SDOの紹介

Java 環境での次世代データ・プログラミング

Bertrand Portier
Software Engineer
IBM

Frank Budinsky
Eclipse EMF Project Lead
IBM

2004年 9月 28日

J2EE プログラミング・モデルと API が原因で、開発者が技術固有の構成、プログラミング、そしてデバッグにあまりにも多くの時間を費やすことになっていると考えているのなら、ぜひ、この記事を読んでください。Java 開発者の多くが、異種のデータに一様にアクセスできるという方法には懐疑的で、この問題の解決策を提案する各種のプログラミング・フレームワークに失望した経験があります。この記事では、Java 開発者である Bertrand Portier と Frank Budinsky が、Service Data Objects (SDO) を使用した次世代のデータ・プログラミングを紹介します。

簡単に言うと、SDO とはアーキテクチャーと API が組み込まれたデータ・アプリケーション開発用フレームワークのことです。SDO は、以下のことを行います。

- J2EE データ・プログラミング・モデルの簡素化
- サービス指向アーキテクチャー (SOA) でのデータの抽象化
- データ・アプリケーション開発の統一化
- XML のサポートおよび統合
- J2EE パターンとベスト・プラクティスの取り込み

In this introduction to the SDO framework, we will try to explain the motivation behind the SDO effort

SDO フレームワークを紹介するこの記事では、SDO への取り組みの背後にある動機、そして SDO とその他の仕様との違いについて説明したいと思います。次に、SDO を構成するコンポーネントを取り上げ、最後にサンプル SDO アプリケーションを用いて実際の SDO の動作を説明します。

なぜ SDO なのか

Service Data Objects (SDO) について大抵の開発者が抱く最初の疑問は、「なぜ SDO なのか」ということでしょう。J2EE は現状のままでもかなり大規模で学習するのに苦労するほど複雑な上、他のフレームワークがすでに Java 環境で XML をサポートしているからです。でも、そんな疑問に対

する答えは、ほとんどの開発者にとって朗報となります。つまり、SDO は J2EE データ・プログラミング・モデルを簡素化するために登場した手段であり、J2EE 開発者がより多くの時間をアプリケーションのビジネス・ロジックに費やすことができるようにします。

SDO フレームワークは、データ・アプリケーション開発用の統合フレームワークです。SDO では、技術固有の API を熟知していなくても、データにアクセスして利用することができます。知っておく必要がある API はたった 1 つ、SDO API だけです。SDO API を知っていれば、リレーショナル・データベース、エンティティー EJB コンポーネント、XML ページ、Web サービス、Java Connector Architecture、JavaServer Pages ページなど、さまざまなデータ・ソースのデータを操作できます。

ここで、「フレームワーク」という言葉を使っているのは、これが Eclipse フレームワークに似ているためです。Eclipse はその頑丈で拡張可能なベースのおかげで、さまざまなツールを統合できるように設計されています。複数のアプリケーションが貢献可能なフレームワークを提供するという点では SDO も同様で、これらのアプリケーションはすべて SDO モデルとの一貫性を持つことになります。

一部のデータ統合モデルとは異なり、SDO はデータの抽象化だけにはとどまりません。SDO フレームワークは多数の J2EE パターンとベスト・プラクティスも取り込むため、実証済みのアーキテクチャーと設計をアプリケーションに簡単に取り込むことができます。例えば、今日の Web アプリケーションの大多数は四六時中バックエンド・システムに接続しているわけでなないため (また、それは不可能です)、SDO は切断されたプログラミング・モデルをサポートします。同様に、今日のアプリケーションは関連する多数の関連層を含んで極めて複雑になる傾向があります。そんな多くの層で構成されたアプリケーションでは、データはどのように保管/送信されるのでしょうか。また、GUI フレームワークでは、どのようにエンド・ユーザーに表示されるのでしょうか。SDO プログラミング・モデルでは使用パターンを規定するため、それぞれの層は明確に区別されます。

分散アプリケーションでは、XML があらゆるところで使用されるようになっています。例えば、アプリケーションのデータ・フォーマットでビジネス・ルールを定義するには、XSD (XML Schema) が使用されます。また、Web サービスでは XML ベースの SOAP をメッセージング技術として使用するなど、XML 自体も相互作用を容易にするために使用されています。XML は非常に重要な SDO ドライバーであるため、フレームワークでは XML をサポートし、統合しています。

技術の比較

前述したように、分散アプリケーションでのデータ統合の問題を解決するために提案されている技術は SDO だけではありません。このセクションでは、SDO を JDO、JAXB、EMF などの同様のプログラミング・フレームワークと比較してみます。

SDO とWDO

WDO (Web Data Object) は、IBM WebSphere Application Server 5.1 および IBM WebSphere Studio Application Developer 5.1.2 に搭載された初期の SDO リリースの名前です。WebSphere Studio 5.1.2 を使用した経験があるなら、すでに SDO を多少は使い慣れているはずです。ただし、ライブラリー名などでは WDO として示されるため、その表示のほうに馴染みがあるかもしれません。WDO は忘れてください。今では SDO と呼ばれています。

SDO と JDO

JDO は Java Data Object の略語です。JDO は、2003年5月の JCP (Java Community Process) 1.0 およびメンテナンス・リリース 1.0.1 で標準化されました。現在バージョン 2.0 に向けて、JCP エキスパート・グループが結成されています。JDO は Java 環境でのデータ・プログラミングを対象として、各種データ・ソース (データベース、ファイル・システム、またはトランザクション処理システムなど) に保管されたデータにアクセスするための共通 API を提供します。JDO は Java オブジェクト (グラフ) 間のリレーションシップを保持すると同時に、データへの同時アクセスを許可します。

JDO の目的は、Java データ・プログラミングを単純化および統一化して、開発者が基礎技術ではなくビジネス・ロジックに専念できるようにするという点では、SDO の目的と同様です。主な違いは、JDO はパーシスタンス問題 (J2EE データ層またはエンタープライズ情報システム (EIS) 層) のみを対象とする一方、SDO はそれよりも汎用的で、すべての J2EE 層間 (プレゼンテーション層とビジネス層間など) を行き来できるデータを表しているという点です。

興味深いことに、SDO は JDO と併用することができます。その場合、JDO は SDO がアクセス可能なデータ・ソースとなり、DTO (Data Transfer Object) 設計パターンが適用されます。同様に、一律のデータ・アクセスを提供する目的で SDO をエンティティー EJB コンポーネントおよび JCA (Java Connector Architecture) と併せて使用することもできます。

SDO と EMF

EMF は Eclipse Modeling Framework の略語です。EMF は、Java インターフェース、XML スキーマ、または UML クラス・ダイアグラムによって定義されたデータ・モデルに基づいて統一メタモデル (Ecore) を生成します。このメタモデルとフレームワークと併用することにより、高品質のモデル実装を作成できます。EMF はパーシスタンス、非常に効率的な自己反映汎用オブジェクト操作 API、そして変更通知フレームワークを提供します。EMFには、EMF モデル・エディターをビルドするための一般的な再利用可能なクラスも組み込まれています。

EMF と SDO はどちらもデータ表現に対処します。この記事でこの後で使用する IBM の SDO 参照実装は、実は SDO の EMF 実装です。EMF コード生成は、SDO 自体の UML モデル定義に基づいた SDO 実装の一部を作成するためにも使用されています。SDO の実装は本質的には EMF の上にある薄い層 (ファサード) で、EMF プロジェクトの一部としてパッケージ化されて出荷されます。EMF についての詳細は、「参考文献」を参照してください。

SDO と JAXB

JAXB は Java API for XML Data Binding の略語です。JAXB 1.0 は 2003年1月、JCP によりリリースされ、バージョン 2.0 の初期ドラフトが JCP エキスパート・グループによって作成されています。JAXB は XML データ・バンディングに関するもので、XML データをメモリー内で Java オブジェクトとして表します。Java 言語の XML バインディング・フレームワークである JAXB によって、XML 文書を構文解析したり作成する手間を省くことができます (実際は、XML を処理する必要を完全に取り除いてくれます)。JAXB は (Java から XML への) マーシャル/シリアライズ、および (XML から Java への) 案マーシャル/デシリアライズを自動で行います。

SDO は独自の Java バインディング・フレームワークを定義していますが、さらに一歩進んでいます。JAXB は Java と XML のバインディングのみを対象としていますが、SDO にバインドされる

データは XML だけではありません。前述したように、各種データへの統一アクセスを提供する SDO にとっては、XML はデータ・タイプのうちの 1 つでしかないのです。また、SDO が静的 API と動的 API* の両方を提供する一方、JAXB では静的バインディングしか提供しません。

* この記事のサンプル・アプリケーションでは動的 SDO のみを使用していますが、EMF コード生成プログラムはデータ・オブジェクトの静的コード生成も完全にサポートしています。

SDO と ADO .NET

今まで ActiveX Data Object の略語は ADO でしたが、.NET コンテキストには当てはまりません。ADO .NET は .NET フレームワークの異なる層間での統一データ・アクセスを提供します。

ADO .NET と SDO は同様の目的を共有しています。それは、複数の層に分散された XML およびアプリケーションをサポートすることです。技術的な違いの他に大きな相違点となっているのは、ADO .NET は Microsoft .NET プラットフォームを対象とした専有技術である一方、SDO は Java (J2EE) プラットフォームを対象とした技術で、JCP によって標準化されているということです。

SDO コンポーネント

SDO アーキテクチャーの概要についてのこのセクションでは、フレームワークを構成するそれぞれのコンポーネントを取り上げ、これらのコンポーネントがどのように連動するかを説明します。最初に紹介する 3 つのコンポーネントは、SDO の「概念的な」機能です。これらのコンポーネントに対応するインターフェースは API にはありません。

SDO クライアント

SDO クライアントは、SDO フレームワークを使用してデータを操作します。技術固有の API やフレームワークを使用する代わりに、クライアントは SDO プログラミング・モデルと API を使用します。SDO クライアントは SDO データ・グラフ (図 1 を参照) 上で機能し、操作中のデータが持続またはシリアライズされる方法を認識しません。

データ・メディエーター・サービス

データ・メディエーター・サービス (DMS) は、データ・ソースからデータ・グラフを作成し、データ・グラフに加えられた変更に基づいてデータ・ソースを更新します。データ・メディエーター・フレームワークは SDO 1.0 仕様の適用範囲ではありません。つまり、SDO 1.0 は特定の DMS について規定していないということです。DMS の例には、JDBC DMS、エンティティー EJB DMS、XML DMS などがあります。

データ・ソース

データ・ソースは、バックエンド・データ・ソース (パーシスタンス・データベースなど) だけに限りません。データ・ソースは、その固有の形式でデータを収容します。データ・ソースにアクセスするのは DMS だけで、SDO アプリケーションはアクセスしません。SDO アプリケーションが操作できるのは、データ・グラフ内のデータ・オブジェクトのみです。

以下のそれぞれのコンポーネントは、SDO プログラミング・モデルの Java インターフェースに対応します。SDO 参照実装 (「参考文献」を参照) は、これらのインターフェースを EMF ベースで実装しています。

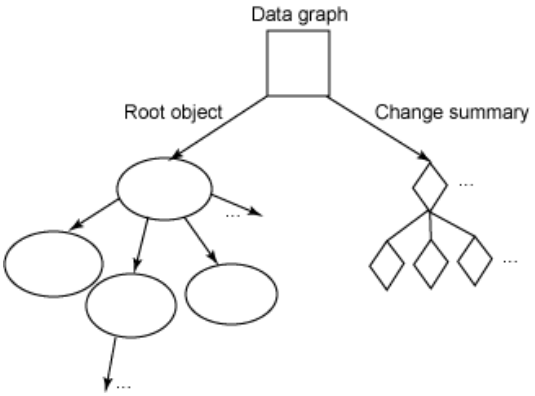
データ・オブジェクト

データ・オブジェクトは SDO の基本的なコンポーネントで、実際には、この仕様自体の名前にも示されているサービス・データ・オブジェクトです。データ・オブジェクトは、構造化データの SDO 表現です。データ・オブジェクトは一般的なもので、DMS によって作成された構造化データを共通に表示します。例えば JDBC DMS ではパーシスタンス技術 (リレーショナル・データベースなど) とその構成方法およびアクセス方法を認識する必要がありますが、SDO クライアントはそのような認識を一切必要としません。データ・オブジェクトが、オブジェクト自体の「データ」をプロパティーに保持するからです (プロパティーについての詳細は、この後説明します)。データ・オブジェクトは、作成メソッドと削除メソッド (さまざまなシグニチャーを持つ createDataObject() および delete())、そしてオブジェクトのタイプ (インスタンス・クラス、名前、プロパティー、および名前空間) を取得するための自己反映メソッドを提供します。データ・オブジェクトはお互いにリンクされてデータ・グラフ内に含まれます。

データ・グラフ

データ・グラフは、データ・オブジェクトのツリーのためのコンテナーを提供します。DMS はデータ・グラフを作成して、SDO クライアントが操作できるようにします。データ・グラフが変更されると、データ・ソースを更新するために DMS に戻されます。SDO クライアントはデータ・グラフをトラバースし、そこに含まれるデータ・オブジェクトの読み取りや変更を行います。SDO が切断されたアーキテクチャーである理由は、SDO クライアントが DMS およびデータ・ソースから切断されているためで、SDO クライアントはデータ・グラフのみを参照します。さらに、データ・グラフには異なるデータ・ソースからのデータを表すオブジェクトを含めることができます。データ・グラフには、ルート・データ・オブジェクト、ルートのすべての関連データ・オブジェクト、そして変更サマリー (変更サマリーについての詳細は、この後説明します) が含まれます。データ・グラフは、アプリケーション・コンポーネント間 (サービス呼び出しの際の Web サービス要求者とプロバイダー間など) で送信される際、DMS に送信される際、またはディスクに保存される際に XML にシリアライズされます。SDO 仕様では、このシリアライズの XML スキーマを規定しています。図 1 に、SDO データ・グラフを示します。

図 1. SDO データ・グラフ



変更サマリー

変更サマリーは、データ・グラフに含まれ、DMS によって戻されたデータ・グラフに対する変更を表すために使用されます。データ・グラフがクライアントに戻される時点では変更サマリーは

空で、データ・グラフが変更されるごとにデータが入力されます。DMS はバックエンドの更新時に変更サマリーを使用して、変更をデータ・ソースに適用します。変更サマリーは、変更されたプロパティーとそのプロパティーの古い値、そしてデータ・グラフで作成されたデータ・オブジェクトと削除されたデータ・オブジェクトのリストを提供することによって、DMS が効率的かつインクリメンタル方式でデータ・ソースを更新できるようにします。情報は、変更サマリーのロギングがアクティブな場合にのみ、データ・グラフの変更サマリーに追加されます。サンプル・アプリケーションのセクションで詳細を説明するように、変更サマリーはDMS がロギングのオン/オフを切り替える方法を提供します。

プロパティー、型、およびシーケンス

データ・オブジェクトの一連のプロパティーには、オブジェクト自体のコンテンツが含まれます。それぞれのプロパティーには型があり、この型はプリミティブ (例えば int) または一般的に使用されるデータ型 (例えば Date) などの属性型のいずれか、あるいは参照の場合は別のデータ・オブジェクト型となります。それぞれのデータ・オブジェクトは、プロパティーの読み取り/書き込みアクセス・メソッド (geter および setter) を提供します。これらのアクセサーにはいくつかのオーバーロードされたバージョンが用意されているため、プロパティー名 (String)、数値 (int)、またはプロパティーのメタ・オブジェクト自体を渡すことによってプロパティーにアクセスできます。String アクセサーは、XPath のような構文によるプロパティーへのアクセスもサポートします。例えば、会社のデータ・オブジェクトで get("department[number=123]") と呼び出すと、番号 123 の最初の部門を取得することができます。一方、シーケンスはより高度で、プロパティーと値のペアのさまざまなリストの順番を保持できます。

SDO とサンプル・インストール

概念と理論を十分に説明したところで、いよいよ実際の例を紹介します。嬉しいことに、SDO は今すぐ無料で使用できます。このセクションでは、IBM の SDO 参照実装で実行する SDO サンプル・アプリケーションを紹介します。これは、EMF (Eclipse Modeling Framework) の一部としてパッケージ化されています。最初に SDO が含まれる EMF 2.0.1 のインストール方法を説明し、次にこの記事で紹介するサンプル・アプリケーションのセットアップ方法を説明します。

EMF 2.0.1 をインストールする

EMF 2.0.1 がすでにインストール済みの場合、またはインストール方法を知っている場合は、次のセクションにスキップしてください。

IBM の SDO 1.0 実装は、EMF 2.0.1 とパッケージ化されています。SDO を使用するには、EMF 2.0.1* をインストールする必要があります。EMF サイトに記載されている Eclipse Update Manager メソッドを使用するか、次のステップに従ってください。

* SDO 1.0 の実装は、EMF 2.0.0 でも入手できます。

EMF ホーム・ページにアクセスすると、Quick Nav セクションに一連のダウンロード・リンクが記載されています。ここで必要なのは、「v2.x: EMF and SDO」ダウンロード・オプションです。EMF をインストールする前に必ず、インストール要件を読んでください。基本的には、Eclipse 3.0.1 と Java Development Kit (JDK) 1.4 がインストールされていないと、EMF 2.0.1 をインストールすることはできません。必ず EMF 2.0.1 リリース・ビルドを選択してください。パッ

ケージのタイプにはソース、ランタイム、および文書をまとめて 1 つのファイルで入手できる emf-sdo-xsd-SDK-2.0.1.zip を選択することをお勧めします。必要に応じて、「EMF & SDO RT」というラベルが付いた SDO の最小パッケージ、emf-sdo-runtime-2.0.1.zip をダウンロードしても構いません。

この zip ファイルを、Exlipse が解凍された場所に解凍します (このアーカイブ内のファイルは、eclipse/plugins/... として構成されます)。EMF が正常にインストールされたことを確かめるには、Eclipse を起動して、About the Eclipse Platform を選択します。次に Plug-in Details ボタンを押します。org.eclipse.emf.* プラグインが 2.0.1 のレベルであることを確認してください。SDO に関連するのは、以下の 6 つのプラグインです。

- org.eclipse.emf.commonj.sdo
- org.eclipse.emf.ecore.sdo
- org.eclipse.emf.ecore.sdo.doc
- org.eclipse.emf.ecore.sdo.edit
- org.eclipse.emf.ecore.sdo.editor
- org.eclipse.emf.ecore.sdo.source

実行時に必要なプラグインは、org.eclipse.emf.commonj.sdo と org.eclipse.emf.ecore.sdo だけです。ランタイム・プラグインのみをインストールすることを選択した場合は、この 2 つのプラグインだけが表示されます。EMF のインストールは以上です。

サンプル SDO アプリケーションをインストールする

次のステップは、この記事の SDO サンプル・アプリケーションをワークススペースに追加することです。以下のステップに従ってください。

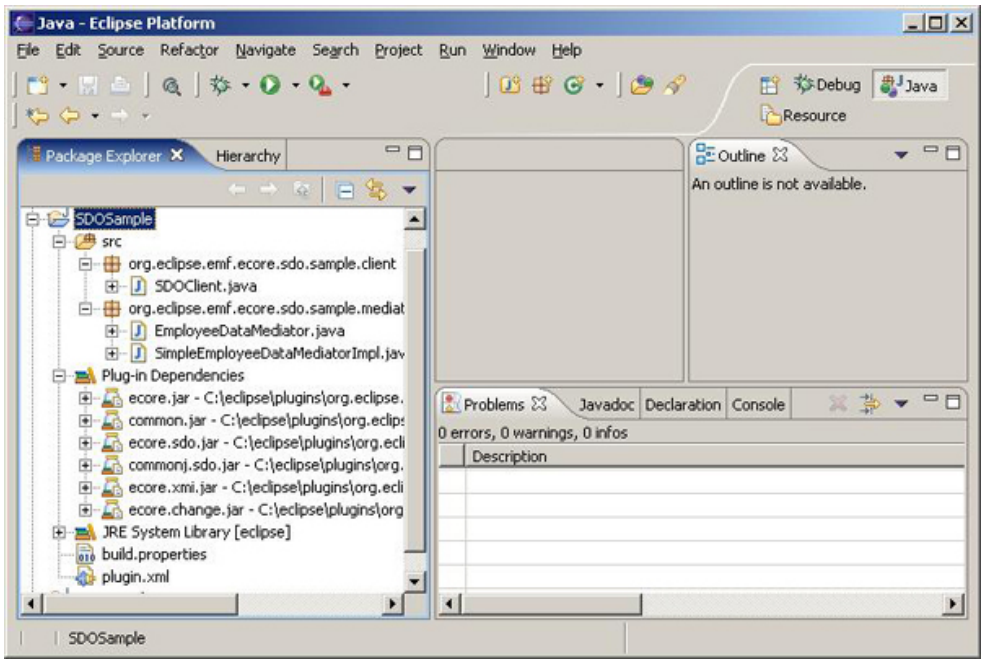
1. Eclipse を起動して、新しいプラグイン・プロジェクトを作成します。
2. プロジェクトに SDOSample という名前を付け、ソース・フォルダー src と出力フォルダー bin を持つ Java ソース・プロジェクトを作成します。
3. Next をクリックします。
4. 「Generate the Java class that controls the plug-in's life cycle (プラグインのライフ・サイクルを制御する Java クラスを生成する)」オプションのチェック・マークを外し、「Finish (完了)」をクリックします。

次に、この記事の先頭または末尾にあるコード・アイコン (または、「ダウンロード」セクションを参照) をクリックして、j-sdoSample.zip を取得します。このファイルを SDOSample ディレクトリーに解凍します (Eclipse のプロジェクト内からは、「Import... (インポート...)」を選択してから、Zip ファイルを選択)。必ずフォルダー構造を維持して、既存のファイルを上書きしてください。これで、SDOSample プロジェクトに j-sdoSample.zip のファイルが取り込まれます。

注: SDOSample は Eclipse プラグイン・プロジェクトとしてパッケージ化されているため、ライブラリーの従属関係を設定する必要はありません。ただし、このサンプルは単なる Java コードなので、CLASSPATH に EMF ライブラリーと SDO ライブラリー (JAR ファイル) が含まれている限り、スタンドアロン・アプリケーションとして実行される可能性もあります。

この時点で、Eclipse 環境は図 2 に示すスクリーン・ショットのようになっているはずです。

図 2. Eclipse 環境



これで、サンプル SDO アプリケーションを使用する準備が整いました。

単純な SDO アプリケーション

以降で使用するサンプル・アプリケーションは、機能性の点では限りがありますが、SDO をより明確に理解するのに役立ちます。このアプリケーションは 2 部構成になっていて、それぞれ対応する dms および client パッケージに分けられています。

SDO 1.0 では標準 DMS API を指定していないため、この例のために、2 つのメソッドを提供する独自の DMS インターフェースを設計しました。リスト 1 を参照してください。

リスト 1. DMS インターフェース

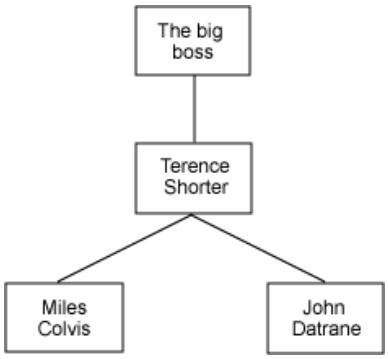
```
/**
 * A simple Data Mediator Service (DMS) that builds
 * SDO Data Graphs of Employees and updates
 * a backend data source according to a Data Graph.
 */
public interface EmployeeDMS
{
  /**
   * @param employeeName the name of the employee.
   * @return an SDO Data Graph with Data Objects for
   * that employee's manager, that employee,
   * and that employee's "employees".
   */
  DataGraph get(String employeeName);

  /**
   * updates backend data source according to dataGraph.
   * @param dataGraph Data Graph used to update data source.
   */
  void update(DataGraph dataGraph);
}
```

クライアントが DMS をインスタンス化し、The Big Boss、Wayne Blanchard および Terence Shorter という特定の従業員に対して get() メソッドを呼び出します。これらの従業員に関する情報がユーザー・フレンドリーな方法でコンソールに出力されてから、Terence Shorter と彼の部下の部門情報が更新されます。最後にクライアントは、DMS の update() メソッドを呼び出し、Terence Shorter の更新されたデータ・グラフを渡します。

ここでは説明しやすいように、データ・ソース・コンポーネントは実装していません。その代わり、DMS には、クエリーに基づいたデータ・グラフの作成方法についての知識が「ハードコーディング」されています。図 3 に、DMS が使用している従業員階層を示します。

図 3. Big Boss 企業の従業員



上記の図から分かるように、DMS の背後にある架空の会社には 4 人の従業員がいます。この会社の階層は、以下のとおりです。

- The Big Boss には上司はなく、Terence Shorter が彼の直属の部下です。
- Terence Shorter の上司は The Big Boss で、直属の部下には John Datrane と Miles Colvis がいます。
- John Datrane の上司は Terence Shorter で、直属の部下はいません。
- Miles Colvis の上司は Terence Shorter で、直属の部下はいません。

サンプルを実行する

このサンプル・アプリケーションを実行するには、SDOClient.java を右クリックして、Run > Java application を選択します。コンソールに、リスト 2 のような結果が表示されます。

リスト 2. アプリケーションのコンソール出力

```
********* EMPLOYEE INFORMATION *********

Name: John Datrane
Number: 4
Title: Mr.
Department: Procurement
Is manager?: no

DIRECT MANAGER:

Name: Terence Shorter
Number: 2
Title: Mr.
Department: Financing
```

```
Is manager?: yes

**************************************

NO INFORMATION AVAILABLE ON EMPLOYEE Wayne Blanchard

********* EMPLOYEE INFORMATION *********

Name: Terence Shorter
Number: 2
Title: Mr.
Department: Financing
Is manager?: yes

DIRECT MANAGER:

Name: The Big Boss
Number: 1
Title: Mr.
Department: Board
Is manager?: yes

DIRECT EMPLOYEES:

Name: Miles Colvis
Number: 3
Title: Mr.
Department: Accounting
Is manager?: no

Name: John Datrane
Number: 4
Title: Mr.
Department: Procurement
Is manager?: no

[Total: 2]
**************************************

DMS updating Terence Shorter
 (changed department from "Financing" to "The new department")
DMS updating Miles Colvis
 (changed department from "Accounting" to "The new department")
DMS updating John Datrane
 (changed department from "Procurement" to "The new department")
```

それでは、アプリケーションのそれぞれのコンポーネントがどのように機能するかを見てみましょう。

クライアント

SDO クライアントは DMS をインスタンス化し、さまざまな従業員のデータ・グラフを取得します。データ・グラフを取得すると、以下のようにルート・オブジェクトをナビゲートしてデータ・オブジェクトにアクセスします (SDO の動的 API を使用)。

```
// Get the SDO DataGraph from the DMS.
DataGraph employeeGraph = mediator.get(employeeName);
...
// Get the root object
DataObject root = employeeGraph.getRootObject();
...
// get the employee under the manager
employee = theManager.getDataObject("employees.0");
```

次にクライアントは以下のように動的 SDO アクセサー API を呼び出し、データ・オブジェクトから情報を取得してコンソールに出力します。

```
System.out.println("Name: " + employee.getString("name"));
System.out.println ("Number: " + employee.getInt("number"));
...
System.out.println ("Is manager?: " +
  (employee.getBoolean("manager") ? "yes" : "no") + "\n");
```

クライアントが情報を取得する (読み出す) 方法は上記のとおりですが、書き込みについてはどうでしょう。具体的には、クライアントはどのようにしてオブジェクトを変更するのでしょうか。それには、DataObject 書き込みアクセサー・メソッドを使用します。SDO クライアントは通常、このメソッドを使用してデータ・オブジェクトを更新します。例えば、クライアントは以下のようにして、従業員 Terence Shorter について取得したデータ・グラフを変更します。

```
employee.setString("department", newDepartmentName);
```

ここで、クライアントがロギング・メソッドを呼び出していないことに注意してください。ロギングは、DMS がデータ・グラフの変更サマリーで beginLogging() および endLogging() を呼び出すことによって行います。

データ・グラフ

データ・グラフのデータ形式 (モデル) は、DMS とクライアントとの間の契約と考えることができます。クライアントは DMS からこのモデルを期待し、DMS はこのモデルのビルド方法 (および、バックエンド・データ・ソースを更新するために読み出す方法) を認識しています。XML や Web サービスを使い慣れている場合は、データ・オブジェクトを定義する XML スキーマ (XSD) としてこのデータ・グラフ・モデルを考えてみてください。そうすれば、データ・グラフ自体は XML インスタンス文書のようなものであることが分かるでしょう。実際、XML スキーマは、SDO モデルを定義する方法の 1 つとして使用できます。

データ・グラフとそれぞれのモデルは常に、XML にシリアライズされます。SDOClient.java で debug 変数を true に設定すると、実行時にシリアライズされたバージョンの結果データ・グラフがコンソール上に表示されます。これは、リスト 3 のように表示されるはずです。

リスト 3. データ・グラフのシリアライズ・バージョン

```
<?xml version="1.0" encoding="UTF-8"?>
<sdo:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:company="http://com.example.company.ecore"
  xmlns:sdo="commonj.sdo"
  xmlns:sdo_1="http://www.eclipse.org/emf/2003/SDO">
  <changeSummary>
    <objectChanges key="#//@eRootObject">
      <value xsi:type="sdo_1:EChangeSummarySetting"
        featureName="department" dataValue="Board"/>
    </objectChanges>
  </changeSummary>
  <company:Employee name="The Big Boss"
    number="1" department="The new department" title="Mr."
    manager="true">
    <employees name="Terence Shorter" number="2"
      department="The new department" title="Mr." manager="true">
      <employees name="Miles Colvis" number="3"
        department="The new department" title="Mr."/>
```

```
      <employees name="John Datrane" number="4"
        department="The new department" title="Mr."/>
    </employees>
  </company:Employee>
</sdo:datagraph>
```

この例でのデータ・グラフは、Employee データ・オブジェクト (および変更サマリー) で構成されています。Employee には、name、number、department、title、manager (その従業員の上司である別の従業員)、および employees (その従業員の部下である別の従業員) などの属性があります。この例では、従業員がハードコーディングされたデータ・ソースに存在する場合、DMS によって戻されるデータ・グラフは常に、従業員の上司 (存在する場合)、要求対象の従業員、およびその直属の部下 (存在する場合) という形式になります。

DMS: グラフを作成する

SDO 1.0 では、データ・グラフ・モデル自体の設計および作成が組み込まれる DMS API を指定していません。データ・ソースへのアクセスを作成する場合に考慮する必要のあるシナリオは数多くあり、データ・グラフの設計自体を主題にした記事が書けるほどです。

この例では、動的 EMF API を使用して、DMS で定義された従業員モデルを扱います。このサンプル・データ・グラフには、XSD などのモデル文書はありません。データ・オブジェクトは動的に生成されているため、Employee の Java クラスは生成されていません。静的メソッドを使用した場合は、その逆になります。

DMS は、さまざまなデータ・アクセス API (JDBC、SQL など) を使用して、各種のデータ・ソースから情報を取得します。ただし、情報がバックエンド(この例では、ハードコーディングされた知識) から取得されると、DMS は SDO の API ではなく、EMF の API (eGet、eSet) を使用してデータ・オブジェクトのデータ・グラフを作成します。この方法によって最適なパフォーマンスが得られますが、SDO 実装の間では移植できないという欠点があります。

パフォーマンスはあまり問題ではないという場合は、まったく同じ DMS 設計を SDO の API を使用して実装することも可能です。その場合、DMS クラス内のキャッシュされたメタ・オブジェクト (employeeClass、employeeNameFeature など) のタイプは、EMF のタイプ EClass、EAttribute、EReference ではなく、commonj.sdo.Type および commonj.sdo.Property になります。さらに、パフォーマンスをまったく気にしないという場合は、便利なストリング・ベースの SDO API (setBoolean(String path, boolean value) など) を使用して、メタ・オブジェクトをキャッシュする必要をなくすこともできます。ただし、より便利な一方、このソリューションの処理速度はかなり遅くなります。

以下のコード・スニペットに、SimpleEmployeeDataMediatorImpl.java での従業員モデルの定義方法を示します。これは SDO オブジェクトをビルドするためのコードではなく、単なる SDO オブジェクトのモデルです。

```
protected EClass employeeClass;
protected EAttribute employeeNameFeature;
protected EReference employeeEmployeesFeature;
...

employeeClass = ecoreFactory.createEClass();
employeeClass.setName("Employee");

EAttribute employeeNameFeature = ecoreFactory.createEAttribute();
...

// employees (that the employee manages)
employeeEmployeesFeature = ecoreFactory.createEReference();
employeeEmployeesFeature.setContainment(true);
...

EPackage employeePackage = ecoreFactory.createEPackage();
employeePackage.getEClassifiers().add(employeeClass);
...
```

employees EReference では値を true に設定して setContainment を呼び出しているため、各従業員にはそれぞれの上司/部下が含まれることになります。このようにしなかった場合、ネストされた従業員階層はデータ・グラフに含まれず、変更サマリーにはグラフのルート・オブジェクト以外の従業員に対する変更が含まれません。

SDO をモデル化する

この時点で、おそらくこう思っていることでしょう。「でも、これでは SDO データ・オブジェクトではなく EMF オブジェクトが作成されてしまう。何かトリックがあるのだろうか」。答えは単純明快です。Employee EClass は employeePackage EPackage に属し、以下を呼び出すからです。

```
// Have the factory for this package build SDO Objects
employeePackage.setEFactoryInstance(
  new DynamicEDataObjectImpl.FactoryImpl());
```

実行時に、ファクトリーは DynamicEDataObjectImpl タイプのオブジェクトを作成します。このオブジェクトは、通常の EMF オブジェクトのみを作成するデフォルトの DynamicEObjectImpl ではなく、DataObjectinterface (つまり、SDO データ・オブジェクト) を実装します。これは、SDO オブジェクトと EMF オブジェクトの関係を際立たせています。つまり、SDO オブジェクトは SDO DataObject インターフェースも実装する、単なる EMF オブジェクトなのです。実際、これらの追加メソッドは、コアとなる EMF メソッドへの委譲によって実装されます。

SDO インスタンスを作成する

データ・オブジェクトのモデルが準備できたところで、Employee のインスタンスを作成して各種のプロパティーを設定してみましょう。前述したように、パフォーマンスを最大限にするため EMF API を使用します。

```
EObject eObject = EcoreUtil.create(employeeClass);

// Note: we could cast the object to DataObject,
// but chose to use EObject APIs instead.
eObject.eSet(employeeNameFeature, name);
eObject.eSet(employeeNumberFeature, new Integer(number));
... ...
```

次に、「employees」参照を使用して、例えば以下のように従業員同士を「リンク」します。

```
((List)bigBoss.eGet(employeeEmployeesFeature)).add(terence);
```

データ・オブジェクトを作成したら、データ・グラフに追加する必要があります。そのためには、データ・グラフの setRootObject() メソッドを呼び出し、ルートに配置するデータ・オブジェクトを渡します。この例では、そのデータ・オブジェクトは Employee The Boss です。

```
EDataGraph employeeGraph = SDOFactory.eINSTANCE.createEDataGraph();
... ...
employeeGraph.setERootObject(rootObject);
```

データ・グラフを戻す前にもう 1 つ必要なことは、変更のロギングを開始することです。SDO の機能を使用して開始する場合は、データ・グラフに変更が加えられる前に、変更サマリーで beginLogging() を呼び出します。このメソッドは基本的に、前の変更をすべてクリアした後に変更のリッスンを開始します。

```
// Call beginLogging() so that the Change Summary is
// populated when changes are applied to the Data Graph.
// The DMS should call beginLogging() and endLogging(),
// not the client.
employeeGraph.getChangeSummary().beginLogging();
```

EmployeeDataMediator インターフェースで定義されている DMS のもう 1 つのタスクは、SDO クライアントによって提供されたデータ・グラフに基づいて、バックエンドのデータ・ソースを更新することです。

DMS: ソースを更新する

DMS はバックエンド・データ・ソースを更新するために、SDO の強力な機能、つまり変更サマリーを使用します。1 つのデータ・グラフの変更サマリーを使用するだけでも、いろいろな方法があります。この例では、変更サマリー・ツリーから参照されるすべてのデータ・オブジェクトを調べて、新しいデータ・オブジェクトを取得します。

リスト 4. データ・グラフに従ってバックエンドを更新する DMS

```
/**
 * Update the DMS's backend data to reflect changes
 * in the data graph.
 * Since this DMS has no actual backend data and therefore
 * has nothing to update, we will just navigate
 * the change summary and report (print) what's changed.
 */
public void update(DataGraph dataGraph)
{
  ChangeSummary changeSummary = dataGraph.getChangeSummary();

  // Call endLogging to summarize changes.
  // The DMS should call beginLogging() and endLogging(),
  // not the client.
  changeSummary.endLogging();

  // Use SDO ChangeSummary's getChangedDataObjects() method.
  List changes = changeSummary.getChangedDataObjects();
```

```
  for (Iterator iter = changes.iterator(); iter.hasNext();)
  {
    DataObject changedObject = (DataObject)iter.next();
    System.out.print("DMS updating " +
      changedObject.getString("name"));

    for (Iterator settingIter = changeSummary.getOldValues(
      changedObject).iterator(); settingIter.hasNext();)
    {
      ChangeSummary.Setting changeSetting =
        (ChangeSummary.Setting)settingIter.next();
      Property changedProperty = changeSetting.getProperty();
      Object oldValue = changeSetting.getValue();
      Object newValue = changedObject.get(changedProperty);

      System.out.print(" (changed: " + changedProperty.getName() +
        " from \"" + oldValue + "\" to \"" + newValue + "\")");
      // If not a simple example, we could update the backend here.
    }

    System.out.println();
  }
}
```

上記の例では、バックエンドの更新は行われていません。実際には、このメソッドでバックエンドが更新されることになります。

DMS がバックエンド更新のためにクライアントからデータ・グラフを取り戻したときに最初に行うことは、データ・グラフの変更サマリーで endLogging() を呼び出すことです。これによって変更の記録が停止され、beginLogging() が呼び出された後 (通常は、グラフが作成された後) にデータ・グラフに行われた変更の要約が提供されます。この形式によって、DMS はバックエンド・データ・ソースを効率的かつインクリメンタル方式で更新できます。変更サマリーに含まれる変更には、以下の 3 つのタイプがあります。

- **オブジェクト変更** には、データ・グラフ内のプロパティーが変更されたデータ・オブジェクトへの参照、および変更されたプロパティーとそのプロパティーの古い値が含まれます。DMS はこの古い値を使用して、誰かによってバックエンド・データが目下、変更されていないことを確認できます。
- **オブジェクト作成**には、データ・グラフに追加されたデータ・オブジェクトが含まれます。これらのオブジェクトは、バックエンド・データ構造に追加する必要のある新規データを表します。
- **オブジェクト削除** には、データ・グラフから削除されたデータ・オブジェクトが含まれます。これらのオブジェクトは、バックエンド・データ構造から除去する必要のあるデータを表します。

ここでは、標準 SDO API を使用して、データ・グラフの変更を点検しました。一方、SDO の ChangeSummary の代わりに EMF ChangeDescription API を使用することもできます。単純な属性の値を更新するこの例では、パフォーマンスへの影響は顕著にはなりません。多様性のある多数のプロパティーを変更するような場合は、EMF API によってパフォーマンスが大幅に改善されます。例えば、数百人の従業員リストから 1 人の従業員を削除するとします。この場合、ChangeSummary は古い値にしかアクセスしません。つまり、数百人の従業員が含まれる古いリストです。一方、EMF の ChangeDescription インターフェースは、「インデックスでの従業員の削除」など、より正確な情報を提供するため一層便利です。

この例では、変更サマリーにはオブジェクトの変更だけが含まれ、削除や追加は含まれていないことも注意してください。SDO 実装を操作して、データ・グラフからオブジェクトを除去すると、objectsToAttach というタイプのエレメントがあることに気づくはずです。これは実際、オブジェクト削除に対する EMF ChangeDescription の名前です。これらは削除されたデータ・オブジェクトで、ロールバックが行われた場合にはグラフに戻される (EMF から見た変更) 必要があります。つまり要約すると、objectsToAttach == deleted objects ということになります。

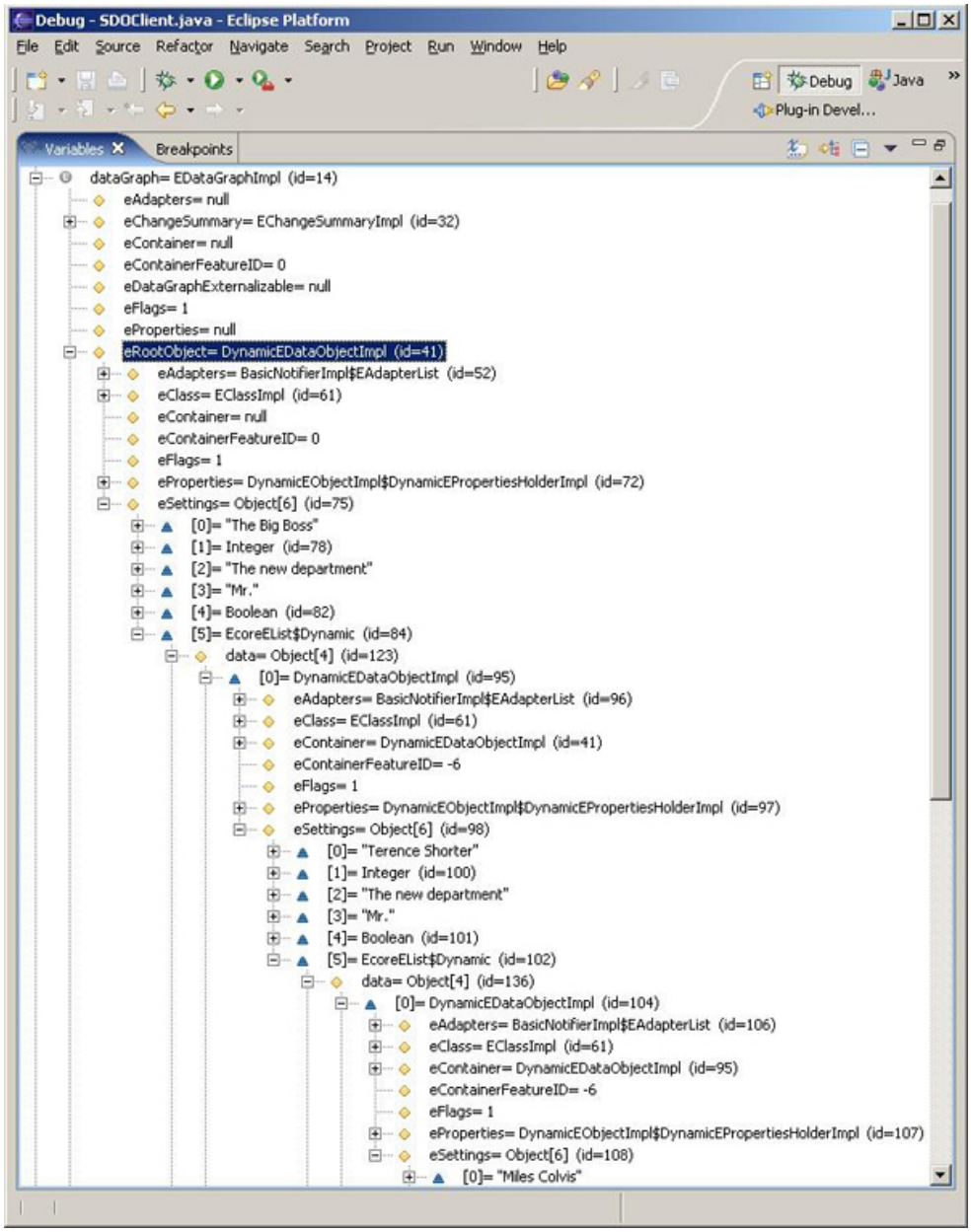
アプリケーションをデバッグする

サンプル・アプリケーションで debug 変数を true に設定すると、以下のような呼び出しが使用可能になり、シリアライズ・バージョンのデータ・グラフを表示できます

```
((EDataGraph) dataGraph).getDataGraphResource().save(System.out, null);
```

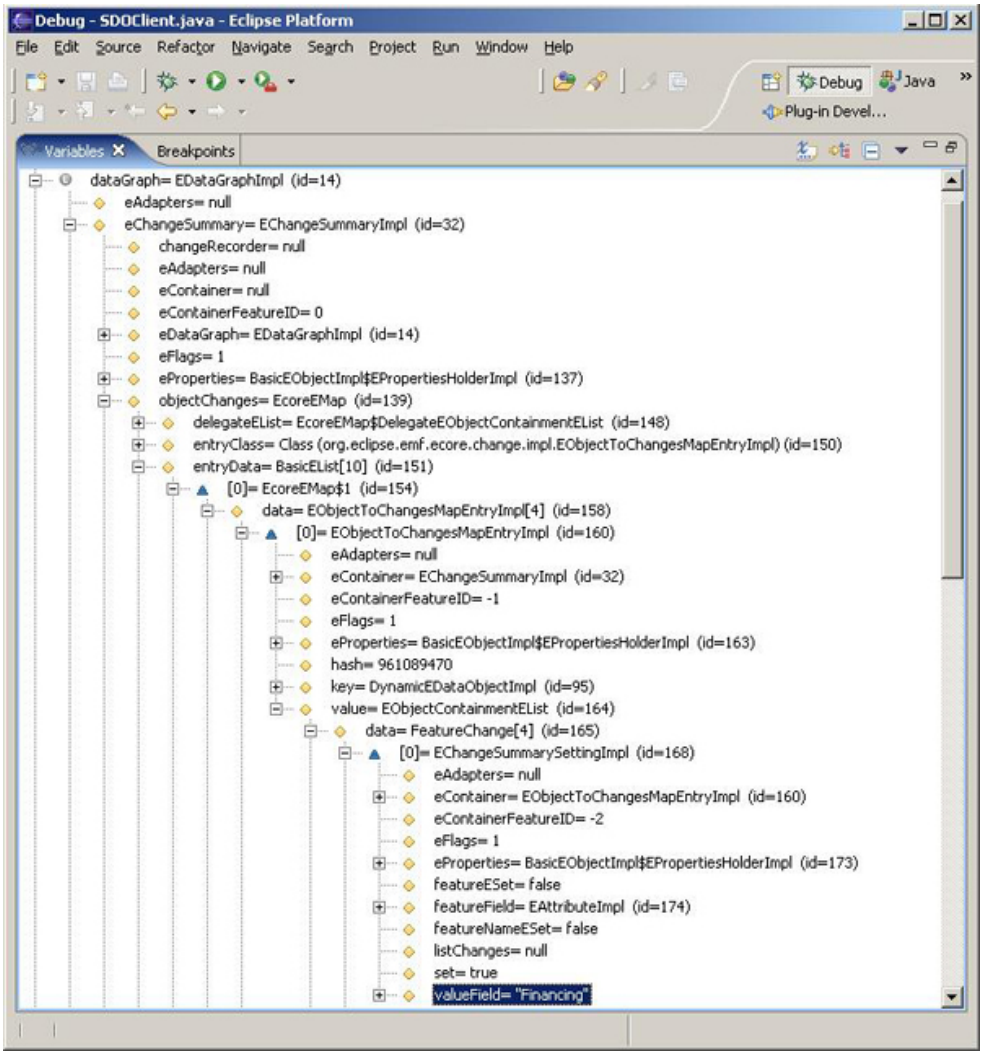
Eclipse デバッグ環境を使用することもできます。例えば、SDOClient.java のブレークポイントを行 110 に設定し、SDOClient を (Java アプリケーションとして) デバッグしてみてください。図 4 に示すように、デバッグ・パースペクティブには (変数の下に) メモリー内のデータ·グラフがデータ·オブジェクト (The Boss、Terrence Shorter など) とともに表示されます。

図 4. デバッグ・モードでのデータ・オブジェクトの表示



この方法で、変更サマリーを表示することもできます。図 5 を参照してください。

図 5. デバッグ・モードでの変更サマリーの表示



上記のスクリーン・キャプチャーは複雑で、現時点では役に立ちそうに見えませんが、SDO アプリケーションをデバッグしてデータ・オブジェクトと変更サマリーの内容を探すときには必要になるはずです。

まとめ

この記事では、SDO とその機能の概要を説明し、SDO のいくつかの機能を使用するサンプル・アプリケーションを紹介しました。詳細については、Eclipse ヘルプ・システムにある SDO API の資料を参照してください。仕様はまだ進化し、拡張されている最中です。例えば、SDO 1.0 では SDO クライアントの観点に焦点が絞られていたため、DMS API が指定されていませんでした。SDO は現在 JCP によって標準化が進められているので、その発表をお待ちください。SDO は非常に柔軟なため、SDO アプリケーションを設計する際に決定しなければならない事項がたくさんありますが、その決定によって再利用性とパフォーマンスが左右されます。コーディングを開始する前には、アプリケーション・データの使用パターンと特徴を十分に考慮してください。

ダウンロード

内容	ファイル名	サイズ
	j-sdoSample.zip	13KB

著者について

Bertrand Portier

Bertrand Portier は IBM のソフトウェア開発者で、Eclipse.org のSDO 参照実装を作成するEMF 開発チームの主要なメンバーです。J2EE での幅広い経験を持つ彼は、Web サービス分野における IBM 製品の開発とオファリングの開発に携わる他、IBM カスタマーの分散アプリケーションの開発も支援しています。

Frank Budinsky

Eclipse.org での Eclipse Modeling Framework プロジェクト・リーダーである Frank Budinsky は共同アーキテクトで、SDO 参照実装を含め、EMF フレームワークの実装に携わっています。彼は IBM のソフトウェア・グループのエンジニアで、数年にわたってフレームワークと生成プログラムの設計に従事しています。権威ある EMF に関する書著「Eclipse Modeling Framework, A Developer's Guide」の主執筆者でもあります。