentProfile）を取得し、それに対して SetUserProperty メソッドを実行します。第一の引数がプロファイル名、第二の引数がデータ型です。ただし、この独自プロパティの定義は、後で解説する実装プロファイルにおいて、ユーザ固有プロパティの定義を行った場合には、不要となります。

```
DocumentProfile profile = manager.Profile.GetDocument("Product");
profile.SetUserProperty("user:group-name", Property.DataTypes.Char);

// 製品の内容に関する通知方法（独自プロパティ）

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

DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "K002";
obj["product-name"] = "製品ABC";
obj["user:group-name"] = "独自の情報";
```

定義した業務アプリケーション固有の業務プロパティは、他の業務プロパティと同様に業務オブジェクトの属性として値を指定することができます。ただし、この新たに定義した業務プロパティは複数型にはできません。また、必須とすることもできません。
上記のプログラムを実行すると、以下のような XML ファイルが生成されます。ここで分かるとおり、追加された業務プロパティは、常に Spec 要素の下位に定義されます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="106" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234">
        <Document name="Product" id="212" action="Notify">
            <Item id="K002">
                <Spec type="pps:name">
```



```
                    <Char value="製品ABC" />
                </Spec>
                <Spec type="user:group-name">
                    <Char value="独自の情報" />
                </Spec>
            </Item>
        </Document>
    </Transaction>
</Message>
```

◆ ヘッダにおける独自プロパティの利用

ヘッダには、ヘッダ用プロパティとして、業務オブジェクトと同様に業務プロパティとその値を設定することができます。ただし、ヘッダ用プロパティは複数型であっても、値は１つしか設定できません。以下に、ヘッダ用の業務プロパティを設定する方法を示します。

```
    Header header = doc.CreateHeader();
    header.Title = "製品ABCの部品表";

    header["display-name"] = "X社向け製品";
    header["product-category"] = "特注品";

    header["standard-price"] = 200;
    header["date-create"] = DateTime.Today;

    header.CreateProperty("user:my-property");
    header["user:my-property"] = "区分";
```

上の例では、表示名（display-name）と製品カテゴリ（product-category）の値が設定されています。これらは、あらかじめプロファイルに存在する業務プロパティです。一方、my-property という名称の業務プロパティは、プロファイルには存在しません。したがって、そのままでは、例外が発生します。しかしながら、上のケースのように、業務プロパティ名の先頭に “user:” を付加し、さらにあらかじめ CreateProperty メソッドでそのプロパティ名を登録することで、プロファイルにない業務プロパティも設定することができるようになります。このような拡張された業務プロパティのデータ型は常に文字型となります。以下は、上記の場合の XML メッセージの内容です。

```
<?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="Product" id="190" action="Notify">
            <Header class="Product" title="製品ABCの部品表">
                <Property name="display-name" type="Target" value="X社向け製品" />
                <Property name="product-category" type="Target" value="特注品" />
                <Property name="standard-price" type="Target" value="200" />
                <Property name="date-create" type="Target"
                    value="2009-02-23T00:00:00" />
                <Property name="user:my-property" type="Target" value="区分01" />
            </Header>
        </Document>
    </Transaction>
</Message>
```

◆ アプリケーション用拡張領域の利用方法

クライアントからサーバに対して、またはサーバからクライアントに対して、アプリケーション固有の設定情報などを送信したい場合があります。そのような個別の情報は、プロファイルには定義されていませんので、アプリケーション用拡張領域を利用します。アプリケーション用拡張領域は、属性名と属性値のペアで構成されており、それぞれについてアプリケーション側で自由に設定できます。ただし、データ形式はともに文字列です。

以下の例は、クライアントからサーバに対して、クライアント固有のユーザ ID とサービスの利用形態を指定しています。ここで、“UserID”および“ServiceMode”というキーワードが利用されていますが、これらはクライアントおよびサーバの二者間であらかじめ合意されたものである必要があります。

```
    Document doc = process.CreateDocument("OperationSchedule");
    doc.Action = Document.ActionTypes.Add;

    doc.SetAppData("UserID", "HOSEI-001");
    doc.SetAppData("ServiceMode", "Trial");

    DomainObject obj = doc.CreateDomainObject();
    obj["operation-id"] = "L09F0001";
```

上のプログラムによって生成された XML メッセージは以下のとおりです。App 要素の下位に指定されたパラメータ情報を設定するタグとして Parameter 要素が生成され、その name

属性および value 属性に指定した値が設定されていることが分かります。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="104" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" confirm="Always">
        <Document name="OperationSchedule" id="210" action="Add">
            <App>
                <Parameter name="UserID" value="HOSEI-001" />
                <Parameter name="ServiceMode" value="Trial" />
            </App>
            <Operation id="L09F0001" />
        </Document>
    </Transaction>
</Message>
```

5. サーバによるメッセージ処理の基本形

◆ 返信のためのトランザクションの生成

サーバは、クライアントから送信された XML メッセージの内容を解釈し、指定されたアクション区分に従ってサービスを実施し、その結果を必要に応じてクライアントへ返信しなければなりません。XML メッセージの受信および返信の方法については、別途、PPS メッセージサービス実装マニュアルにて解説してありますので、ここでは省略します。ここでは、受信した XML メッセージを解釈し、返信用の XML メッセージを作成するための処理方法について説明します。

サーバは、受信した XML メッセージをもとに、返信用の業務トランザクション処理を開始します。これによって、業務トランザクション処理オブジェクトが生成され、そこの DocumentsReceived というプロパティに受信した業務ドキュメントのリストが設定されます。したがって、業務アプリケーションは、このリストの先頭から一つずつ業務ドキュメントを処理していくことになります。

処理した結果として、返信用の業務ドキュメントは、この業務トランザクション処理オブジェクト内にすでに生成されていますが、その個々の返信用業務ドキュメントへのポインタが、受信した各業務ドキュメントから Reference というプロパティを介して取得することができます。したがって、処理結果は、この返信用の業務ドキュメント内に設定してください。

受信した各業務トランザクションは、確認要求が Always、OnError、Never のいずれかに設定されており、それに対応して、コンポーネント内で返信メッセージを生成し、実際に返信すべきかどうかを判断しています。業務トランザクションとしては確認要求が Never の場合であっても、トランザクション内に照会ドキュメントが含まれている場合には、返信を必要とします。

サーバ側のプログラムでは、以下のように、まず、受信した XML メッセージまたはテキストを引数として、CreateProcess メソッドを実行します。そして、DocumentsReceived にある受信した業務ドキュメントのリストを順に展開し、個々の業務ドキュメントのアクション区分によって、該当する処理を実行します。こうして、すべての業務ドキュメントの処理が完了し、すべての業務トランザクションの処理が完了したら、最後に、ResponseRequired プロパティの値をチェックし、もし真の場合には、返信メッセージをクライアントに送信します。


```
TransactionProcess process = manager.CreateProcess(messageReceived);
if (process.Result != TransactionProcess.TentativeResults.Failure)
{
    foreach (Document doc in process.DocumentsReceived)
    {
        switch (doc.Action)
        {
            case Document.ActionTypes.Add:
                ServiceAdd(doc);
                break;
            case Document.ActionTypes.Change:
                ServiceChange(doc);
                break;
            case Document.ActionTypes.Remove:
                ServiceRemove(doc);
                break;
            case Document.ActionTypes.Get:
                ServiceGet(doc);
                break;
            case Document.ActionTypes.Notify:
                ServiceNotify(doc);
                break;
            case Document.ActionTypes.Sync:
                ServiceSync(doc, process);
                break;
        }
    }
}
if (process.IsResponseRequired())
{
    TransactionMessage message = process.CreateMessage();
    // ここにxmlの返信処理を追加してください。
}
```

◆ サーバによる業務オブジェクトの追加

サーバのメイン処理によって、受信した XML メッセージの業務ドキュメントごとに、それぞれのアクション区分にしたがった処理が実施されます。追加依頼ドキュメントに対応した以下の ServiceAdd メソッドの中では、受信したそれぞれの業務オブジェクトについて、内部のデータベースに登録します。

以下の例では、サンプル用に作成した MyTable という DB 上のテーブルへ追加するための

NewRow というメソッドを実行しています。ここでは、便宜的にサーバが持っているデータベースの各レコードを DataRow によって扱います。業務オブジェクトである obj の内容を、新しく生成した myObj というレコードに設定していきます。ここでは、業務プロパティの名称とローカルな DB のフィールド名称とが異なるために、変換のための GetLocalName メソッドを利用しています。

ここで重要な点は、サーバ側で DB への追加処理が成功した場合に、返信用の業務ドキュメントにその業務オブジェクトの ID を設定することです。ここで返信用の業務ドキュメントは、受信した業務ドキュメントがもつ Reference プロパティに設定されています。したがって、以下の例のように、この返信用の業務ドキュメントに対して、CreateDomainObject メソッドを実行し、生成された業務オブジェクトの Id プロパティに、追加した業務オブジェクトの ID を設定します。

```
// 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);
// コンディション情報にある業務プロパティはすべてに共通して設定します。
Dictionary<string, object> defaults= new Dictionary<string, object>();
foreach (Condition cond in doc.Conditions)
    foreach (Pps.Documents.Constraint cst in cond.Constraints)
        if (!defaults.ContainsKey(cst.PropertyName))
            defaults.Add(cst.PropertyName, cst.Value);
foreach (DomainObject obj in doc.DomainObjects)
{
    DataRow myObj = MyTable.NewRow();
    // コンディション情報はあらかじめデフォルト値として設定します。
    foreach (string key in defaults.Keys)
    {
        string localName = GetLocalName(MyTable.TableName, doc.DocumentName, key);
        SetValue(MyTable, myObj, localName, defaults[key]);
    }
    // 対象オブジェクトに対応するテーブルに情報を設定します。
    Property [] propertyList = obj.GetProperties();
    foreach (Property prop in propertyList)
    {
        // 単数型の業務プロパティに値を設定します。
        string localName =
            GetLocalName(MyTable.TableName, doc.DocumentName, prop.Name);
        SetValue(MyTable, myObj, localName, prop.Value);
    }
    MyTable.Rows.Add(myObj);
    // 成功した場合に、返信用業務ドキュメントにIDを設定します。
    if (obj.Id != null) doc.Reference.CreateDomainObject().Id = obj.Id;
}
```


クライアントから受取るXMLは、以下のような形式となっています。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="86" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001308" confirm="Always">
        <Document name="OperationSchedule" id="113" action="Add">
            <Operation id="L09F0001">
                <Start type="pps:production">
                    <Time type="pps:schedule" value="2009-01-10T10:00:00" />
                </Start>
                <Assign type="pps:general" resource="R001" />
            </Operation>
            <Operation id="L09F0002">
                <Start type="pps:production">
                    <Time type="pps:schedule" value="2009-01-10T10:30:00" />
                </Start>
                <Assign type="pps:general" resource="R005" />
            </Operation>
        </Document>
    </Transaction>
</Message>
```

サーバは、正常に追加した場合に、必要に応じて以下のように、追加した業務オブジェクトのIDのリストを返信します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="91" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001312">
        <Document name="OperationSchedule" id="121" action="Confirm" ref="119">
            <Operation id="L09F0001" />
            <Operation id="L09F0002" />
        </Document>
    </Transaction>
</Message>
```

◆ サーバによる業務オブジェクトの修正

修正依頼ドキュメントに対応する ServiceChange メソッドに対しては、同様にして、修正対象となるサーバ上の業務オブジェクトのリストを得た後に、さらにセレクション情報に対応した修正内容を適用します。セレクション情報は、修正すべき業務プロパティ名とその修正値が設定されています。プログラムは以下のようになります。

```
    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;
        foreach (Selection sel in doc.Selections)
        {
            foreach (Property prop in sel.Properties)
            {
                string localName = GetLocalName(MyTable.TableName,
                    doc.DocumentName, prop.Name);
                if(SetValue(MyTable,myObj,localName,prop.Value)) modified = true;
            }
        }
        // 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;
    }
```

クライアントから受取るXMLは、以下のような形式となっています。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="92" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001314" confirm="Always">
        <Document name="OperationSchedule" id="122" action="Change">
            <Condition id="L09F0001" />
            <Selection>
                <Property name="start-time-schedule">
                    <Time value="2009-01-10T10:05:00" />
                </Property>
                <Property name="progress-status" value="遅延" />
            </Selection>
        </Document>
    </Transaction>
</Message>
```


サーバは、正常に修正が行われた場合に、必要に応じて以下のように、追加した業務オブジェクトのIDのリストを返信します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="93" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001314">
        <Document name="OperationSchedule" id="124" action="Confirm" ref="122">
            <Operation id="L09F0001" />
        </Document>
    </Transaction>
</Message>
```

◆ サーバによる業務オブジェクトの削除

上記のプログラムは、追加依頼ドキュメントに対する処理ですが、修正依頼ドキュメント、削除依頼ドキュメントの場合も、処理の基本的な構造はまったく同じです。ただし、削除および修正の場合には、業務ドキュメントに業務オブジェクトが存在しません。代わって、削除および修正対象を選択または限定するコンディション情報が対象となります。

対応するプログラムは以下のようになります。まず、コンディション情報としてコンディションオブジェクトが存在しない場合には、何も行われません（照会の場合は、コンディションオブジェクトがない場合には、サーバが持つすべての業務オブジェクトが対象となりますが、削除および修正では、対象オブジェクトはないものと解釈されます。）

業務ドキュメントの Conditions プロパティに設定されたリストには、複数のコンディションオブジェクトが定義されており、それぞれが OR の関係で対象となる業務オブジェクトを限定します。したがって、以下のプログラムのように、それぞれのコンディションオブジェクトによって選択された DB 上のレコードのリストを FilterDatabase メソッドによって得たうえで、それらの内容に重複がないようにマージします。以下のプログラムでは、このリスト（targetList）の個々の要素に対して、テーブル MyTable から削除を実行しています。

```
    DataTable MyTable = GetMyTable(doc);
    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);
    }
    foreach (DataRow myObj in targetList)
    {
        object id = GetKeyValue(MyTable, myObj, doc);
        string idString = (id == null) ? "IDが設定されてません。" : id.ToString();

        // 該当するオブジェクトをテーブルから削除します。
        MyTable.Rows.Remove(myObj);

        // 成功した場合に、返信用業務ドキュメントにIDを設定します。
        doc.Reference.CreateDomainObject().Id = idString;
    }
```

クライアントから受取るXMLは、以下のような形式となっています。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="88" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001310" confirm="Always">
        <Document name="OperationSchedule" id="116" action="Remove">
            <Condition id="L09F0001" />
            <Condition id="L09F0002" />
        </Document>
    </Transaction>
</Message>
```

サーバは、正常に削除が実行された場合に、必要に応じて以下のように、追加した業務オブジェクトのIDのリストを返信します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="89" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001310">
        <Document name="OperationSchedule" id="118" action="Confirm" ref="116">
            <Operation id="L09F0001" />
            <Operation id="L09F0002" />
        </Document>
    </Transaction>
</Message>
```

◆ エラー情報の返信方法

サーバ上で業務オブジェクトの追加、修正、削除を行おうとした際に、正常に処理できない場合もあります。そのような場合には、正常に処理できなかったことをクライアントに

知らせる必要があります。

このためには、以下のように、DB への処理が成功しなかった場合の処理として、返信用の業務ドキュメントに対してエラー情報を生成し、そのエラーオブジェクトに対して、エラーの内容を設定します。こうして生成されたエラーオブジェクトは、正常に処理できなかった業務オブジェクトの数だけ追加されます。ここで ErrorType プロパティの値が Error の場合は、確認要求が OnError となっている場合にエラー処理として返信されます。Warning の場合には、Always の場合のみ返信されます。

```
    DataTable MyTable = GetMyTable(doc);
    if (MyTable == null)
    {
        Error error = doc.Reference.CreateError();
        error.ErrorType = Error.ErrorTypes.Error;
        error.ReferenceInfo = doc.DocumentName;
        error.ErrorCode = "009";
        error.Description = "該当する業務オブジェクトは存在しません。";
        return;
    }
```

エラー情報を含んだ XML メッセージは、以下のような構成となります。エラーの ID は、コンポーネントによって自動生成されます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="109" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001235">
        <Document name="OperationSchedule" id="216" action="Confirm" ref="214">
            <Error id="000000" ref="L09F0001" code="010" status="Warning"
                description="データがすでに存在します。" />
            <Error id="000000" ref="L09F0002" code="010" status="Warning"
                description="データがすでに存在します。" />
        </Document>
    </Transaction>
</Message>
```

後で説明するトランザクション処理が定義されていない場合には、処理中にエラーが発生した場合、そのまま処理を継続するか、中止するかは、アプリケーションプログラム側（サーバ側）の機能として自由に判断することができます。また、トランザクション処理が定

義されている場合は、必要に応じてロールバックを行います。

6. 業務オブジェクト内容の照会

◆ 照会メッセージの作成方法

クライアントとしてサーバがもつ DB の内容を紹介する場合には、照会ドキュメントを作成してサーバに送信します。照会ドキュメントは、コンディション情報とセレクション情報によって、照会したい情報を指定したものであり、アクション区分として Action プロパティの値に Get を設定します。照会ドキュメントには、業務オブジェクトを含むことはできません。なお、照会メッセージは、常に返信メッセージが伴うために、確認要求のための Confirm プロパティを設定する必要はありません。

以下の例は、製品 ABC を生産する手順を照会するための最もシンプルなプログラムです。照会ドキュメントにおいて、もしコンディション情報が存在しない場合には、サーバがもつすべての業務オブジェクトが返信されることとなります。一方、セレクション情報が存在しない場合には、返信メッセージには業務オブジェクトが含まれず、ヘッダ情報のみからなる回答ドキュメントとなります。(ヘッダ情報の Counter プロパティには、サーバがもつ該当する業務オブジェクト数が設定されています。) したがって、以下の例では、SelectionType プロパティの値に All を設定することで、サーバが持っているすべての業務プロパティの値を照会したいことを示しています。

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

一方、個別に対象とする業務プロパティを指定する場合は、以下のようになります。

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

Selection sel = doc.CreateSelection();
sel.CreateProperty("process-id");       // プロセスID
sel.CreateProperty("process-item-id");  // プロセス品目ID
sel.CreateProperty("capability-value"); // 機能数量
```


◆ IDおよびワイルドカードによる業務オブジェクトの限定

対象とする業務オブジェクトの ID があらかじめ分かっている場合には、その ID を指定して、一回の照会ドキュメントで複数の業務オブジェクトを照会することができます。たとえば、作業者について、その配属場所を知りたい場合には、以下のようなプログラムとなります。

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

    doc.CreateCondition("P00H033");
    doc.CreateCondition("P00K005");
```

上記のプログラムの実行結果として、以下のような XML メッセージが得られます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="114" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001235">
        <Document name="RoutingRecord" id="221" action="Get">
            <Condition id="P00H033" />
            <Condition id="P00K005" />
            <Selection>
                <Property name="process-id" />
                <Property name="process-item-id" />
                <Property name="capability-value" />
            </Selection>
        </Document>
    </Transaction>
</Message>
```

サーバは、この要求に対して、照会結果を以下のように、クライアントに対して返信します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="107" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001328">
        <Document name="RoutingRecord" id="143" action="Show" ref="141">
            <Header class="ProductionProcess" count="2">
                <Property name="production-id" />
```

```
                <Property name="production-item-id" />
                <Property name="capability-value" />
            </Header>
            <Process id="P00H033" item="P001">
                <Capacity type="pps:capability">
                    <Qty type="pps:general" value="7" />
                </Capacity>
            </Process>
            <Process id="P00K005" item="P002" />
        </Document>
    </Transaction>
</Message>
```

対象とする業務オブジェクトがもつ業務プロパティの中で、文字型の場合には、ワイルドカードを用いて対象オブジェクトを限定することができます。以下の例は、作業者 ID の先頭文字列が K90 で始まる業務オブジェクトをすべて照会しています。

```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("PersonnelRecord");
doc.Action = Document.ActionTypes.Get;
doc.CreateSelection(Selection.SelectionTypes.All);

Condition cond = doc.CreateCondition();
cond.Wildcard = "personnel-id";
cond.Value = "K90*";
```

◆ 条件の設定方法（ＡＮＤ・ＯＲの関係）

対象業務オブジェクトの限定には、ID による設定と、ワイルドカードによる設定のほかに、任意の業務プロパティに対して、その値を以下の演算子によって制約することで、その条件にあったもののみを対象とする方法があります。設定可能な演算子は、等号（EQ）、不等号（NE）、以上（GE）、以下（LE）、より大きい（GT）、より小さい（LT）があります。業務プロパティが文字列型の場合は、等号または不等号のみが設定可能です。日時型の場合は、大小関係の小さいほうが過去、大きいほうが未来として解釈されます。

以下のプログラムでは、作業区 03 に所属する総作業時間が 2000 時間以上のベテラン作業者を照会するための XML メッセージを生成します。このように、ひとつのコンディションオブジェクトに対して複数設定された制約条件は、AND の関係であるとみなされ、それらがともに満たされる業務オブジェクトが選択されます。


```
TransactionProcess process = manager.CreateProcess();
Document doc = process.CreateDocument("PersonnelRecord");
doc.Action = Document.ActionTypes.Get;
doc.CreateSelection(Selection.SelectionTypes.All);

Condition cond = doc.CreateCondition();
cond.SetConstraint("personnel-area-id", "作業区",
    Pps.Documents.Constraint.ConstraintTypes.EQ);
cond.SetConstraint("total-time", 2000,
    Pps.Documents.Constraint.ConstraintTypes.GE);
```

一方、コンディションオブジェクトが異なる場合には、OR の関係にあると解釈されます。すでに説明した ID による対象業務オブジェクトの選択は、OR の関係であることを考慮し、複数のコンディションオブジェクトを生成しています。コンディション情報のなかで、この AND の関係と OR の関係を組み合わせることで、さまざまな複雑な条件を指定することも可能となります。これらの AND や OR の制約の設定は、ID による指定やワイルドカードによる指定と併用することが可能です。

以下の例では、作業者のランクが A であるか、またはランクが B でかつ総作業時間が 2000 時間以上の者を照会しています。

```
Condition cond;
cond = doc.CreateCondition();
cond.SetConstraint("personnel-rank", "A",
    Pps.Documents.Constraint.ConstraintTypes.EQ);

cond = doc.CreateCondition();
cond.SetConstraint("personnel-rank", "B",
    Pps.Documents.Constraint.ConstraintTypes.EQ);
cond.SetConstraint("total-time", 2000,
    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 XML メッセージが生成されます。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="116" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001235">
        <Document name="PersonnelRecord" id="223" action="Get">
            <Condition>
```



```
                <Property name="personnel-rank" type="Condition">
                    <Char value="A" condition="EQ" />
                </Property>
            </Condition>
            <Condition>
                <Property name="personnel-rank" type="Condition">
                    <Char value="B" condition="EQ" />
                </Property>
                <Property name="total-time" type="Condition">
                    <Qty value="2000" condition="GE" />
                </Property>
            </Condition>
            <Selection type="All" />
        </Document>
    </Transaction>
</Message>
```


7. サーバによる照会に対する回答

◆ 基本的な回答メッセージの構成

サーバでは、照会ドキュメントを受け取った際には、ただしにその照会内容を解釈し、回答ドキュメントを作成してクライアントに返信しなければなりません。XML メッセージの受信および返信の方法については、PPS メッセージサービス実装マニュアルにて解説していますので、そちらを参照してください。ここでは、照会ドキュメントを解釈し、それに対応する回答ドキュメントの作成方法を説明します。サーバの基本的な処理の流れは、「サーバによるメッセージ処理の基本形」の章で説明しました。ここでは、そこで、受信した XML メッセージの中に、照会ドキュメントが見つかり、その業務ドキュメントを引数とした照会のためのメソッド（たとえば ServiceGet）内の処理方法を示します。

以下のプログラムは、照会のためのメソッドの内部処理を示しています。まず、最初にコンディション情報をもとに、各コンディションオブジェクトについて DB の内容をフィルタ実行し、その結果として候補となる業務オブジェクトのリストを収集します。ここでは、同一の ID が重複しないようにチェックをおこなうことで、OR の関係を実現しています。プログラムでは、targetList に、対象となる業務オブジェクトの ID が設定されます。なお、ここでは、FilterDatabase メソッドによって、コンディションオブジェクトの内容をもとに DB から該当する業務オブジェクトが選択されています。

続いて、セレクション情報から、回答する業務プロパティ名のリスト selections を作成します。先頭のセレクションオブジェクトの SelectionType プロパティの値によって、All の場合にはすべての業務プロパティが、Typical の場合は、アプリケーションで定義した標準的な業務プロパティが、そして None または Undefined の場合には、セレクションオブジェクト内で個々に指定された業務プロパティが対象となります。すべての業務プロパティの列挙のために、プロファイルの定義内容を調べています。なお、セレクションオブジェクトがひとつも定義されていない場合には、対象業務プロパティは空となります。

そして、対象とする業務オブジェクトを個々に取り出し、返信用の業務ドキュメントに追加された新規の業務オブジェクトに対して、その内容をコピーします。この際に、コピーする業務プロパティは、さきほど生成したリスト selections にある業務プロパティ名のみが対象となります。最後に、ヘッダ情報を追加します。

```
private void ServiceGet(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);
    // 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);

    // セレクション情報から、回答する業務プロパティ名のリストを作成します。
    List<string> selections = new List<string>();
    if (doc.Selections.Count == 0)
    {
        // リストは空となります。
    }
    else if (doc.Selections[0].SelectionType == Selection.SelectionTypes.All)
    {
        // リストは複数型として定義されたもの以外のすべての業務プロパティを含みます。
        selections = GetAllProperties(doc);
     }
    else if (doc.Selections[0].SelectionType == Selection.SelectionTypes.Typical)
    {
        // リストは標準的な業務プロパティを含みます。
        selections = GetTypicalProperties(doc);
    }
    else
    {
        // 先頭のセレクションオブジェクトで指定された業務プロパティのみを含みます。
        foreach (Property prop in doc.Selections[0].Properties)
            selections.Add(prop.Name);
    }
    // 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];
        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;
        }
    }
    // 最後にヘッダ情報を生成します。
    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;

    // コンディション情報をヘッダにコピーします。
    SetHeaderConditionInfo(header, doc.Conditions);
    // セレクション情報をヘッダにコピーします。
    SetHeaderSelctionInfo(header, doc.Selections);
}
```

コンディション情報をもとに対象業務オブジェクトを選択する FilterDatabase メソッドの内容の以下のようになっています。このメソッドによって、条件に合う業務オブジェクトを DB からさがし、該当する業務オブジェクトの ID をリストに設定します。ここでは、各条件はすべて AND の関係となります。したがって、すべての条件をクリアした場合について、該当するテーブル上のレコードがリストに登録されます。

```
private List<DataRow> FilterDatabase(DataTable table, List<DataRow> baseList, Condition
cond, Document doc)
{
    string keyLocalName = GetKeyName(table, doc);
    List<DataRow> list = new List<DataRow>();
    foreach (DataRow myObj in baseList)
    {
        // それぞれの制約はAND関係なので、ひとつでも違反したら除外する。
        if (keyLocalName != null && cond.Id != null &&
            cond.Id != (string)myObj[keyLocalName]) continue;
        if (cond.Wildcard != null)
        {
            string localName = GetLocalName(table.TableName,
                doc.DocumentName, cond.Wildcard);
            object value = GetValue(table, myObj, localName);
            if (value != null && !(value is DBNull) &&
                !Regex.IsMatch((string)value, cond.Value)) continue;
        }
        bool isSatisfied = true;
        foreach (Pps.Documents.Constraint constraint in cond.Constraints)
        {
            string localName = GetLocalName(table.TableName,
                doc.DocumentName, constraint.PropertyName);
            if (localName == null) continue;
            object value = GetValue(table, myObj, localName);
            switch (constraint.DataType)
            {
                case Property.DataTypes.Qty: // 数値型の場合の比較
                    isSatisfied = CompareQty(constraint.ConstraintType,
                        value, constraint.Value);
                    break;
```



```
                case Property.DataTypes.Time: // 日付型の場合の比較
                    isSatisfied = CompareTime(constraint.ConstraintType,
                        value, constraint.Value);
                    break;
                default: // その他（文字列）の比較
                    isSatisfied = CompareChar(constraint.ConstraintType,
                        value, constraint.Value);
                    break;
            }
            if (!isSatisfied) break;
        }
        if (!isSatisfied) continue;

        // 登録する
        list.Add(myObj);
    }
    return list;
}
```

ここで、各制約の定義に対して、数値型、日時型、文字列型を分けて比較演算を実施しています。この中で、通知型の例のみを以下に示します。この例にある CastToDecimal メソッドは、任意の数値型のデータを decimal にキャストするもので、その定義は省略します。

```
public static bool CompareQty(Pps.Documents.Constraint.ConstraintTypes type, object
target, object constraint)
{
    // 数値制約を調べ、制約を満たす場合にtrueを返します。
    decimal targetValue = CastToDecimal(target);
    decimal constraintValue = CastToDecimal(constraint);
    switch (type)
    {
        case Pps.Documents.Constraint.ConstraintTypes.EQ:
            if (targetValue == constraintValue) return true;
            break;
        case Pps.Documents.Constraint.ConstraintTypes.NE:
            if (targetValue != constraintValue) return true;
            break;
        case Pps.Documents.Constraint.ConstraintTypes.GE:
            if (targetValue >= constraintValue) return true;
            break;
        case Pps.Documents.Constraint.ConstraintTypes.LE:
            if (targetValue <= constraintValue) return true;
            break;
        case Pps.Documents.Constraint.ConstraintTypes.GT:
            if (targetValue > constraintValue) return true;
            break;
```



```
        case Pps.Documents.Constraint.ConstraintTypes.LT:
            if (targetValue < constraintValue) return true;
            break;
    }
    return false;
}
```

◆ 回答ドキュメントにおけるHeaderの設定方法

照会ドキュメントに対応して返信される回答ドキュメントでは、常にヘッダ情報を設定しなければなりません。このヘッダ情報には、サーバ側に存在し、照会の対象となった業務オブジェクトの総数が常に TotalObjectCount プロパティに設定されます。データ数制約があり、実際に回答ドキュメントに含まれない業務ドキュメントがある場合には、回答ドキュメントに含まれる業務ドキュメント数と、このデータ総数は異なります。

回答ドキュメントのヘッダ情報には、これ以外に、照会ドキュメントにおいて指定されたコンディション情報、セレクション情報、そして集計結果情報が設定されます。以下に、そのプログラム例を示します。集計情報は、すでにセレクション情報内の業務プロパティに設定すみであるために、ここでは単に、セレクション情報の内容をヘッダにコピーしているだけです。ヘッダでは、同一の業務プロパティに対してコンディション情報やセレクション情報を重複して定義する場合があるため、同一の業務プロパティ名のものが複数存在する場合があります。

```
private void SetHeaderConditionInfo(Header header, List<Condition> conditions)
{
    // コンディション情報をヘッダにコピーします。
    if (conditions.Count > 0)
        foreach (Pps.Documents.Constraint constraint in conditions[0].Constraints)
            header.Properties.Add(constraint.GetProperty());
}
private void SetHeaderSelctionInfo(Header header, List<Selection> selections)
{
    // セレクション情報をヘッダにコピーします。
    if (selections.Count > 0)
        foreach (Property prop in selections[0].Properties)
            header.Properties.Add(prop);
}
```


8. トランザクション処理

◆ クライアント側の処理

トランザクション処理とは、１つあるいは複数のトランザクションオブジェクトに分けて送信または受信した同一のトランザクション ID をもつ業務ドキュメント群を、一連の処理としてまとめて行うための機能です。これは、リレーショナル・データベースのトランザクション処理に対応した機能であり、一連の処理に対する確定（コミット）や取消（ロールバック）などの処理を実装することが可能となります。

トランザクション処理を実行するためには、クライアントは DocumentManager がもつ以下の種類のメソッドによってトランザクションオブジェクトを生成する必要があります。これらのメソッドによって生成されたトランザクションオブジェクトには業務ドキュメントを追加しないでください。

種別	メソッド	説明
開始	TransactionStart	トランザクション処理を開始する
確定	TransactionCommit	トランザクション処理を確定する
取消	TransactionCancel	トランザクション処理を取り消す

プログラムは以下のようになります。まず、TransactionStart メソッドでトランザクション処理の開始を宣言します。そして、以降では、そこで得られたトランザクション ID を用いて、CreateTransaction メソッドを実行することで、同一のトランザクション処理として認識されます。開始したトランザクション処理は、TransactionCommit メソッドか、TransactionCancel メソッドまで有効となります。

```
// トランザクション処理を開始します。
process.TransactionStart();

Document doc = process.CreateDocument("Product");
DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "P001";
obj["product-name"] = "製品ABC";

// この時点で処理をコミット（確定）させます。
process.TransactionCommit();
```

トランザクション処理の開始を依頼するトランザクションオブジェクトと、確定または取消を依頼するトランザクションオブジェクトを、異なる業務メッセージに設定してサーバに送信することができます。ただし、トランザクション処理の開始は、同一トランザクション ID をもつ他のメッセージよりも先に業務アプリケーションに到着する必要があります。また、トランザクション処理の確定または取消以降は、トランザクション処理が行われません。伝送の順序を保証しない通信処理の場合には注意が必要です。

上記のプログラムを実行すると、以下のような XML が生成されます。以下では、同一の id 属性値をもつ Transction 要素が 3 つ存在し、先頭の type 属性に“Start”が、最後の type 属性に“Commit”が設定されています。トランザクション処理を実行するサーバは、この Transaction 要素にある type 属性を見て、トランザクション処理を実行します。なお、この例では、トランザクションの確定（Commit）が同一の XML テキスト（メッセージ）内に存在していますが、これらは異なるメッセージに分けて送信することも可能です。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="110" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <Transaction id="PPS001234" type="Start" />
    <Transaction id="PPS001234">
        <Document name="Product" id="217" action="Notify">
            <Item id="P001">
                <Spec type="pps:name">
                    <Char value="製品ABC" />
                </Spec>
            </Item>
        </Document>
    </Transaction>
    <Transaction id="PPS001234" type="Commit" />
</Message>
```

以下の例では、一回のメッセージ送信ではトランザクション処理の確定は行わずに、なんらかの処理（送受信を含む）を行った後にあらためてトランザクションの確定をサーバに送信しています。すでに送信処理をおこなったトランザクション処理を、引き続き継続して実施する場合には、トランザクション処理オブジェクトがもつ TransactionResume メソッドを利用するか、ドキュメントマネージャクラスがもつ ResumeProcess メソッドを利用します。後者の場合には、同一のトランザクション ID をもつ別のインスタンスを生成します。

```
    TransactionProcess process = manager.CreateProcess();
    // トランザクション処理を開始します。
    process.TransactionStart();
```



```
Document doc = process.CreateDocument("Product");
DomainObject obj = doc.CreateDomainObject();
obj["product-id"] = "P001";
obj["product-name"] = "製品ABC";

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

// ここに確認のための処理を記述します。

// トランザクションを再開します。
process.TransactionResume();

// これ以降の定義内容が次のメッセージで送信されます。
// ... 確認手順

// この時点で処理をコミット（確定）させます。
process.TransactionCommit();

// このメッセージの送信手順を記述します。
TransactionMessage message02 = process.CreateMessage();
// ... 送信手順
```

◆ サーバ側の処理

サーバ側の処理は、すでに説明したサーバ側処理の基本形に加えて、以下のように、ExitTransaction メソッド、そして CheckTransaction メソッドを挿入します。また、CreateProcess メソッドは、トランザクションの開始依頼が設定されている場合に、新規にトランザクション ID を登録するとともに、開始イベントを発生させます。また、もしそのトランザクションが失敗し、内部的に取消となっている場合には、内部的にそれ以降のトランザクション処理をスキップさせます。続いて、CheckTransaction メソッドでは、設定されている業務ドキュメントの処理が正常に実行されたかをチェックし、もしエラーが存在する場合には、トランザクション処理を中断するために true を返します。最後に、ExitTransaction では、トランザクションの確定または取消依頼が設定されている場合にイベントを生成させます。CheckTransaction においてエラーが認識された場合にも取消イベントが生成されます。

```
// トランザクションが読み込まれた場合の処理を記述します。
```



```
    TransactionProcess process = manager.CreateProcess(messageReceived);
    if (process.Result != TransactionProcess.TentativeResults.Failure)
    {
        foreach (Document doc in process.DocumentsReceived)
        {
            switch (doc.Action)
            {
                case Document.ActionTypes.Add:
                    ServiceAdd(doc);
                    break;
                case Document.ActionTypes.Change:
                    ServiceChange(doc);
                    break;
                case Document.ActionTypes.Remove:
                    ServiceRemove(doc);
                    break;
                case Document.ActionTypes.Get:
                    ServiceGet(doc);
                    break;
                case Document.ActionTypes.Notify:
                    ServiceNotify(doc);
                    break;
                case Document.ActionTypes.Sync:
                    ServiceSync(doc, process);
                    break;
            }
            if (process.CheckTransaction(doc)) break;
        }
        process.ExitTransaction();
    }
```

上記の処理によって、トランザクション処理の開始、確定、そして取消のイベントがコンポーネント内部で生成されます。これらのイベントに対応したメソッドを以下のように定義しておきます。

トランザクション処理用イベントの設定

```
    manager.TransactionStart += new
        DocumentManager.TransactionStartHandler(manager_TransactionStart);
    manager.TransactionCommit += new
        DocumentManager.TransactionCommitHandler(manager_TransactionCommit);
    manager.TransactionCancel += new
        DocumentManager.TransactionCancelHandler(manager_TransactionCancel);
```

同時に、コールバック用のメソッドを用意します。以下の例では、具体的な処理の内容は省略されています。データベースの環境等に合わせて作成してください。なお、トランザ

クション処理の ID は、トランザクション処理オブジェクト（TransactionProcess）がもつ起動者名（Initiator）プロパティと、トランザクション ID（Id）によってユニークとなります。また、これまでに処理をおこなった業務ドキュメントが、Dosuments プロパティおよび DosumentsReceived プロパティに設定されていますので、確定（Commit）および取消（Cancel）で利用することができます。

トランザクション処理の実行結果は、Result プロパティの値に、実行結果が Success または Failure として設定されています。また、各業務オブジェクトの TentativeResult プロパティに同様の値が設定されます。トランザクションの途中で Failure となった場合には、以降のトランザクション処理はおこなわないため、業務オブジェクトのリストには含まれていません。また、トランザクションオブジェクトに含まれる業務ドキュメントについては、途中で Failure となった場合には、それ以降は Undefined となり、処理が行われなかったことを意味します。

```
private void manager_TransactionStart(TransactionProcess transaction)
{
    // トランザクション処理の開始のための処理をここに追加する。
}
private void manager_TransactionCommit(TransactionProcess transaction)
{
    // トランザクションの確定処理をここに追加する。
}
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。
}
```


9. 実装プロファイルの作成と照会

◆ 実装プロファイルの生成

すべての業務アプリケーションは、その業務アプリケーションがもつ PSLX プラットフォーム対応機能を、実装プロファイルとして宣言しなければなりません。実装プロファイルに定義する内容としては、対応する業務ドキュメント名およびオプション名、そして、各業務ドキュメントに対して対応可能なアクション種別（追加、修正、削除、照会など)、業務プロパティ、そして公開イベントがあります。

アクション種別としては、それぞれに対して以下の項目を指定します。

表：実装アクションの定義情報

番号	項目名	説明
1	アクション種別	追加、修正、削除、照会、同期、通知のいずれかを指定します。
2	サーバ／クライアント種別	サーバまたはクライアントのいずれかを指定します。
3	実装レベル	機能の実装レベル（１が部分実装、２がフル実装）を指定します。

業務プロパティの定義では、以下の情報を指定してください。ただし、これらはすべて前提となるアプリケーション・プロファイルに定義されているものとしてください。

表：実装プロパティの定義情報

番号	項目	説明
1	プロパティ名	アプリケーション・プロファイルの該当する業務ドキュメントに定義された業務プロパティ名
2	表示タイトル	表示上のタイトル。ヘッダ等に利用します。
3	必須区分	必須の場合には true を設定します。省略値は false です。
4	標準区分	標準の属性項目の場合には true を設定します。標準の場合には照会時に Typical パラメータを設定した場合にその値が返されます。省略値は false です。
5	複数型区分	複数型の業務プロパティの場合は true を設定します。省略値は false です。

6	説明	業務プロパティの説明やマッピング上の注意などを記述します。

```
ImplementDocument doc;
manager.Implement.ClearDocuments();

// 業務ドキュメントの登録
doc = manager.Implement.AddDocument("Product", null);

// アクション種別の登録
doc.AddAction(Document.ActionTypes.Add, ImplementAction.RoleTypes.Server, 1);
doc.AddAction(Document.ActionTypes.Change, ImplementAction.RoleTypes.Server, 1);
doc.AddAction(Document.ActionTypes.Remove, ImplementAction.RoleTypes.Server, 1);
doc.AddAction(Document.ActionTypes.Get, ImplementAction.RoleTypes.Server, 1);
doc.AddAction(Document.ActionTypes.Notify, 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を設定してください。");
doc.AddProperty("product-name", "名称", true, true, true,
    "製品名を設定してください。");
doc.AddProperty("product-category", "製品カテゴリ", false, false, false,
    "製品カテゴリを設定してください。");
doc.AddProperty("standard-price", "価格", false, false, false,
    "標準価格を設定してください。");

// 以下、同様に設定します。
```

◆ ユーザ固有プロパティの定義

業務アプリケーションが独自に利用するプロパティが存在する場合に、それをローカルに定義することが可能です。業務アプリケーション独自で業務プロパティを設定する場合には、実装プロパティとして宣言し、外部に対して公開する必要があります。以下の例は、AddPropertyExtended メソッドによって、独自プロパティを定義しています。この場合の設定情報は、複数型が不可であるためにその識別情報がなくなり、代わりにプロパティのデータ型情報が加わります。

表：実装プロパティ（独自拡張）の定義情報

番号	項目	説明
1	プロパティ名	アプリケーション・プロファイルの該当する業務ドキュメ

		ントに定義された業務プロパティ名
2	表示タイトル	表示上のタイトル。ヘッダ等に利用します。
3	必須区分	必須の場合には true を設定します。省略値は false です。
4	標準区分	標準の属性項目の場合には true を設定します。標準の場合には照会時に Typical パラメータを設定した場合にその値が返されます。省略値は false です。
5	データ型	プロパティがとるデータの値の型として、文字列、日時、数値のいずれかを指定します。
6	説明	業務プロパティの説明やマッピング上の注意などを記述します。

```
ImplementDocument doc;
manager.Implement.ClearDocuments();

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

// アクション種別の登録
doc.AddAction(Document.ActionTypes.Get, ImplementAction.RoleTypes.Server, 1);
doc.AddAction(Document.ActionTypes.Notify, 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を設定してください。");
doc.AddProperty("product-name", "名称", true, true, true,
    "製品名を設定してください。");
// ... （中略）

// 業務プロパティ（独自拡張）の登録
doc.AddPropertyExtended("my-property", "区分", false, false,
    Property.DataTypes.Char, "拡張を設定してください。");

// 業務プロパティ（リンク拡張）の登録
doc.AddPropertyLinked("assign-name", "ProductionProcess", "product-process-id",
    "設備", false, false, "設備を設定してください。");
```

◆ 連結（リンク）拡張プロパティの場合

選択した業務ドキュメントには、該当する業務プロパティがない場合であっても、意味的に連結が可能な他の業務ドキュメントには存在する場合があります。たとえば、受注オー

ダ（SalesOrder）に製品 ID があるが、その製品 ID に対応する詳細な内容は、Product という業務ドキュメントに別途定義されているような場合です。このような場合には、連結拡張プロパティとして、通常の独自の拡張とは区別して定義することができます。

連結拡張プロパティを定義するには、まず連結先の業務ドキュメント名を指定し、その業務ドキュメントのキー（インデックス）を保持する自身の業務ドキュメント内の業務プロパティ名を指定します。そして、連結先の業務ドキュメントの業務プロパティ名を指定します。この連結先の業務プロパティ名は、“user:”を付加した上で、自身の連結拡張プロパティ名となります。連結拡張プロパティを定義する場合の必要項目をまとめると以下のようになります。なお、データ型の定義は、連結先の業務ドキュメントですでに定義されているため不要です。

表：実装プロパティ（連結拡張）の定義情報

番号	項目	説明
1	プロパティ名	連結先の業務ドキュメントに定義された業務プロパティ名
	連結先ドキュメント名	連結先の業務ドキュメント名
	連結キー名	連結のキーとなる自身の業務プロパティ名
2	表示タイトル	表示上のタイトル。ヘッダ等に利用します。
3	必須区分	必須の場合には true を設定します。省略値は false です。
4	標準区分	標準の属性項目の場合には true を設定します。標準の場合には照会時に Typical パラメータを設定した場合にその値が返されます。省略値は false です。
6	説明	業務プロパティの説明やマッピング上の注意などを記述します。

以下に、プログラムの記述例を示します。この結果、連結情報として、XML の内容として ImplementProperty 要素の link 属性に、業務プロパティ単位で連結先ドキュメント、連結キーの情報が設定されます。リンク情報は、“連結キー名”＋“:”＋“連結先ドキュメント名”として表現されます。

```
// 業務プロパティ（リンク拡張）の登録
doc.AddPropertyLinked("assign-name", "ProductionProcess", "product-process-id",
    "設備", false, false, "設備を設定してください。");
```


◆ 実装プロファイルの照会

すべての業務アプリケーションプログラムは、そのプログラムが起動している間は、常に実装プロフィルの照会に対して回答します。相手の業務アプリケーションがどのような機能を有しているのかを知りたい場合に、以下のようにして実装プロファイルを照会することができます。

```
// 実装プロファイルの照会（依頼）メッセージを生成する。（クライアント側）
TransactionMessage message =
    manager.CreateProfileMessage(Document.ActionTypes.Get);
// ここに送信処理を記述します。
```

上記のプログラムによって、以下のような XML が生成され、これを相手の業務アプリケーションに送信します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="122" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <ImplementProfile action="Get" />
</Message>
```

◆ 照会に対する回答

PSLX プラットフォームに対応するすべての業務アプリケーションは、実装プロファイルをテキストファイルとして提供できるか、あるいは以下のプログラムを実装することで実装プロファイルの照会に対して回答できなければなりません。
以下のプログラムでは、受け取ったメッセージ xml を解釈し、その内容が業務プロファイルであり、かつ照会を依頼するものである場合に、自身の業務プロファイルを生成し返信します。

```
TransactionMessage messageReceived = manager.Parse(xml);
if (messageReceived.MessageType == TransactionMessage.MessageTypes.Profile)
{
    // 実装プロファイルが読み込まれた場合の処理を記述します。
    if (messageReceived.ImplementObtainedAction == Document.ActionTypes.Get)
    {
        TransactionMessage message =
            manager.CreateProfileMessage(Document.ActionTypes.Show);
```



```
            string profileXml = message.XmlString();
            // ここにxmlの返信処理を追加してください。
        }
    }
```

上記のプログラムを実行することで、以下のような XML が生成されます。この内容は、本節の最初で定義して「実装プロファイルの生成」の内容にしたがっています。実際には、該当する業務アプリケーションが適用可能なすべての業務ドキュメントについて以下のようにアクションやプロパティが定義されたものが提示されます。なお、実装プロファイルには、イベント情報も定義可能ですが、これについては別の章で解説します。

```
<?xml version="1.0" encoding="utf-8"?>
<Message id="3" sender="PSLX001" xmlns="http://docs.oasis-open.org/pps/2009">
    <ImplementProfile id="PSLX001" action="Show">
        <ImplementDocument name="Product" option="General-01"
profile="pslx-platform-1.0">
            <ImplementAction action="Add" role="Server" />
            <ImplementAction action="Change" role="Server" />
            <ImplementAction action="Remove" role="Server" />
            <ImplementAction action="Get" role="Server" />
            <ImplementAction action="Notify" role="Server" />
            <ImplementProperty name="product-id" title="ID" use="Required"
                type="Typical" description="製品IDを設定してください。" />
            <ImplementProperty name="product-name" title="名称" multiple="Unbounded"
              use="Required" type="Typical" description="製品名を設定してください。" />
            <ImplementProperty name="my-property" title="拡張区分" extend="user"
                dataType="Char" description="拡張01を設定してください。" />
            <ImplementProperty name="assign-name" title="設備" extend="user"
                link="product-process-id:ProductionProcess" dataType="Char"
                description="設備を設定してください。" />
        </ImplementDocument>
    </ImplementProfile>
</Message>
```


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。

