ATF 3.0 プログラマーズガイド: ドキュメントオブジェクトモデル (DOM)

ATF ドキュメントオブジェクトモデル (DOM) は、大小を問わずすべてのアプリケーションに精巧なデータ管理を提供する、モデリング、管理、および永続化のフレームワークです。DOM は、ゲームやアプリケーションのデータの読み込み、格納、検証、および変更の管理を行うための強力で柔軟、かつ拡張性のあるフレームワークです。

本書では、これらの概念の包括的な概要と ATF DOM を使用するツールの開発テクニックについて説明します。

> **注意:** ATF 3 で提供される DOM は ATF 2 の DOM とは大幅に異なります。バージョン 3 よりも前のバージョンの ATF で作業している場合は、以前のバージョンの『ATF プログラマーズガイド: ドキュメントオブジェクトモデル (DOM)』を参照してください (ATF 2 ソフトウェアに付属)。

バージョン	改訂日	作成者	コメント
3.0	2010/4/30	Laura Lemay	初版
3.01	2010/7/20	Laura Lemay	若干の編集
3.1	2011/2/25	Laura Lemay	DOM アダプタ、検証、永続化、検証、プロパティ記述子、DOM エクスプローラー、イベントに関する大幅な追加

本書に対するご意見、ご要望について

皆様のご意見、ご要望は弊社にとって重要です。本書に対するご意見やご質問は、以下のリンクを使用して T&T ドキュメントフィードバックフォーラムにご投稿ください。

ATF Documentation Feedback Forum

目次

1. はじめに

Authoring Tools Framework (ATF) は、機能豊富で製品レベルの Windows クライアントアプリケーションおよびゲーム関連のツールの作成に使用できる C# の .NET コンポーネントの集まりで構成されています。2D の Visio タイプのツールから複雑な 3D 入力および編集ツールにいたるすべてが ATF で作成可能です。ATF は現在まで、サウンドモデリングとオーディオバンクのツール、レベルエディタ、キャラクターアニメーションのブレンドツール、スクリプト言語デバッガ、タイムライン/シーケンスツールのベースとして使用されてきました。

ATF ドキュメントオブジェクトモデル (DOM) は、大小を問わずすべてのアプリケーションに精巧なデータ管理を提供する、モデリング、管理、および永続化のフレームワークです。DOM は、ゲームやアプリケーションのデータの読み込み、格納、検証、および変更の管理を行うための強力で柔軟、かつ拡張性のあるフレームワークです。

本書について

『ATF プログラーマーズガイド: ドキュメントオブジェクトモデル (DOM)』は ATF DOM の主要な機能と概念の概要、Sce.Atf.Dom 名前空間の主要なクラスとインタフェース、および ATF アプリケーションで DOM を使用するための共通のタスクについて説明します。

本書は ATF を使用するツールディベロッパーを対象にしています。読者は一般に、C#、Visual Studio、および .NET 開発に習熟している必要があります。

本書の内容

本書は、次の章で構成されています。

はじめに: この章です。

ドキュメントオブジェクトモデルについて: ATF DOM の概要および DOM の主要コンポーネントと機能がどのように整合するかを説明します。

データモデリング、XML スキーマ、および DOM: アプリケーションのデータモデルを定義する方法、型定義ファイルを作成する方法、および型ローダで型定義ファイルを読み込む方法について説明します。型定義言語に XML スキーマを使用する場合に使用可能なユーティリティと基底クラスの情報が得られます。

DOM ノードと DOM アダプタ: DOM ノードツリーにはアプリケーションデータが格納されます。DOM ノードと DOM ノードツリーに関する情報、および DOM 拡張とアダプタがこれらのノード内のデータにオブジェクト API を提供する方法について説明します。データモデルの型に独自の DOM アダプタを実装し登録する方法を説明します。

その他の DOM の機能とユーティリティ: イベント、検証、プロパティ記述子、DOM エクスプローラー、DOM 永続化など、上記の章以外の DOM の機能と特徴を説明します。

関連資料

最新の ATF ソースコード配布、ニュース、機能、使用者の声、およびドキュメントなど、ATF の詳細は、SHIP の Authoring Tools Framework ページを参照してください。

ATF 配布には、ATF アーキテクチャの多様な面を示す、実行可能なサンプルアプリケーション一式が含まれています。これらのチュートリアルのドキュメントは、SHIP のコードサンプルのページにあります。サンプルのビルドに関する情報は、ATF3¥Components¥wws_atf¥ReadMe.txt ファイルを参照してください。

ATF の API クラスとメソッドの完全なリファレンス資料は、ソースコード本体内のコメントおよび Visual Studio の Intellisense の一部として入手できます。

ATF ドキュメントはすべて、SHIP の ATF SourceForge プロジェクトの [Documents] タブか、ATF 配布の ATF¥Docs フォルダにあります。

2. ドキュメントオブジェクトモデルについて

ATF のドキュメントオブジェクトモデル (DOM) は、オブジェクトのモデリング、管理、および永続化のフレームワークです。DOM は、アプリケーションデータ管理の処理が、単一のアプリケーションにおいては容易になるように、複数のプロジェクト間では一貫性が向上するように設計されています。DOM を使用することで、アプリケーションデータの設計、読み込み、格納、検証、および変更の管理をコードの残りの部分すべてから独立させられるため、ユーザは内部のデータ変更を気にする必要がなくなります。

DOM はデータをメモリにオブジェクトグラフとして格納し、検証、データ変更時のイベント発生、および必要に応じて変更内容を元に戻すインフラを提供します。ATF の MVC (Model-View-Controller/モデルビューコントローラ) 設計の「モデル」部分として、DOM を使用する GUI コントロールやアプリケーション (ATF 自体が提供するものすべてを含む) に対して、DOM で発生した変更が自動的に通知されます。

DOM には多くの利点がありますが、ATF の使用には必須ではなく、データバインディングや「単純な旧型のオブジェクト」などの、従来のデータ格納方法を使用できます。

> **注意:**「ドキュメントオブジェクトモデル」は、XML の用語で、XML 要素のメモリ内階層を表現したものです (XML 用語における「ドキュメント」とは、定義された構造を持つデータの集合体です)。ATF DOM には、この基本データ表現以上のものが含まれています。ATF DOM は元は XML から取り入れた多くの概念に基づいており、デフォルトで XML ファイルの読み書きができますが、XML には依存しません。データモデル設計とデータ格納の両方に任意のフォーマットを使用できます。
>
> また、ATF DOM は W3C DOM とは関連がないことにも注意してください。

本書に記載されている、中核となる DOM を使用するためのクラスとインタフェースはすべて、Sce.Atf.Dom 名前空間に格納されます。これらのクラスの大半は、Atf.Core アセンブリに含まれており、少数が Atf.Gui に含まれています。

DOM の構成要素

DOM の 3 つの主要構成要素を次に示します。

- データモデルには、アプリケーションのデータ型の表現が含まれています。データモデルを表現する型定義ファイルは、実行時にアプリケーション内に読み込まれます。型メタデータオブジェクトのセットはそのファイルから生成されます。

- DOM ノードのマネージドツリーには、実際のアプリケーションデータが格納されます。DOM ノードツリーには豊富なイベント管理機能があり、ツリーとその子ノードの任意の特定ノードのイベントをリッスンできます。一意なノードの命名、参照追跡、トランザクション、DOM ツリーと基本ドキュメントの同期、および名前検索の機能はすべて含まれていますがオプションです。DOM ノードツリーは、ファイルなどの永続的データストアから読み書き可能です。
- DOM 拡張は、定義済みのデータモデル型とアプリケーション間のオブジェクト API を定義します。拡張は任意のオブジェクトですが、一般的には DomNodeAdapter 基底クラスから派生します。アプリケーションのコードは、DOM 拡張を介して DOM ノードのデータにアクセスします。アダプタやその他の拡張を実装し、モデルが読み込まれたら、データモデルの型に登録します。

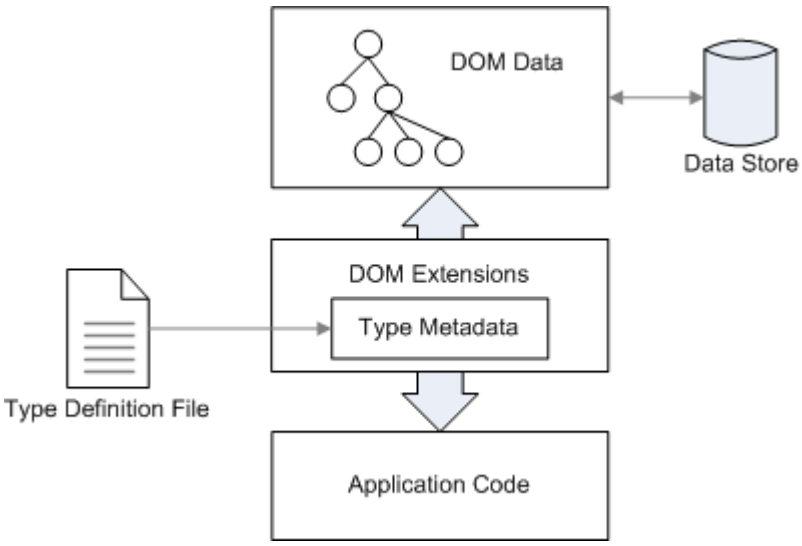

図 1:　DOM の構成要素

データモデル

ATF DOM を使用するには、最初にアプリケーションデータのデータモデルを定義します。そのデータ型の表現および、型の属性と型同士の関係も定義します。データにアダプタを作成する際、データモデルのこれらの定義が、実装するクラスとプロパティを決定します。

アプリケーションのデータモデルは、アプリケーションの外部にある型定義ファイルで定義します。選択したデータモデルの記述において、DOM では XML スキーマ定義言語 (XSD) がネイティブにサポートされていますが、任意の型定義言語を使用できます。

作成した型ローダは型定義ファイルをインポートし、そのファイル内の型に対応した内部 DOM メタデータオブジェクト (DomNodeType) を生成します。XML スキーマを型定義ファイルに使用する場合、XML スキーマの型ローダ基底クラス (XmlSchemaTypeLoader) で大半の処理が行われるので、特定の型に応じて型ローダをカスタマイズするだけで済みます。他の型定義言語を使用する場合、独自の型ローダを実装する必要があります。

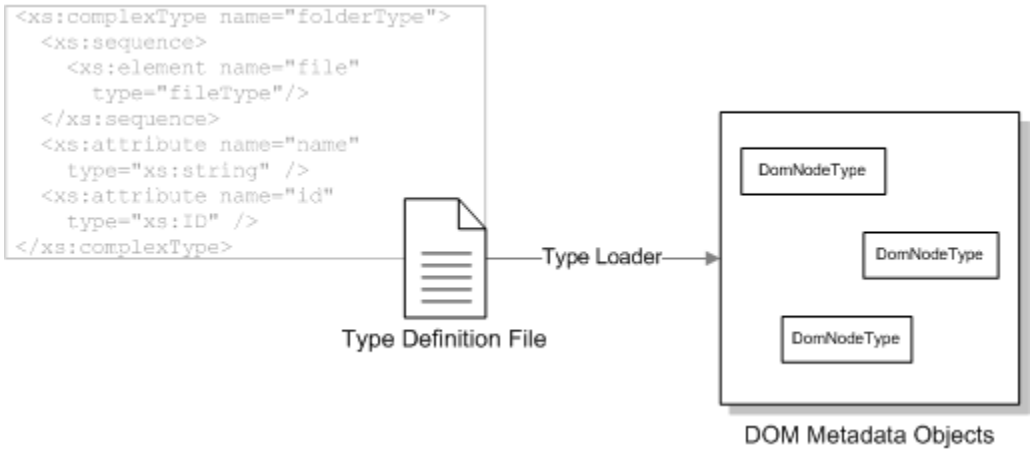

図 2:　型定義ファイルと型ローダ

また型ローダは、使用する型に応じた DOM アダプタの登録、型定義ファイルに存在していた可能性があるその他のデータ (検証規則、注釈、識別子など) の解析や管理を行います。

DOM ノードツリー

アプリケーションデータは、個々の DOM ノード (DomNode クラスのインスタンス) のツリーとして DOM に格納されます。各 DOM ノードには、基本 DOM メタデータオブジェクト (DomNodeType) があります。メタデータオブジェクトは、データモデルと型定義ファイルで定義した型から作成されますが、そのノードの構造 (属性、子ノード) を定義します。

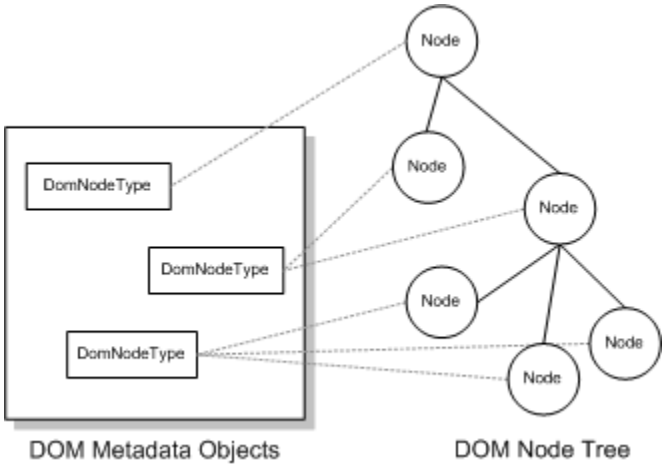

図 3:　DOM ノードツリーと基本 DOM メタデータオブジェクト

DOM ノードツリーのデータを管理するために、DOM には多くの機能が含まれています。次にその一部を示します。

- 機能豊富なイベント管理: ツリー内のノードのイベントをリッスンし、そのノードとすべての子のイベントを受信します。
- データ検証: 検証規則 (たとえば、属性のサイズや子要素の数に関する制約) は、データモデルと型定義ファイルで定義します。ツリーでノードが追加、削除、または変更されると、データ検証が実際に行われます。
- 一意性: 一意性を要する型には、DOM がこれらのノードに一意な ID または名前を確保します。オプションのアダプタによりこの振る舞いが有効になります。
- 参照追跡: DOM は、ツリーの複数の DOM ノード間における参照を有効にします (技術的に DOM ノードツリーは実際はグラフです)。オプションのアダプタにより複数のノード間の参照追跡が有効になります。
- DOM ノードツリーは、個々のドキュメントまたは、個別に管理されたサブドキュメントを持つマスタードキュメントのいずれかのドキュメントを表現できます。また DOM は、ノードツリーに変更があるとドキュメントが「変更されている」とマークしたり、元に戻すとやり直すの変更がそのドキュメントと同期するようにします。
- DOM ノードツリーは、ファイルなどの永続的データストアから読み書き可能です。

DOM 拡張とアダプタ

DOM 拡張は、メモリ内の DOM ノードツリーのデータに対して柔軟で型安全な C# オブジェクト API を提供します。拡張は任意の型のオブジェクトですが、たいていは DomNodeAdapter 基底クラスから派生するクラスです。この基底クラスは、DOM ノードツリーのデータをオブジェクト API に適応させるための機能を数多く提供しています。DomNodeAdapter クラスを使用する拡張は DOM アダプタと呼ばれます。

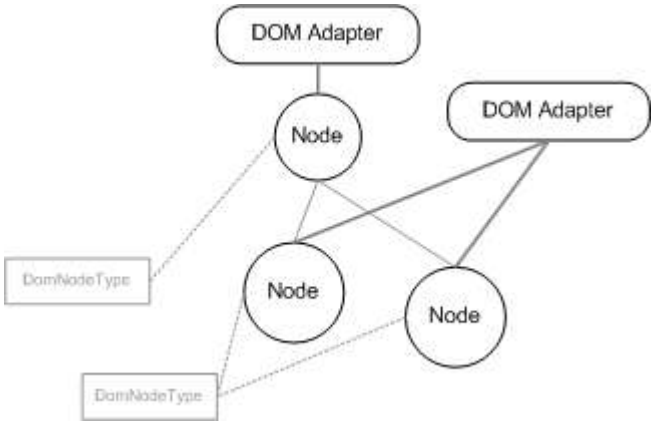

図 4:　DOM ノードと DOM アダプタ

DOM アダプタを実装するには、DomNodeAdapter 基底クラスから派生させ、データモデルから作成した DOM メタデータオブジェクト (DomNodeType) と、DOM ノードツリー (DomNode) の機能を使用します。

また、自分で開発する DOM 拡張とアダプタはいずれも、処理対象となる型に実行時に登録する必要があります。データモデルの型ローダ内に DOM 拡張とアダプタを登録します。

DOM アダプタは多くの場合、データモデルの型と 1 対 1 の対応関係にあります。簡略な DOM アダプタは多くの場合、その基本の型のオブジェクトラッパーです。たとえば、データモデルの「ステート」型は、対応する State クラスによって変換されます。ただし、DOM アダプタを型に登録すると、登録したアダプタはその特定の型とすべての下位の型に適用され、型とアダプタ両方の継承が可能になります。また、複数の DOM アダプタを同じ型に登録できるので、異なる API を同じ基本データに提供できます。多くの異なる DOM ノード型にわたってデータを使用する複雑なアダプタを作成し、これらのノードの複数の異なるイベントをリッスンするか、または DOM ツリー構造をまとめて隠すことができますが、これは処理対象の基本データと完全に同期を保ちます。

DOM アダプタクラスはいずれも、ATF のコンテキストインタフェースを介して一般的な振る舞いを追加できます。これらのインタフェースは基本的な処理に特有で、容易に実装できるように設計されています。たとえば、INamingContext インタフェースは、DOM アダプタのオブジェクトが命名可能かどうかを示します。名前付きノードには名前によ

り高速に取得されるメリットがあり、ツリーとリストのコントロールにより動的に収集できます。

ワークフロー

DOM をアプリケーションで使用する場合、アプリケーション実装時とアプリケーション実行時の手順があります。次の表にそれらの手順を示します。また、本書における参照セクションも記載しています。

実装時

表 1: DOM ワークフロー: 実装時

手順	説明	参照先
1. データモデルの定義	データモデルにより、アプリケーションで必要な型のセットを定義します。オーディオアプリケーションは、ベクター描画アプリケーション（線、長方形、色）とは大幅に異なるデータ型（ストリーム、チャンネル、バッファ）を使用します。	データモデリング、XML スキーマ、および DOM
2. データモデル用に型定義ファイルを作成	データモデルのデータ型は、アプリケーションの外部にある型定義ファイルで定義します。XML スキーマ定義言語（XSD）や独自言語など、任意の型定義言語を使用できます。	型定義ファイルの作成と読み込み または XML スキーマを使用したデータモデルの定義
3. 型定義ファイル用の型ローダクラスを実装	型ローダは実行時にデータモデルを DOM に取り込みます。データモデルの定義に XML スキーマを使用する場合、DOM は拡張可能な XmlSchemaTypeLoader 基底クラスを提供します。	型定義ファイルの作成と読み込み または スキーマ型ローダの実装
4. データモデル用の DOM 拡張クラスを実装	DOM 拡張とアダプタは、DOM ノードツリー内のデータと型をアプリケーション自体に対応付けるグルーコードです。 型定義ファイルを読み込んだ直後に、DOM 拡張とアダプタを型ローダクラスの基本の型に登録します。	DOM 拡張とアダプタ

手順	説明	参照先
5. 永続的ストレージを実装	アプリケーションファイルを保存するファイル形式を決めます。必要に応じて、DOM からこれらのファイルに読み書きするためのクラスを作成します。アプリケーションデータを XML に格納する場合、ATF DOM は、使用可能なデフォルトの DomXmlReader クラスと DomXmlWriter クラスを提供します。	DOM 永続化

実行時

表 2: DOM ワークフロー: 実行時

手順	説明	参照先
1. 型定義ファイルを読み込み	アプリケーションの初期化の一部として、型定義ファイル用の型ローダを起動します。	型定義ファイルの作成と読み込み または XML スキーマの読み込み
2. DOM 拡張クラスを登録	データモデルで型に実装した DOM 拡張クラスとアダプタを、これらの型に対して登録する必要があります。一般的に、型定義ファイル用のローダ内に DOM 拡張を登録します。	スキーマ型ローダの実装 または DOM 拡張とアダプタの登録
3. 必要に応じて、アプリケーションのデータファイルを読み込んで保存	Reader および Writer クラスを使用して、永続的データの DOM への入出力を処理します。	DOM 永続化

3. データモデリング、XML スキーマ、および DOM

データモデリングは、アプリケーションが使用するデータ型とそれらがどのように関連し合うかを定義する処理のことです。DOM を使って、外部の型定義ファイルにデータモデルを記述します。アプリケーションはその型定義ファイルを読み込むと、アプリケーションが使用する型を記述した内部メタデータオブジェクトのセットを生成します。

ATF は、データモデルを記述する型定義言語として XML スキーマをデフォルトでサポートし、XML スキーマを読み込み、DOM メタデータオブジェクトを生成するためのユーティリティとクラスを提供します。XML スキーマ以外の任意の型言語 (DDL、IDL、カスタム言語) を使用してデータモデルを記述することもできますが、これらの型定義を DOM にインポートするために、独自の型ローダまたは他のユーティリティを実装する必要が生じる場合があります。

本章は、次のセクションで構成されています。

- データモデリングについて: DOM でデータモデリングを使用する方法、型定義ファイルを読み込む方法、および型定義ファイルから DOM メタデータオブジェクトを作成する方法の概要を説明します。
- XML スキーマと DOM について: ATF DOM でデータモデリングに XML スキーマを使用する方法および XML スキーマを使用するために DOM で使用可能なツールとクラスについて説明します。
- 主要なクラス: この章に記載された最も重要なクラスとインタフェースのまとめ。
- XML スキーマの定義: XML スキーマファイルを作成してデータモデルに型を記述する方法および XML スキーマ型を DOM メタデータオブジェクトにマップする方法について説明します。
- DomGen を使用してスキーマスタブクラスを生成する: DomGen ユーティリティは XML スキーマファイルを入力として使用し、スキーマスタブクラスを生成します。スキーマ型ローダはそのスキーマタブクラスを使用して、アプリケーションのデータモデル用の DOM メタデータオブジェクトを作成します。
- スキーマ型ローダの実装: 型ローダは、スキーマが定義した型を実行時にアプリケーションに取り込みます。特定のスキーマのスキーマ型ローダクラスを作成するには、XmlSchemaTypeLoader クラスから派生します。
- XML スキーマの読み込み: 実行時にスキーマが読み込まれるように、エディタクラスコンストラクター内でスキーマ型ローダを呼び出して一つにまとめます。

XML スキーマ定義言語の完全なチュートリアルは、本書の範囲外です。本セクションの内容は、単純型と複合型、要素、属性、名前空間、URI などの XSD の概念と用語における基本的な経験が読者にあることを想定しています。XML スキーマの概念について詳しくない場合は、Web Developer's Virtual Library (http://www.wdvl.com/Authoring/Languages/XML/Schema.html) に、W3C 仕様書、書籍、記事、チュートリアルなど、XML スキーマ資料が多数紹介されています。

データモデリングについて

データモデリングは、アプリケーションが使用するデータ型とそれらがどのように関連し合うかを定義する処理のことです。これは DOM または ATF に特有の処理ではなく、大小にかかわらずすべてのアプリケーションに、何らかのデータモデルがあります。

ATF DOM は、データ型が個別の型定義ファイルに記述されているデータモデルレイヤーを使用します。その型定義ファイルは、型ローダを使用して実行時にアプリケーション内に読み込まれます。型ローダもユーザが定義しますが、これは型定義ファイル内の型を読み込み、その型の内部メタデータオブジェクトを生成し、その型の API を登録して、型定義ファイルが提供する他のデータへのアクセスを提供します。

デフォルトで、ATF は型定義ファイルに XML スキーマを使用し、XML スキーマの読み込みと管理を処理するインフラを提供します。型定義言語には任意の言語を使用できますが、型を DOM にインポートするために独自の型ローダを実装する作業が必要になります。

データモデリングの処理で定義した型はすべて、アプリケーションデータを保持する DOM ノードツリー内のノードの型も定義することに注意してください。dwarf 型と ogre 型またはステート型と遷移型もすべて、単なる DOM ノードの型です。「DOM 拡張とアダプタ」では、ツリー内の未処理の DOM ノードを高レベルのアプリケーションコードに適合させる実際の ATF クラスをアプリケーションに実装します。

データ型とエンティティ関係モデリング

データモデルを調べるための一般的な方法として、エンティティと、エンティティ間の関係から調べる方法があります (エンティティ関係モデリング)。エンティティ関係モデリングは、データモデルを汎用的かつ抽象的な意味で記述することに役立ちますが、DOM にはデータモデルを設計するときに検討する必要がある機能がいくつかあります。これらの機能を次に示します。

エンティティと属性

データモデル内のエンティティは、自己完結型の情報です。エンティティのコンポーネントは、名前、サイズ、位置などのような、そのエンティティの属性です。エンティティと属性は、要素と属性、またはオブジェクトとプロパティに相当します。

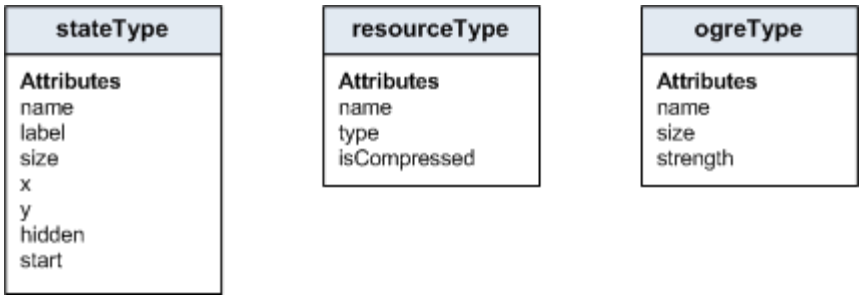

図 5:　エンティティ

クラス設計と同様に、データ型は、他のより一般的な型から継承したり、拡張することができます。

関係

データモデル内のエンティティは、関係を介して関連付けられます。DOM は管理されたオブジェクトグラフ（オブジェクトノードのツリー）なので、エンティティ間の最も一般的な関係は親子関係です。格納する要素が親、内部の要素が子になります。このように、要素はツリー内で整理され、親から子および子から親へトラバースできます。

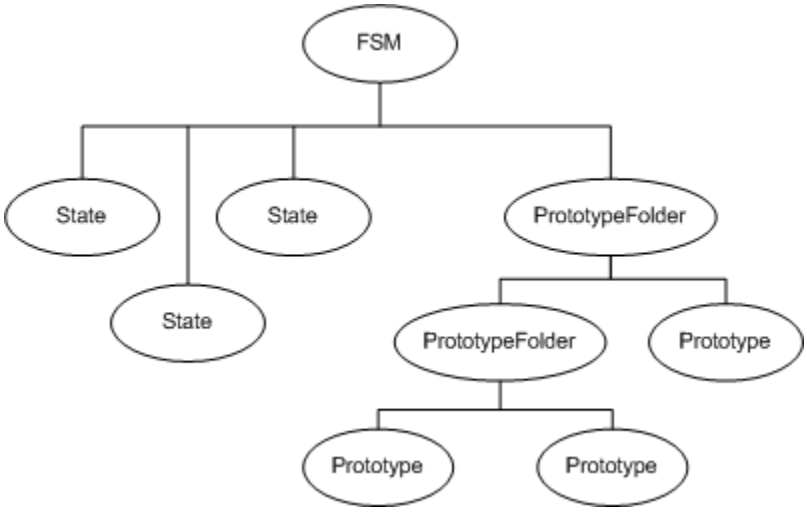

図 6:　親子関係

子エンティティは、データモデル内の各エンティティごとに、属性とは切り離して定義されます (図 7:)

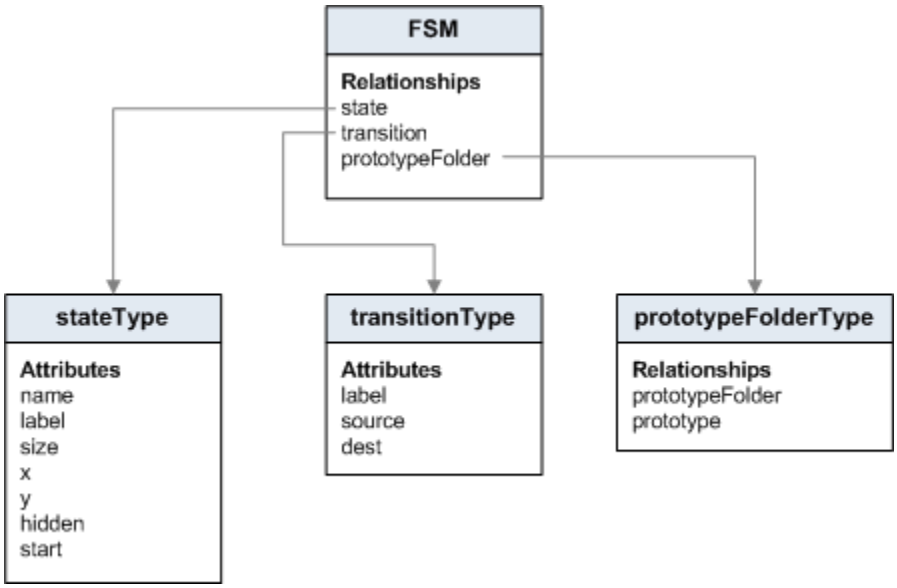

図 7:　関係としての子エンティティ

参照は、2 つの要素が親子同士でない関係です。参照先は、DOM ノードツリーの他のオブジェクトの場合も、必要な場合にのみ時間をかけて読み込まれる外部ファイルまたは他のアセットの場合もあります。通常、参照は属性ごとに定義します。

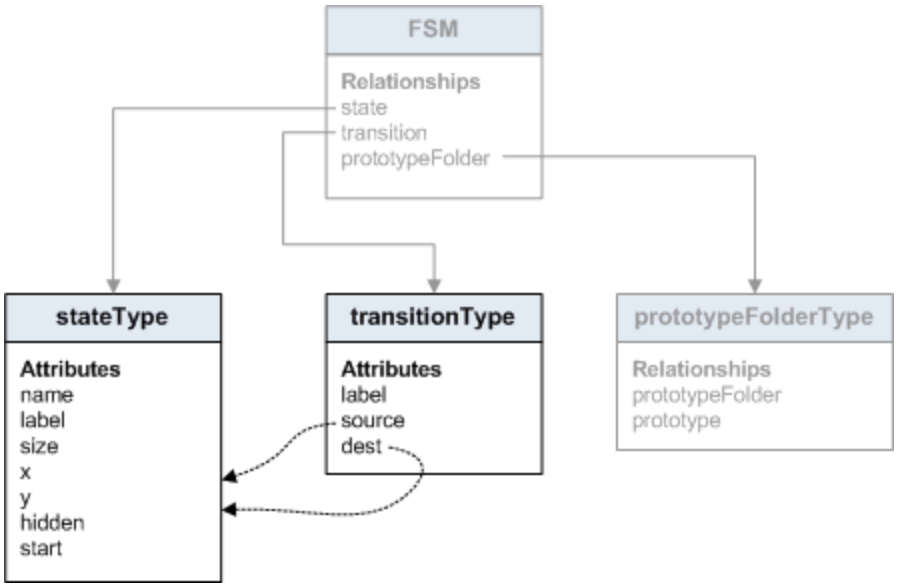

図 8:　関係としての参照

ID 属性

データモデルの主要なデータ型には多くの場合、その型のオブジェクトを一意に識別する ID が含まれています。この一意の識別子は、ID 属性と呼ばれます。すべての型に ID 属性が必要というわけではなく、一般的にコードが直接命名またはアクセスするオブジェクトのみが ID 属性を持つ必要があります。DOM が ID 属性の一意性を管理しますが、識別する必要があるオブジェクトはユーザが選択する必要があります。

DOM が ID 属性の一意性を管理するので、表示名などのユーザー側属性を ID 属性として使用するのではなく、データ型に特有の内部 ID を作成してください。DOM はオブジェクトの一意性を確保するために ID 属性を変更するため、ユーザー側の値を使用するとこの値が変化し、アプリケーションの UI に混乱が生じる可能性があります。

制限

DOM 内のデータ検証により、モデルが格納するデータの規則と制限を定義できます。たとえば、サイズ属性を、値を 1000 未満のサイズに制限する規則で定義したり、Vector3F 型を 3 つの float 型配列として定義できます。型が格納する子エンティティの数の最小値や最大値の要件を指定できます（少なくとも 1 つ、1 つまで、など）。

DOM は XML スキーマ制限ファセットに基づく多くの制限をサポートします。詳細は、「DOM データの検証」で説明します。また、独自の DOM 検証を定義して独自の制限を作成できます。データモデルを定義する際は、そのモデルのデータの型だけでなく、データに対する制限も考慮してください。

注釈

型定義ファイルには、データモデルが定義する実際の型に加えて、型注釈も作成できます。注釈を使用して、次のような、型に関する追加のメタデータを指定できます。

- パレットでデータ型を表現するために使用されるアイコン
- 3D 表示でプロキシオブジェクトに使用される画像
- データ型のカテゴリまたはソート情報
- 属性の表示名または他のプロパティエディタ情報
- 型またはプロパティのツールヒント情報

通常、注釈は型に関するランタイムメタデータを含むので、アプリケーションコード自体にアクセスまたは再コンパイルしなくても、注釈の変更や型定義ファイルへの追加が容易にできます。

DOM メタデータオブジェクト

型定義ファイルが読み込まれると、そのファイルに定義されている型はメモリ内 DOM メタデータオブジェクトに変換されます。ATF は、次の 2 つの主要な型のメタデータオブジェクトを定義します。

- DomNodeType: 属性、子 DMO ノード、および拡張を順に含むエンティティを記述します。
- AttributeType: プリミティブ値またはその値の配列となる属性を記述します。XmlAttributeType サブクラスは、XML スキーマ属性に固有の属性型を記述します。

メタデータの 3 番目の汎用クラスであるフィールドデータは、DomNodeType 内の次のフィールドを記述します。

- AttributeInfo: DomNodeType の属性に関するメタデータ。属性型 (AttributeType) と最大値など属性に定義された制限が含まれます。XmlAttributeInfo サブクラスは、XML スキーマ属性に固有の情報を追加します。
- ChildInfo: DomNodeType オブジェクトでもある、この DomNodeType の子エンティティに関するメタデータ。子情報には、子ノード型および子の数などの制限規則が含まれます。
- ExtensionInfo: この型に対して登録された DOM データ拡張 (一般的には DOM ノードアダプタ)。各型に DOM 拡張を実装し、これらの拡張をデータモデルの型ローダに登録します。

アプリケーションが型定義ファイルを読み込むと、そのファイルに定義されている型はそれぞれ、適切な内部 DOM メタデータオブジェクトに変換されます。XML スキーマを使用している場合、XmlSchemaTypeLoader 基底クラスがこの変換を行います。他の型定義言語を使用している場合は、これらのオブジェクトを自分で作成する必要があります。

型定義ファイルの作成と読み込み

データモデルの表現は、型ローダが実行時にアプリケーションに読み込む外部ファイルに定義し格納します。デフォルトでは、この外部ファイルは XML スキーマ (.xsd) ファイルです。他の形式を選択する場合は、実際の型定義のほかに、プログラマー以外が追加または編集できる注釈内に、プロパティや他のメタデータ情報が、型定義ファイルに含まれる可能性があることに注意してください。型定義言語を選択するときは、型およびその型のユーザーの実際の要件を考慮してください。型定義ファイルは通常テキスト形式で、アプリケーションの外部にある追加ツールまたは手動で編集できます。

アプリケーションが型定義ファイルを認識できるようにするには、型ローダを実装する必要があります。実行時、アプリケーションの初期化の際に、型定義ファイルへのパスで型ローダを起動して、データモデルを読み込みます。

型定義ファイルに XML スキーマを使用する場合、型ローダを記述する作業の多くは XmlSchemaTypeLoader 基底クラスによってすでに完了しています。このクラスの詳細は、「スキーマ型ローダの実装」を参照してください。独自の型定義ファイルを使用する場合、XmlSchemaTypeLoader (Atf.Core.Dom 内) 内のコードを作業の開始点にすることができますが、型ローダは自分で実装する必要があります。

型ローダが行うタスクの一部を次に示します。

- 型定義ファイルを開く、読み込む、および内容を解析する。必要に応じてそのファイル内のデータを検証する。
- 各エンティティおよび属性を対応する DOM メタデータ型に変換する。
- 独自の型に対して定義した制限規則の検証オブジェクトを作成する。
- ID 属性を定義する型に対して ID 属性を割り当てる。
- 注釈があれば、読み込み解析する。
- 定義した型の拡張または DOM アダプタ、および ID や参照検証などのユーティリティアダプタを登録する。

XML スキーマと DOM について

XML スキーマ定義言語 (XSD) は、XML 文書の記述に使用されます。XML スキーマは型定義として使用して、アプリケーションデータモデルを記述することができます。ATF は

デフォルトで XML スキーマをサポートしており、使用するためのツールを提供しています。

> **注意:** 型定義ファイルは、実際のアプリケーションデータに使用される永続化形式から独立していることに留意してください。型定義ファイルに XSD を使用しても、そのアプリケーションデータの格納に XML を使用する必要はありません。詳細は、「DOM 永続化」を参照してください。

ノード型、属性、およびノード間の関係を含むアプリケーションのデータモデルを記述するには、XML スキーマを使用します。スキーマには、データに関するプロパティを追加する注釈や、そのデータの表示方法や編集方法に関するアプリケーションへのヒントも含めることができます。たとえば、色に関する属性は、色エディタ管理をするユーザーが変更する必要がある、などです。すべての型定義ファイルと同様に、スキーマ内のデータに関連のある情報を指定することにより、アプリケーションコードの変更や多くの場合にプログラムの再コンパイルを行うことなく、そのデータを素早く変更できるようになります。

XML スキーマにより定義された型は、DOM メタデータオブジェクトに直接マップされます。たとえば、スキーマ内の複合型は、DomNodeType オブジェクトになります。単純型および単純型の配列は AttributeType オブジェクトになります。スキーマ内の制限ファセットは、適切な DOM 検証規則に変換され、複合型の制限と拡張もサポートされます。「XML スキーマの定義」では、XML スキーマファイルを作成してアプリケーションデータモデルを定義する方法および XML スキーマ型を DOM メタデータオブジェクトにマップする方法について詳細に説明します。

ATF アプリケーション内で XML スキーマを使用する手順を次に示します。

1. DomGen ユーティリティを実行して、XML スキーマからスキーマスタブクラスを生成します。
2. XmlSchemaTypeLoader から派生し、スキーマ型ローダを実装します。
3. アプリケーションの初期化の際に、スキーマ型ローダを起動します。

DomGen ユーティリティは XML スキーマを解析しスキーマスタブクラスを生成します。このスタブクラスには、DOM メタデータに直接アクセスするショートカットとしてスキーマ型ローダおよび DOM 拡張実装で使用するクラスとプロパティ定義のセットが含まれます。アプリケーションのデータモデルとスキーマが安定している場合は、DomGen は一度実行するだけで済みます。スキーマ内の型（または型の名前）を変更する場合、DomGen を再実行してスタブファイルを再生成する必要があります。DomGen ユーティリティの詳細は、「DomGen を使用してスキーマスタブクラスを生成する」を参照してください。

前のセクション（「データモデリングについて」）で説明したように、実行時に型ローダを使用してアプリケーションのデータモデルの型定義および他のメタデータをアプリケーションにインポートできます。XML スキーマを型定義言語として使用する場合、型

ローダを作成する作業の多く (XML スキーマ自体の読み込みおよび解析を含む) は、XmlSchemaTypeLoader 基底クラスによってすでに完了しています。

スキーマ型ローダを作成するには、XmlSchemaTypeLoader クラスから派生して、以下をアプリケーションに定義します。

- スキーマが定義した注釈を処理して、プロパティ記述子、パレットデータ、ツールヒント、その他の情報を作成します。
- スキーマに定義された型に対して拡張または DOM アダプタを登録します。

スキーマ型ローダの実装の詳細は、「スキーマ型ローダの実装」を参照してください。

最後に、アプリケーションの実行時にスキーマが読み込まれるように、アプリケーション内のスキーマへのパスでスキーマ型ローダを実際に起動する必要があります。詳細は、「XML スキーマの読み込み」を参照してください。

主要なクラス

Sce.Atf.Dom 名前空間には、DOM 内のメタデータを記述するためのクラスが数多く含まれています。表 3: にこれらのクラスを示します。

表 3:　データモード用 Sce.Atf.Dom の主要なクラス

クラス	説明
AttributeInfo	属性メタデータ。属性は AttributeType により定義されます。FieldMetadata を拡張します。
AttributeType	属性メタデータの型を記述します。属性はプリミティブ型またはプリミティブ型の配列です。NamedMetadata を拡張します。
AttributeTypes	属性単純型の列挙型 (プリミティブ)。
ChildInfo	子メタデータ。FieldMetadata を拡張します。
DomNodeType	DOM ノード型を記述します。NamedMetadata を拡張します。
ExtensionInfo	DOM データ型の拡張。拡張は任意の型ですが、一般的には DomNodeAdapter のサブクラスです。FieldMetadata を拡張します。
FieldMetadata	DOM ノードフィールドメタデータの抽象基底クラス (属性、子、拡張)。NamedMetadata を拡張します。
NamedMetadata	DOM ノード名/値 (プロトタイプ) メタデータの抽象基底クラス。

XML スキーマを型定義ファイルに使用している場合、DOM には、XML 属性メタデータを記述するための追加クラス、およびスキーマにより記述された DOM メタデータオブジェクトを読み込みアクセスするための追加クラスが数多く含まれます。さらに、XML スキーマに関連する型データの ATF サンプルアプリケーションでは、慣習的に 2 つのクラス名 (Schema と SchemaLoader) が使用されます。表 4: にこれらのクラスを示します。

表 4: XML スキーマ用 Sce.Atf.Dom の主要なクラス

クラス	説明
XmlAttributeInfo	XML 属性メタデータ。属性は XmlAttributeType により定義されます。AttributeInfo を拡張します。
XmlAttributeType	XML 属性メタデータの型を記述します。AttributeType を拡張します。
XmlSchemaTypeCollection	名前空間、要素、属性、および子要素を含む XML スキーマにより定義された型を表現する、DOM メタデータオブジェクトのコレクション。
XmlSchemaTypeLoader	スキーマ型ローダの基底クラス。アプリケーションのスキーマ型ローダは、このクラスから派生して作成します。
Schema	DomGen ユーティリティで XML スキーマから生成されるスキーマスタブクラスの標準的な名前。
SchemaLoader	独自のスキーマ型ローダに定義するクラスの標準的な名前。XmlSchemaTypeLoader から派生します。

DOM (DomNodeType、AttributeType、AttributeInfo、ChildInfo) のメタデータを定義する主要なクラスはすべて、抽象基底クラスの NamedMetadata と FieldMetadata から派生します。これらすべてのメタデータ型のクラス階層については、0 を参照してください。

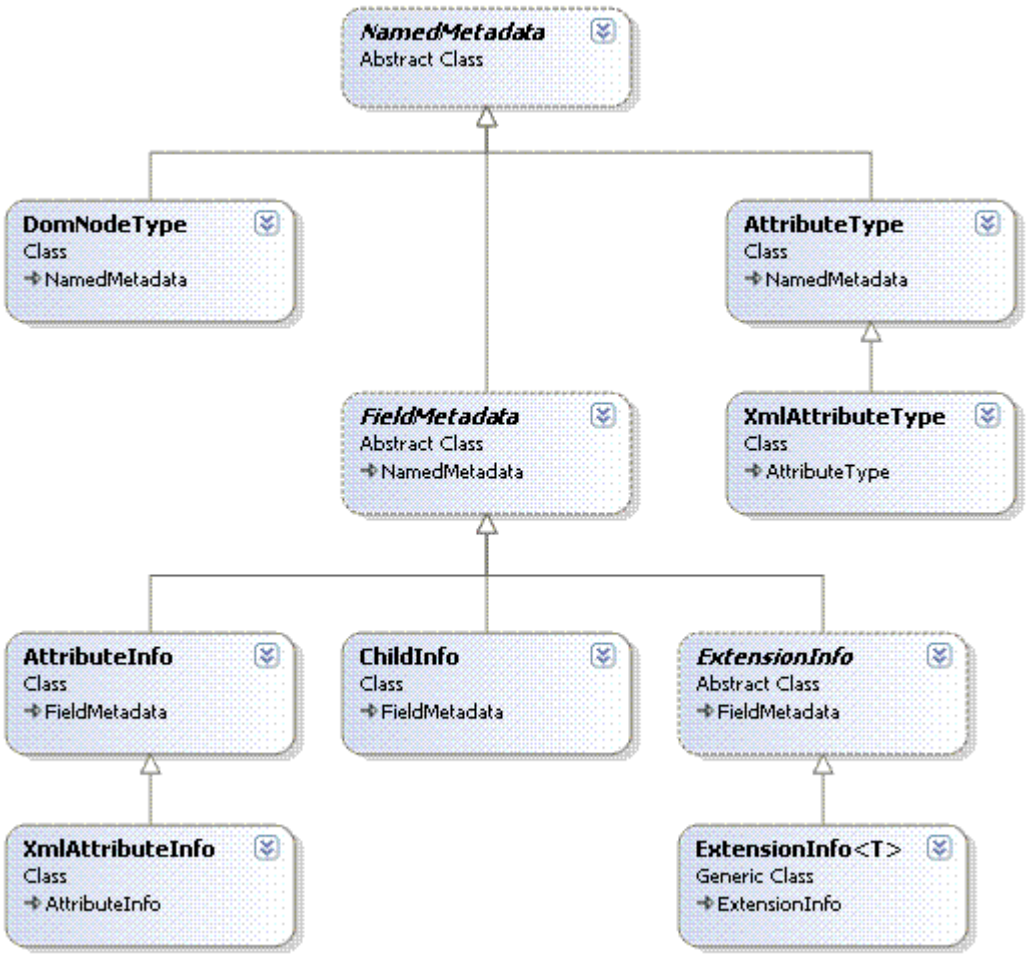

図 9:　DOM メタデータ型

XML スキーマを使用したデータモデルの定義

XML スキーマ定義言語 (XSD) は XML 文書の構造を定義するために設計されていますが、アプリケーションのデータモデルの定義にも使用できます。ATF には、型定義言語としての XML スキーマの組み込みサポートが含まれます。また、スキーマ型ファイルの読み込みと解析に役立ち、スキーマ内の型を DOM メタデータオブジェクトに変換するための基底クラスとユーティリティも含まれます。

XML スキーマファイルには、データモデルを定義するための次の機能が含まれます。

- 要素と属性
- 単純型

- 複合型
- 継承
- 親要素と子要素
- 参照
- 注釈

このセクションの終わりに、これらの機能すべてを示すスキーマの例を説明します。

要素と属性

XML スキーマの主要コンポーネントは要素と属性です。要素と属性はどちらもデータ型に関連付けられます。要素は単純型または複合型で定義できます。属性は単純型のみです。データ型の詳細は、以下の「単純型」および「複合型」を参照してください。

XML スキーマ内の要素は、データモデル内のエンティティで、情報の特定の単位です。XML スキーマ内の属性は、エンティティ関係モデリングにおける属性と同じ役割を果たします。つまり、エンティティのプロパティまたはコンポーネントを表します。

要素間の関係には、親子関係と参照の 2 つの形があります。これらの関係は以下の「親要素と子要素」および「参照」で説明します。

単純型

単純型は、整数、浮動小数点数、日付と時刻、文字列などのプリミティブ型、および ID や URI などの特殊型です。XML スキーマ仕様で定義された、定義済みの組み込み単純型のセットがあります。単純型は主に要素の属性を記述します。要素自体が単純型を持つことがありますが、これは他の内容 (子要素または属性) を含んでいない場合のみです。

また、プリミティブ単純型のリスト (<xs:list>) および、最大値、最小値、列挙などの値の制限 (<xs:restriction>) を含めることもできます。

XML 型ローダが XML スキーマを読み込む際に、単純型定義はすべて XmlAttributeType DOM メタデータオブジェクトに変換されます。制限付きの単純型定義は、DOM 検証規則に変換され、他のメタデータ (属性デフォルト値、注釈) が保存されます。

xs:boolean や xs:integer などの組み込み単純型は、.NET 型に変換されてから、同じまたは類似の型の XmlAttributeType オブジェクトに変換されます (0)。AttributeTypes 列挙は、サポートする DOM 属性型を一覧表示します。XML スキーマ型とそれに対応する .NET プリミティブ型の詳細は、「XML スキーマ (XSD) 型と .NET Framework 型の間でのデータ型のサポート」の表を参照してください。

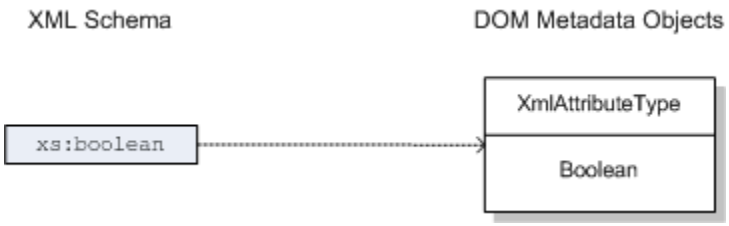

図 10: XML 型ローダによる単純型の変換

単純型には特別に処理されるものがあります。たとえば、xs:ID は要素に固有の ID 属性を示し、xs:IDREF と xs:anyURI は参照型です。これらの型の詳細は、「参照」を参照してください。

リスト型 (<xs:list>) は基になる単純型の配列に変換されます (0)。AttributeType 列挙には、配列の個別の型が含まれます。

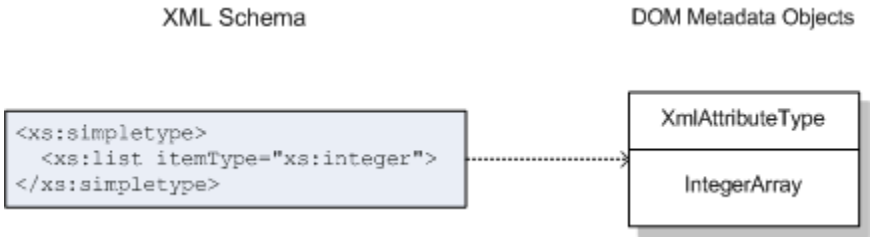

図 11: リストを有する単純型の変換

maxExclusive、minInclusive、enumeration などの制限ファセット付きの単純型は、適切な単純型の XmlAttributeType オブジェクトに変換されます。また、XML スキーマ型ローダは、DOM データ検証規則を作成して、制限規則を管理します (図 12:)。DataValidator クラスを型ローダに含めた場合、単純型の値が変化するとデータ検証が自動的に行われます。DOM 検証の詳細は、「DOM データの検証」を参照してください。

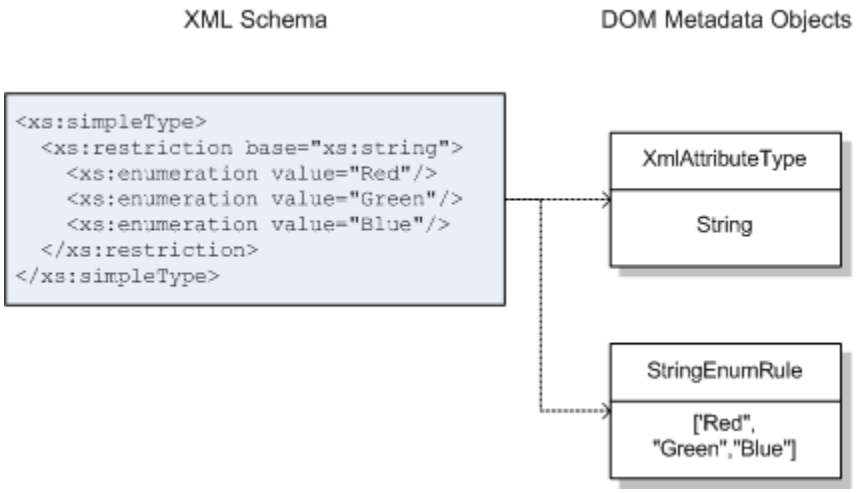

図 12: 制限規則付きの単純型の変換

複合型

複合型は、属性や他の子要素を持つ要素を記述します。一般的なデータモデルでは、複合型はエンティティを記述し、その属性はそのエンティティの属性であり、その複合型の子要素はその型と他の型との関係を定義します。

また、複合型は基底型を持つ場合もあり、これにより他の型を拡張する複合型を作成できます。XML スキーマにおけるこの形式の型継承は、スキーマ内にモジュール方式で再利用可能なコンポーネントを作成可能にするだけでなく、使用する型に応じた DOM アダプタを作成する際にも重要です。スキーマ内の派生した型により、アプリケーション内の DOM アダプタに対してクラス継承を活用できます。

XML 型ローダが XML スキーマを読み込む際に、複合型定義は DomNodeType オブジェクトに変換されます。型が定義した属性と子メタデータのすべてはオブジェクトに含まれます (0)。

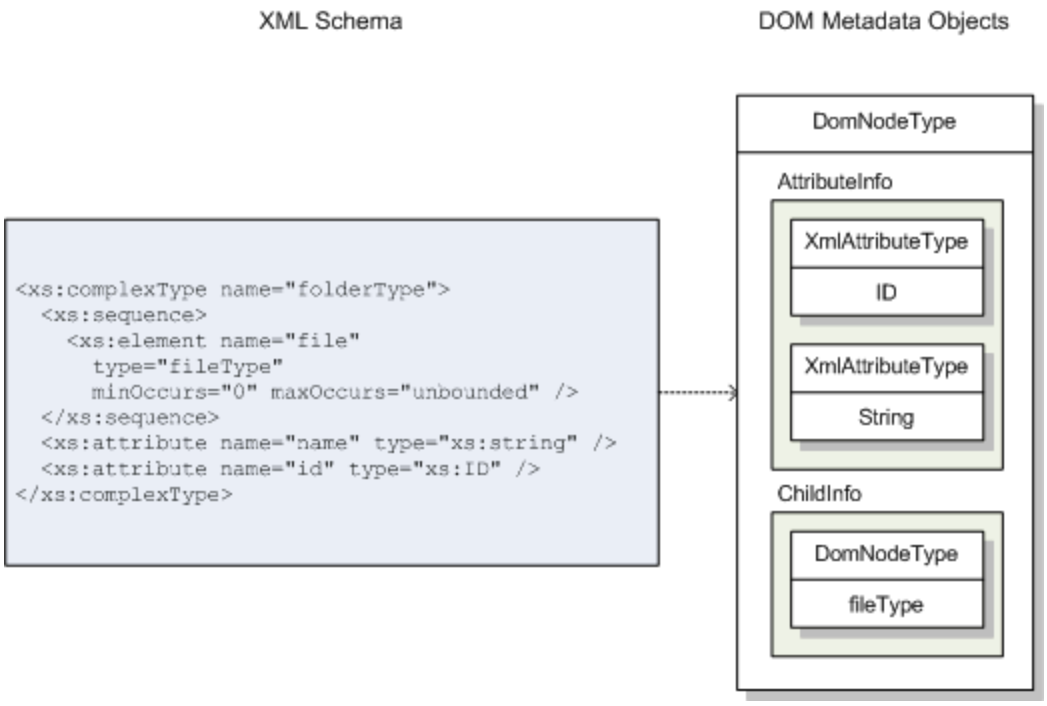

図 13: XML 型ローダによる複合型の変換

DomNodeType オブジェクトには、自身の属性 (AttributeInfo)、子 (ChildInfo)、ID 属性 (idAttribute)、拡張 (ExtensionInfo) など、スキーマにより定義されたあるいは型ローダにより追加された複合型の特性が含まれます。

継承

スキーマ内に型拡張を作成することによって (<xs:extension>) 型継承を実装します。複合型定義は基底型を持つことが可能ですが、その基底型を拡張し、属性、要素、またはその両方を追加します。

```
<xs:complexType name="subCircuitInstanceType">
    <xs:complexContent>
      <xs:extension base="moduleType">
        <xs:attribute name="type" type="xs:IDREF" use="required" />
      </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

要素は、ブール抽象属性を使用して宣言できます。抽象的な要素は、次のようにサブの型にして使用する必要があります。

```
<xs:complexType name="moduleType" abstract="true">
    <xs:attribute name="name" type="xs:ID" use="required" />
```

```
    <xs:attribute name="label" type="xs:string" />
    <xs:attribute name="x" type="xs:int" use="required" />
    <xs:attribute name="y" type="xs:int" use="required" />
</xs:complexType>
```

スキーマにおけるこの形式の型継承は、スキーマ内にモジュール方式で再利用可能なコンポーネントを作成可能にするだけでなく、使用するスキーマ型に応じた DOM アダプタを設計する際にも重要です。DOM アダプタの詳細は、「DOM ノード、拡張、およびアダプタ」を参照してください。

ID 属性

型に一意の識別子があることマークするには、次のように xs:ID 単純型を持つ属性を指定します。

```
<xs:complexType name="stateType">
    <xs:attribute name="name" type="xs:ID" use="required" />
    <xs:attribute name="label" type="xs:string" />
    <xs:attribute name="x" type="xs:int" use="required" />
    <xs:attribute name="y" type="xs:int" use="required" />
</xs:complexType>
```

この例では、値が「name」である属性が、stateType 複合型の識別子として機能します。この型属性の場合、「name」属性は「label」属性とは異なることに注意してください。name (ID) は内部識別子であり、label はオブジェクトの表示名として使用されます。

ID 属性の一意性を自分で管理する必要はありません。ATF には、UniqueIdValidator と UniquePathIdValidator という 2 つの ID 検証が含まれています。これにより、特定の型のオブジェクトの ID および DOM ノードツリー内のパスが一意であることが保証されます。これらの検証のいずれかを拡張としてスキーマローダに登録します。詳細は、「スキーマ型ローダの実装」を参照してください。

親要素と子要素

XML スキーマの要素間の関係は、一般的に親子関係として定義されます。複合型要素の定義には他の要素を含めることができます。すべての要素 (親と子の両方) は、スキーマがスキーマ型ローダによって読み込まれるときに DomNodeType オブジェクトに変換されます。

他のデータ型すべてを含む単一のルート要素、または複数のルート要素を使用してスキーマを設計できます。型ローダがスキーマをインポートするとき、スキーマのルート要素の DomNodeType を保持するために DomNodeType.BaseOfAllTypes プロパティが定義されます。データモデルに複数のルートを使用する場合、すべての型が Sce.Atf.Dom.Object を基底とします。BaseOfAllTypes プロパティは、すべての型の汎用 DOM アダプタ (プロパティ記述子など) をデータモデルに登録するのに役立ちます。

次のように maxOccurs と minOccurs 制限ファセットを使用して、要素が含む子要素の数を指定できます。

```
<xs:complexType name ="circuitType">
    <xs:sequence>
      <xs:element name="module" type="moduleType"
          minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="connection" type="connectionType"
          minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
```

これらの制限はスキーマ型ローダにより DOM データ検証規則 (ChildCountRule) に変換され、子 DOM ノードが親ノードに追加または削除されると検証が行われます。DOM 検証の詳細は、「DOM データの検証」を参照してください。

参照

ATF は XML スキーマを使用して、次の 2 種類の参照を定義します。

- xs:IDREF 型の属性に含まれる、識別子がある (つまり、xs:ID 属性を含む) 他の要素への参照。識別子の詳細は、「識別子」を参照してください。
- xs:anyURI 型の属性に含まれる、現在の文書の外部 (アセットまたはサブドキュメント) にあるリソースへの参照。

ATF には、参照を管理するための検証拡張、ReferenceValidator が含まれています。この拡張は、内部参照と外部参照を検証し、外部参照が解決される (読み込み) または解決されない (解放) ときにイベントを発生させます。この検証をスキーマの参照を使用する型に登録します。詳細は、「スキーマ型ローダの実装」を参照してください。

注釈

XML スキーマのすべてのコンポーネント (単純型、複合型、要素、および属性) には、カスタムプロパティで注釈を付けることができます。注釈は、ATF アプリケーションで使用したり、XML スキーマ自体では指定されない情報を追加します。

注釈は、次のように標準の <xs:annotation> タグと <xs:appinfo> タグで作成されます。

```
<xs:annotation>
    <xs:appinfo>
        <scea.dom.editors menuText="Marker" description="Marker"
            image="TimelineEditorSample.Resources.marker.png"
            category="Timelines" />
        <scea.dom.editors.attribute name="name"
            displayName="Name" description="Name" />
        <scea.dom.editors.attribute name="color"
            displayName="Color" description="Display Color"
            editor="Sce.Atf.Controls.PropertyEditing.ColorEditor"
```

```
            converter="Sce.Atf.IntColorConverter" />
    </xs:appinfo>
</xs:annotation>
```

以前のバージョンの ATF には、特定の名前とパラメータが付いた定義済みの注釈セットが含まれていました。ATF 3 では、任意の形式の注釈を付けるか、またはまったく注釈を付けません。

XML スキーマがスキーマ型ローダによって読み込まれるとき、注釈はそのまま読み込まれます。XML スキーマの注釈を解析し解釈するには、独自のスキーマローダに ParseAnnotations () メソッドを実装する必要があります。ATF 2 で使用されていた旧式の、プロパティ記述子およびパレット用注釈の一部は、後方互換性を保つために ATF3 でもサポートされますが、ParseAnnotations() メソッドで明示的に読み込む必要があります。詳細は、「スキーマ型ローダーの実装」を参照してください。

DomGen を使用してスキーマスタブクラスを生成する

DomGen は XML スキーマファイルを入力とし、スキーマスタブクラスを生成するユーティリティです。このスタブクラスには、DOM メタデータに直接アクセスするショートカットとしてスキーマ型ローダおよび DOM 拡張実装で使用するクラスとプロパティ定義のセットが含まれます。

DomGen について

スキーマ型ローダは、XML スキーマをアプリケーションに読み込む際に、独自の型定義をすべて内部 DOM メタデータオブジェクトに変換します。これらのオブジェクトは型ローダ自体が内部的に格納します。

DOM アダプタを作成するとき (実装時) およびアダプタを DOM ノード型に登録するとき (実行時) に、その DOM メタデータにアクセスする必要があります。XmlSchemaTypeCollection クラスにより、GetNodeType()、GetAttributeInfo()、および GetChildInfo() などのメソッドを介したそのメタデータへのアクセスが提供されます。これらのメソッドは、スキーマ内でその型、属性、または要素の名前を表す文字列引数で呼び出されます。次に例を示します。

```
DomNodeType keyType = typeCollection.GetNodeType("keyType");
```

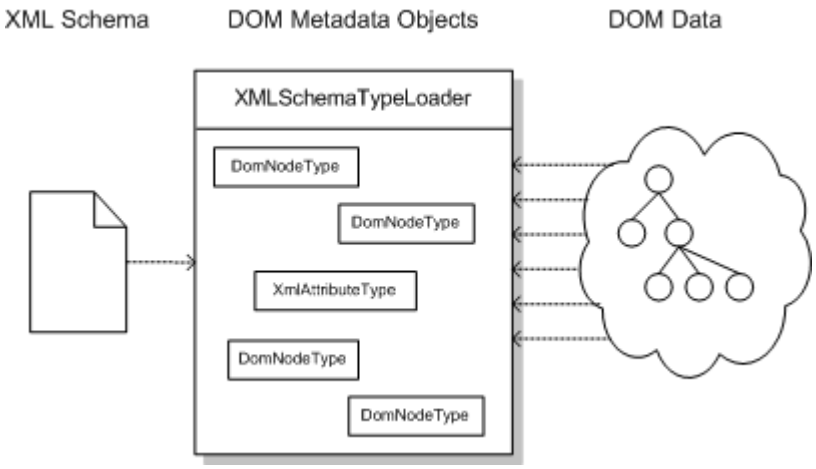

図 14: XML スキーマの読み込み

しかし、XmlSchemaTypeCollection メソッドを介してメタデータに直接アクセスする場合、次の 2 つの問題点があります。

- スキーマメタデータへの直接アクセスは、特に型、属性、および要素名の観点から、コードをスキーマに強く結び付けます。これらの名前は文字列であり、コンパイル時に確認されず、アプリケーションを実行するまで不一致またはスペルミスが判明しない場合があります。スキーマの型名の変更は、コード全体で複数ファイルの変更が必要になる場合があります。
- XML スキーマに数多くの型を定義している場合、XmlSchemaTypeCollection メソッドを使いやすくするためにラップするプロパティのテーブルを自分で作る必要があります。スキーマ内の型を変更する場合、これは非常に煩雑で繰り返しが多く、管理が困難になります。

DomGen は、スキーマを入力とし、スキーマと残りのコードとの間にクラスレイヤーを生成することにより、これらの問題を解決します。このレイヤーは、内部クラスおよびプロパティを多数含むクラス、Schema を定義します。内部クラスおよびプロパティの名前は各スキーマ型、属性、および要素に由来し、それぞれが XmlSchemaTypeCollection の適切な値に初期化されます。このクラスはスキーマスタブクラスと呼ばれます。

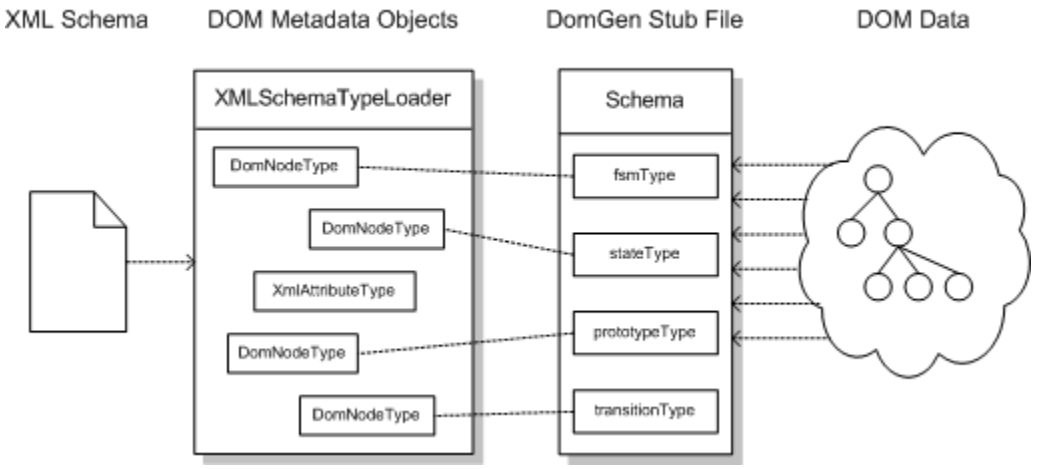

図 15: DomGen により生成される Schema クラス

DomGen の実行

アプリケーションをビルドするとき、およびスキーマ内の型、属性、または要素を変更するたびに、DomGen を実行します。スキーマ内で、いずれかの名前に変更があった場合も DomGen を再実行する必要があります。DomGen が生成する情報には、注釈ファセットまたは制限ファセットは含まれないことに注意してください。スキーマ型ローダは実行時にその情報を解析します。その情報はいつでも変更できます。アプリケーションのデータモデルとスキーマが安定している場合は、DomGen を何度も実行する必要はありません。

DomGen は、ATF に付属するコマンドラインツールで、ATF/DevTools ディレクトリにあります。DomGen は次のように複数の方法で実行できます。

- Windows コマンドプロンプトから
- Visual Studio 内から

コマンドプロンプトから DomGen を実行する

1. ATF/DevTools/DomGen の DomGen ユーティリティをコンパイルします (デバッグまたはリリース)。

   DomGen は、Debug または Release モードのいずれかで使用できます。

2. Windows コマンドプロンプトを起動し、スキーマファイルのある場所に移動します (例: C:¥SceaDev¥SCEA_WIP¥ATF¥Samples¥TimelineEditor¥schemas)。

3. コマンドプロンプトで、DomGen 実行ファイルのフルパスをコピーまたは入力します (例: C:¥SceaDev¥SCEA_WIP¥ATF¥DevTools¥DomGen¥bin¥Debug¥DomGen.exe)。

4. スキーマファイルの DomGen オプションを追加します。

   DomGen には次の 4 つのオプションが必須です。

   - スキーマファイルの名前。「timeline.xsd」など。
   - 生成するスキーマスタブクラスの名前。便宜上、このクラスを「Schema.cs」と呼びます。
   - スキーマ内の型の XML ターゲット名前空間。「timeline」、「http://sce/timeline」など。
   - 生成されたスキーマスタブクラスの C# 名前空間。「TimelineEditorSample」、「Sce.Atf.Samples」など。

5. リターンキーを押して DomGen コマンドを実行します。

   Schema.cs ファイル（または、ファイル名オプションに指定した名前）がスキーマディレクトリに生成されます。ファイルの場所は、スキーマディレクトリのままでも、都合のよい場所に移動してもかまいません。

   0 に、Timeline Editor での DomGen コマンドの使用を示します。

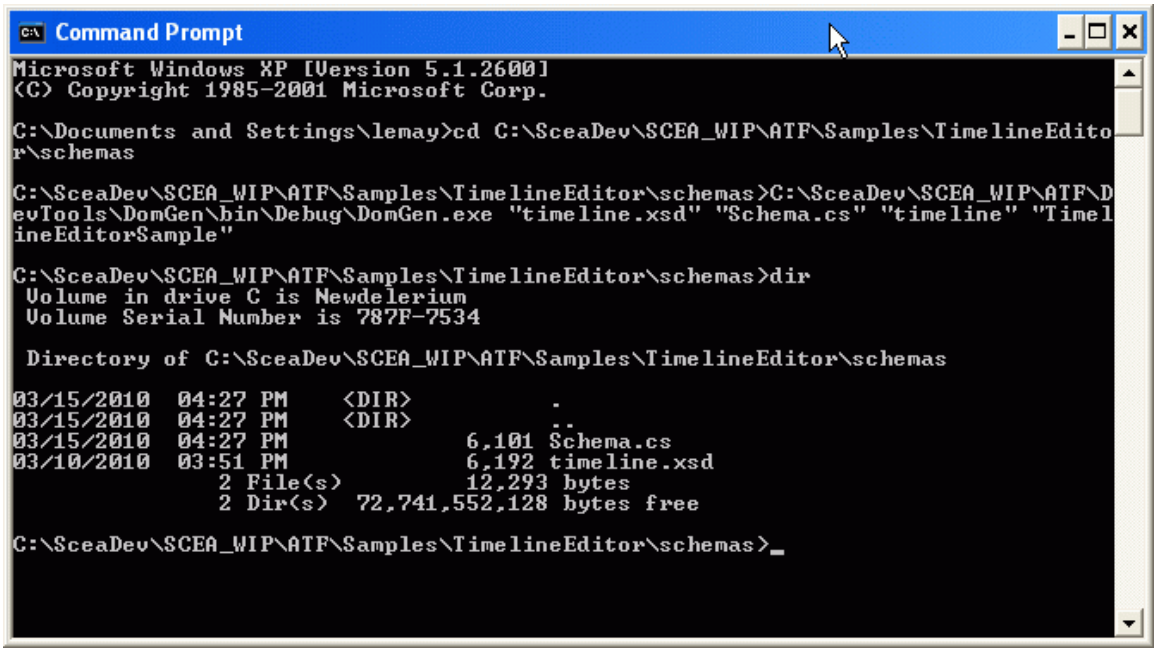

図 16: Windows コマンドプロンプトから DomGen を実行する

6. Visual Studio の［プロジェクト］>［既存項目の追加］コマンドで、Schema.cs ファイルを ATF プロジェクトに追加します。

Visual Studio 内から DomGen を実行する

Visual Studio 内から直接 DomGen を実行できます。この場合、入力が少なくて済みます。

1. [プロジェクト] > [DomGen のプロパティ] をクリックして、DomGen プロジェクトのプロパティを編集します。

   また、DomGen プロジェクトを右クリックして、プロパティを編集することもできます。

2. ビルドタイプに応じて、プロパティの [Debug] ページまたは [Release] ページを選択します。

   DomGen は、Debug または Release モードのいずれかで使用できます。

3. [作業ディレクトリ] をスキーマファイルを含むディレクトリに変更します。

4. スキーマファイルの DomGen オプションを [コマンドライン引数] に追加します。DomGen プログラムの名前は含めないでください。

   DomGen には次の 4 つのオプションが必須です。

   - スキーマファイルの名前。「timeline.xsd」など。
   - 生成するスキーマスタブクラスの名前。便宜上、このクラスを「Schema.cs」と呼びます。
   - スキーマ内の型の XML ターゲット名前空間。「timeline」、「http://sce/timeline」など。
   - 生成されたスキーマスタブクラスの C# 名前空間。「TimelineEditorSample」、「Sce.Atf.Samples」など。

   0 に、プロジェクトのプロパティページへの Timeline Editor の入力例を示します。

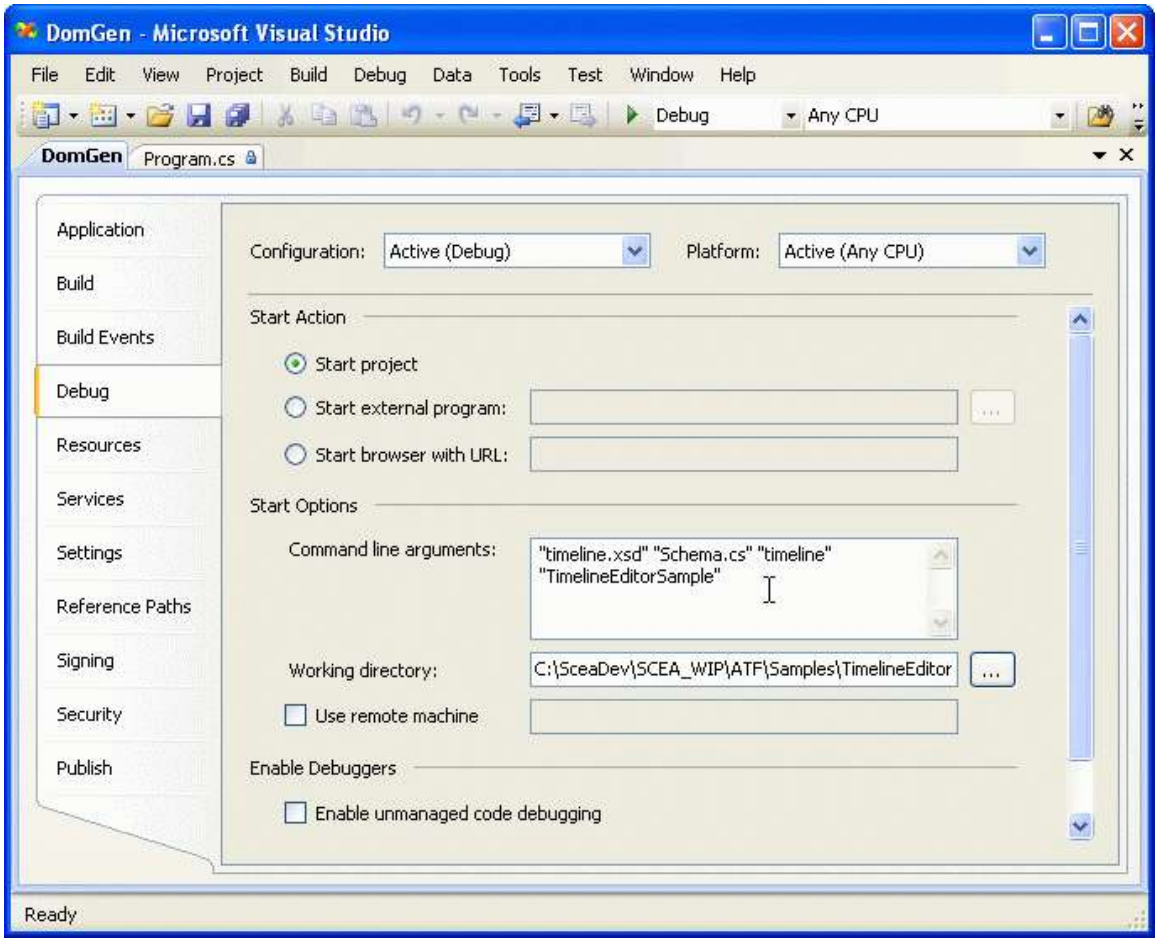

図 17: Visual Studio 内から DomGen を実行する

5. DomGen プロジェクトを保存してビルドします。

   Schema.cs ファイル（または、ファイル名オプションに指定した名前）がスキーマディレクトリに生成されます。ファイルの場所は、スキーマディレクトリのままでも、都合のよい場所に移動してもかまいません。

6. Visual Studio の［プロジェクト］>［既存項目の追加］コマンドで、Schema.cs ファイルを ATF プロジェクトに追加します。

スキーマ型ローダの実装

スキーマ型ローダは、実行時にスキーマにより定義された型のアプリケーションへの取り込み、これらの型のメタデータの定義、DOM 拡張およびアダプタの登録、注釈の処理（存在する場合）を行います。特定のスキーマのスキーマ型ローダクラスを作成するには、

XmlSchemaTypeLoader クラスから派生し、OnSchemaSetLoaded() メソッドと ParseAnnotations() メソッドを実装します。

XmlSchemaTypeLoader 基底クラス

XmlSchemaTypeLoader 基底クラスから派生してスキーマ型ローダを実装します。このクラスは、次のような XML スキーマの読み込みと管理に関する多くの作業を提供します。

- ローカルファイルから 1 つ以上のスキーマを読み込みます。これには、他のインポートされたスキーマ文書など外部リソースへの参照の解決が含まれます。
- スキーマ内の型、属性、要素の内部 DOM メタデータオブジェクト (DomNodeType、XmlAttributeType) を作成し、これらのオブジェクトを XmlSchemaTypeCollection オブジェクトに格納します。
- スキーマが定義する注釈のコレクション (IDictionary) を作成します。
- xs:ID を定義する型に対して ID 属性を定義します。
- スキーマにより定義された制限ファセットに対し、DOM 検証規則を作成します。DOM 検証の詳細は、「DOM データの検証」を参照してください。

注意: XmlSchemaTypeLoader クラスは .NET スキーマオブジェクトモデル (SOM) を使用して、XML スキーマを読み込み解決します。System.XML 参照をプロジェクトに含める必要があります。また、「using」C# キーワードを持つ System.Xml 名前空間と System.Xml.Schema 名前空間を含める必要もあります。.NET XML フレームワークの詳細は、.NET Framework Developer's Guide の XML スキーマ オブジェクト モデル (SOM) セクションを参照してください。

スキーマ型ローダは、この基底クラスを拡張し、特定のスキーマに必要な振る舞いを追加します。規則により、スキーマ型ローダクラスは、一般的に SchemaLoader またはその名前のバリエーションで呼ばれます。

派生したスキーマローダクラスにおいて、次の 2 つのメソッドをオーバーライドします。

- OnSchemaSetLoaded(): スキーマ自体が読み込まれ、XmlSchemaTypeCollection が利用可能になり、DOM メタデータオブジェクトが作成された後で実行する振る舞い。このメソッドでは、DomGen が生成したスキーマスタブファイルを結び付け、DOM 拡張またはアダプタを登録します。
- ParseAnnotations(): スキーマが定義した注釈を処理して、プロパティ記述子、パレットデータ、ツールヒント、その他の情報を作成します。

これらのメソッドの詳細は、次のセクションを参照してください。

スキーマ型ローダは、次のような一般的に使用されるプロパティを定義する場合があります。

- TypeCollection プロパティ。(スキーマメタデータを格納する) XmlSchemaTypeCollection を保持します。
- Namespace プロパティ。(XmlSchemaTypeCollection から取得する) 現在のスキーマの名前空間を保持します。

カスタムスキーマ型ローダの例は、ATF サンプル (ATF/Sample) を参照してください。

OnSchemaSetLoaded() メソッド

特定の XML スキーマの振る舞いを追加するには OnSchemaSetLoaded() メソッドをオーバーライドします。OnSchemaSetLoaded() メソッドは、スキーマが読み込まれて解決され、各型の DOM メタデータオブジェクトが作成された後に、XmlSchemaTypeLoader クラスから呼び出されます。このメソッドにはデフォルトの振る舞いはありません。

このメソッドに実装するタスクを次に示します。

- TypeCollection と Namespace プロパティの型コレクションと名前空間を取得します (定義されている場合)。
- Schema.Initialize() を呼出し、DomGen によって生成されたスキーマスタブクラス定義を結び付けます。
- DomGen から使用可能な型に DOM 拡張を登録します。
- ID 検証や参照検証などのユーティリティアダプタを登録します。

OnSchemaSetLoaded() のサンプルを次に示します。

```
protected override void OnSchemaSetLoaded(XmlSchemaSet schemaSet)
{
    foreach (XmlSchemaTypeCollection typeCollection
        in GetTypeCollections())
    {
        // Initialize fields for use by properties
        m_namespace = typeCollection.TargetNamespace;
        m_typeCollection = typeCollection;

        // Initialize DomGen types
        Schema.Initialize(typeCollection);

        // register extensions
        Schema.fsmType.Type.Define(new ExtensionInfo<Fsm>());
        Schema.fsmType.Type.Define(new ExtensionInfo<FsmContext>());
        Schema.fsmType.Type.Define(new ExtensionInfo<FsmDocument>());
        Schema.fsmType.Type.Define(new
            ExtensionInfo<ReferenceValidator>());
```

```
            Schema.fsmType.Type.Define(new
                ExtensionInfo<UniqueIdValidator>());
            // ...
            Schema.prototypeFolderType.Type.Define(new
                ExtensionInfo<PrototypeFolder>());
            Schema.prototypeType.Type.Define(new ExtensionInfo<Prototype>());

            break;
        }
    }
```

このメソッドの中核として機能している foreach ループに注意してください。スキーマとその XmlSchemaTypeCollection との間には 1 対 1 の対応関係があるため、データモデルにスキーマを 1 つしか使用しない場合、このループは不要に思われます。ループがある理由は、型コレクションが名前空間の名前で XmlSchemaTypeLoader 基底クラスによって登録されるためです。GetTypeCollections() メソッドとループにより、その名前空間名をコードにハードコーディングしたりスキーマへ自体への依存を作成することなく、型コレクションを取得することができます。

最初の 2 行は、SchemaLoader クラスにより定義された Namespace と TypeCollection プロパティが使用する内部フィールドを初期化します。

```
m_namespace = typeCollection.TargetNamespace;
m_typeCollection = typeCollection;
```

この行は、DomGen により生成されたスキーマスタブクラスを初期化します。

```
Schema.Initialize(typeCollection);
```

最後の一連の行はすべて、スキーマスタブクラスに定義された型に DOM アダプタを登録します。最初の数行は fsm 型、残りの 2 行は prototype 型です。DOM アダプタの作成と登録の詳細は、「DOM 拡張とアダプタ」を参照してください。

特に次の 2 つの DOM アダプタの登録に注意してください。

```
Schema.fsmType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.fsmType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
```

これらの DOM アダプタは、それぞれ参照および識別子のユーティリティ検証です。参照検証は、DOM ノード間の内部参照および外部参照の検証を管理します。ID 検証は、DOM ノードの ID が一意であることを確認します。この場合、どちらの検証もスキーマ内のどのサブ要素でも動作するように、スキーマ内のルート要素に登録されます。検証の詳細は、「DOM データの検証」を参照してください。

ParseAnnotations() メソッド

注釈メタデータに基づいてプロパティ記述子、パレットアイテム、またはツールヒントを作成するなど、スキーマに定義された注釈を処理するには、ParseAnnotations() を

オーバーライドします。ParseAnnotations() メソッドは、OnSchemaSetLoaded() の直後に XmlSchemaTypeLoader 基底クラスで呼び出されます。基底の ParseAnnotations() メソッドは、従来の idAttribute 注釈のみを管理します。他の注釈はすべてこのメソッド内でユーザが管理する必要があります。

> **注意:** 従来の ATF 2 注釈の一部、scea.dom.editors.attribute と scea.dom.editors.child は ATF 3 でサポートされます。注釈の解析およびプロパティ記述子の作成には PropertyDescriptor.ParseXml() メソッドを使用します。次の例および「プロパティ記述子の定義と使用」を参照してください。

この Timeline Editor からのサンプル ParseAnnotations() メソッドは、注釈を解析し、プロパティ記述子と DOM パレットアイテムを作成します。

```
protected override void ParseAnnotations(
    XmlSchemaSet schemaSet,
    IDictionary<NamedMetadata, IList<XmlNode>> annotations)
{
    base.ParseAnnotations(schemaSet, annotations);

    IList<XmlNode> xmlNodes;

    foreach (DomNodeType nodeType in m_typeCollection.GetNodeTypes())
    {
        // parse XML annotation for property descriptors
        if (annotations.TryGetValue(nodeType, out xmlNodes))
        {
            PropertyDescriptorCollection propertyDescriptors =
                Sce.Atf.Dom.PropertyDescriptor.ParseXml(
                    nodeType, xmlNodes);

            nodeType.SetTag<PropertyDescriptorCollection>(
                propertyDescriptors);

            // parse type annotation to create palette items
            XmlNode xmlNode = FindElement(xmlNodes, "scea.dom.editors");
            if (xmlNode != null)
            {
                string menuText = FindAttribute(xmlNode, "menuText");
                if (menuText != null) // must have menu text and category
                {
                    string description =
                        FindAttribute(xmlNode, "description");
                    string image = FindAttribute(xmlNode, "image");
                    NodeTypePaletteItem item =
                        new NodeTypePaletteItem(
                            nodeType, menuText, description, image);
                    nodeType.SetTag<NodeTypePaletteItem>(item);
                }
            }
        }
    }
}
```

XML スキーマの読み込み

XML スキーマ、スキーマスタブクラス、およびスキーマ型ローダクラスがすべてアプリケーションに実装されたら、最終の手順として、アプリケーションが実行時にスキーマ型ローダを呼び出すことを確認します。一般的に、これは主要なエディタクラス (例: TimelineEditor、FSMEditor) のコンストラクター内で行われます。

スキーマ型ローダを呼び出すには、SchemaLoader クラスのインスタンスを作成して、XML スキーマファイルのパスで Load() を呼び出します。

```
// load schema from debug location if available, otherwise
// assume that the schema is in release location
string schemaPath = "..\\..\\schemas\\Game.xsd";
if (!File.Exists(schemaPath))
    schemaPath = Path.Combine(Application.StartupPath,
    "schemas\\Game.xsd");

s_schemaLoader = new SchemaLoader();
s_schemaLoader.Load(schemaPath);
```

名前空間がそれぞれ一意であれば、アプリケーションに必要な数のスキーマを読み込むことができます。使用するスキーマはそれぞれ、そのターゲットの名前空間を使用して、XmlSchemaTypeLoader 基底クラスによって登録されます。個別の XmlSchemaTypeCollection がスキーマごとに作成されます。

DOM 内の XML スキーマ型ローダは .NET スキーマオブジェクトモデル (SOM) を使用して、XML スキーマを読み込み解決します。System.XML 参照をプロジェクトに含める必要があります。また、「using」C# キーワードを持つ System.Xml 名前空間と System.Xml.Schema 名前空間を含める必要もあります。.NET XML フレームワークの詳細は、.NET Framework Developer's Guide の XML スキーマ オブジェクト モデル (SOM) セクションを参照してください。

4. DOM ノードと DOM アダプタ

アプリケーションのランタイムデータは、DOM ノード (DomNode クラスのインスタンス) のツリーに格納されます。DOM 拡張とアダプタにより、アプリケーションとそのデータモデル (DOM ノードツリー内のデータ) 間のオブジェクト API が定義されます。DOM ノードツリーを使用するには、拡張とアダプタを実装して、それらをデータモデルの型ごとに登録します。アプリケーションは、これらの DOM 拡張を介して DOM ノードツリー内のデータにアクセスします。

本章は、次のセクションで構成されています。

- DOM ノードと DOM アダプタについて: 拡張とアダプタが DOM ノートおよび DOM ノードツリーとどのように連携するかについて説明します。また、拡張とアダプタを独自のデータ型に実装および登録する方法についても説明します。
- 主要なクラス: この章に記載された最も重要なクラスとインタフェースのまとめ。
- DOM ノードとアダプタの使用: DOM ノードと DOM アダプタを使用するための一般的なタスク。親ノードと子ノードの作成と削除、DOM アダプタ間の切り替えなどを含みます。
- DOM アダプタの実装: DomNodeAdapter 基底クラスから派生させて、独自の DOM アダプタを作成します。
- DOM 拡張とアダプタの登録: データモデルの型に登録することにより、DOM ノードへのオブジェクト拡張および実装する DOM アダプタを登録します。

DOM ノードと DOM アダプタについて

ATF DOM は次の 3 つのレイヤーで構成されます。

- データモデル。型定義ファイルに定義し、型ローダに読み込みます。
- DOM ノードツリー。ランタイムアプリケーションデータが含まれます。
- DOM 拡張とアダプタのセット。アプリケーションコードと DOM ノードツリー内のノードの間に C# API レイヤーを提供します。

前の章では、DOM のデータモデルの作成について説明しました。残り 2 つのレイヤー、DOM ノードツリーと DOM 拡張は緊密に連携しているため、この章で両レイヤーについて説明します。

DOM ノードと DOM ノードツリー

ランタイムアプリケーションデータは、DOM ノード (DomNode クラスのインスタンス) のツリーとして DOM に格納されます。各 DOM ノードには、基本メタデータ型 (DomNodeType のインスタンス) があります。DomNodeType メタデータオブジェクトは、「データモデリング、XML スキーマ、および DOM」に記載されているように、作成した型定義ファイルの情報に基づいて型ローダにより作成されます。DomNodeType メタデータオブジェクトは、その型のすべての DOM ノードで共有されます。

DomNode クラスと DomNodeType クラスは、そのツリー内でツリーとノードを管理するための基本的な機能を提供します。これには、豊富なイベント管理、一意なノードの命名、参照追跡、トランザクション、および基本ドキュメントとの同期が含まれます。

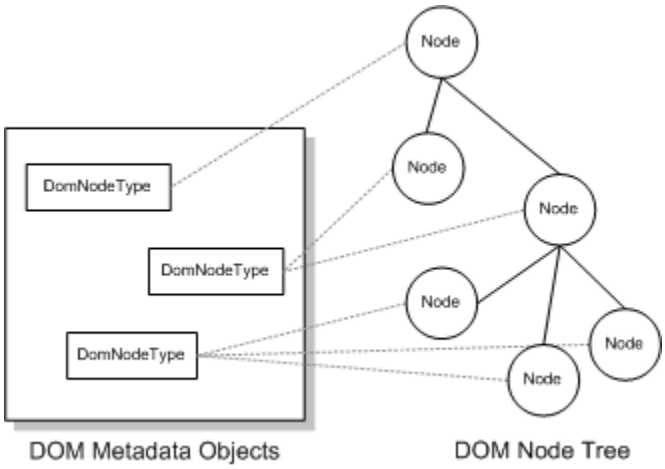

図 18: DOM ノードと DOM ノードツリー

DOM ノードと DOM ノードツリーの詳細は、「DOM ノードとアダプタの使用」を参照してください。

DOM 拡張と DOM アダプタ

アプリケーションは、DOM ノードツリー内のノードを直接操作しません。代わりに、DOM 拡張により、アプリケーションコードと DOM 内のノード間に柔軟な適応レイヤーが提供されます。適応レイヤーを介して、アプリケーションはノードの作成と削除、ノードプロパティの編集、および子ノードの追加と削除ができます。

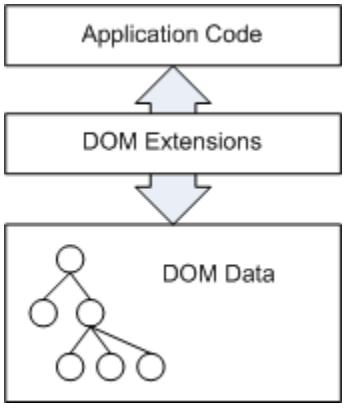

図 19: DOM 拡張、DOM データ、およびアプリケーション

DOM 拡張は任意の型のオブジェクトです。しかし多くの場合は、DomNodeAdapter 基底クラスから派生した DOM 拡張クラスを作成し、データモデルの型にオブジェクト API を提供します。DomNodeAdapter クラスから派生する拡張は DOM アダプタと呼ばれます。

DOM アダプタの実装

DomNodeAdapter 基底クラスから派生して DOM アダプタを実装します。このクラスは、アダプタをその基本 DOM ノードに関連付けるためのインフラおよび多くの有益な共通メソッドとプロパティを提供するインフラを提供します。DOM アダプタの実装で可能なことを次に示します。

- DOM ノード内のデータに API を提供します (プロパティ内の属性と要素のラップなど)。
- アダプタが作成されるとき、または基本 DOM ノードに関連付けられるときに特定の初期化振る舞いを提供します。
- 子の型を含め、この型と関連する型に対して DOM ノードツリーのデータの変更を管理します。
- コンテキストインタフェースを追加して DOM 型全体に共通の振る舞いを実装します。これらの実装が容易なインタフェースにより、基本的な共通の DOM ノードの振る舞いが提供されます。たとえば、INameable インタフェースにより、DOM ノードに命名できます (パレットでの収集を容易にします)。

DOM アダプタの作成の詳細は、「DOM アダプタの実装」を参照してください。

DOM アダプタの登録

DOM 拡張とアダプタはいずれも、実行時に処理対象となる型に登録されます。データモデルの型ローダ内に DOM 拡張とアダプタを登録します。

一般的に、データモデル内の型ごとに 1 つの DOM アダプタを実装し登録して、型とそのアダプタを 1 対 1 で対応させます。ただし、登録されたアダプタは特定の型とそのサブ型すべてに自動的に適用され、型の継承が利用可能になります。また、複数のアダプタを同じ型に登録できるので、異なるインタフェースまたはビューを DOM ノードツリー内の同じデータに提供できます。

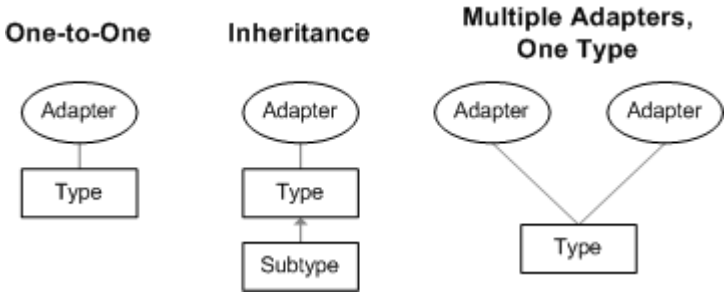

図 20: アダプタ登録

拡張とアダプタの登録の詳細は、「DOM 拡張とアダプタの登録」を参照してください。

DOM アダプタのライフサイクル

DOM アダプタインスタンスは、DOM ノードが作成されるときに、その DOM ノードのコンストラクター内で自動的に作成されます。また、DOM アダプタは OnNodeSet() メソッドにより初期化されます。このメソッドは、アダプタがその基本ノードに関連付けられるたびに呼び出されます。ノードには複数のアダプタがある場合があり、ノードアダプタはいつでも変更できるため、OnNodeSet() はアプリケーションの実行時に何度も呼び出される可能性があります。

アダプタインスタンスは、参照先のノードが削除されると破棄されます。

DOM ノードのアダプタの設定

DOM ノードは、そのノード上で動作する DOM アダプタを複数持つことができるので、As<T>()、Is<T>()、および Cast<T>() メソッドを利用して、あるアダプタから別のアダプタに DOM ノードを「変換」できます (T は新しいアダプタ)。これらのメソッドは、C# の同等メソッドと同じように動作します。DOM ノードがあるアダプタから別のアダプタに「変換」されるとき、ノードとその型は変わらず、そのノードの現在のアダプタのみが変更されることに注意してください。

ノードがそのアダプタを変更するたびに、そのアダプタの OnNodeSet() メソッドが呼び出されます。

ワークフロー

次の表に、DOM ノードとアダプタを操作するための実装時および実行時の手順と、本書内での参照先を示します。

実装時

表 5: DOM アダプタのワークフロー: 実装時

手順	説明	参照先
1. データモデルの型に対して DOM 拡張クラスを実装	DOM 拡張とアダプタは、DOM ノードツリー内の型とデータをアプリケーション自体の間を取り持つ API を提供します。	DOM アダプタの実装

実行時

表 6: DOM アダプタのワークフロー: 実行時

手順	説明	参照先
1. DOM 拡張とアダプタを登録	DOM 拡張とアダプタは、処理対象となる型に対して登録する必要があります。DOM 拡張は型ローダで登録します。	DOM 拡張とアダプタの登録
2. 必要に応じて DOM ノードを作成および初期化	DomNode の新しいインスタンスを作成し、As<T>() メソッドを使用して DOM アダプタをそのノードに関連付けます。 ドキュメントを読み込むときなど、DOM ノードの大きなツリーを一度に初期化するには、InitializeExtensions() を使用します。	DOM ノードの作成と初期化

主要なクラス

Sce.Atf.Dom 名前空間には、DOM 拡張とアダプタを作成および使用するためのクラスが数多く含まれています。表 7: にこれらのクラスの概要を示します。

表 7: Sce.Atf.Dom に存在する、DOM 拡張とアダプタに用の主要なクラス

クラス	説明
DomNode	DOM データツリー内のノードを記述します。各 DOM ノードには、基本メタデータ型 (DomNodeType) があります。
DomNodeAdapter	DOM アダプタの基底クラスです。
DomNodeType	DOM ノードの基本型を記述します。DomNodeType メタデータオブジェクトは、型ローダによって作成されます。NamedMetadata を拡張します。DomNodeType の詳細は「DOM メタデータオブジェクト」を参照してください。
DomNodeListAdapter<T>	DOM ノードの子データに関連付けられているアダプタのリストを含むラッパークラスです。IList<T> から派生しています。
TypeAdapterCreator	DOM ノード型から DOM ノードアダプタを生成するための内部クラスです。IAdapterCreator を実装します。

DOM ノードとアダプタの使用

アプリケーションデータは、個々の DOM ノード (DomNode クラスのインスタンス) のツリーとして DOM に格納されます。各 DOM ノードには、基本 DOM メタデータオブジェクト (DomNodeType) があります。また、各型には、その型用の DOM アダプタが 1 つまたは複数登録されています。アプリケーションは、それらのアダプタを (一度に 1 つ) 介して、DOM ノードツリー内にあるデータにアクセスできます。

このセクションでは、DOM アダプタを使用して DOM ノードツリー内の DOM ノードを操作する方法を説明します。説明では、各 DOM ノードには、DOM 拡張またはアダプタが登録されているものと想定します。新しい型に対する DOM アダプタの実装および登録方法の詳細は、「DOM アダプタの実装」と「DOM 拡張とアダプタの登録」を参照してください。

DOM ノードの作成と初期化

DOM ノードツリーに新しいノードを作成して初期化する手順を次に示します。

- ノードまたはノードのリストを作成します
- 拡張またはアダプタをそのノードに関連付けます

ノードのツリー全体を一度に作成する場合（ドキュメントを読み取る場合など）は、それらのノードの拡張を明示的に初期化することも必要となります。

DOM ノードの作成

DomNode コンストラクタで新しいノードを作成します。

```
DomNode node = new DomNode(Schema.prototypeFolderType.Type)
```

デフォルトの DomNode コンストラクタは、DomNodeType オブジェクトを引数にします。この例では、DomGen で生成したスキーマスタブクラスに記述したメタデータオブジェクトを使用しています。

このノードが他のノードの子である場合、DomNode コンストラクタ内で ChildInfo メタデータオブジェクトを追加指定することもできます。この ChildInfo オブジェクトは、このノードの親を示します。

```
DomNode root = new DomNode(Schema.fsmType.Type, Schema.fsmRootElement);
```

拡張をノードに関連付ける

DOM 拡張とアダプタのインスタンスは、DOM ノードの作成時に自動的に作成されます。同じ型に対して複数の DOM アダプタが登録されている場合や、この型に対して登録されているアダプタが別のアダプタから継承している場合、それらのアダプタもすべて作成されます。アダプタのインスタンスは、初期化処理の一部として DOM ノードに関連付けられます。

ただし、DOM アダプタが最初に使用され、そのアダプタに対して OnNodeSet() メソッドが呼び出されるまでは、DOM アダプタの初期化は完了しません。これは通常、As<T>() メソッドが呼び出され、アダプタがノードに関連付けられるときに実行されます。たとえば、次のコードは、prototype に新しい DOM ノードを作成してから、そのノードへの Prototype DOM アダプタを初期化します。

```
DomNode node = new DomNode(Schema.prototypeType.Type);
Prototype prototype = node.As<Prototype>();
```

多くの場合、ノードの作成と対応するアダプタの関連付けは同時に行われます。

```
Pin pin = new DomNode(Schema.pinType.Type).As<Pin>();
```

拡張のツリーの初期化

DOM 拡張とアダプタには、バリデータなどのように直接使用されないものがあるため、DOM ノードツリー全体またはそのサブツリーの拡張を明示的に初期化する必要がある場合があります。これには、ドキュメントを開き、ドキュメント内のデータに対して DOM

ノードツリーを作成する場合などが挙げられます。DOM ノードのツリー全体のアダプタを初期化するには、DomNode.InitializeExtensions() メソッドを使用します。

```
node.InitializeExtensions();
```

このメソッドを呼び出す場所は、ドキュメントクライアントクラス (通常は Editor クラス) の Open() メソッドの中が一般的です。DOM ノードの新しいツリーが作成されるのは多くの場合、ドキュメントを開いたり作成するときです。

子の DOM アダプタのリストの取得

DOM ノードは、子ノードのリストの親であることがよくあります。たとえば、フォルダ型には、多くのファイル型とサブフォルダ型が含まれている場合があります。ATF は DomNodeListAdapter<T> クラスを提供しており、毎回ループ処理を作成し、それらの子を反復処理して DOM アダプタをそれぞれに関連付けなくても済むようになっています。このクラスは DOM アダプタのリストのラッパーで、DOM ノードの基本子リストに関連付けられます。

DomNodeListAdapter を作成するには、子要素の DomNode および DOM メタデータオブジェクト (DomNodeType) を使用します。

```
m_states = new DomNodeListAdapter<State>(DomNode,
    Schema.fsmType.stateChild);
```

DomNodeListAdapter<T> クラスは IList<T> から派生し、他のリストと同様に使用できます。

DOM ノードアダプタの設定

多くの場合、DOM ノードの型には、多くの DOM 拡張とアダプタが登録されています。一つの型に DOM 拡張が複数あることで、同じ基本 DOM データにさまざまな API を提供できます。ただし、DOM ノードに一度に使用できる DOM アダプタは 1 つだけです。ノードとアダプタを一緒に使用するために、アダプタをノードに関連付けまたは設定します。DOM ノード上で別のアダプタが提供するインタフェースを使用するには、ノードをそのアダプタに設定します。

0 では、各 DOM アダプタ (丸ボックス) が fsm 型に対して登録され、その型を表す DOM ノードで使用可能になっていますが、そのノードには現在、FSM アダプタが関連付けられています。異なるアダプタに切り替えると、それらの API を介してノードを使用できます。

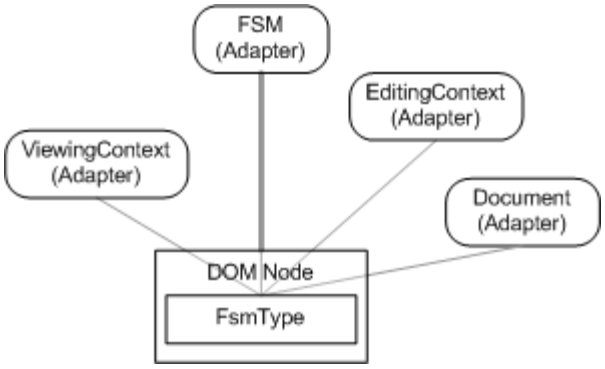

図 21: DOM アダプタと DOM ノード

DOM アダプタと DOM ノードの互換性をテストしたり、DOM アダプタを DOM ノードに接続するには、As<T>()、Is<T>()、Cast<T>() メソッドを使用します。これらのメソッドは、C# の同等のメソッドと同じように動作し、DOM ノードと DOM アダプタの型「変換」を実行します。これらのメソッドには、DomNode クラスのメソッドと、Sce.Atf.Adaptation.Adapters クラスの静的メソッドの 2 つがあります。

各メソッドのパラメータはジェネリック型パラメータ T で、これは、DOM アダプタの型またはコンテキストインタフェースを示します。静的アダプタのバージョンの引数は、テストするオブジェクトです。

DOM 拡張またはアダプタの型が DOM ノードと互換性があるかどうかをテストするには、Is<T>() を使用します。

```
bool INamingContext.CanSetName(object item)
{
    return
        Adapters.Is<Prototype>(item) ||
        Adapters.Is<PrototypeFolder>(item);
}
```

Is<T>() と同じテストを実行して、DOM アダプタを DOM ノードに関連付けるには、As<T>() を使用します。As<T>() メソッドは DOM アダプタに OnNodeSet() メソッドを呼び出し、DOM アダプタインスタンスを返します。

```
Prototype prototype = node.As<Prototype>();
```

Cast<T>() メソッドの機能は、As<T>() とほぼ同一です。As<T>() は、DOM アダプタが見つからない場合、NULL を返します。Cast<T>() メソッドは、AdaptationException 例外を渡します。

コンテキストインタフェースを使用すると、As<T>() を使用してインタフェースごとにアダプタを選択できます。たとえば、FSM サンプルの EditorContext クラスにある次の Center() メソッドは、現在の編集コンテキスト (DOM ノードのツリー) が、そのツリー

のルートに対して As<ILayoutContext>() を呼び出すことで ILayoutContext が実装されるようにしています。

```
public void Center(IEnumerable<object> items, Point p)
{
    ILayoutContext layoutContext = this.As<ILayoutContext>();
    if (layoutContext != null)
    {
        // get bounds, convert to world coords
        Rectangle bounds;
        LayoutContexts.GetBounds(layoutContext, items, out bounds);
        Matrix transform =
            m_viewingContext.Control.As<ITransformAdapter>().Transform;
        p = GdiUtil.InverseTransform(transform, p);

        LayoutContexts.Center(layoutContext, items, p);
    }
}
```

ノードの属性の変更

DOM ノードの属性を変更するには、DOM アダプタのプロパティを使用します。DOM アダプタは、DOM ノードデータに対して使い慣れた C# 構文を提供するだけでなく、基本データモデルに変更があってもアプリケーションコードを保護します。

```
Prototype prototype = node.As<Prototype>();
if (prototype != null)
{
    prototype.Name = name;
}
```

DOM アダプタを実装するとき、DOM アダプタコード内で DomNode.GetAttribute() および DomNode.SetAttribute() メソッドを使用して、DOM ノード内のデータを直接変更します。詳細は、「DOM アダプタの実装」の「属性と要素のプロパティへのラッピング」を参照してください。

子ノードの追加

子 DOM ノードを作成し、それを親に追加するには、DomNode.SetChild() メソッドを使用します。SetChild() メソッドの引数は、子メタデータに対する ChildInfo メタデータ型と、子自体に対する DOM ノードの 2 つです。子のノードが存在しない場合は、新しく作成する必要があります。

FSM サンプルからの次のコードは、最初に FSM 型のルートノードを作成してから、そのルートの子の PrototypeFolder を作成しています。

```
node.SetChild(
    Schema.fsmType.prototypeFolderChild,
    new DomNode(Schema.prototypeFolderType.Type));
```

子ノードがすでに存在して、それが要素のリストである場合、新しいノードをコレクション同様にリストに追加できます。

```
Prototype prototype = node.As<Prototype>();

PrototypeFolder folder = active_item.As<PrototypeFolder>();
folder.Prototypes.Add(prototype);
```

ノードの削除

ノードを削除するには、DomNode.RemoveFromParent() メソッドを使用します。

```
node.RemoveFromParent();
```

ノードを削除すると、そのノードに関連付けられているすべての拡張とアダプタも削除されます。DOM ノードツリーのルートノードは削除できないので注意してください。

DOM アダプタの実装

独自の DOM アダプタを作成するには、DomNodeAdapter 基底クラスを拡張します。このクラスは、アダプタを基本 DOM ノードに関連付けるためのインフラストラクチャおよび、多くの便利な一般的なメソッドとプロパティを提供します。独自の DOM アダプタの実装では、次を追加します。

- アダプタを基本ノードに関連付け（再関連付け）するときの初期化または構成の振る舞いを提供する、OnNodeSet() のオーバーライドメソッド。
- DOM ノードの属性と要素リストを取得および設定するプロパティ。
- DOM ノード間の参照と、外部リソースへの参照を管理する機能。
- その他の振る舞いを、アダプタ内のメソッドとして。
- 編集コンテキストでもあるアダプタの場合、コンテキストインタフェースの実装。

DomNodeAdapter 基底クラス

DomNodeAdapter クラスは、DOM アダプタを基本 DOM ノードに関連付けるためのインフラストラクチャおよび、DOM アダプタの実装や、アダプタを介した DOM ノードの作業に役立つ多くの一般的なメソッドを提供します。

DomNodeAdapter 基底クラスの機能を次に示します。

- IAdapter、IAdaptable、IDecorator インタフェースを実装して、DOM ノードの適応を提供します。
- DomNode プロパティを提供して、アダプタの基本 DOM ノードにアクセスできるようにします。
- 空の OnNodeSet() メソッドを提供します。このメソッドは、アダプタの基本 DOM ノードが設定または変更されたときに呼び出されます。DOM アダプタクラス内で、このメソッドをオーバーライドします。詳細は「OnNodeSet() メソッド」を参照してください。
- DOM ノードのアダプタを取得したり、DOM ノードアダプタが存在するかテストしたり、異なる DOM アダプタを設定するためのメソッドを提供します (As<T>()、Is<T> ()、Cast<T>())。これらのメソッドの詳細は、「DOM ノードアダプタの設定」を参照してください。
- DOM ノードの子および親データを操作したり、そのノードの属性を取得および設定したり、DOM ノード間の参照を管理したりするためのユーティリティメソッドを定義します。

OnNodeSet() メソッド

DOM アダプタが基本 DOM ノードに関連付けられるときに、初期化や同期タスクを実行するには、OnNodeSet() メソッドを使用します。OnNodeSet() メソッドは、最初の 1 回だけでなく、アダプタがノードに関連付けられるたびに呼び出されます。As<T> と Cast<T>() メソッドはどちらも OnNodeSet() を呼び出します。

OnNodeSet() メソッドに引数はなく、void を返します。DomNodeAdapter 基底クラスのデフォルトの実装は空ですが、この基底クラス内の以降のコードが必ず呼び出されるようにするために、base.OnNodeSet() を呼び出してください。

```
protected override void OnNodeSet()
{
    // your implementation here

    base.OnNodeSet();
}
```

OnNodeSet() メソッドには次のような一般的なタスクがあります。

- 位置、境界ボックス、内部状態プロパティ (非表示、ロックなど) など、同期していない可能性のあるアダプタやノードのデータを更新します。
- As<T>() を呼び出してノードを他のアダプタに関連付けてから、新しいアダプタをプライベートフィールドに割り当てます。同じ DOM データに対して複数の DOM アダプタが頻繁に併用されます。OnNodeSet() メソッドは、アダプタ間の初期の関連付けを確立する場となります。
- イベントハンドラを作成し、DOM ノード内のデータの変更を追跡します。

次の OnNodeSet() メソッドは、ATF の FSM サンプルの Fsm クラスのものです。Fsm クラスは、FSM 型の DOM ノードを適応させる DOM アダプタです。FSM 型には、ステート、遷移、注釈を含めることができます。これらはすべてそれ自体が DOM 型、DOM ノード、および DOM アダプタです。この OnNodeSet() メソッドは、DomNodeListAdapter クラスを使用して、ノードおよびスキーマ型から子アダプタのリストを取得します。このラッパークラスの詳細は、「子の DOM アダプタのリストの取得」を参照してください。

```
protected override void OnNodeSet()
{
    m_states = new DomNodeListAdapter<State>(DomNode,
        Schema.fsmType.stateChild);
    m_transitions = new DomNodeListAdapter<Transition>(DomNode,
        Schema.fsmType.transitionChild);
    m_annotations = new DomNodeListAdapter<Annotation>(DomNode,
        Schema.fsmType.annotationChild);
}
```

次の OnNodeSet() メソッドも FSM サンプルからですが、EditingContext DOM アダプタのものです。EditingContext クラスも FSM ノードを適応させて、選択、検証、トランザクション、履歴などの編集機能を追加します。この OnNodeSet() メソッドは、FSM 型を他の 2 つのアダプタ (Fsm および ViewingContext) に型変換し、それらのアダプタを保存してから、FSM データの変更を監視するために DOM ノードのイベントハンドラのセットを定義しています。

```
protected override void OnNodeSet()
{
    m_fsm = DomNode.Cast<Fsm>();

    m_viewingContext = DomNode.Cast<ViewingContext>();

    DomNode.AttributeChanged += new
       EventHandler<AttributeEventArgs>(DomNode_AttributeChanged);
    DomNode.ChildInserted += new
       EventHandler<ChildEventArgs>(DomNode_ChildInserted);
    DomNode.ChildRemoved += new
       EventHandler<ChildEventArgs>(DomNode_ChildRemoved);

    base.OnNodeSet();
}
```

属性と要素のプロパティへのラッピング

データモデルの属性と子要素 (エンティティ) は、C# のプロパティに簡単にマップできます。プロパティを使用すると、基本 DOM ノード内の属性とエンティティデータのオブジェクトモデルを提供したり、型定義に変更があってもアプリケーションコードが保護されます。

基本 DOM ノード内の属性値を取得および設定するには、GetAttribute<T>() と SetAttribute() メソッドを使用します。

```
public string Name
```

```
{
    get { return GetAttribute<string>(Schema.stateType.labelAttribute); }
    set { SetAttribute(Schema.stateType.labelAttribute, value); }
}
```

GetAttribute<T>() と SetAttribute() の引数はいずれも、属性を表す AttributeInfo DOM メタデータオブジェクトです。ジェネリック型パラメータ T は string や int など属性の型で、プロパティの型と一致する必要があります。ほとんどの場合、これらの DOM メタデータオブジェクトは、DomGen プログラムによって XML スキーマから生成され、スキーマスタブクラスにリストされます。この例の Schema.stateType.labelAttribute は AttributeInfo オブジェクトで、ステート型のラベル属性を表しています。このサンプルコードは、その属性を Name プロパティにラッピングします。

データモデル内の子要素またはエンティティを表す子の DOM ノードのリストを、読み取り専用のジェネリック IList プロパティとしてラップします。子のリストを取得するには、GetChildList<T>() メソッドを使用します。

```
public IList<Prototype> Prototypes
{
    get { return GetChildList<Prototype>(
        Schema.prototypeFolderType.prototypeChild); }
}
```

GetChildList<T>() メソッドの引数は、ChildInfo DOM メタデータオブジェクトです。ジェネリック型パラメータ T は、それぞれの子が適応すべき型です (IList と同じ型)。属性と同様に、この場合の ChildInfo オブジェクトは XML スキーマの DomGen によって生成されたスキーマスタブクラスから取得されます。

この例は FSM Editor チュートリアルの PrototypeFolder クラスからのもので、プロトタイプフォルダ型には、1 つまたは複数のプロトタイプ型を子要素として含めることができます。別の Prototypes DOM アダプタクラスは、プロトタイプ自体を定義します。したがって、この Prototypes プロパティは、この PrototypeFolder に含まれるすべての子 Prototype オブジェクトの IList オブジェクトです。

GetChildList<T>() で要素をラップする場合、ラップされた要素は読み取り専用の「get」プロパティであることに注意してください。基本 DOM オブジェクトから子要素を追加または削除する場合、任意の DOM インタフェースを使用して子要素を変更するメソッドまたは、子要素を基本 DOM オブジェクトに追加または削除する DOM オブジェクトメソッドを定義します。

参照の使用

参照は、親子の関係でない 2 つの DOM ノード間の関係を定義します。次の参照があります。

- 同じ DOM ツリーの 2 つのノード間の内部参照。

- DOM ツリー外部のリソース（アセットまたはサブドキュメント）への外部参照。外部参照は、外部リソースの URI（存在する場合）を含む属性です。リソースは、アプリケーションに必要なときにだけ読み込まれます。それらのリソースの作成、読み込み、管理の振る舞いはユーザが実装する必要があります。

内部参照

同じ DOM ツリー内にあるノード間の内部参照は、DOM アダプタ内でプロパティとしてラッピングできる単純な属性です。

DOM 間の内部参照を解決または定義するには、SetReference() と GetReference<T>() メソッドを使用します。

```
public State FromState
{
    get { return GetReference<State>(
        Schema.transitionType.sourceAttribute); }
    set { SetReference(Schema.transitionType.sourceAttribute, value); }
}
```

GetReference<T>() と SetReference() の引数はいずれも、属性を表す AttributeInfo DOM メタデータオブジェクトです。ジェネリック型パラメータ T は参照の型で、プロパティの型と一致する必要があります。属性同様に、これらの DOM メタデータオブジェクトは、DomGen プログラムによって XML スキーマから生成され、スキーマスタブクラスにリストされます。SetReference() には、この参照の対象である DOM アダプタも設定します。

次の例は FSM サンプルからのものです。これは、この Transition オブジェクトからソースステートを表す State オブジェクトへの参照を表す FromState プロパティを作成します。ここでは、Schema.transitionType.sourceAttribute が AttributeInfo オブジェクトで、遷移型のソース属性を示します。また、ToState 参照に対応するプロパティもあります。

```
public State FromState
{
    get { return
        GetReference<State>(Schema.transitionType.sourceAttribute); }
    set { SetReference(Schema.transitionType.sourceAttribute, value); }
}
```

外部参照

外部参照は、サブドキュメントなど、DOM ノードツリーの外部にあるリソースへの参照です。外部参照は、データモデル内で URI として定義されます。アプリケーションで、外部参照内のデータの検索、読み込み、解放、管理を実行する必要があります。

外部参照を使用している一例は、Timeline サンプルです。Timeline サンプルでは、外部のタイムラインドキュメントをマスタータイムラインドキュメント内に含めることができます。外部ドキュメント参照を独自のタイムライン参照として操作したり、ドキュメントを読み込み、サブドキュメント内のトラックおよびマーカーを編集できます。

タイムラインデータモデルには、TimelineRefType 型が含まれます。これは、タイムラインイベントデータを含む外部サブドキュメントへの参照です。TimelineRefType 型の 1 つの属性は、外部ドキュメントを示す URI です。

```
<xs:complexType name="timelineRefType">
    <xs:attribute name="name" type="xs:string" />
   <xs:attribute name="start" type="xs:float"/>
   <xs:attribute name="description" type="xs:string"/>
   <xs:attribute name="color" type="xs:integer" default="-32640"/>
   <xs:attribute name="ref" type="xs:anyURI" />
</xs:complexType>
```

TimelineReference DOM アダプタは、タイムライン参照型を表します。この DOM アダプタは、URI プロパティ内の ref 属性をラップします。

```
public Uri Uri
{
    get { return DomNode.GetAttribute(
        Schema.timelineRefType.refAttribute) as Uri; }
    set { DomNode.SetAttribute(
        Schema.timelineRefType.refAttribute, value); }
}
```

また、要求されたときに ATF のドキュメントレジストリを使用して TimelineReference.Uri でドキュメントを読み込む Target プロパティを追加します。

```
public IHierarchicalTimeline Target
{
    get
    {
        IHierarchicalTimeline target = null;
        TimelineDocument doc =
          (TimelineDocument)DocumentRegistries.GetDocument(
              TimelineEditor.TimelineDocumentRegistry, Uri);
        if (doc != null)
            target = doc.Timeline as IHierarchicalTimeline;
        return target;
    }
}
```

TimelineEditor クラスは、TimelineReference オブジェクトと Uri および Target プロパティを利用して、タイムラインサブドキュメントを必要に応じて読み込み、管理します。

DOM 拡張とアダプタの登録

DOM 拡張とアダプタを適用する型に対して登録するには、DomNodeType.Define() メソッドを使用します。DOM 拡張とアダプタの登録は通常、アプリケーションの型ローダで行われます。XmlSchemaTypeLoader から派生した型ローダを使用している場合、DOM 拡張の登録を OnSchemaSetLoaded() メソッド内で行います。詳細は、「スキーマ型ローダの実装」を参照してください。

```
Schema.fsmType.Type.Define(new ExtensionInfo<Fsm>());
Schema.fsmType.Type.Define(new ExtensionInfo<EditingContext>());
Schema.fsmType.Type.Define(new ExtensionInfo<ViewingContext>());

Schema.prototypeFolderType.Type.Define(new
    ExtensionInfo<PrototypeFolder>());
Schema.prototypeType.Type.Define(new ExtensionInfo<Prototype>());
Schema.stateType.Type.Define(new ExtensionInfo<State>());
Schema.transitionType.Type.Define(new ExtensionInfo<Transition>());
Schema.annotationType.Type.Define(new ExtensionInfo<Annotation>());
```

この例は FSM サンプルからのもので、異なる種類の DOM アダプタをいくつか定義しています。いずれも、Schema.fsmType、Schema.stateType、Schema.prototypeType など、スキーマスタブファイルで定義されている DOM メタデータ型 (DomNodeType) を使用しています。

この例の最初の 3 行は、基底 FSM 型の DOM アダプタを定義しています。これは、同じ DOM 型に対して複数の DOM アダプタを登録する方法を示しています。この場合、基底 FSM オブジェクトモデルを定義する DOM アダプタと、編集および表示コンテキストの両方を作成するアダプタを登録しています。またサンプルでは、この型に対して検証、一意の ID、ドキュメント、印刷可能ドキュメント、その他の機能に対する DOM アダプタも登録されていますが、ここには表示されていません。

次の例では、FSM データモデル内のその他の型が登録されています。これらの登録は型と DOM アダプタ間の対応が 1 対 1 なので、1 つの型に対する登録は 1 つだけです。

DOM アダプタは、型の継承を利用できます。ある型と 1 つ以上のサブ型に適用する DOM アダプタがある場合、その DOM アダプタを適用するそれらすべての型に登録する必要はないことに注意してください。基底型に登録するだけで済みます。たとえば、基底型 の polygonType と、派生した triangleType、rectangleType、octagonType の各型をサポートする DOM インタフェースがある場合、DOM インタフェースは polygonType に対してのみ一度登録する必要があります。

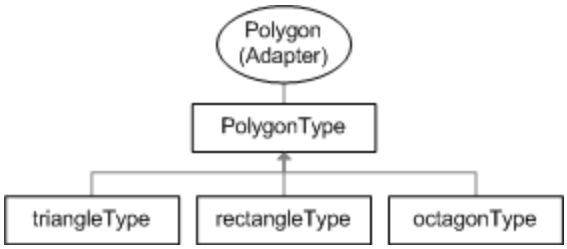

図 22: DOM アダプタと型の継承

DomNodeType.BaseOfAllTypes 静的プロパティには、データモデル内で定義したすべての DOM メタデータ型のルートが含まれています。このプロパティを使用すると、データモデル内にあるすべての型に適用する DOM アダプタを登録できます。ただし、BaseOfAllTypes は静的であるため、すべての型に対する DOM アダプタの追加には、Define() ではなく、AddAdapterCreator() メソッドと AdaptorCreator クラスを使用します。次の行はほとんどのサンプルに含まれていますが、DOM アダプタを追加してプロパティ記述子を管理します。

```
DomNodeType.BaseOfAllTypes.AddAdapterCreator(new
     AdapterCreator<CustomTypeDescriptorNodeAdapter>());
```

5. その他の DOM の機能とユーティリティ

この章では、これまでの章で説明されていない、DOM のその他の特徴や機能を説明します。本章は、次のセクションで構成されています。

DOM イベント

DomNode クラスは、ノードの属性と子ノードへの変更を管理するためのイベントのセットとイベントハンドラを定義します。DOM ノードツリー内の任意の DOM ノードに対するイベントを購読し、そのノードとすべての子のイベントを受信することができます。DOM ノードのイベントは、編集コンテキストやデータバリデータなど、DomNodeAdapter から継承した任意のクラスで管理できますが、イベントの購読とイベントハンドラの定義はどちらも通常、DOM アダプタの実装の中で行います。

DOM ノードのイベント

表 8: は、DOM ノードで利用できるイベントをまとめたものです。この表には、各イベントに対応するイベント引数オブジェクト内で利用できるフィールドの概要も示しています。慣例的に、イベントハンドラを定義してイベントを処理する場合、イベント引数パラメータは e と呼ばれます。

DomNode は、変更発生前に発生するイベント（プレイベント）と変更後に発生するイベント（ポストイベント）の 2 つのグループを定義することに注意してください。

表 8:　DOM ノードのイベント

イベント	イベント引数 (e)	説明
AttributeChanged	AttributeEventArgs	DOM ノードの属性の値が変更されました。 ▪ e.DomNode: この属性が属するノード ▪ e.AttributeInfo: 属性を表すメタデータオブジェクト ▪ e.OldValue: 属性の古い値 ▪ e.NewValue: 属性の新しい値
AttributeChanging	AttributeEventArgs	DOM ノードの属性の値が変更されようとしてます。AttributeChanged を参照してください。
ChildInserted	ChildEventArgs	DOM ノードの子が挿入されました。子は、このノードがルートであるツリーの任意の場所に挿入されています。 e.Parent: この子の直接の親ノード e.Child: 挿入された子ノード e.ChildInfo: 子を表すメタデータオブジェクト e.Index: 子が挿入されたリストインデックス
ChildInserting	ChildEventArgs	DOM ノードの子が挿入されようとしています。ChildInserted を参照してください。
ChildRemoved	ChildEventArgs	DOM ノードの子が削除されました。その子は、このノードがルートであるツリー内の任意の場所から削除できます。 e.Parent: この子の直接の親ノード e.Child: 削除された子ノード e.ChildInfo: 子を表すメタデータオブジェクト e.Index: 子が削除されたリストインデックス

イベント	イベント引数 (e)	説明
ChildRemoving	ChildEventArgs	DOM ノードの子が削除されようとしています。ChildRemoved を参照してください。

イベントの購読

DOM アダプタクラスでイベントを使用するには、OnNodeSet() メソッド内でイベントを購読し、イベントハンドラのコンストラクタでイベントハンドラのメソッド名を指定します。次の例では、そのメソッドの名前は DomNode_AttributeChanged です。

```
protected override void OnNodeSet()
{
    DomNode.AttributeChanged +=
        new EventHandler<AttributeEventArgs>(DomNode_AttributeChanged);
   // ...

    base.OnNodeSet();
}
```

また、購読するイベントのイベントハンドラを、引数がイベントを送信するオブジェクトとイベント引数パラメータ (e) の 2 つであるプライベート void メソッドとして定義します。

```
private void DomNode_AttributeChanged(object sender,
    AttributeEventArgs e)
{
    if (IsFsmItem(e.DomNode, e.DomNode.Parent))
        Event.Raise(ItemChanged, this,
            new ItemChangedEventArgs<object>(e.DomNode));
}
```

この例は FSM サンプルの一部で、EditingContext クラスからのものです。このクラスでは、OnNodeSet() メソッド内で AttributeChanged イベントを購読します。イベントハンドラメソッドでは、属性が属するノードが FSM 型の場合、ItemChanged イベントが発生します。アイテム関連イベントが、編集コンテキストフレームワークの一部である IObservableContext インタフェースによって定義されています。

オブザーバおよびバリデータイベント

データを監視および検証する ATF の基底クラスは、トランザクションの管理、選択およびデータの検証に役立つ追加イベントを DOM に追加します。これらのイベントは通常、アイテムやトランザクションなど、個別の属性や子要素の変更ではなく、変更のより大きなグループを処理します。実行するタスクによっては、これらの追加イベントを利用するバリデータを実装する方が、個別ノードで DOM イベントをリッスンするよりも簡単な場合があります。

Observer および Validator 基底クラスは、イベント自体の他に、それらのメソッドのリスニングを容易にするユーティリティイベントメソッドも追加します。それらのクラスから派生してからイベントメソッドをオーバライドすることで、特定のメソッドを購読したりコールバックを作成したりしなくても、イベントの発生時に処理を実行することができます。カスタムイベントメソッドの詳細と例は、「DOM データの検証」を参照してください。

DOM データの検証

DOM 内にあるデータの一貫性と整合性を確認するには、バリデータを使用します。DOM 内のデータが変更されると、バリデータは新しいデータがそのデータに定義した規則に従っていることを確認します。最も一般的なバリデータの形式は属性値と子要素の単純なデータ検証ですが、内部参照と外部参照を確認したり、ツリー内にある複数のノード間でカスタムの依存性を設定したりするなど、DOM バリデータを他の種類の妥当性に使用することもできます。

このセクションには、次の 3 つのサブセクションがあります。

- DOM バリデータについて: DOM バリデータとデータ検証規則の概要。
- 主要なクラス: プロパティ記述子で使用されるクラスとインタフェースの概要。
- コア ATF バリデータクラス:
- データ検証規則クラス:
- カスタムバリデータの実装:

DOM バリデータについて

DOM に含まれるデータがアプリケーションにとって常に正しく、一貫性があり、意味があるようにするには、DOM バリデータを使用します。DOM 内のデータ (属性) が変化するたびに、DOM ノードが追加または削除されたり、その他の DOM の内容が変更されます。DOM の型に登録した DOM バリデータは、変更を確認し、データに割り当てた妥当性の条件が必ず満たされるようにします。

DOM 検証は、検証コンテキスト (IValidationContext) の中で行われる必要があります。EditingContext クラス (Sce.Atf.Applications アセンブリ) がこのコンテキストを提供します。編集コンテキスト内でも、検証がトランザクション内で発生します。トランザクションは、必要に応じて一組の変更を追跡およびロールバック可能にします。トランザクション内で検証が発生するとき、変更が無効ならば、その変更をキャンセルして元の状態に戻すことができます。

ATF には、アプリケーション内で使用できる一般的な検証処理のためのクラスが含まれています。これらのクラスには、規則ベースのデータ検証、参照管理、および一意の DOM ノード ID などが含まれています。これら基本的な DOM バリデータを使用するには、他の DOM アダプタと同様に、スキーマ型ローダ内でバリデータを登録します。

データ検証は、DOM が使用する最も一般的な検証です。DOM データバリデータは、元のデータモデルが定義した規則を使用して、DOM の属性と子要素の妥当性を保証します。

データ定義言語として XML スキーマを使用している場合、これらの規則は、スキーマ型ローダによってスキーマが読み込まれる際に自動的に作成されます。型に対して DOM データバリデータを登録していれば、データモデル内で使用する制限はすべて、DOM 内の検証に自動的に変換されます。異なる型定義言語を使用していて、独自の型ローダを作成している場合、データ制限を DOM 検証規則に変換し、それらを型の属性と子メタデータに追加する必要があります。既存の規則が要件に合わない場合は、新しい規則を簡単に作成することもできます。

新しい DOM バリデータクラスを作成して、新たな種類のデータ検証を実行することもできます。Validator 基底クラスから派生して新しいバリデータを作成し、DOM 内のさまざまなイベントに対応するメソッドを実装します。

主要なクラス

Sce.Atf.Dom 名前空間には、バリデータ自体のクラスやデータ検証規則に関連するクラスなど、バリデータに関連するクラスが数多く含まれています。

DOM 検証のクラス

表 9: は、DOM 内の検証に関連するクラスを示しています。0 は、これらの型のクラスダイアグラムを示しています。

表 9:　バリデータに関連した Sce.Atf.Dom の主要なクラス

クラス	説明
DataValidator	データモデルで定義された属性と子の規則の DOM バリデータ。以下の「データ検証規則クラス」を参照してください。Validator を拡張します。
DependencyValidator	DOM ノード間の依存性を追跡する DOM バリデータ。Sce.Atf.DependencySystem を参照してください。Validator を拡張します。
IValidationContext	検証を実行するコンテキストのインタフェース。

クラス	説明
IdValidator	ID バリデータの抽象基底クラス。DOM ノードが一意の ID を持つことを確認します。Validator を拡張します。
LockingValidator	DOM ノード内のロックされたデータに対する DOM バリデータ。ILockingContext を実装したアダプタが必要です。Validator を拡張します。
ReferenceValidator	DOM ノード間の内部参照や外部リソースへの参照に対する DOM バリデータ。Validator を拡張します。
UniqueIdValidator	DOM ノードが一意の ID を持つことを確認する DOM バリデータ。IdValidator を拡張します。
UniquePathIdValidator	DOM ノードが DOM ノードツリー内で一意のパスを持つことを確認する DOM バリデータ。
Validator	DOM バリデータの抽象基底クラス。このクラスから派生して、独自のバリデータを作成します。Observer から派生しています。

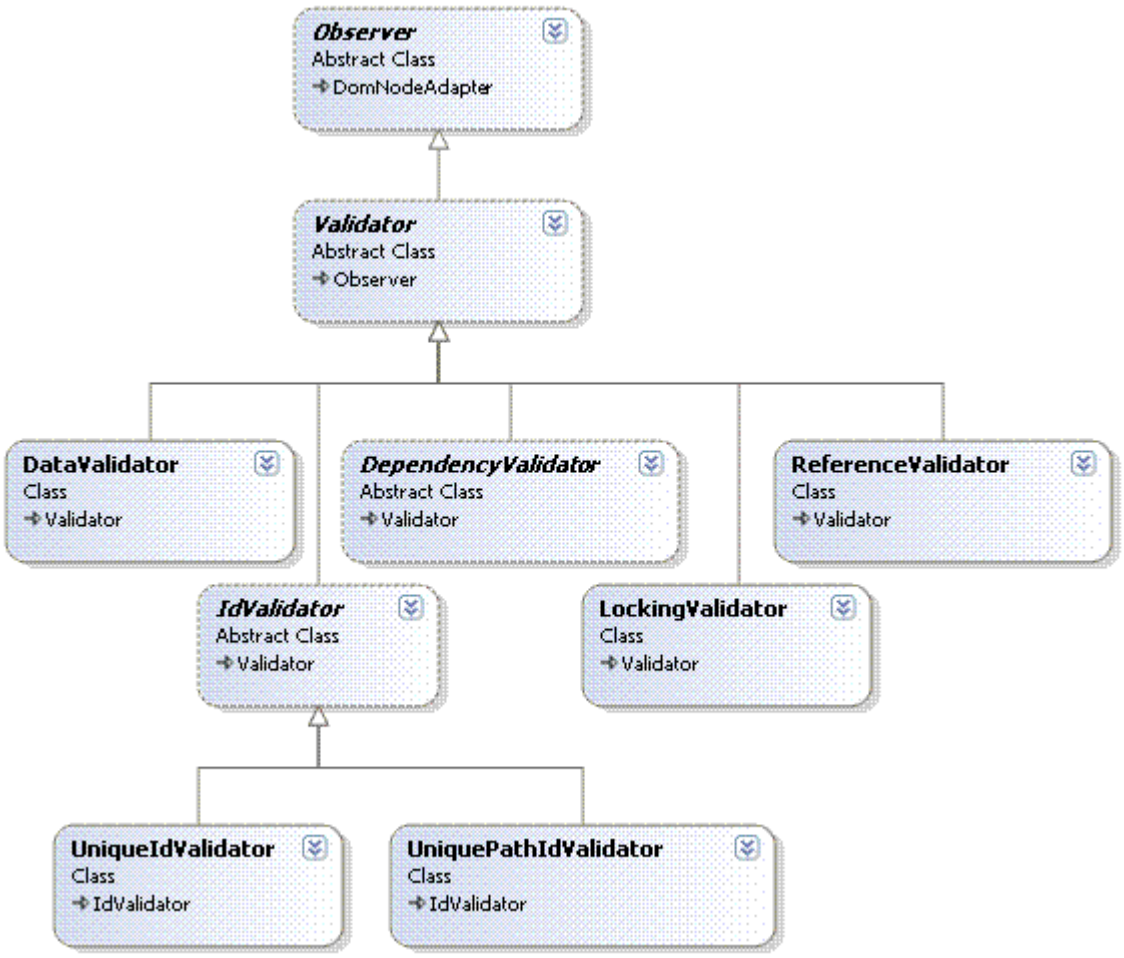

図 23: DOM バリデータのクラス

データ検証規則のクラス

表 10: は、データ検証に特有の規則を定義する Sce.Atf.Dom 内のクラスを示しています (DataValidator クラス)。0 は、これらの型のクラスダイアグラムを示しています。これらのクラスの詳細は、「データ検証規則クラス」を参照してください。

表 10: データ検証規則に関連した Sce.Atf.Dom の主要なクラス

クラス	説明
AttributeRule	属性制限のバリデータの抽象基底クラス。
ChildCountRule	子の数のバリデータ。ChildRule を拡張します。
ChildRule	子制限のバリデータの抽象基底クラス。

クラス	説明
NumericMaxRule	属性の最大値制限のバリデータ。AttributeRule を拡張します。
NumericMinRule	属性の最小値制限のバリデータ。AttributeRule を拡張します。
StringEnumRule	属性の列挙制限のバリデータ。

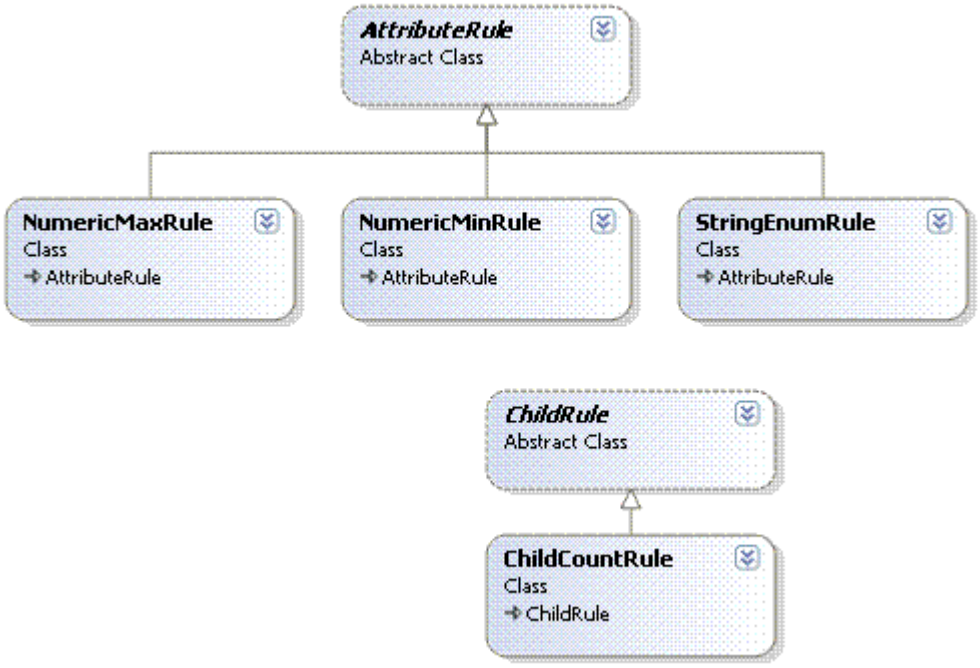

図 24: データ検証規則の種類

コア ATF バリデータクラス

ATF には、アプリケーション内で使用できる基本的なバリデータのセットが含まれています。DOM データに対して最もよく使用されるバリデータを、次に示します。

- DataValidator: 属性と子のデータが、データモデルで定義されている最大値/最小値、子要素の数、文字列列挙などの制限に従っていることを確認します。「データ検証の規則」を参照してください。
- UniqueIdValidator: すべての DOM ノードが一意の内部 ID を持っていることを確認します。
- ReferenceValidator: 内部参照と外部参照を管理します。

DOM バリデータ拡張のリストについては、表 9: を参照してください。

これらのバリデータを使用するには、スキーマローダ内でバリデータを型に対する拡張として登録します。通常、データモデルのプライマリデータ型に対してバリデータを登録します。

```
Schema.fsmType.Type.Define(new ExtensionInfo<DataValidator>());
Schema.fsmType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.fsmType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
```

DOM バリデータは、DOM アダプタでもあります。DOM アダプタが完全に初期化されるのは、最初に使用されるときだけです。DOM バリデータはアプリケーション内で間接的に使用されるため、アプリケーションの起動時や、ファイルを開いたり新規作成したりするときなどは、DOM ノードの新しいツリーに対して明示的に初期化する必要があります。DOM ノードのツリー全体に対して拡張を初期化するには、DomNode.InitializeExtensions() メソッドを使用します。この例のノードは、ツリーのルートです。

```
node.InitializeExtensions();
```

データ検証規則クラス

DataValidator クラスは、データモデル内で定義した属性と子要素の制限に対する検証機能を提供します。たとえば、Size 属性の制限を 100 未満に定義したり、City 属性を文字列の列挙に制限したりできます。

制限は、データモデルの型定義ファイル内に作成します。XML スキーマを使用している場合、これらの制限は標準の XSD 制約ファセットを使用して作成します。独自の型定義ファイルを作成する場合、独自の制限を定義する必要があります。

制限は、スキーマ型ローダが型定義ファイルを読み込むときに、データ検証規則に変換されます。表 10: にリストされたさまざまな規則クラスのインスタンスが、型の属性または子メタデータの一部として追加されます (AttributeInfo または ChildInfo)。型ローダ内で DataValidator を拡張として登録することで、データ検証をアプリケーションに追加すると、これらの規則は、DOM 内でデータが変更されたときに、属性と子の新しいデータを検証するために使用されます。詳細は、AttributeInfo と ChildInfo の AddRule() および Validate() メソッドを参照してください。

検証規則は単純なクラスで、独自のデータモデルや型定義ファイルを使用している場合は、独自の検証規則を容易に作成できます。AttributeRule または ChildRule 基底クラスを拡張し、Validate() メソッドを実装して、独自のデータ検証規則を作成します。例は、既存の規則クラスを参照してください。

カスタムバリデータの実装

既存の ATF バリデータクラスが要件に合わない場合、抽象基底 Validator クラスを拡張して、独自のバリデータクラスを作成することもできます。この基底クラスには、検証トランザクションが発生したときに特定のイベントを監視したり、DOM データが変更されたときにチェックするための、一組の一般的なトランザクションメソッドが含まれています。

Validator 基底クラスから派生して、対象のイベントメソッドをオーバーライドします。表 11: は、Validator 内で利用できるイベントメソッドとそのメソッドのパラメータをまとめたものです。イベントメソッドはすべて void を返します。

表 11: Validator のイベントメソッド

イベントメソッド	説明
AddNode()	任意のノード (新規ノードまたは子ノード) がツリーに追加されたときに呼び出されます。引数: ▪ DomNode: 追加されたノード。
OnAttributeChanged()	ツリー内にある DOM ノードの属性が変更されたときに呼び出されます。このメソッドをオーバーライドするのは、AttributeChanged イベントをリッスンするのと同じことです。引数: ▪ オブジェクト (ツリーのルートにある DomNode) ▪ AttributeEventArgs
OnBeginning()	検証トランザクションの開始時に呼び出されます。引数: ▪ オブジェクト (IValidationContext) ▪ EventArgs
OnCancelled()	検証がキャンセルされた場合に呼び出されます。引数: ▪ オブジェクト (IValidationContext) ▪ EventArgs
OnChildInserted()	DOM ノードツリー内に子が挿入されたときに呼び出されます。このメソッドをオーバーライドするのは、ChildInserted イベントをリッスンするのと同じことです。引数: ▪ オブジェクト (ツリーのルートにある DomNode) ▪ ChildEventArgs

イベントメソッド	説明
OnChildRemoved()	DOM ノードツリー内の子が削除されたときに呼び出されます。このメソッドをオーバーライドするのは、ChildRemoved イベントをリッスンするのと同じことです。引数: ▪ オブジェクト (ツリーのルートにある DomNode) ▪ ChildEventArgs
OnEnding()	検証トランザクションが終了する直前に呼び出されます。引数: ▪ オブジェクト (IValidationContext) ▪ EventArgs
OnEnded()	検証の終了後に呼び出されます。引数: ▪ オブジェクト (IValidationContext) ▪ EventArgs
RemoveNode()	ツリーからノードが削除されたときに呼び出されます。引数: DomNode: 削除されたノード。

Validator クラスには Validating プロパティもあります。これは、検証コンテキストが処理中の場合に true になります。Validating は、OnBeginning() メソッドが呼び出された後に true になり、OnCancelled() または OnEnding() の後に false になります。検証トランザクションが行われているかどうかを確認するには、DOM イベントメソッドの中で Validating プロパティを使用します。

```
protected virtual void OnAttributeChanged(object sender,
                                          AttributeEventArgs e)
{
    if (Validating)
    {
        m_locked = m_lockingContext.IsLocked(e.DomNode);
    }
}
```

また、バリデータがテストしているデータが条件を満たしておらず、トランザクションが実行されている場合、InvalidTransactionException 例外を渡します。こうすることで、トランザクションがロールバックされ、元のすべての DOM データが保持されます。

この例では、Timeline Editor チュートリアルサンプルは、バリデータを使用して次を確認しています。

- Timeline イベントの開始時間が 0 より大きく、整数に丸められる。
- 間隔長が 1 より大きく、整数に丸められる。

TimelineValidator クラスは Validator 基底クラスから派生しています。これらのテストは共に OnAttributeChanged() メソッドを介して管理されます (単純化するため、タイムラインイベントの開始時間だけを次に示します)。

```
protected override void OnAttributeChanged(object sender,
    AttributeEventArgs e)
{
    BaseEvent _event = e.DomNode.As<BaseEvent>();
    if (_event != null)
    {
        if (e.AttributeInfo == Schema.eventType.startAttribute)
        {
            float value = (float)e.NewValue;

            // >= 0, snapped to nearest integral frame number
            float constrained = Math.Max(value, 0);
            constrained = (float)MathUtil.Snap(constrained, 1.0);
            if (constrained != value)
                throw new InvalidTransactionException(Localizer.Localize(
                  "Timeline events must have a positive
                   integer start time"));
            return;
        }

        Interval interval = _event.As<Interval>();
        if (interval != null)
        {  // further validation for intervals ...
        }
    }
}
```

カスタムバリデータは、Timeline Editor、Dom Tree Editor および Circuit Editor のサンプルにあります。それらのコードを参考にしてください。

DOM プロパティ記述子の定義と使用

プロパティ記述子は .NET の機能で、プロパティエディタなどのコントロールで使用するために、クラスプロパティにメタデータを追加します。DOM プロパティ記述子は、DOM ノードのプロパティ (要素と子ノード) に同一の操作をするカスタム記述子です。アプリケーションでプロパティの編集を可能にするには、データモデル内にプロパティの記述子を作成します。

このセクションには、次の 3 つのサブセクションがあります。

- DOM プロパティ記述子について: プロパティ記述子の DOM 内での使用方法の概要と、プロパティ記述子に含まれる情報の種類。
- 主要なクラス: プロパティ記述子で使用されるクラスとインタフェースの概要。
- スキーマローダでのプロパティ記述子の定義: スキーマローダを使用して、コード内にプロパティ記述子を直接定義する方法。

- XML スキーマ注釈でのプロパティ記述子の定義: コード自体の外部にプロパティ記述子を定義する別の方法です。XML スキーマへの注釈を使用してプロパティ記述子を定義します。この方法は、ATF 2 で使用されていました。

DOM プロパティ記述子について

プロパティ記述子は、クラスプロパティの抽象です。これは、外部コンポーネントやプロパティエディタなどのコントロールが使用できる、プロパティについての追加情報を提供します。プロパティ記述子が提供する情報には、プロパティの表示名、ツールヒントの説明テキスト、そのプロパティの編集に使用されるエディタコントロールなどがあります。

.NET System.ComponentModel 名前空間は、コアの PropertyDescriptor と PropertyDescriptorCollection クラスを定義します。ATF は、DomNode オブジェクトの属性と子に対して、System.ComponentModel.PropertyDescriptor を拡張するカスタムの PropertyDescriptor クラス (Sce.Atf.Dom.PropertyDescriptor) を提供します。ATF の PropertyEditor (0) と GridPropertyEditor プラグインはいずれも、これらのプロパティ記述子を利用しています。

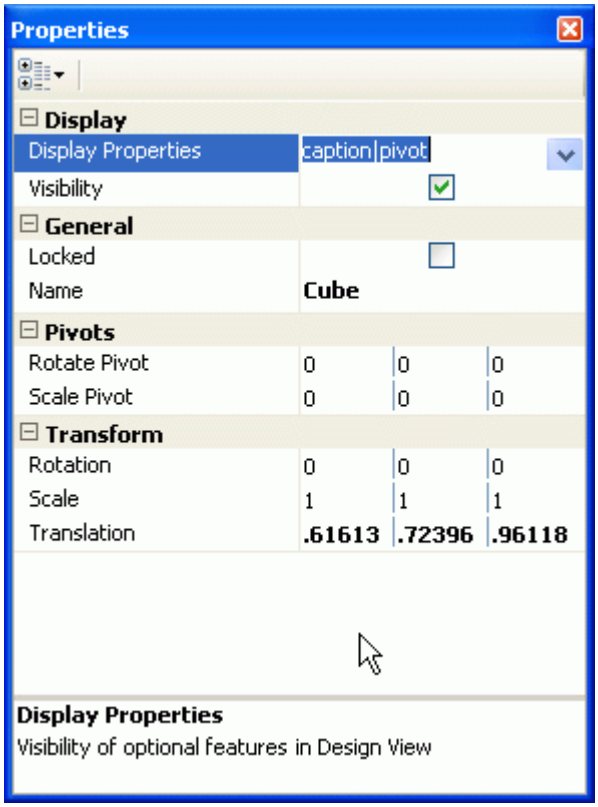

図 25: プロパティエディタ

ATF は、次の 3 種類のプロパティ記述子を定義しています。

- 属性: DomNode 属性はプロパティと直接マッピングされているため、これが最も一般的なプロパティ記述子です。AttributePropertyDescriptor クラスがこれらの記述子を定義します。
- 子属性: 子属性は、通常の属性と同じですが、子ノードに定義され、子のメタデータ (ChildInfo) を含みます。ChildAttributePropertyDescriptor クラスがこれらの記述子を定義します。
- Child ノード: 子ノード自体もプロパティ記述子を含むことができます。ChildPropertyDescriptor クラスがこれらの記述子を定義します。

3 つのプロパティ記述子クラス (,AttributePropertyDescriptor、ChildAttributePropertyDescriptor、ChildPropertyDescriptor) はすべて、抽象 Sce.Atf.Dom.PropertyDescriptor クラスから派生しています。

プロパティ記述子は、アプリケーションのデータモデルに追加として作成され、DOM メタデータ型のメタデータへの追加として定義されます。プロパティ記述子を作成する最も簡単な場所は、アプリケーションのスキーマローダクラス内です。この方法によるプ

ロパティ記述子の定義の詳細は、「スキーマローダでのプロパティ記述子の定義」を参照してください。

ATF 2 では、プロパティ記述子の定義に、データモデルの XML スキーマ内で注釈を使用していました。これらの注釈は、XML スキーマが実行時に解析されたときに、自動的にコードに変換されていました。ATF 3 は、これら従来の XML 注釈の多くをサポートしています。必要に応じて、プロパティ記述子の独自の外部注釈をサポートするために、このコードを基として使用することができます。詳細は、「XML スキーマ注釈でのプロパティ記述子の定義」を参照してください。

DOM が定義するカスタムプロパティ記述子は、ATF に特有の一組のプロパティ（パラメータ）を使用します。表 12: は、それらプロパティ記述子のプロパティをまとめたものです。必要なのは、名前と型だけです。異なるコンストラクタと NULL パラメータを使用して、オプションのパラメータを定義できます。詳細は、さまざまな DOM プロパティ記述子の API ドキュメントを参照してください。

表 12: DOM プロパティ記述子のパラメータ

記述子	型	説明
name	string	プロパティの名前。
type	Type	プロパティの型。通常は AttributeInfo や ChildInfo など、このプロパティを定義するメタデータオブジェクト。
category	String	プロパティが、プロパティエディタでグループにまとめられる場合の、プロパティのカテゴリ。
description	String	プロパティの説明。ヘルプテキストやツールヒントとしてプロパティグリッドまたはプロパティエディタに表示されます。
isReadOnly	Boolean	プロパティが変更可能かどうか。プロパティグリッドでプロパティを編集できる機能だけに影響します。基本 DOM 属性も読み取り専用の場合があります。デフォルトでは、すべてのプロパティが編集可能です。
editor	object	このプロパティの値を変更するためにプロパティグリッドで使用されるエディタ。プロパティエディタは、Controls/PropertyEditing フォルダにある Atf.Gui アセンブリ内にあります。サンプルは、LevelEditor を参照してください。
typeConverter	TypeConverter	型を変換するためのクラス。DOM プロパティ関連の型コンバータは、Controls/PropertyEditing フォルダにある Atf.Gui アセンブリ内にあります。サンプルは、LevelEditor を参照してください。

主要なクラス

表 13 は、プロパティ記述子を定義する Sce.Atf.Dom 内のクラスを示しています。

表 13: プロパティ記述子に関連した Sce.Atf.Dom の主要なクラス

クラス	説明
AttributePropertyDescriptor	DOM ノードの属性のプロパティ記述子を定義します。Sce.Atf.Dom.PropertyDescriptor を拡張します。
ChildAttributePropertyDescriptor	子 DOM ノードの属性のプロパティ記述子を定義します。Sce.Atf.Dom.PropertyDescriptor を拡張します。
ChildPropertyDescriptor	子 DOM ノードのプロパティ記述子を定義します。Sce.Atf.Dom.PropertyDescriptor を拡張します。
PropertyDescriptor	DOM プロパティ記述子の抽象基底クラス。System.ComponentModel.PropertyDescriptor を拡張します。

スキーマローダでのプロパティ記述子の定義

データモデルの型ローダ (通常、スキーマローダ) 内でプロパティ記述子を直接定義します。この方法を使用すると、1 つのクラス内にスキーマとプロパティ記述子の定義を一緒に保持できます。ただし、プロパティ記述子を変更すると、コードを再コンパイルする必要があります。

```
Schema.prototypeFolderType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                Schema.prototypeFolderType.nameAttribute,
                null,
                Localizer.Localize("Prototype folder name"),
                false)
    }));
```

各プロパティ記述子は、この型ローダクラスに定義されているように、特定の DOM メタデータオブジェクト (DomNodeType) に属しています。型に対して追加のメタデータ (プロパティ記述子を含む) を定義するには、DomNodeType.SetTag() メソッドを使用します。この例では、prototypeFolderType メタデータ型にプロパティ記述子を定義しています。

プロパティ記述子に対して、SetTag() メソッドに新しい System.Componentmodel.PropertyDescriptorCollection オブジェクトを渡しています。

PropertyDescriptorCollection のコンストラクタは、PropertyDescriptor オブジェクトの配列を取ります。

個々のプロパティ記述子は、表 13: にリストしたように、PropertyDescriptor のインスタンスか、その派生クラスの 1 つです。表 12: は、さまざまな PropertyDescriptor コンストラクタのパラメータを示しています。詳細は、プロパティ記述子クラスの API ドキュメントを参照してください。

この例では、プロパティフォルダ型には、名前属性に対するプロパティ記述子が 1 つだけあります。この場合、AttributePropertyDescriptor のコンストラクタには 5 つの引数があり、その内の 3 つだけが使用されています。

- プロパティの表示名 (この例でのはローカライズされた文字列「Name」)
- このプロパティを定義するメタデータ。この場合、Schema.prototypeFolderType.nameAttribute によって定義された AttributeInfo オブジェクト。
- プロパティのカテゴリ (存在する場合。この例では NULL)。
- プロパティグリッドのツールヒントに表示されるプロパティの説明 (ローカライズされた文字列「Prototype folder name」)
- 読み取り専用の Boolean。この例では false (名前は編集可能)。

次は DomTreeEditor の ChildAttributePropertyDescriptor の例です。このプロパティ記述子は、UIControlType 型に定義されています。

```
UISchema.UIControlType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new ChildAttributePropertyDescriptor(
                Localizer.Localize("Translation"),
                UISchema.UITransformType.TranslateAttribute,
                UISchema.UIControlType.TransformChild,
                null,
                Localizer.Localize("Item position"),
                false,
                new NumericTupleEditor(typeof(float),
                    new string[] { "X", "Y", "Z" }),
                new FloatArrayConverter()),
      }));
```

この例のコンストラクタでは、次の AttributePropertyDescriptor の例に 2 つの引数を追加しています。

- この属性を適用する子情報を定義するメタデータ。この場合、UISchema.UIControlType.TransformChild によって定義された ChildInfo オブジェクト。

- プロパティエディタでこのプロパティを編集するためのエディタコントロール (および、そのエディタコントロールに渡すパラメータ)。この例では、エディタは Sce.Atf.Controls.PropertyEditing の NumericTupleEditor。
- このプロパティ内のデータの型コンバータ。この例では、Sce.Atf.Controls.PropertyEditing の FloatArrayConverter。

XML スキーマ注釈でのプロパティ記述子の定義

アプリケーションのデータモデルの XML スキーマ内の注釈を使用して、プロパティ記述子を定義することができます。プロパティ記述子を XML スキーマ内に格納することの (スキーマローダ内に直接定義する方法と比較したときの) 利点は、データモデルを定義するすべての情報が 1 つの場所に存在することです。また、プロパティ記述子の注釈は、アプリケーションの実行時に XML スキーマとともに読み込まれるため、プログラマ以外のユーザやデザイナーも、コードにアクセスすることなく、これらのプロパティに変更を加えることができます。

ATF PropertyDescriptor クラスには、ATF 2 の従来の注釈の一部をサポートするために、プロパティ記述子に対する基本的な XML 注釈処理が含まれています。これらの注釈を解析し、PropertyDescriptor クラスの新しいインスタンスを作成するには、スキーマローダクラスで PropertyDescriptor.ParseXml() メソッドを使用します。

> **注意:** 注釈の書式が ATF 2 の従来の注釈と異なる場合や、追加のプロパティ記述子の型をサポートする場合、これらの注釈を解析する追加コードを記述し、PropertyDescriptor クラスのインスタンスを (通常はスキーマローダの ParseAnnotations() メソッド内に) 作成する必要があります。これは、データモデルに XML スキーマ以外の型定義ファイルを使用している場合にも当てはまります。

プロパティ記述子の注釈

PropertyDescriptor クラスがサポートする XML スキーマ注釈の書式は、従来の ATF 2 アプリケーションで使用されていたものと同じです。PropertyDescriptor クラスは属性 (scea.dom.editors.attribute) および子要素の値 (scea.dom.editors.child) の注釈をサポートしています。

```
<xs:annotation>
  <xs:appinfo>
    <scea.dom.editors menuText="Interval" description="Interval"
     image="TimelineEditorSample.Resources.interval.png"
     category="Timelines" />
    <scea.dom.editors.attribute name="name"
     displayName="Name" description="Name" />
    <scea.dom.editors.attribute name="length"
     displayName="Length" description="Length or Duration" />
    <scea.dom.editors.attribute name="color"
```

```
      displayName="Color" description="Display Color"
      editor="Sce.Atf.Controls.PropertyEditing.ColorEditor"
      converter="Sce.Atf.Controls.PropertyEditing.IntColorConverter" />
  </xs:appinfo>
</xs:annotation>
```

この Interval 型 (Timeline Editor のサンプル) の注釈には、name、length、color という属性の 3 つのプロパティ記述子の注釈が含まれています。

scea.dom.editors という追加注釈は、型自体を説明し、パレットでこの型を表示するために使用されます。PropertyDescriptor クラスはこの注釈を処理しません。この注釈を SchemaLoader クラスの ParseAnnotations() メソッドの中で管理する方法は、Timeline Editor のサンプルを参照してください。

プロパティ記述子の注釈の読み込み

プロパティ記述子の注釈は、スキーマローダクラスの ParseAnnotations() メソッド内で読み込まれます。XML スキーマ注釈を解析し、PropertyDescriptorCollection のインスタンスを作成して、そのコレクションをメタデータとして DOM メタデータオブジェクトに追加するには、PropertyDescriptor.ParseXml() と DomNodeType.SetTag() を使用します。

```
protected override void ParseAnnotations(
    XmlSchemaSet schemaSet,
    IDictionary<NamedMetadata, IList<XmlNode>> annotations)
{
    base.ParseAnnotations(schemaSet, annotations);

    IList<XmlNode> xmlNodes;

    foreach (DomNodeType nodeType in m_typeCollection.GetNodeTypes())
    {
      // parse XML annotation for property descriptors
      if (annotations.TryGetValue(nodeType, out xmlNodes))
      {
          PropertyDescriptorCollection propertyDescriptors =
             Sce.Atf.Dom.PropertyDescriptor.ParseXml(nodeType, xmlNodes);

          // Create additional property descriptors here

          nodeType.SetTag<
             PropertyDescriptorCollection>(propertyDescriptors);

          // parse additional annotations here
      }
   }
}
```

XML 注釈の解析 (ParseXml() を使用) と型のメタデータへの PropertyDescriptorCollection の追加 (SetTag() を使用) の間に、注釈では作成されない追加のプロパティ記述子を手動で定義することもできます。詳細は、「スキーマローダでのプロパティ記述子の定義」を参照してください。

このメソッド内で、PropertyDescriptor クラスによって管理されないその他の注釈（パレットまたはプロキシオブジェクトの注釈など）を解析する必要もあります。

その他の例は、Timeline Editor および Level Editor のサンプルを参照してください。

DomExplorer の使用

DOM Explorer は Atf.Gui アセンブリの一部で、DOM ノードツリーの内容を視覚化するために開発中に使用できるプラグインです。DomTreeEditor のサンプルでは、0 に示すように DomExplorer を使用しています。

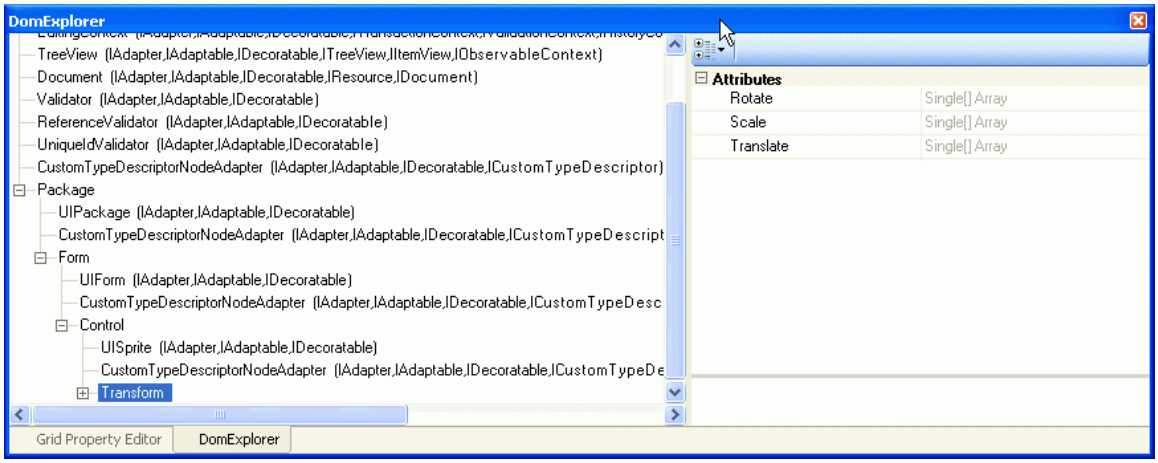

図 26: Dom Explorer

エディタの左側には、DOM ノードツリー内にあるすべてのノードのツリービューが表示されます。ノード名（ノードの ChildInfo.Name プロパティ）がリストされます。ノードの登録済みアダプタはすべて子としてリストされます。アダプタ名の後に、ノードが実装しているすべてのインタフェースが括弧付きで表示されます。たとえば、0 では Package ノードに次の 2 つのアダプタがあります。

- UIPackage。このノードのプライマリ DOM アダプタ。
- CustomTypeDescriptorNodeAdapter。プロパティ記述子を管理するためにすべてのノードに追加されたアダプタ。

Package ノードにはこれらのアダプタに加え、Form 子ノード（対応する UIForm および CustomTypeDescriptorNodeAdapter アダプタも）が含まれています。この中には、Control ノード（UISprite および CustomTypeDescriptorNodeAdapter アダプタも）を含みます。

DOM Explorer プラグインの右側は、ノードの属性の型を確認できるプロパティグリッドです。プロパティグリッドは表示のみで、これらの属性の値は編集できません。この例では、Transform ノードには、Rotate、Scale、Translate という 3 つの属性があり、すべてが Single 型の配列です。

DOM Explorer をエディタに追加するには、他のプラグインと同様に、通常は Main() メソッドの Program.cs ファイルにあるプログラムの型カタログにプラグインを追加します。

```
TypeCatalog catalog = new TypeCatalog(
    typeof(SettingsService),        // persistent settings
                                    //and user preferences dialog
    typeof(StatusService),          // status bar at bottom of main Form
    typeof(CommandService),         // menus and toolbars
    typeof(ControlHostService),     // docking control host
    // ....
    typeof(DomExplorer)             // diagnostic view of DOM
);
```

DOM 永続化

永続的データは、アプリケーションが終了しても何らかの形で維持されるアプリケーションデータで、アプリケーションが再び起動すると、また読み込むことが可能です。通常、永続的データはファイルに保存されますが、データベースや他のサービスに格納することもできます。

ATF では、DOM 内のデータは、元のデータモデルの設計に使用されたのと同じスキーマを使用する XML ドキュメントとして、永続的に格納されるのが最も一般的です。ATF は、ファイルなどの .NET ストリームに XML データを容易に読み書きできるように、DomXmlWriter と DomXmlReader という 2 つのクラスを提供しています。永続的データの形式として XML を使用しない場合は、永続的データストアの読み書きおよび DOM とのやりとりを実行する独自のクラスを記述する必要があります。

このセクションには、次の 3 つのサブセクションがあります。

- 主要なクラス: DOM 永続化の最も重要なクラスの概要。
- XML の読み取り: DomXmlReader を使用した、ストリームから DOM への XML データの読み取り。
- XML の書き込み: DomXmlWriter を使用した、DOM からストリームへの XML データの書き込み。
- XML を使用しない永続的データの使用: XML 以外の永続的データ形式を使用する場合の独自のリーダークラスとライタークラスの記述。

主要なクラス

Sce.Atf.Dom 名前空間には、表 14: に説明するように、DOM 内の永続的データのためのクラスが含まれています。

表 14:　DOM 永続化関連した Sce.Atf.Dom の主要なクラス

クラス	説明
DomXmlReader	XML スキーマで定義された XML データのストリームを DOM ノードツリーに読み取ります。
DomXmlWriter	DOM ノードツリーを XML ストリームに書き込みます。
XmlSchemaTypeCollection	名前空間、要素、属性、および子要素を含む XML スキーマにより定義された型を表現する、DOM メタデータオブジェクトのコレクション。 DomXmlWriter は XmlSchemaTypeCollection データを使用して DOM ノードツリーから XML ファイルを書き込みます。
XmlSchemaTypeLoader	スキーマ型ローダの基底クラス。アプリケーションのスキーマ型ローダは、このクラスから派生して作成します。 DomXmlReader はスキーマローダのインスタンスを使用して、入力内の XML タグを適切な型の DOM ノードに変換します。DomXmlWriter はスキーマローダ (XmlSchemaTypeCollection) 内に含まれるスキーマメタデータ情報を使用して、DOM ノードツリーから XML ファイルを書き込みます。

XML の読み取り

入力ストリームから DOM ノードツリーへ XML コンテンツを読み込むには、DomXmlReader クラスを使用します。読み込み元のストリームは、多くの場合はファイルですが、任意の入力デバイスも使用できます。ATF は、System.IO から標準 .NET ストリームクラスを使用します。

DOM アプリケーションのための XML ファイルの読み取り (および書き込み) は、普通は IDocument インタフェースの実装の一部で、通常はアプリケーションの Editor クラス内にあります。ストリームから XML コンテンツを読み取るには、プログラムを実行してデータモデルを作成するときに、スキーマローダによって XmlSchemaTypeLoader オブジェクトを作成する必要があります。Editor クラスは、コンストラクタに XmlSchemaTypeLoader のインスタンスを取得します。

次のコードは FSM チュートリアルの例にある IDocument.Open() メソッドからのものですが、すべての ATF の例でほとんど同じです。

```
public IDocument Open(Uri uri)
{
    DomNode node = null;
    string filePath = uri.LocalPath;
    string fileName = Path.GetFileName(filePath);

    if (File.Exists(filePath))
    {
        // read existing document using standard XML reader
        using (FileStream stream = new FileStream(filePath,
            FileMode.Open, FileAccess.Read))
        {
            DomXmlReader reader = new DomXmlReader(m_schemaLoader);
            node = reader.Read(stream, uri);
        }
    }
    else
    {
        // create new document by creating a Dom node of
        // the root type defined by the schema
        // ...
    }

     // Initialize the document, contexts, controls
     // ...
}
```

この例では、System.IO.Path、File および FileStream クラスを使用して、ローカルファイルで指定した XML ドキュメントを開いています。DomXmlReader の新しいインスタンスがそのファイルストリーム内のデータを読み取り、そのファイル内に XML データの DomNode インスタンスを作成します。DomXmlReader コンストラクタは DomNode の新しいインスタンスを返します。これは、新しい DOM ノードツリーのルートノードです。

この if 文の 2 番目の部分は、要求されたファイルが存在しない場合に使用されます。IDocument.Open() メソッドは、新しいドキュメントを作成する方法として繰り返します。このブロックでは、空の DOM ノードツリー用に新しいルート DOM ノードを作成します。

XML ファイルを読み取った後には、DOM データがドキュメントとして実行するように初期化するために、通常他の手順を実行する必要があることにも注意してください。たとえば、DOM データのドキュメントや、表示または編集可能なコンテキストを作成したり、As() または Cast() メソッドを使用してそれらのオブジェクトのアダプタを初期化したりする必要があります。新しいドキュメントやコンテキストは、追加して初期化する必要のあるコントロールやアダプタを使用します。これは、InitializeExtensions() メソッドを呼び出して、この DOM ノードツリーを使用するその他すべてのアダプタ (バリデータなど) を確実に初期化するにも適した場所です。

単純化するために、この追加コードは上記の例では省略していますが、FSM サンプル (または任意の ATF サンプル) で参照できます。

XML の書き込み

DOM ノードツリーを出力ストリームに書き込むには、DomXMLWriter クラスを使用します。書き込み元のストリームは、多くの場合はファイルですが、任意の入力デバイスも使用できます。ATF は、System.IO から標準 .NET ストリームクラスを使用します。

DomXmlReader と同様に、DomXmlWriter は IDocument インタフェースの実装の一部として、通常はアプリケーションの Editor クラス内で使用します。次のコードは、FSM チュートリアルの例にある IDocument.Save() メソッドからのものですが、すべての ATF の例でほとんど同じです。

```
public void Save(IDocument document, Uri uri)
{
    string filePath = uri.LocalPath;
    FileMode fileMode = File.Exists(filePath) ? FileMode.Truncate :
                            FileMode.OpenOrCreate;
    using (FileStream stream = new FileStream(filePath, fileMode))
    {
        DomXmlWriter writer = new
            DomXmlWriter(m_schemaLoader.TypeCollection);
        Document fsmDocument = (Document)document;
        writer.Write(fsmDocument.DomNode, stream, uri);
    }
}
```

この例ではまず新規または既存のファイルのパスを、uri 引数から抽出します。次にスキーマローダから型コレクションを使用して DomXmlWriter の新しいインスタンスを作成し、ライターがどの XML タグを書き込むかを認識できるようにします。最後に、入力 IDocument パラメータをアプリケーションのドキュメント型 (Document) に型変換してから、ドキュメントのルートにある DOM データツリーを、開いたファイルストリームに書き込みます。

XML を使用しない永続的データの使用

DOM ノードツリーの永続的形式として XML を使用しない場合、独自のリーダークラスとライタークラスを作成する必要があります。DomXmlReader および DomXmlWriter を独自コードの開始点として使用できます。SimpleDomNoXmlEditor チュートリアルサンプルも、DOM は使用するがデータモデルや永続的形式には XML を使用しないアプリケーションの例を提供しています。リーダーおよびライターのメソッドは IDocument クラス (EventSequenceDocument) に含まれています。詳細は、その例を参照してください。