



オブジェクトとリレーショナル間の障害：
インピーダンス・ミスマッチ

著：Dan Shusman

オブジェクトとリレーショナル間の障害：インピーダンス・ミスマッチ

はじめに

インピーダンス・ミスマッチとは、一般的に、オブジェクト指向 (OO) アプリケーションがそのデータを従来のリレーショナル・データベース (RDBMS) に格納する際に生じる問題を説明するのに使用される用語です。C++ プログラマは長年、この問題を経験してきましたが、今では Java を始めとする他の OO プログラマにとっても、この問題は日常的なものとなってきました。

インピーダンス・ミスマッチは、オブジェクト・モデルとリレーショナル・モデル間に、本質的に親和性がないことに起因します。インピーダンス・ミスマッチに関連する問題には、クラス階層のリレーショナル・スキーマへのバインディング (オブジェクト・クラスのリレーショナル・テーブルへのマッピング)、ID 生成、並行処理などに関する問題があります。これらの問題については、この後の項で説明します。

これらの問題による悪影響は、主に OO アプリケーションとリレーショナル・スキーマを併用することが原因で生じます。一方、これらによって生じる結果は、アプリケーションの市場投入期間、設計、開発、品質保証にかかるコスト、コードの保守容易性と拡張性の折衷的な問題、必要なレスポンスとスループット時間の確保に必要なハードウェア・サイズとトポロジなどの観点からすると明白です。

OO と RDBMS 間のインピーダンス・ミスマッチ、あるいは将来的に SQL ベース・アプリケーションとオブジェクト・データベース (OODBMS) 間のミスマッチが拡大し続けることを考えると、その結果として生じる問題の回避方法を検討しておくことは、時期を得ているだけでなく大きな価値があるといえます。

オブジェクト開発言語

C++、Microsoft Visual Basic、Borland Delphi などの長く親しまれてきた多数のテクノロジー、現在も進化を続ける Java 言語、およびオープン・ソース言語のホストによって、ビジネス・ロジックとユーザ・インタフェースの実装に使用するオブジェクト環境が実現されています。これらの OO 環境には、多少の差はあるものの、カプセル化、多態性、継承がいずれも実装されています。アプリケーションを開発および拡張する際にこれらを有効活用することの利点は、よく知られています。

データをどこに格納するか

他のプログラミング言語と同じようにオブジェクト言語でも、永続化が要求される場合は、データ・ストアへのバインディングが必要となります。通常、データ・ストアはデータベースです。最もよく使用されるデータ・モデルには、リレーショナル、オブジェクト、およびポスト・リレーショナル (トランザクション処理用多次元モデルとも呼ばれる) の 3 種類があります。

リレーショナルおよびオブジェクト・データベース・モデル：基本的な違い

RDBMS モデルと OODBMS モデルの基本的な違いと、それぞれのプログラミング手法を対比的に理解しておくことは重要です。

簡単に言うと、リレーショナル・モデルでは、情報を格納する列と情報を編成する行によってテーブルが構成されます。データ構造が複雑な場合は、必要なテーブル数が多くなります。テーブル間のリレーションシップ (一対一、一対多、および多対多) は、外部キーに基づいています。

ビジネス・ロジック操作は、テーブルの外部にあるソースから適用されます。例えば、埋め込み SQL や、

事前にコード記述された静的なストアド・プロシージャやトリガが使用されます。リレーショナル・モデルを使用して効率的かつ効果的なアプリケーションを構築するには、開発者は、テーブル、テーブル間のリレーションシップ、および外部ロジック・コンポーネントのそれぞれについて包括的な知識が必要になります。対照的に、オブジェクト・モデルのクラスは自己完結型のエンティティです。リレーショナル・テーブルと共通するのは、クラスのプロパティに情報が格納される点です。一方、大きな違いは、関連するデータ（つまり、埋め込みクラスとコレクション）が、“外部キー”型の構造を必要とする別テーブルとしてではなく、“コンテナ”クラス内に格納可能な点です。

もう 1 つの大きな違いは、オブジェクト・モデルでは、ビジネス・ロジックが外部から適用されないことです。その代わりに、クラスのプロパティに対して操作を実行するコードが記述されたメソッドが、クラス内に実装されます。メソッドは、その呼び出し時に使用するインタフェースを公開します。それによって、アプリケーション開発者は、スキーマの持つ複雑さから解放されます。

請求書アプリケーション以外の例

これらの 2 モデルを使用したデータベース設計とコーディング例を紹介することにより、前項で説明した違いを具体的に示します。

自動車の登録アプリケーションを想定します。各自動車には、型式、モデル、内装タイプ、登録年度、車両登録番号 (VIN) などの固有のデータがあり、1 人以上の所有者、1 人以上の運転者、修理歴などが記録されます。また、登録、廃車などの処理は必須です。

オブジェクト・モデルで自動車のデータを表すには、Car、Person、Owner、Driver、RepairHistory の各クラスを作成します。

Owner および Driver クラスは、Person クラスからプロパティとメソッドを継承します。必要に応じて、これらの両クラスに固有のプロパティとメソッドを追加して、拡張します。

Owner、Driver、および RepairHistory は、Car クラス内でコレクションになると想定されます。

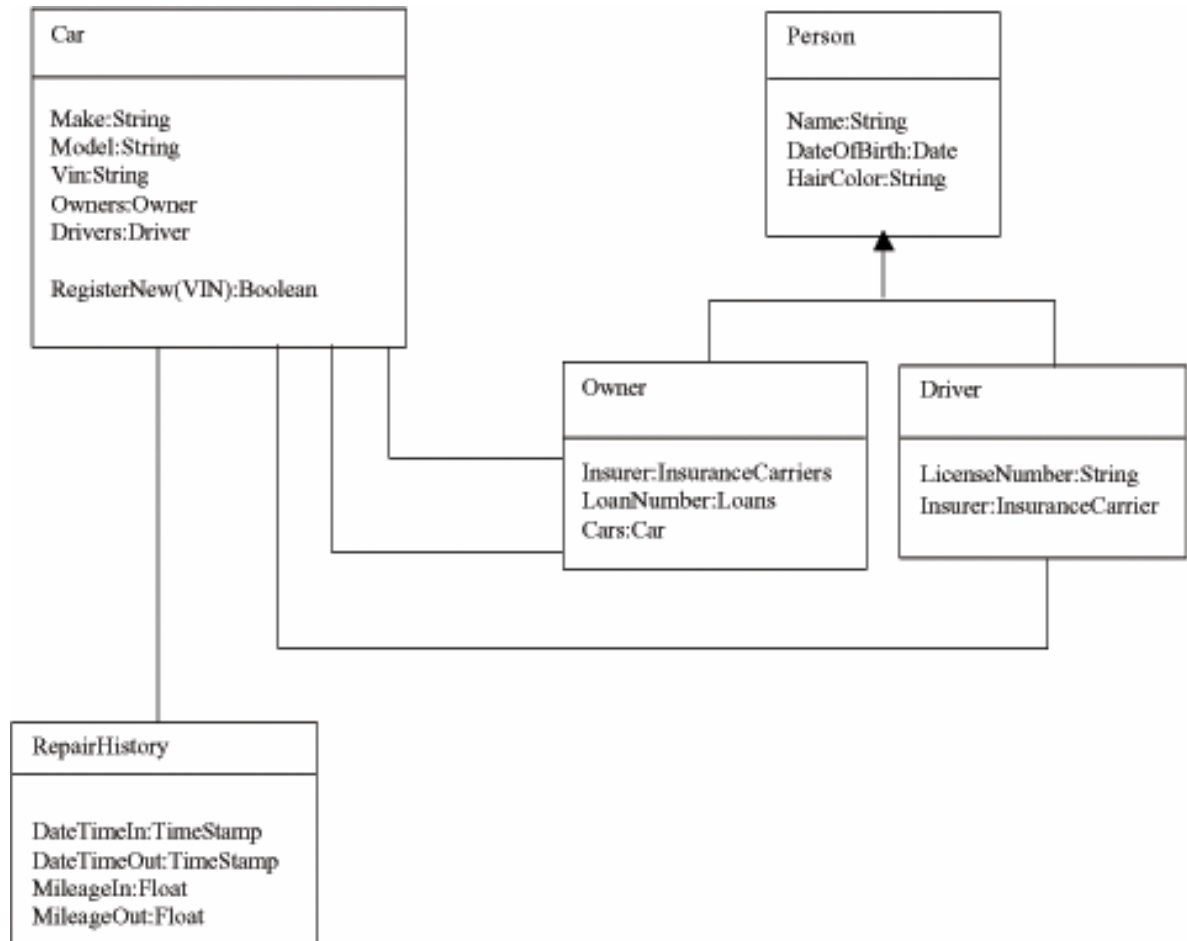
Owner および Driver は参照コレクションで、RepairHistory は埋め込みインスタンスとなります。

より現実的にするには、1 人の Owner が複数の Car を所有し、1 台の Car が複数の Owner によって所有されるリレーションシップを指定可能にします。こうした多対多のリレーションシップを実装するには、Owner クラスに Car クラスへの参照コレクションが必要です（前の説明に従って、Car には Owner のコレクションが既存するとします）。また、Car と Driver 間にも、多対多のリレーションシップを実装できます。

最後に、Car クラスのメソッドとして、パラメータ VIN を使用して呼び出す RegisterNew() メソッドを指定します。新車の登録に必要なすべての処理は、このインタフェースの背後で実行されます。

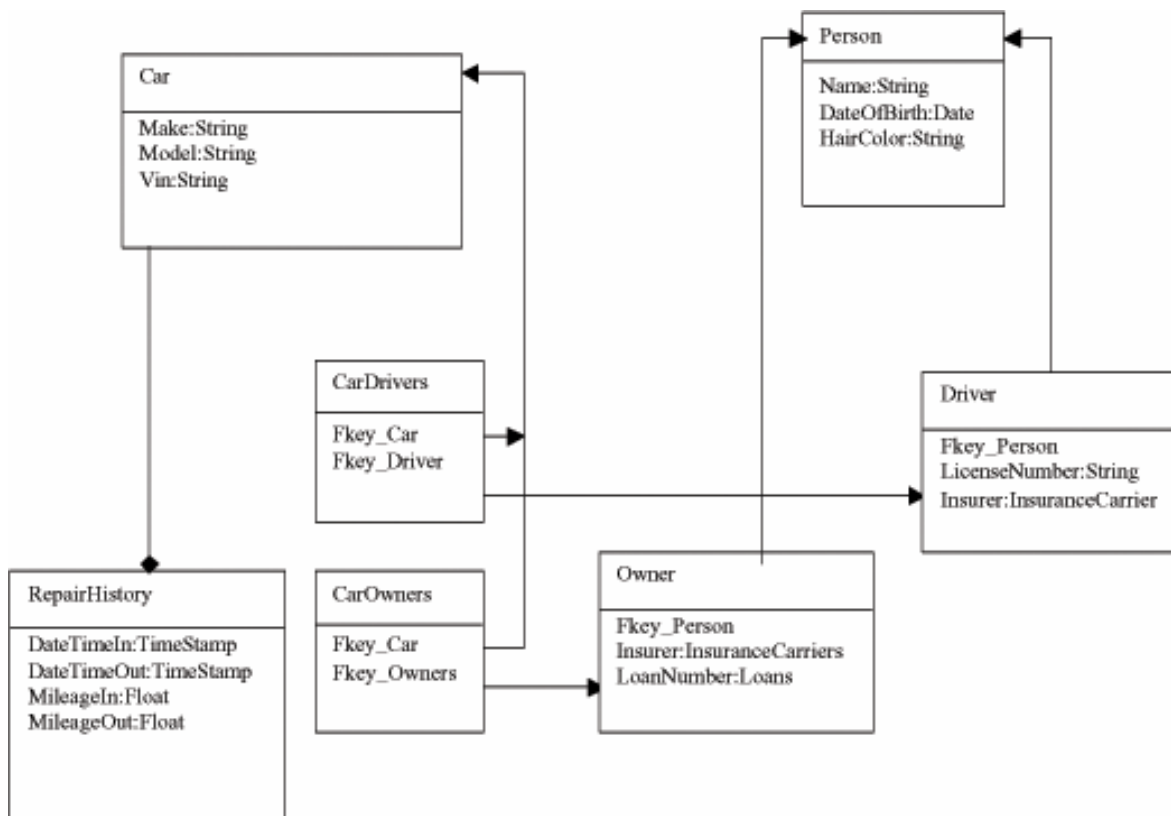
これらの仕様を図式化すると、図 1 のようになります。

図 1 リレーショナル・モデルでの同様の例



リレーショナル・モデルでこの例を表すと、図 2 に示すような複数のテーブルの作成が必要になります。

図 2



Owner、Driver、および Car 間に多対多のリレーションシップを構築するには、追加のテーブル (CarDriver と CarOwner) が必要になることに注意してください。

コードの記述

2 つの作業を実行します。

例 1a および 1b は、指定した VIN の Car を所有するすべての Owner を検索表示するコードを、オブジェクト・プログラミングと SQL プログラミングのそれぞれで実装する方法を示します。

例 1a - Car の VIN とその Owner の一覧表示 : OO

```
objCar=Open Car(vin);
owner_count=objCar.Owners.Count()
For (i=0; i<owner_count; i++) {

    owner_name=objCar.Owners.Get(i).Name;
}
}
```

例 1b - Car の VIN とその Owner の一覧表示 : SQL

```
Select Car.VIN,Person.Name
From Car, CarOwners, Owner, Person
Where CarOwners.Fkey_Car=Car.CarID

    And CarOwners.Fkey_Owner=Owner.OwnerID
    And Owner.Fkey_Person=Person.PersonID
    And Car.VIN=:vin
```

例 2a および 2b は、オブジェクト・プログラミングと SQL プログラミングでの新車の登録方法を示す擬似コードです。

例 2a - 新車の登録 : OO

```
// somehow we acquired the car's VIN, make, model and list of owners // use a class
method rather than an instance method to validate rc = Car.IsValidVin(vin) // if rc
indicates an error, logic to reject goes here // now assume an owners[i] array and this
is where the logic for validating them goes // instantiate a new car objCar = New Car;
// assign the properties objCar.VIN = vin; objCar.Model=model; objCar.Make=make; //
now register the car objCar.RegisterNew(vin)
// assign the Owners
For (i = 1; i < count; i++) {

    obj.Car.Owners.SetAt[i] = owner[i]
}
// make it persistent
objCar.Save()
```

例 2a - 新車の登録 : SQL

```
// somehow we acquired the car's VIN, make, model and list of owners
//validation of information is external to the table
If (vin == "") ! (vin=0){

    Exception("VIN required")
}
//Other fields would be validated here
//
&sql(Insert Into Car(Make, Model, VIN) Values(:make, :model, :vin)
// recover the new assigned Car ID
&sql(Select CarId Into :car_id from Car Where Car.VIN=:vin)
// and assign the owners to the Car
For (i = 1; i < count; i++) {

    &sql(Insert Into CarOwners(CarKey,OwnerKey) Values(:car_id,:owner[i])
}
```

データベース操作時のインピーダンス・ミスマッチ

オブジェクト・プログラマが純粋な OO 環境で作業を行う場合は、例 1a および例 2a に示したコードを記述する開発手法を採用できます。SQL プログラマが RDBMS を使用する場合は、例 1b および例 2b に概略を示したコードを記述する開発手法を採用できます。

オブジェクト・プログラマが RDBMS を操作する場合は、データベースを処理する例 2b の箇所と共に、何らかの方法で例 1a と例 1b を結合する必要があります。言い換えると、各クラスの永続化メソッド (Open()、Save()、Delete() など) を SQL内 (または、ストアド・プロシージャなどの他の RDBMS テクノロジ内) にコード化して、必要に応じてデータを読み書きする必要があります。例えば、Car の例では以下のようにコード化します。

- Open() メソッドでは、Car テーブルの各列をリカバリし、そのデータを Car クラスの属性にバインドする SQL SELECT クエリを実行します。
- Owner 全体に対する繰り返し処理に使用する Count() メソッドは、以下のようなコードにする必要があります。
Select Count(*) From CarOwners Where CarOwners.Fkey_Car=Car.CarID AND Car.VIN=:vin)
- 各 Owner インスタンスを調べて Name をリカバリする Get(index) メソッドには、以下のようなクエリの '結果セット' を格納するコードが必要になります。

Select Person.Name From Car, CarOwners, Owner, Person

Where CarOwners.Fkey_Car=Car.CarID

And CarOwners.Fkey_Owner=Owner.OwnerID

And Owner.Fkey_Person=Person.PersonID

さらなる問題：より困難なインピーダンス・ミスマッチの複雑さ

OO アプリケーションを RDBMS と連携させる際には、以下のような問題についても考慮する必要があります。

クラスと 1 つ以上の既存テーブルの関連付け

最初のタスクは、各クラスの属性を特定のテーブルの列に関連付けることです。クラスとテーブルの関係が一对一の場合は容易に対応できますが、最適化されたオブジェクト・モデルでは、1 つのクラスが複数のテーブルにまたがり、各テーブルのサブセット列が必要となる場合があります。または、複数のクラスを 1 つのテーブルにマップし、テーブルの意味体系を個々のニーズに合わせて再解釈する必要がある場合もあります (この場合は、各行に対応付けられているクラス名を保持するために、新しい列がテーブルに必要になります)。オブジェクト・モデルのプロジェクト・データに合わせてリレーショナル・スキーマを再解釈および操作する方法では、オブジェクトとリレーショナル・スキーマが大きく分離することになります。

ID の生成 - オブジェクト ID と RDBMS への挿入

オブジェクト ID の生成と、オブジェクトの Save() メソッドによって RDBMS に一意キーを生成するプロセスは、非常に重要です。クラスとテーブルの関係が一对一の場合は、RDBMS エンジンを使用して次に有効な ID 値を生成し、その値をオブジェクト ID にする方法が一般的に最適です。一方、1 つのクラスが複数のテーブルにまたがる場合は、その Save() メソッドに複数のリレーショナル ID が必要となり、さらにはそれらをグループとして 1 つのオブジェクト ID に関連付ける必要があります。この関連付けは、オブジェクトのオープン時に、複数のテーブルから行をリカバリしてインスタンスを配置できるように永続的である必要があります。そのため、オブジェクト ID と複数のリレーショナル ID の関連付けに使用するテーブルを別に構築し、保持する必要があります。

検証およびその他のチェック

RDBMS 用の元の SQL ベース・アプリケーションでデータの外部検証が実行されている場合は、そのコードを識別してオブジェクト・アプリケーションに移行する必要があります。

永続化メソッドおよび操作順序の手作業の構築

開発者は、アプリケーション・レベルでのデータの整合性に、常に注意する必要があります。Save() メソッドを実装する場合は、

Insert、Update、および Delete 操作に特定の順序を適用し、アプリケーション・レベルでのデータの整合性を保証する必要があります。

並行処理の問題

アプリケーション・レベルでのデータの整合性に対するもう 1 つの影響として、並行処理の問題があります。RDBMS によって実行されるロック・レベル (特定のエンティティによる排他的な読み取り/書き込み) が、クラスの永続化メソッドにも確実に反映されるようにする必要があります。

スキーマ拡張時の問題

オブジェクトとリレーショナルがマッピングされているため、スキーマを拡張するときは、オブジェクト・スキーマとリレーショナル・スキーマの調和の維持に配慮する必要があります。オブジェクト・スキーマと元のリレーショナル・スキーマが分離していると (前の説明のように)、この問題がさらに複雑になります。

同時処理の実現

これまで説明したすべての問題は、元のアプリケーションと新しく構築したオブジェクト・アプリケーションが元の RDBMS に対して同時に処理を実行できるかどうかに関係しています。

市場で提案されている解決策

インピーダンス・ミスマッチの問題を解決する方法としては、以下の 4 つのアプローチが一般的です。

1. リレーショナル・データベースのオブジェクト機能を使用する。
2. オブジェクト/リレーショナル・マッピング・ツールを使用する。
3. オブジェクト・データベースを導入する。
4. ポスト・リレーショナル・データベースを導入する。

1. リレーショナル・データベースのオブジェクト機能の使用

RDBMS ベンダは、自社のエンジンにオブジェクト機能を追加するよう多大な努力をしてきましたが、根本的な問題解決にいたっていません。また、それらはいずれも、本当のオブジェクト実装ではありません。多くの点で、RDBMS ベンダはインピーダンス・ミスマッチの本質そのものに苦しんでいます。継承、多態性、およびカプセル化は、RDBMS ベンダが提供するコア・テクノロジーに容易に融合できる概念ではありません。結局、開発者は、既存する RDBMS の最上位にある擬似的なオブジェクト・レイヤを操作するのが精一杯である状態に止まっています。

2. オブジェクト/リレーショナル・マッピング・ツールの使用

多くのデータベース・ベンダやサード・パーティによって、アプリケーション・クラスを RDBMS の既存テーブルに関連付ける“オブジェクト/リレーショナル・マッピング”ソフトウェア製品が提供されています。これらのツールには、データベース操作 (挿入、更新、削除、選択、および関連する並行処理要件) のパフォーマンスと整合性を向上させるランタイム・キャッシング・コンポーネント、一意な ID を生成するメソッド、サポート言語で使用するバインディング (JavaBeans、COM オブジェクト) を生成する機能などが含まれるものもあります。こうした支援ツールは、開発者がリレーショナル・スキーマをオブジェクトベース・アプリケーションに公開するコードを開発する際に非常に有効です。こうしたパッケージには、評判のよいものも多数あります。ただし、オブジェクト/リレーショナル・マッピング・ツールには、問題がより複雑になると、それを開発者の責任に転嫁する傾向があります。この問題は特に、1 つのクラスを複数のテーブルにマッピングするとき、複数の

クラスを 1 つのテーブルにマッピングするとき、およびこうしたマッピングで必要となる ID 生成の困難さに対処するときなどに該当します。

3. オブジェクト・データベースの導入

インピーダンス・ミスマッチを解決する第 3 のアプローチとして、純粋なオブジェクト・データベースを導入する方法があります。オブジェクト・データベースを使用する利点は、OO 概念との親和性にあります。

その導入に対しては、以下の点の検討が必要です。

- この種のデータベースの市場シェアが小さいこと。
- RDBMS 用のトランザクションが OODBMS でも利用可能であること。
- SQL ベース・アプリケーションとオブジェクト・データベース間にインピーダンス・ミスマッチが発生すること。

その他の検討事項としては、エンド・ユーザがデスクトップで使用する高度な意志決定支援 (DS) ツールの例があげられます。一般的な意志決定支援ツールでは、ODBC を使用してターゲット・データベースに接続するインタフェースで SQL クエリが構成されるため、オブジェクトベース・データのリレーショナルなビューが必要になります。一部の OODBMS ベンダでは、中間層のコンピュータにリレーショナル/オブジェクト・マッピング・ソリューションを配置することにより、このミスマッチの解決が図られています。

4. Cache、ポスト・リレーショナル・データベースの導入

Cache などのポスト・リレーショナル (トランザクション処理用多次元とも呼ばれる) データベースは、連想配列テクノロジーを実装することで、マッピング・ツールの介在や中間層でのキャッシングがなくても、オブジェクト・モデルおよびリレーショナル・モデルに対して同時に投影できます。

Save() などの永続化メソッドは、連想配列を操作するネイティブ・コマンドを通して、オブジェクト開発者に直接的に投影され、データベース自体に実装されます。同時に、これらのエンジンは、ODBC および JDBC を通して公開された同一の (連想配列) データのリレーショナルなビューを投影できます。Insert などの SQL DDL、DML、および DCL コマンドの実装は、結果的に、オブジェクトの永続化メソッドを実装する同一のネイティブ・データベース・コマンドとなります。多次元エンジンは各投影による同時的で直接的なアクセスを実装できるため、ID 生成、並行処理、検証などに関する問題が、完全に除去されないにしても最小化されます。

各投影による同時的で直接的なアクセス (並行処理の管理を含む) は、SQL ベース・アプリケーション (ODBC を介した VB、C++、または Delphi) がオブジェクトベース・アプリケーション (VB、Java、または C++) としても同時に操作可能であることを意味します。また、リレーショナルへの投影によって、エンド・ユーザは意志決定に、使い慣れた SQL ベースのフロントエンド・ツールを使用可能になります。

まとめ

前の項では、OO 言語と RDBMS 間のインピーダンス・ミスマッチを解決する主要なアプローチとして、RDBMS が提供する部分的なオブジェクト・サポートを使用する方法、オブジェクト/リレーショナル・マッピング・ツールを使用する方法、オブジェクト・データベースを導入する方法、およびポスト・リレーショナル (トランザクション処理用多次元) データベースを導入する方法について検証しました。

これらのテクノロジーを評価する際に検討すべき項目には、以下のものがあります。

- 現実的なニーズに対して、テクノロジーの導入が容易かどうか (学習曲線など)、または統合が可能かどうか。
- アプリケーションやデータベースの将来的なニーズに対して、テクノロジーが拡張可能かどうか。
- XML や SOAP などの新しいテクノロジーとの統合に対して、テクノロジーが柔軟かどうか。
- 適切なロジック層へのコードの実装が可能かどうか。言い換えると、データベース・サーバで効率的に実行されるコードを、テクノロジーの厳密な実装のために、アプリケーション・サーバなどの他の中間層に移行する必要が生じるかどうか。
- Web または企業全体、あるいはその両方に対してテクノロジーを配備できるほど、そのテクノロジーにスケールビリティとパフォーマンスがあるかどうか。または、その配備に際して、インフラストラクチャやオペレーションに過度の影響を与えることはないか。
- 開発段階の特定の問題解決には役立つが、下流の実行段階で別の問題が生じることがないか。

明白なことが 1 つあります。それは、新規アプリケーションの開発とレガシー・アプリケーションの拡張には OO が最も適したアプローチであることを、市場が確信していることです。RDBMS に多大な投資を行ってきた組織は、オブジェクトとリレーショナル・パラダイム間のインピーダンス・ミスマッチを軽減するために、短期的、中期的、または長期的な展望のいずれかを決定する必要があります。

* この文章は、米 InterSystems 社より出されている "Oscillating Between Objects and Relational: The Impedance Mismatch" の翻訳文です。詳しくは英文の本文をご参照ください。

インターシステムズジャパン株式会社
<http://www.intersystems.co.jp/>