ATF Programmer’s Guide: Document Object Model (DOM)

Revision History

This guide provides information about using the ATF DOM.

Version	Revision Date	Author(s)	Comments
3.0	4/30/10	Laura Lemay	Initial Draft
3.01	7/20/10	Laura Lemay	Minor edits
3.1	2/25/11	Laura Lemay	Substantial additions for DOM adapters, validators, persistence, property descriptors, DOM explorer, events
3.3	2/29/12	Gary Staas, Ron Little	Made corrections and clarifications, fixed URLs, updated various info, augmented introduction
3.5	11-June-2013	Gary Staas	Added Overview chapter. Extensively edited, made corrections, added clarifications and examples, reordered material. Improved and expanded discussion of schema annotations. Replaced Introduction with "About This Guide." Used new document format.
3.6	31-July-2013	Gary Staas	Remove support for Visual Studio 2008.
3.9	29-Jan-2014	Gary Staas	Updated document list.

Table of Contents

About This Guide

The Authoring Tools Framework (ATF) is a collection of .NET components in C# you can use to build rich professional-grade Windows client applications and game-related tools.

The *ATF Document Object Model (DOM)* is a modeling, management, and persistence framework that provides sophisticated data management for any application, large or small. The DOM is a powerful, flexible, extensible framework for loading, storing, validating, and managing changes to application or game data.

This guide discusses the major concepts and features of the ATF DOM, outlines the key classes and interfaces in the `Sce.Atf.Dom` namespace, and describes common tasks in using the DOM with ATF applications.

What this Guide Contains

This guide contains these chapters:

"About This Guide" (this chapter): Tells you about this guide and other ATF documentation, what you need to know to work with the DOM, and tells you about available resources.

"Overview of the Document Object Model (DOM)": High level overview of the DOM.

"About the Document Object Model": Get a medium level overview of the ATF Document Object Model and how the major components and features of the DOM fit together. The DOM describes a model for an application's data types and manages a tree containing application data of these types. It provides an object-oriented way for an application to interact with DOM data through DOM extensions. It shows the steps you take at implementation time and run time to use the DOM.

"Data Modeling, XML Schemas, and the DOM": Learn how to define a data model for your application and create a type definition file describing the data types in your application. Develop a type loader that brings the types into the application and defines DOM extensions. Find out about utilities and classes available for these tasks if you use XML Schema Definition (XSD) for your type definition language.

"DOM Nodes and DOM Adapters": The DOM stores application data in a tree of DOM nodes. See how to create a DOM node tree and use events to track tree changes. Learn how DOM extensions and adapters provide an object API for data in tree nodes. Explore how to implement and define your own DOM adapters for the types in your data model, including validators to check data as it changes. You can use adapters to work with the DOM rather than directly manipulating DOM nodes, so your application's logic is decoupled from the DOM representation.

"Other DOM Features & Utilities": Other features and functions of the DOM that do not fit into any of the previous chapters. This includes how to persist DOM data, defining and using property descriptors, and how to use the DOM explorer to visualize a DOM tree.

ATF Document Set

The ATF document set is described in Table 1.

Table 1 ATF Document Set

Volume	Version	Comments
ATF Programmer's Guide	3.9	Shows using ATF to create applications and tools. It includes a Glossary of ATF technical terms. On a wiki at https://github.com/SonyWWS/ATF/wiki.

Volume	Version	Comments
ATF Programmer's Guide: Document Object Model (DOM)	3.7	This document: Shows how to use the ATF DOM for application data.
ATF API Reference	3.9	Complete reference to the ATF API. (CHM file).
ATF Curve Editor User and Programming Guide	3.7	A guide to using the ATF Curve Editor component and programming it.
ATF Qt Comparison	3.7	Compares the amount of work needed to create GUI game tools using ATF and Qt.

In addition, complete API class and method reference documentation for ATF is available as comments in the source code itself and as part of Visual Studio's Intellisense.

Who Should Use This Guide

This Guide is intended for tools developers working with the Authoring Tools Framework DOM. Programmers should have:

- Experience in C#, Visual Studio, and .NET development. Framework development experience is also helpful.
- Visual Studio 2012 or 2010
- Access to *SHIP*, the *SHared Information Portal* for Sony Computer Entertainment (SCE) worldwide game development.

Supported Tools

ATF supports the following tools:

- Visual Studio 2012 and 2010. Visual Studio 2010 and 2012 projects are interchangeable.
- All of the core code in the foundation compiles with C# 3.0. Over time, there will be additional support for C# 4.0 covariance and contravariance, using the `in` and `out` keywords, but a C# 3.0 equivalent will be available.

Migration From Earlier Releases

The ATF 2.8 components, data models, and so on, are supported in ATF 3. You can continue to build your tools as usual, while taking advantage of new components and features that ATF 3 provides.

There are many class and method name changes from ATF 2.8 to ATF 3. The `AtfRefactor` tool automates the first step in the migration process by updating those class and method names that have well-defined mapping rules and that affect very few lines in the original source code. For more information, see the *ATF Refactor User Guide*.

Resources

Other Reading

Some other publications provide useful information:

- *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition)* by Krzysztof Cwalina and Brad Abrams. This is an excellent book whose guidelines ATF follows in developing a reusable extensible framework. MSDN has much of the book's content as well as the naming guidelines.
- *Programming C# 5.0, Building Windows 8, Web, and Desktop Applications for the .NET 4.5 Framework* by Ian Griffiths, O'Reilly Media.

Typographical Conventions

This document follows the typographical conventions in Table 2.

Table 2 Document typographical conventions

Font	Used for	Examples
monospace	Paths and file names	`wws_atf\Samples\CircuitEditor`
	Scripts	`function ShouldRevive() return m_shouldRevive end`
	Code elements and code	`public Command(string description) { m_description = description; }`
	Formatted text, such as XML	`<?xml version="1.0" encoding="utf-8" standalone="yes"?>` `<StringLocalizationTable>` `<StringItem id="Cut" context="" translation="Cut" />`
bold	User interface elements	**Add** button
	Menu item and other paths	**Build > Build Solution**
italic	New technical terms	*Document Object Model*
	Emphasis	Do *not* do this.
	Document titles	*Getting Started with ATF*
bold italic	Special emphasis	***MEF Catalog***
blue	URLs (external)	ATF project home
	Links inside a document	"Application Shell Framework"
quoted	Exact text	"ActiproSoftare SyntaxEditor"
	Names	"Add Layer"

1 Overview of the Document Object Model (DOM)

This chapter describes the Document Object Model (DOM) at a high level to highlight its key concepts.

Applications typically create and edit data, so the following data-related functionality is usually necessary:

- Define the data format for application data, which can vary widely, from unformatted text to graphics in a scene, such as a level editor application creates.
- Keep multiple views of data in sync as a user creates and edits data.
- Perform operations on data, such as validating it as it changes.
- Persistently save and read data.

The Document Object Model (DOM) is a simple hierarchical in-memory database providing these features.

Overview

You design a DOM data model by specifying the various kinds of data or *types* of data elements in your application data. You specify these data types in some data definition language of your choice and save the definition in a *type definition file*. For simple elements, type specification includes information like the primitive data type, such as a string or floating point number. More complex element types can have *attributes*. For example, a type that defines graphical objects can have “x” and “y” attributes to specify a coordinate location on a canvas. For more information, see “About Data Modeling”.

The term DOM also refers to the actual data stored using a DOM. When application data is read, it is stored in memory as a tree of data elements known as *DOM nodes*, represented by `DomNode` objects. Each node’s data has a type that was defined in the type definition file, represented as a `DomNodeType` object. This DOM node tree has a root `DomNode` whose type you specify, and you can specify more than one root. For more information, see “Creating and Initializing DOM Nodes”.

If a DOM node can have child nodes, its type definition includes the types of its child nodes. For instance, a finite state machine node could have state and transition child nodes. These *child node types* would be specified in the finite state machine node’s type definition. For more information, see “Parent and Child Elements”.

By default, you define data element types in the ATF DOM with the *XML Schema Definition (XSD)* language. XSD provides great flexibility in type definition, allowing types that describe elements with attributes and child elements. You can also describe relationships and restrictions in the data. For instance, you can specify that an element type has child elements of a certain type and the maximum number of these children. Data types can also be based on other types, so there can be a type inheritance hierarchy. For example, a “polygon” type could be extended to “triangle” and “pentagon” subtypes. You can use a type definition language other than XSD if you prefer. For details, see “Defining the Data Model using XML Schema”.

When an application starts, it reads the type definition file with a *type loader*. This loader converts types to *metadata objects* containing the types’ information. ATF provides an XSD type definition file loader, as well as a set of classes of metadata objects. These metadata classes, such as `DomNodeType` and `AttributeInfo`, provide information the DOM needs to handle data of that type appropriately. You can also provide your own type loader. For more information, see “Implementing a Schema Type Loader”.

ATF provides the `DomGen` utility that, given an XSD type definition file, generates a set of classes you can reference—instead of the names inside the schema. This helps ensure that you don’t have to change code if the schema changes. In addition, when you use these class and field names in your code, they are checked at compile time, unlike schema name strings. For details, see “Using DomGen to Generate a Schema Stub Class”.

DOM extensions are classes that “extend” a `DomNode`’s data type. Extensions have two basic functions: *adapting* a `DomNode`’s data type to another type, and listening to events on nodes and their children.

If you adapt a `DomNode` to a DOM extension, the `DomNode` can be dynamically cast to the DOM extension class’s type. As a result, all the extension class’s methods, properties, and events become available to manipulate the data represented by the DOM node, just as if the DOM node were an instance of that class. In other words, extensions provide an API for an application to manipulate DOM data. You can design the extension with an API that is convenient for you to work with and makes it easier to change the data in response to user editing actions.

Extensions that listen to node events can perform a variety of operations, such as validating data as it changes. Such extensions subscribe to events of interest and perform their processing in the event handlers. For instance, validator extensions can listen to attribute change events and check that the new attribute values are valid.

Extensions are specified for data types by *defining* an extension for the type. You can define as many extensions as you want for each type. In particular, you can define extensions for the type of the DOM tree’s root `DomNode`, which allows you to listen to events on all `DomNode`s. Such a DOM extension class could be a viewing context, an editing context, a document, or a validator for DOM data.

Extensions actually operate on individual `DomNode`s or subtrees of `DomNode`s. You initialize an extension for a given `DomNode` for one of the extension types defined for that `DomNode`’s type. You can initialize a single `DomNode` with one of the casting functions `Is<T>()`, `As<T>()`, and `Cast<T>()` and then use the resulting value as if it were an object of the extension’s type. You can also initialize a whole subtree of DOM nodes with the `DomNode.InitializeExtensions()` method. You can initialize a `DomNode` for as many extensions as are defined for that `DomNode`‘s type.

The `DomNodeAdapter` class provides an infrastructure for DOM extensions, as well as many useful methods and properties for extensions. Extensions derived from `DomNodeAdapter` are known as *DOM Adapters*. Practically speaking, all extensions are DOM adapters.

When you define an extension for a type, the extension is also defined for any subtypes of that type.

You can subscribe to events indicating changes to the DOM node tree or DOM nodes, and DOM extensions often do this. This technique allows extensions and adapters to keep the DOM tree structure fully in sync with the underlying data on which adapters operate. As the user changes data in the application, the adapter can change DOM data.

For details on using `DomNode`s and DOM extensions and adapters, see “DOM Nodes and DOM Adapters”.

DOM extensions can perform automatic validation of DOM data as it changes. Some validators are built in, such as checking that a DOM node attribute’s value is between a minimum and maximum value. All you have to do to use such validators is define the appropriate restrictions in your XML Schema. More generally, you can define *DOM validator* classes on your data types to automatically check data as it changes. DOM validators are DOM adapters that check values. Validator extensions listen to node events and are typically defined on the type of the `DomNode` tree’s root, so they can check all nodes in the tree. ATF provides standard DOM validators, such as `UniqueIdValidator`, which checks that an ID attribute in a type always has a unique value in DOM data. You can also use the `DataValidator` class, which is a DOM validator that allows you to define rule classes that perform validation on your data. For example, the classes `NumericMinRule` and `NumericMaxRule` are rule classes that check that the value of a type attribute is between a minimum and maximum value, respectively. For more information, see “Validating DOM Data”.

Data types have attributes, which can be considered properties of objects of these types. Applications often have property editor controls to modify object properties. *Property descriptors* are metadata for class properties that controls like property editors can use. Similarly, *DOM property descriptors* describe properties of DOM types—their attributes and child node types. The DOM provides property descriptor classes for type attributes, child node types, and attributes of child node types. After you create these descriptors, it's easy to implement property editing in your application using ATF property editing control components, such as `PropertyEditor` and `GridPropertyEditor`. For details, see "Defining and Using DOM Property Descriptors".

An application usually saves its data to a file or other persistent storage and can read it back for further user editing. DOM data can be stored in an XML document that uses the same XML schema as the data model. ATF provides two classes, `DomXmlWriter` and `DomXmlReader`, to easily write and read XML data to any .NET stream, such as a file. You can use `DomXmlWriter` to write a DOM tree's `DomNode` data to an XML file, and then use `DomXmlReader` to read the file's data and recreate that entire `DomNode` tree. You can also define your own scheme for reading and writing application data. For details on DOM data storage, see "DOM Persistence".

Terminology Conventions

The term *element* refers to a data element in application data. Every element has a type, and sometimes the term element is used to refer to the element's type. For instance, saying that a type can have child elements means that a type can have child node types. In other words, elements of this type can have child elements. There's a direct mapping between elements and their types.

The term *node* refers to a DOM node, represented by a `DomNode` object.

An *extension* is a DOM extension. *DOM adapter* is short for DOM node adapter. *Adapter* is also used for DOM node adapter.

DOM can refer not only to the DOM data model framework, but to data following the model.

XML Schema means the *XML Schema Definition (XSD)* language.

2 About the Document Object Model

The ATF *Document Object Model (DOM)* is a data modeling, management, and persistence framework. The DOM makes application data management easier and more consistent across multiple applications. With the DOM, you can design, load, store, validate, and manage changes to your application data independently of the rest of your code.

The DOM holds data in memory as an object tree and provides infrastructure for an object API for data, data validation, raising events when data changes, and rolling back changes if necessary. The DOM is the "model" part of ATF's model-view-controller design (MVC), so any GUI controls or applications that use the DOM, including all of those provided by ATF, are automatically notified of changes that occur in application data.

Although the DOM provides many benefits, you don't need to use the DOM to use ATF. You can use any data storage method with ATF, including data binding or "plain old objects".

The classes and interfaces for using the core DOM described in this document are in the `Sce.Atf.Dom` namespace. Most of these classes are contained in the `Atf.Core` assembly, with some in `Atf.Gui` and a handful in `Atf.Gui.WinForms`. In addition, the `Sce.Atf.Rendering.Dom` namespace contains DOM data rendering classes.

Although the term DOM refers to a data modeling framework, this document also uses the term DOM to refer to data that follows the DOM model.

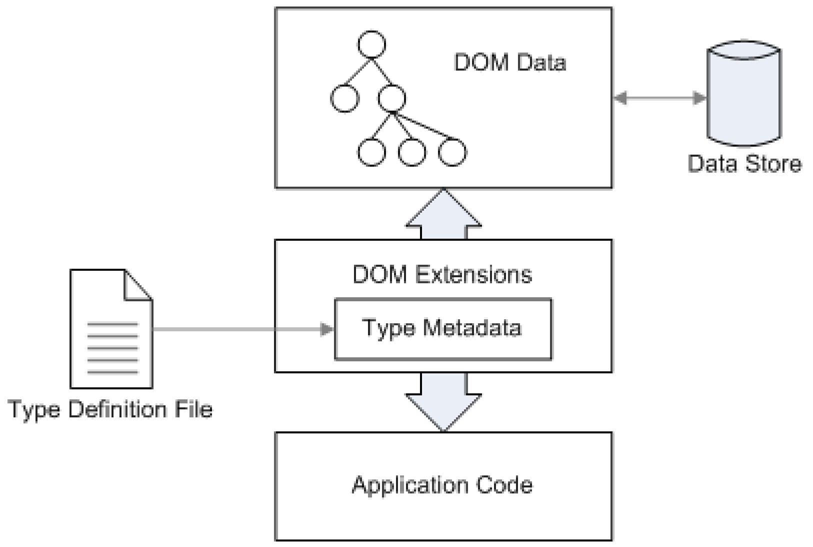
Note: The ATF DOM is unrelated to the W3C DOM. Document Object Model is a term borrowed from XML that describes an in-memory hierarchy of XML elements. A document, in XML terms, is a collection of data with a defined structure. The ATF DOM includes much more than a data representation. In addition, although the ATF DOM was originally based on a number of concepts borrowed from XML and can read and write XML files by default, the DOM is not dependent on XML—you can use any format to specify your data model and store your data.

The Parts of the DOM

The DOM consists of three primary parts, shown in Figure 1.

- The *data model* contains the description of an application's data types. A *type definition file* that describes a data model is loaded into an application at runtime. The type loader generates a set of type metadata objects from that file.
- A *managed tree of DOM nodes* contains actual application data. The DOM includes event management, allowing you to listen to events on any node in the tree and its child nodes. Features for unique naming of nodes, reference tracking, transactions, synchronizing the DOM tree with an underlying document, and name lookup are all included, but optional. The DOM node tree can be read from and written to a persistent data store, such as a file.
- *DOM extensions* define the object API between the application and its data. A DOM node can be cast to the type of the extension's class. Extensions can be any class, but nearly always derive from the `DomNodeAdapter` base class, when they are called *DOM Adapters*. Implement adapters or other extensions and define them for the types in the data model when the data model is loaded. Application code accesses data in DOM nodes through DOM extensions, which provide an object-oriented way to handle application data.

Figure 1 The Parts of the DOM



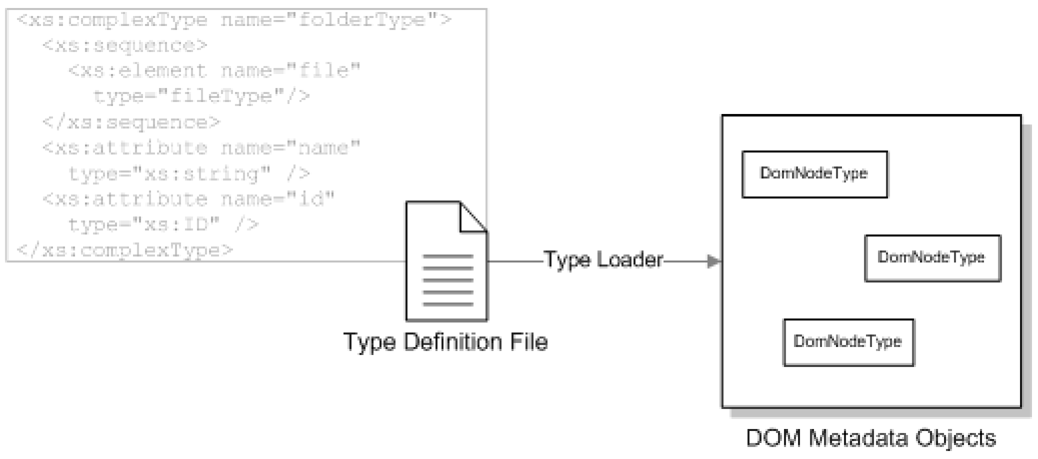
The Data Model

To use the ATF DOM, first define the *data model* for application data, that is, describe its data types, as well as the attributes and relationships between these types (such as parent-child). Data types can also have subtypes. These attributes and relationships can influence what adapter classes you create for types to adapt their data to other types.

Your application's data model is defined in a *type definition file*, external to your application. You can use any type definition language to describe your data model, but the DOM natively supports the XML schema definition language (XSD). An XSD type definition file has a target namespace; an application may have more than one type definition file.

The *type loader* that you create imports the type definition file and generates internal DOM metadata objects, such as `DomNodeType`, for the types in that file, as illustrated in Figure 2. If you're using XML schema for your type definition file, ATF's XML schema type loader base class `XmlSchemaTypeLoader` does most of this work for you. If you use some other type definition language, you must implement your own type loader.

Figure 2 The Type Definition File and the Type Loader



```
<xs:complexType name="folderType">
  <xs:sequence>
    <xs:element name="file"
      type="fileType"/>
  </xs:sequence>
  <xs:attribute name="name"
    type="xs:string" />
  <xs:attribute name="id"
    type="xs:ID" />
</xs:complexType>
```

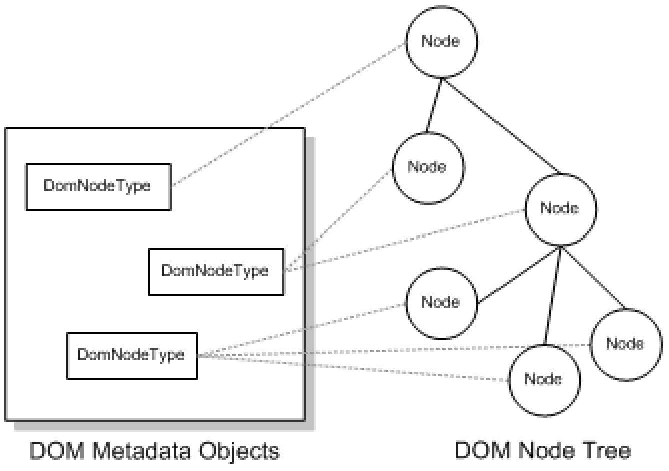
The type loader also defines DOM extensions for types and parses any other information the type definition file contains, such as validator rules and annotations.

For details on using the DOM data model, see "Data Modeling, XML Schemas, and the DOM".

The DOM Node Tree

Application data is stored in the DOM as a tree of individual DOM nodes, instances of the class `DomNode`, as shown in Figure 3. Each DOM node has a metadata type, `DomNodeType`. Each `DomNodeType` is created from a type defined in the type definition file, and specifies the structure of a node: its attributes, child nodes, and extensions.

Figure 3 The DOM Node Tree and the Underlying DOM Metadata Objects



The DOM includes features to manage data in the DOM node tree:

- The DOM node tree can be read from and written to a persistent data store, such as a file. The ATF DOM provides default support for XML data storage, but you can use any format you wish.
- DOM extensions: You can adapt DOM nodes to objects of other types. Several of the following features, such as data validation and uniqueness, are implemented through extensions.
- Event management: You can listen to events for any node in the tree, and receive events for that node and its children.
- Data validation: Validator rules (for example, restrictions on the value of a type's attribute or the number of child elements) can be defined in the type definition file. Data validation occurs when nodes in the tree are added, removed, or changed.
- ID uniqueness: The DOM can ensure that nodes of a given type have unique IDs or names. An optional DOM adapter maintains uniqueness.
- Reference tracking: DOM nodes can reference each other in the tree. An optional DOM adapter tracks references between nodes.
- DOM node trees can represent documents: individual documents or a master document with independently-managed subdocuments. The DOM can ensure that node tree changes mark the document as "dirty", and that Undo/Redo changes are synchronized between the document and data.

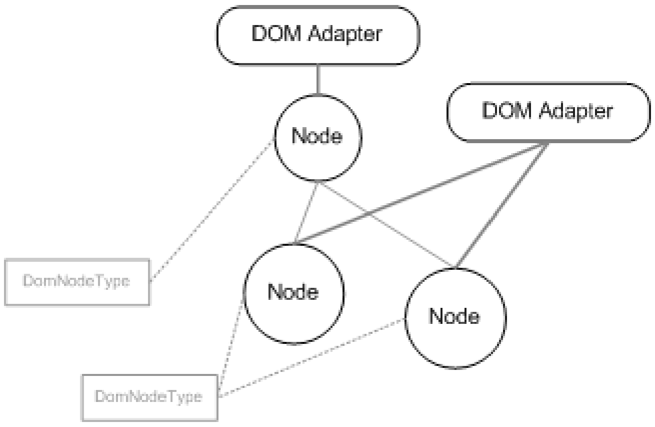
For more information on DOM nodes, see "DOM Nodes and DOM Adapters".

DOM Extensions and Adapters

A *DOM extension* is an object that allows a `DomNode` to be cast dynamically to another type or to track data related events, as indicated in Figure 4. When you *adapt* a `DomNode` to a class using a DOM extension, that class's methods and properties are available to access data represented by the DOM node, just as if the `DomNode` were an instance of an object in the extension's class. Thus DOM extensions provide a flexible, type-safe, C# object API to manipulate data in the DOM node tree. You can design the extension so its API is convenient to work with and so it's easier to change data in response to user editing. You can also use an extension to listen to events on a `DomNode` and its children, to perform data validation, for instance.

An extension can be an object of any type, but DOM extensions are nearly always classes deriving from the `DomNodeAdapter` base class. Extensions that use `DomNodeAdapter` are called *DOM adapters*. In practice, there is no reason to use an extension that is not a DOM adapter.

Figure 4 DOM Nodes and DOM Adapters



You don't directly specify an extension for a DOM node. Instead, you associate the DOM node's type with one or more extensions by defining these extensions for the type at run time. Define DOM extensions and adapters for types in the type loader, as described in "Defining DOM Extensions and Adapters".

DOM adapters often have a one-to-one correspondence with the types in the data model. A simple DOM adapter is often just an object wrapper for its underlying type. For example, a "state" type in a data model could have a corresponding `State` class that serves as its adapter.

You can define multiple DOM extensions for the same type, enabling you to provide different APIs and operations for the same underlying data, to be used in various circumstances. You can create complex adapters that use data across many different DOM node types, listen to different events on those nodes, or hide the DOM tree structure altogether.

When you define a DOM adapter for a type, that adapter applies to that type *as well as all its subtypes*, if any. In other words, a data subtype inherits not only its base type's characteristics, but its base type's adapters as well.

A DOM adapter class can add common behavior through ATF's context interfaces. These interfaces are specific to basic tasks and are intended to be easy to implement. The `INamingContext` interface, for example, is simply a context for objects that can be named. If a DOM adapter implements `INamingContext`, nodes of any type that adapter is defined for can be named. (Named nodes benefit from fast retrieval by name and can be dynamically collected by tree and list controls.) For more information on context interfaces, see *Getting Started with ATF*.

For more information on DOM extensions and adapters, see "DOM Nodes and DOM Adapters".

Workflow

To use the DOM in an application, take the steps described in Table 3 and Table 4 when you implement your application and when your application runs. These steps are discussed in detail in the rest of this document.

At Implementation Time

Table 3 DOM Workflow: Implementation Time

Step	Description	For more Information
1. Define the data model.	The data model defines the set of element data types an application needs. An audio application uses very different data types (stream, channel, buffer) than a vector-illustration application (lines, rectangles, colors).	"Data Modeling, XML Schemas, and the DOM"
2. Create a type definition file for the data model.	The data types in the data model are defined in a type definition file, external to the application. You can use any type definition language you prefer, such as the XML Schema Definition Language (XSD) or one of your own. ATF supports XSD.	"Creating and Loading the Type Definition File" and "Defining the Data Model using XML Schema"
3. Implement a type loader class for the type definition file.	The type loader brings the data model into the DOM at runtime. If you use XML schema to define the data model, the DOM provides an `XmlSchemaTypeLoader` base class you can extend.	"Creating and Loading the Type Definition File" and "Implementing a Schema Type Loader"
4. Implement DOM extension classes for the data model.	DOM extensions and adapters are the glue code between types and data in the DOM node tree and the application, providing a well-structured way for the application to handle various data types. Define DOM extensions and adapters for types in the type loader, right after you load the type definition file.	"DOM Nodes and DOM Adapters"
5. Implement persistent storage.	Choose a file format to save application data in, and create classes to read and write data from the DOM to files. If you store application data in XML, the ATF DOM provides default `DomXmlReader` and `DomXmlWriter` classes you can use.	"DOM Persistence"

At Run Time

Table 4 DOM Workflow: Run Time

Step	Description	For more Information
1. Load the type definition file.	Invoke the type loader with the type definition file when the application initializes.	"Creating and Loading the Type Definition File" and "Loading the XML Schema"

Step	Description	For more Information
2. Define DOM extension and adapter classes for types.	Define DOM extensions and adapters you implemented for their designated types in your data model. Typically you define DOM extensions for types in the type loader class.	"Implementing a Schema Type Loader" and "Defining DOM Extensions and Adapters"
3. Read and write application data from/to files as needed.	Use your data reader and writer classes to handle reading and writing persistent application data from and to files.	"DOM Persistence"

3 Data Modeling, XML Schemas, and the DOM

Data modeling is the process of defining the data types the application uses in its data and describing how types relate to each other. With the DOM, you describe your data model in an external type definition file. When your application loads that file, it generates a set of internal metadata objects that describe your application's data types.

By default, ATF supports the *XML Schema Definition (XSD)* language, or *XML Schema* for short, as the type definition language to describe the application's data model. ATF provides utilities and classes for loading XML schemas and generating DOM metadata objects. You can use a type language other than XML schema to describe your data model (DDL, IDL, or custom languages), but then you need to implement your own type loader or other utilities to import those type definitions into the DOM. You would also need to create your own mechanisms for ATF DOM facilities that are based on XSD, such as data validation rules.

This chapter covers these topics:

- "About Data Modeling": General overview of how data modeling is used in the DOM, how the type definition file is loaded, and how DOM metadata objects are created from that file.
- "About XML Schema and the DOM": Learn how XML schemas are used in the ATF DOM for data modeling, as well as tools and classes available in the DOM for using XML schemas.
- "Data Modeling Key Classes": A summary of the most important classes and interfaces described in this chapter.
- "Defining the Data Model using XML Schema": Learn how to create an XML schema file to describe types in the data model, as well as how XML schema types map to DOM metadata objects.
- "Using DomGen to Generate a Schema Stub Class": The `DomGen` utility generates a *schema stub class* from an XML schema file. The application can use this stub class to access DOM metadata objects for the application's data model.
- "Implementing a Schema Type Loader": The type loader brings the types defined by the schema into the application at runtime. If you are using XML schema for your data model, you can derive from the `XmlSchemaTypeLoader` class to create a schema type loader class.
- "Loading the XML Schema": Bring it all together by invoking the schema type loader when the application starts, so the schema is loaded at runtime.

A complete tutorial of the XML Schema Definition language is beyond the scope of this Guide. The topics in this section assume you have a basic background in XSD concepts and terminology, including simple and complex types, elements, attributes, namespaces, and URIs. If you are unfamiliar with XML schema concepts, W3Schools has a tutorial on XSD at http://www.w3schools.com/schema.

About Data Modeling

The data modeling process specifies data types and their relationships. This is not specific to the DOM or to ATF; all applications, no matter how small, usually have some sort of model for data the application handles.

The ATF DOM data model defines data types in a separate type definition file. That file is loaded into the application at runtime using a type loader. The type loader, which you also create, reads the types in the type definition file, generates internal metadata objects for those types, defines the extensions for those types, and provides access to any other data provided by the type definition file, such as *annotations* that provide additional information about types. (For details, see "Annotations".)

By default, ATF uses an XML schema, i.e., an XSD, for its type definition file, and provides infrastructure to handle loading and managing XML schema. You can use any alternate language you like as a type definition language, but then you need to implement your own type loader to import your types into the DOM.

The types you define in your data model are the types of the nodes in the DOM node tree that holds your application data. You can adapt DOM node data types to your own or to ATF classes, and this allows you to manipulate DOM node data as instances of these classes. To learn more, see "DOM Nodes and DOM Adapters".

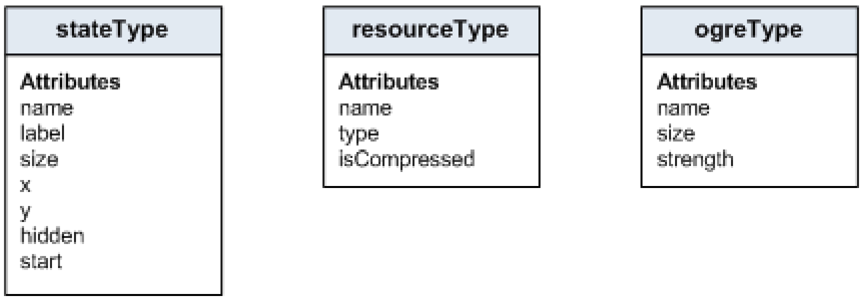
Data Types and Entity-Relationship Modeling

One general way to look at a data model is in terms of entities and the relationships between them: *entity-relationship modeling*. Entity-relationship modeling can help you describe your data model in a generic, abstract sense. The DOM includes features to represent entities and their relationships.

Entities and Attributes

An *entity* in a data model is a self-contained piece of information, such as a state in a state machine diagram. In a DOM, each node can represent an entity. The components of an entity are its *attributes*, such as its name, size, or position. Entities and attributes correspond to elements and their attributes, or objects and their properties, as illustrated in Figure 5.

Figure 5 Entities and Attributes

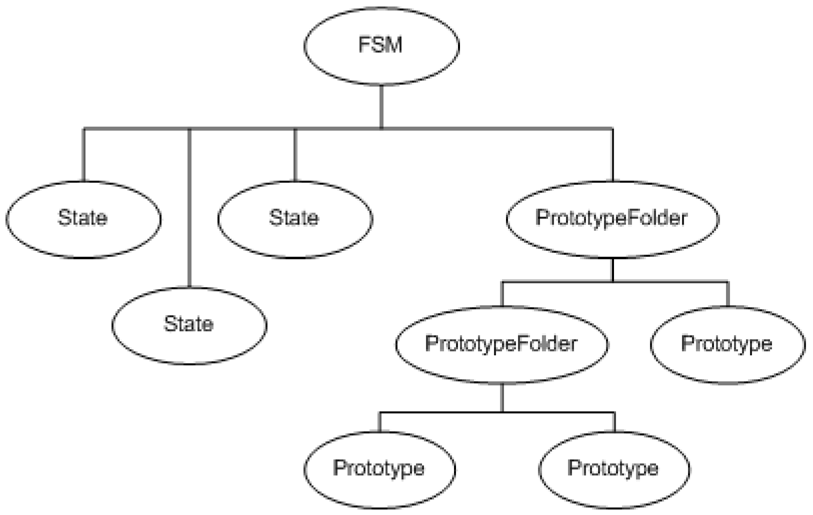


As in class design, data types can inherit from and extend other, more general types, by adding attributes, for instance.

Relationships

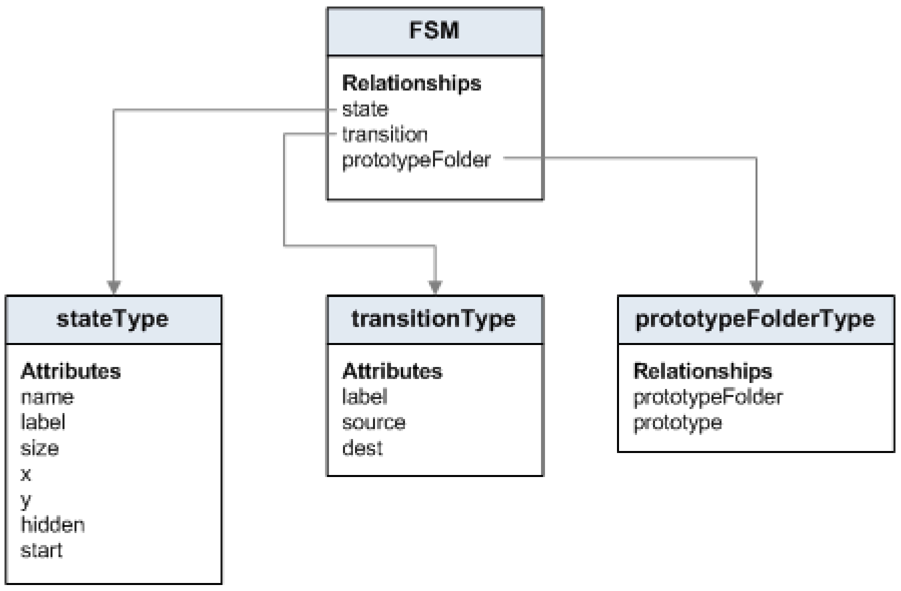
The entities in your data model are connected through relationships. Because the DOM is a managed object graph—a tree of object nodes—the most common kind of relationship between entities is the *parent-child* relationship. The containing element is the parent, and the contained elements are the children. In this way, elements are arranged in a tree, and can be traversed from parent to child and in reverse, as shown in Figure 6.

Figure 6 Parent-Child Relationships



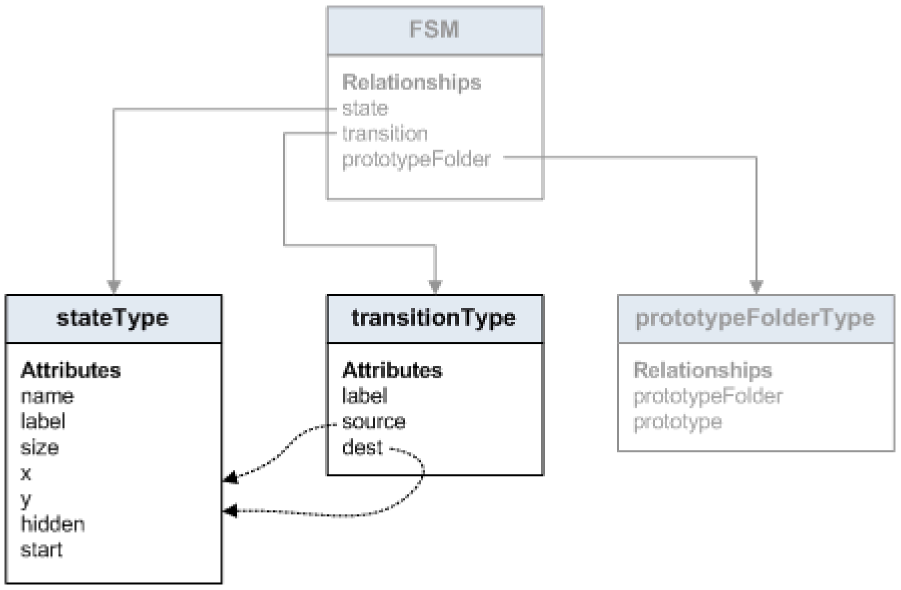
Child entities are defined as relationships, *distinct* from attributes, as indicated in Figure 7, which corresponds to the entities in Figure 6.

Figure 7 Child Entities as Relationships



References are relationships between arbitrary elements—where the first element is not necessarily a parent or child of the second. A reference can be to some other object in the DOM node tree itself, or even to an external file or other asset. (It is useful to make references to objects that should be lazily-loaded only when needed.) References are usually defined by attributes, as illustrated in Figure 8, where “source” and “dest” attributes of a “transitionType” point to a “stateType” object.

Figure 8 References as Attributes



ID Attribute

Major data types often include an ID attribute to uniquely identify objects of these types. This unique identifier is called the *ID attribute*. Not all types require an ID attribute; in general, only those objects that are named or accessed directly by your code need an ID attribute. The DOM manages the uniqueness of the ID attribute's value for you; you choose which types have unique IDs.

Note that because the DOM manages the uniqueness of the ID attribute's value, you should make ID attributes *internal* rather than visible to users. For example, don't use the ID as an object's label. Because the DOM changes the ID attribute's value to ensure object uniqueness, a changing user-visible value may confuse users.

For more information, see "ID Attribute".

Restrictions

Data validators in the DOM enable you to define rules and restrictions that automatically check data. For example, a size attribute can be defined with a rule that limits the value to be less than 1000. Or you could restrict the number of children a type contains: must have at least one but no more than three.

The DOM supports a number of restrictions based on XML schema restriction facets, described in further detail in "Validating DOM Data". You can also write DOM validators to create your own restrictions. As you define your data model, consider not only types of data, but restrictions on data.

Annotations

In addition to types defined in the data model, you can create *annotations* in the type definition file. Annotations can be used to specify additional metadata about a type, such as:

- The icon used to represent a data type in a palette.
- The image used for a proxy object in a 3D view.
- Category or sorting information for data types.
- Display names or other property editor information for attributes.
- Tooltip information for types or properties.
- Controls used to edit properties.

If an application provides support for an annotation, such annotations can easily be changed or added to a type definition file without needing to change or recompile application code, because the type loader supplies the updated annotation information to the application at run time.

For more information, see "Annotations".

DOM Metadata Objects

When a type definition file is loaded, types defined in that file are converted to in-memory DOM metadata objects. ATF defines two primary types of metadata classes:

- `DomNodeType`: type of a `DomNode`. A `DomNodeType` can, in turn, contain attributes, child node types (types of children of a node of this type), and DOM extensions.
- `AttributeType`: describe a `DomNode`'s attribute, which can be a primitive value or an array of those values. The `XmlAttributeType` class, derived from `AttributeType`, describes attribute types specific to XML schema attributes.

A third general class of metadata, *field data*, describes the fields inside `DomNodeType`:

- `AttributeInfo`: Metadata about a `DomNodeType`'s attributes, which includes the attribute type `AttributeType` and any restrictions the attribute was defined with, such as its maximum value. The `XmlAttributeInfo` derived class adds information specific to XML schema attributes.
- `ChildInfo`: Metadata about a child of a `DomNodeType`. Child information includes the child's node type and whether the child is a list of nodes. `ChildInfo` contains any restriction rules, such as limits on the number of children. `ChildInfo` can serve as an identifier for a type of node child.
- `ExtensionInfo`: DOM extensions defined for this type, nearly always DOM node adapters.

When the application loads the type definition file, each of the types defined in that file converts to the appropriate internal DOM metadata object. If you're using XML schema, the `XmlSchemaTypeLoader` base class does this conversion for you. If you're using some other type definition language, you have to create these metadata objects yourself.

Creating and Loading the Type Definition File

You define and store the representation of the data model in an external file, the type definition file, which is loaded into the application during initialization by the type loader. By default, the type definition file is an XML schema (`.xsd`) file. Keep in mind that in addition to the actual type definitions, you may want your type definition file to contain property or other metadata information in annotations that can be added or modified by non-programmers. Consider the requirements for your types and users of those types when you choose a type definition language. The type definition file is typically in a text format that can be edited with tools external to the application, or edited manually.

If you use XML schema for your type definition file, much of the work of writing a type loader has been done for you by the `XmlSchemaTypeLoader` base class (in the `Sce.Atf.Dom` namespace). For details on this class, see "Implementing a Schema Type Loader". If you use your own type definition file, you need to implement a type loader yourself, although the code in `XmlSchemaTypeLoader` provides a starting point.

Tasks the type loader is responsible for include:

- Opening, reading, validating, and parsing contents of the type definition file.
- Converting each entity and attribute to its corresponding DOM metadata type.
- Creating validator objects for any restriction rules defined for your types.
- Assigning the ID attribute, if defined.
- Reading and parsing annotations, if any.
- Defining extensions or DOM adapters for types.
- Define *utility adapters* for types that perform various tasks, such as ID and reference validation.

About XML Schema and the DOM

The XML Schema Definition Language (XSD) specifies the structure of an XML document. XML schemas can also be used for type definitions to describe your application data model, and ATF provides support and tools for using XML schemas by default.

Note: Keep in mind that your type definition file is *independent* of the persistence format used for application data. Even if you use XML Schema for your data's type definition file, you don't have to use XML to store that data. For more information on storing data, see "DOM Persistence".

Use XML schema to describe the application's data model, including data types, attributes, and relationships between types. The schema can also include annotations to add additional properties about data or hints to the application about how to display or edit that data. For example, an annotation could indicate that color-related attributes should be modified with a particular color editor control. As with all type definition files, specifying data types with an XML schema enables you to make changes to that schema without necessarily having to change your application code or even recompile your application.

Types defined by the XML schema map directly to DOM objects the type loader creates. For example, complex types in the schema become `DomNodeType` objects. Simple types, as well as arrays of those types, become `AttributeType` objects. Restriction facets in the schema convert to the appropriate DOM validator rule class objects, complex type restrictions can convert to validator objects, and extensions convert to extension class objects. "Defining the Data Model using XML Schema" describes in detail how to create an XML schema file to define the application data model and how XML schema types map to DOM metadata objects.

To use an XML schema in an ATF application:

(1) Run the `DomGen` utility to generate a schema stub class from the XML schema.

(2) Derive from `XmlSchemaTypeLoader` to implement a schema type loader.

(3) Invoke the schema type loader when the application initializes.

The `DomGen` utility parses the XML schema and generates a schema stub class. This stub class contains a set of class and property definitions you use in your schema type loader and in your DOM extension implementations to access DOM metadata. If the application's data model is stable, you should only need to run `DomGen` once. If you change any of the types or type names in the schema, you need to re-run `DomGen` to regenerate the stub file. For details, see "Using DomGen to Generate a Schema Stub Class".

As you learned in the previous section, "About Data Modeling", you use a type loader to import the type definitions and other metadata for the application's data model into an application at runtime. If you use XML schema as the type definition language, much of the work of creating a type loader, including loading and parsing the XML schema, has been done in the `XmlSchemaTypeLoader` base class.

To create a schema type loader, derive it from `XmlSchemaTypeLoader` and then:

- Handle any annotations defined by the schema to create property descriptors, palette data, tooltips, or other information. For more information, see "Annotations".
- Define the extensions or DOM adapters for the types defined in the schema. For details, see "Defining DOM Extensions and Adapters ".

For more information, see "Implementing a Schema Type Loader".

Finally, you must actually invoke your schema type loader with the path to your schema in your application, so that the schema is loaded when the application runs. For more information, see "Loading the XML Schema".

Data Modeling Key Classes

The `Sce.Atf.Dom` namespace includes classes for describing metadata in the DOM, outlined in Table 5.

Table 5 Key Classes in Sce.Atf.Dom for the Data Model Metadata

Class	Description
`NamedMetadata`	Abstract base class for DOM metadata. All metadata classes listed here ultimately derive from this class.
`DomNodeType`	Describes `DomNode` types. Contains attributes, children, and extensions, all described by `FieldMetadata`. Extends `NamedMetadata`.
`FieldMetadata`	Abstract base class for `DomNodeType` field metadata: attributes, children, and extensions. Extends `NamedMetadata`.
`AttributeInfo`	Metadata for an attribute in a `DomNodeType`. `AttributeInfo` types are defined by `AttributeType`. Extends `FieldMetadata`.
`AttributeType`	Describes the type of attribute metadata, i.e., an `AttributeInfo`. Attributes can be primitive types or arrays of those types. Extends `NamedMetadata`.
`ChildInfo`	Metadata describing child elements, containing an identifying name and `DomNodeType` that applies to the child nodes. Often used to identify kinds of child nodes. Extends `FieldMetadata`.
`ExtensionInfo` `ExtensionInfo<T>`	Extension for a DOM data type `T`. In theory, extensions can be any type, but in practice are derived from `DomNodeAdapter`. Extends `FieldMetadata`.
`AttributeTypes`	`Enum` for attribute simple types, which are primitive types and arrays of primitives, such as `Boolean` and `Int8Array`.

If you are using XML schema for the type definition file, the DOM includes a number of additional classes for describing XML attribute metadata and for loading and accessing DOM metadata objects described by a schema. In addition, two class names, `Schema` and `SchemaLoader`, are used for XML schema-related classes in the ATF sample applications by convention. Table 6 describes these classes.

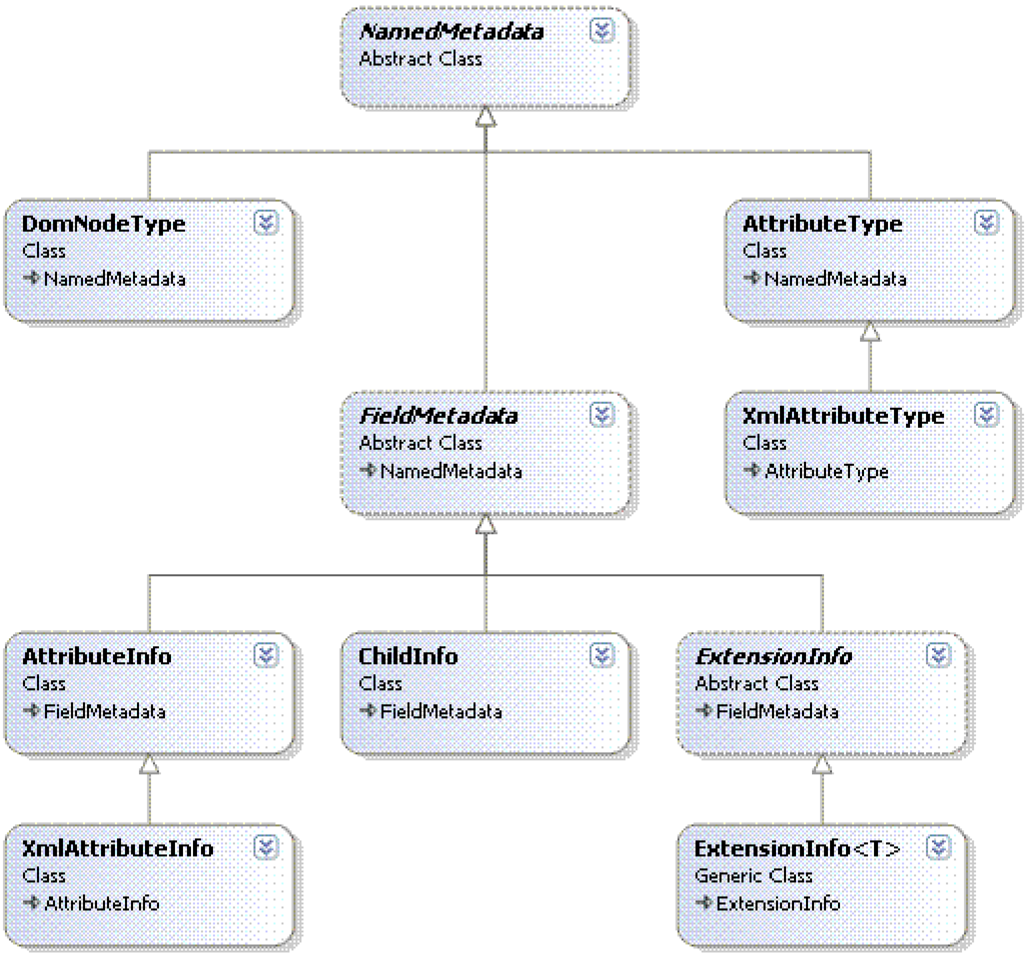
Table 6 Key Classes in Sce.Atf.Dom for XML Schema

Class	Description
`XmlAttributeInfo`	Metadata for an XML attribute in a `DomNodeType`, which adds information from XML schema. Attribute types are defined by `XmlAttributeType`. Extends `AttributeInfo`.
`XmlAttributeType`	Describes the type of XML attribute metadata, i.e., an `XmlAttributeInfo`. Extends `AttributeType`.
`XmlSchemaTypeCollection`	A collection of the DOM metadata objects—`DomNodeType`, `ChildInfo`, and `AttributeInfo` objects—associated with the target namespace in an XML schema file.

Class	Description
`XmlSchemaTypeLoader`	Base class for schema type loaders that creates `DomNodeTypes`, parses annotations, and creates attribute rule objects. Derive from this class to create a schema type loader for your application.
`Schema`	Conventional name for the schema stub class generated from the XML schema type definition file by the `DomGen` utility.
`SchemaLoader`	Conventional name for the class you define for your schema type loader. Derives from `XmlSchemaTypeLoader`.

The key classes that define metadata in the DOM—`DomNodeType`, `AttributeType`, `AttributeInfo`, and `ChildInfo`—all derive from the abstract base classes `NamedMetadata` and `FieldMetadata`. For the class hierarchy of these metadata types, see Figure 9.

Figure 9 DOM Metadata Types



Defining the Data Model using XML Schema

ATF includes built-in support for using the XML Schema Definition Language (XSD) as a type definition language. ATF includes base classes and utilities to help load and parse the schema type files and to convert types in your schema to DOM metadata objects.

An XML schema includes the following features for defining data types:

- "Types and Attributes": main elements of XML schema.
- "Simple Types": primitive types, such as integers and floats, dates and times, and strings.
- "Complex Types": elements with attributes and/or child elements.
- "Inheritance": types can inherit from base types.
- "ID Attribute": unique identifier attribute in type.
- "Parent and Child Elements": types can have child node types.
- "References": attributes can reference other elements.
- "Annotations": custom properties or additional information markup.

This section contains schema examples to illustrate these features.

Note that the conversions described in this section are of schema elements to metadata objects in ATF—not type conversions done by DOM extensions.

Types and Attributes

The main elements of an XML schema are types and their attributes, and these describe all the application data's types. Types can be simple or complex. Attributes have only a simple type. For information on data types, see "Simple Types" and "Complex Types".

In this discussion, the term "element" refers to an XML element in the schema defining a type, and may also be used to refer to the type itself.

The types in an XML schema are the entities in the data model. Attributes in an XML schema play the same role as attributes in entity-relationship modeling do: they represent properties or components of an entity.

Relationships between types come in two forms: parent-child and references, described in "Parent and Child Elements" and "References". Also see "Data Types and Entity-Relationship Modeling".

Simple Types

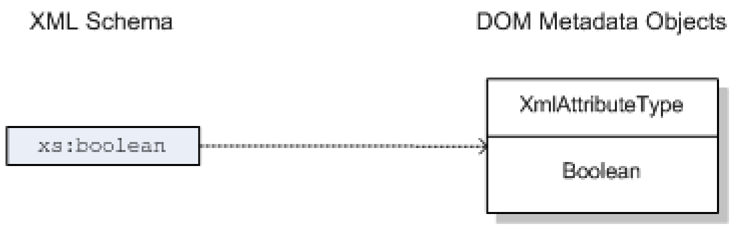
Simple types are primitive types, such as integers and floats, dates and times, and strings—as well as special types like IDs and URIs. There is a defined set of built-in simple types defined by the XML schema specification. Simple types primarily describe attributes of elements. A type itself may have a simple type, but only if it contains no other content: no child node types or attributes.

You can create simple type definitions that include lists of primitive simple types (`<xs:list>`) and value restrictions (`<xs:restriction>`), such as minimum and maximum values and value enumerations.

When the XML type loader reads the XML schema file, all the simple type definitions are converted to `XmlAttributeType` DOM metadata objects. Simple type definitions with restrictions convert to DOM validator rules. Other metadata, such as attribute default values and annotations, is preserved in metadata objects.

Schema simple types, such as `<xs:boolean>` and `<xs:integer>`, correspond to .NET types and are converted to `XmlAttributeType` objects of the same or similar type, as shown in Figure 10. An `XmlAttributeType` object is associated with an `AttributeTypes` enumeration, which lists the supported DOM attribute types. (For information on how XML schema types convert to .NET primitive types, see the table at Data Type Support between XML Schema (XSD) Types and .NET Framework Types.)

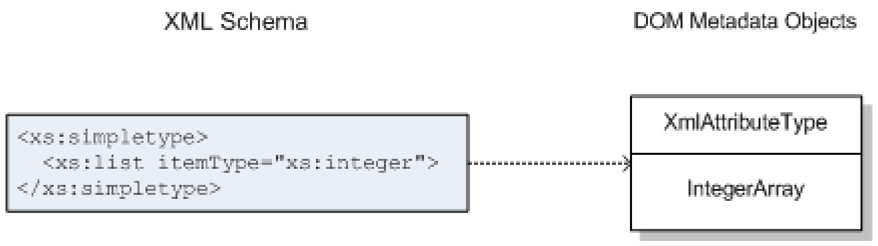
Figure 10 Simple Type Conversions by the XML Type Loader



Some simple types get special handling. For example, `<xs:ID>` indicates a unique ID attribute for an element, and a validator DOM extension is created to handle it. References also receive special handling; both `<xs:IDREF>` and `<xs:anyURI>` are reference types. For information on how these types are handled, see "References".

List types (`<xs:list>`) convert to `XmlAttributeType` objects with associated `AttributeTypes` that specify arrays of the underlying simple type, as illustrated in Figure 11. The `AttributeTypes` enumeration includes separate types for arrays of simple types, such as `BooleanArray` and `Int16Array`.

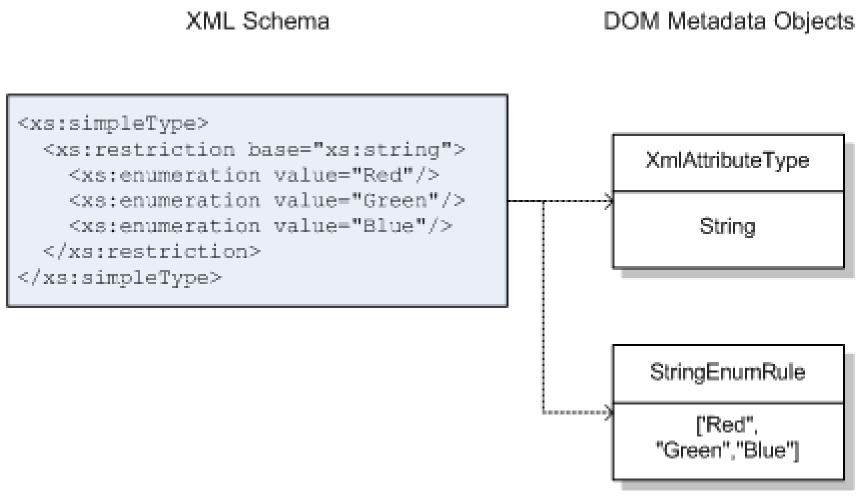
Figure 11 Simple Type Conversions With Lists



```
<xs:simpletype>
  <xs:list itemType="xs:integer">
</xs:simpletype>
```

Simple types with restriction facets, such as `<xs:maxExclusive>`, `<xs:minInclusive>`, and `<xs:enumeration>`, convert to `XmlAttributeType` objects of the appropriate simple type. In addition, the XML schema type loader creates DOM data validator rule objects to manage the restriction rules, as shown in Figure 12. If you define the `DataValidator` class for a type in your type loader, data validation occurs automatically when the value of that simple type changes in application data. For more information, see "Validating DOM Data".

Figure 12 Simple Type Conversions With Restriction Rules



```
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Red"/>
    <xs:enumeration value="Green"/>
    <xs:enumeration value="Blue"/>
  </xs:restriction>
</xs:simpleType>
```

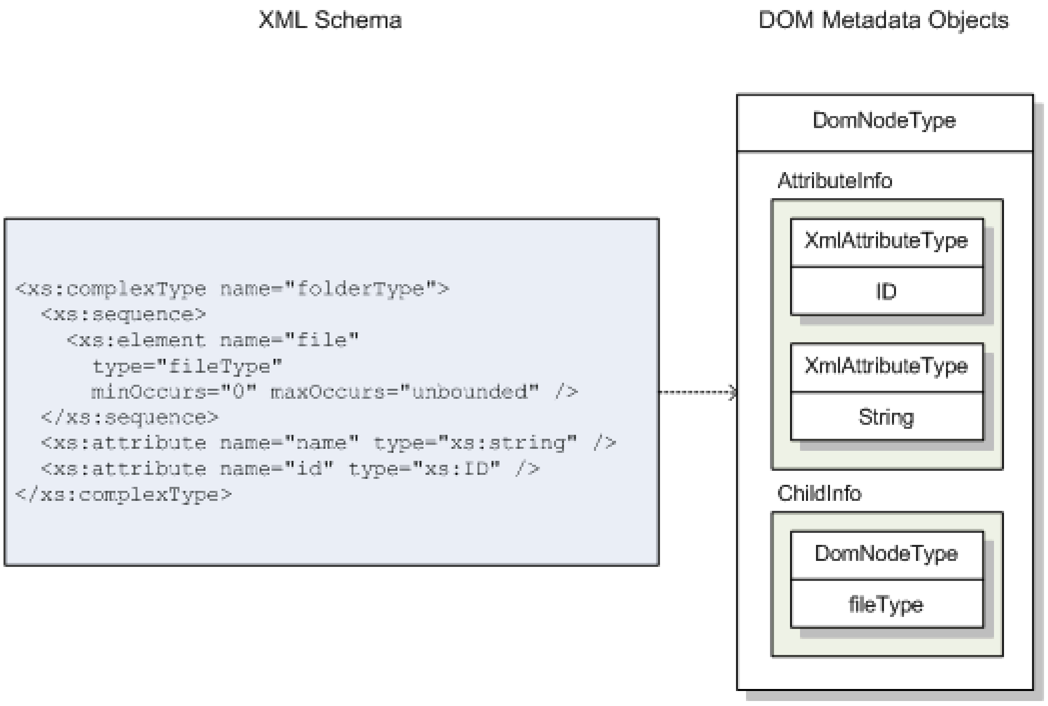
Complex Types

Complex types describe elements with attributes and/or child elements. In the data model, a complex type describes an entity, and the type's attributes are the attributes of that entity. Defining child elements for a complex type defines a relationship between that type and its child node types.

A complex type may have a base type (specified in the "base" attribute of an `<xs:extension>` tag) that is extended to create another complex type. This form of type inheritance in the XML schema not only enables you to create modular, reusable components within the schema itself, but is relevant to DOM adapters for types. A type inherits not only its base type's characteristics, but its base type's adapters as well. For details, see "Inheritance".

When the XML type loader reads the XML schema file, complex type definitions convert to `DomNodeType` objects, which in turn contain all the attribute and child metadata defined by that type, shown in Figure 13. In this example, the `DomNodeType` contains both an `AttributeInfo` and a `ChildInfo`. The `AttributeInfo` has two `XmlAttributeInfo` objects for the two `<xs:attribute>` tags in the schema, one an ID type and the other a string type. The `ChildInfo` contains a `DomNodeType` for the child node type, "fileType", in the `<xs:sequence>` tag in the schema.

Figure 13 Complex Type Conversions by the XML Type Loader



```
<xs:complexType name="folderType">
  <xs:sequence>
    <xs:element name="file"
      type="fileType"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" />
  <xs:attribute name="id" type="xs:ID" />
</xs:complexType>
```

The `DomNodeType` object contains the parts of the complex type defined by the schema, including its attributes (`AttributeInfo`), its children (`ChildInfo`), its ID attribute (`idAttribute`), and parts added by the type loader, such as extensions (`ExtensionInfo`). A `DomNodeType` for a complex type contains at least one of these parts.

Inheritance

Implement type inheritance by creating type extensions in the schema (`<xs:extension>`). A complex type definition can have a base type, which extends that base type and adds attributes, children, or both.

Elements can be declared with the Boolean attribute `abstract`. An abstract element cannot be used by itself, but must be subtyped to be used. In the following example, "moduleType" is an abstract type that is extended by the type "subCircuitInstanceType":

```
<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>

  <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>
```

This form of type inheritance in the schema not only enables you to create new types within the schema, but is useful when you design DOM adapters for your schema types, because a type inherits its base type's adapters. For more information, see "DOM Nodes and DOM Adapters".

ID Attribute

To mark a type as having an unique identifier, give it an attribute with the `<xs:ID>` simple type:

```
<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>
```

In this example, the attribute with the name "name" is the ID attribute for the "stateType" complex type. Note that for this type, the "name" attribute is distinct from the "label" attribute. The "name" attribute is an internal identifier, invisible to users, whereas the "label" attribute is used as the display name for the object. Note that ID attributes should be internal, not visible to users.

You do not have to manage the uniqueness of the ID attribute's value yourself. ATF includes two ID validator classes, `UniqueIdValidator` and `UniquePathIdValidator`, which ensure that objects of a specific type have a unique ID value or a unique path in the DOM node tree, respectively. To use one of them, define the validator as a DOM extension in the schema loader for the root type. For more information, see "OnSchemaSetLoaded() Method" in "Implementing a Schema Type Loader" and "Validating DOM Data".

Parent and Child Elements

The most common relationship between elements in an XML schema is parent-child. A complex element type definition can contain other types as child node types. These types—both the parent and children—convert to `DomNodeType` objects when the schema is loaded by the schema type loader.

You can design your schema with a single root element containing all your other data types, or with multiple root elements. In the DOM, the `DomNodeType.BaseOfAllTypes` static property returns the root `DomNodeType` that is the root for all `DomNodeTypes`. (The name of this root-of-all-types type is "`Sce.Atf.Dom.Object`".) The `BaseOfAllTypes` property is useful to define general-purpose DOM adapters that apply to *all* types in the data model.

The following schema example, from the Fsm (Finite State Machine) Editor sample, shows the "fsmType" and its child node types, defined in the `<xs:sequence>` tag":

```
<!--
An FSM contains states, transitions among them, annotations, and a prototype
folder hierarchy.
-->
<xs:complexType name ="fsmType">
```

```
      <xs:sequence>
        <xs:element name="state" type="stateType" minOccurs="0"
          maxOccurs="unbounded"/>
        <xs:element name="transition" type="transitionType" minOccurs="0"
          maxOccurs="unbounded"/>
        <xs:element name="annotation" type="annotationType" minOccurs="0"
          maxOccurs="unbounded" />
        <xs:element name="prototypeFolder" type="prototypeFolderType" />
      </xs:sequence>
    </xs:complexType>

    <!--
    The root element of an FSM document.
    -->
    <xs:element name="fsm" type ="fsmType"/>
```

In this schema, the "fsmType" has child node types of "stateType", "transitionType", "annotationType", and "prototypeFolderType". This means that a DOM node of type "fsmType" can have child nodes of types "stateType", "transitionType", "annotationType", and "prototypeFolderType". Note that the schema also defines a single root element of type "fsmType".

As the previous example also shows, you can limit the number of child elements an element can contain with `minOccurs` and `maxOccurs` restriction facets:

```
<xs:element name="state" type="stateType" minOccurs="0"
  maxOccurs="unbounded"/>
```

These restrictions are converted to DOM data validator rule extension objects (`ChildCountRule` in this example) by the schema type loader. Validation occurs when child DOM modes are added to or removed from the parent node, as when a user edits application data. For more information, see "Validating DOM Data".

References

ATF uses XML schema to define two kinds of references:

- Attributes of type `<xs:IDREF>` contain references to other elements that have a unique identifier, that is, that include an `<xs:ID>` attribute. For more information on identifiers, see "ID Attribute".
- Attributes of type `<xs:anyURI>` contain references to resources external to the current document, such as assets or subdocuments, specified by a URI.

ATF includes a validator extension for managing references, `ReferenceValidator`. This extension validates both internal and external references and raises events when external references are resolved (loaded) or de-resolved (unloaded). Define this validator for types that use references in the schema. For information on defining extensions, see "OnSchemaSetLoaded() Method" in "Implementing a Schema Type Loader".

Annotations

All components of an XML schema—simple types, complex types, elements, and attributes—can be annotated with custom properties or additional information that is not otherwise specified by the XML schema itself.

Annotations are placed in a type definition in an XML Schema. Annotations themselves are defined inside the standard XSD `<xs:annotation>` and `<xs:appinfo>` tags. A type can contain any number of annotations or none at all. You are not required to use annotations.

Each kind of annotation has its own tag name and contains attributes specifying information for that kind of annotation. In the following example from the Timeline Editor sample's schema file, the "markerType" definition contains three annotations with two different tags, `<scea.dom.editors />` and `<scea.dom.editors.attribute />`:

```
<xs:complexType name="markerType">
  <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,Atf.Gui"
        converter="Sce.Atf.Controls.PropertyEditing.IntColorConverter" />
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:complexType>
```

You can use an annotation to specify whatever you want, defining the tag name and attributes you need. Annotations are read as-is when the XML schema is loaded by the schema type loader. In general, you need to write code to process the annotations you create, but a few annotation types already have programming support in ATF.

Previous versions of ATF supported a defined set of annotations with specific names and parameters. ATF 3 offers support for some of the legacy annotations from ATF 2, mainly for creating property descriptors, as described later.

To parse and interpret XML schema annotations, you must implement the `ParseAnnotations()` method in your schema loader and call the `PropertyDescriptor.ParseXml()` method there. For more information, see "ParseAnnotations() Method" in "Implementing a Schema Type Loader" and "Defining Property Descriptors in XML Schema Annotations".

The standard annotations ATF 3 provides support for are:

- `idAttribute`: specify a unique ID attribute.
- `scea.dom.editors.attribute`: create a property descriptor for a type's attribute.
- `scea.dom.editors.child`: create a property descriptor for a child node type.
- `scea.dom.editors.enumeration`: create editor or type converter for enumerated values.

In the previous example, the `scea.dom.editors` annotation is a custom annotation that the application needs to provide its own support for.

The annotations `scea.dom.editors.attribute` and `scea.dom.editors.child` both create property descriptors, which contain metadata about type attributes that property editor controls can use. In the following descriptions, these annotation's attributes are essentially the same as the DOM property descriptor parameters described in Table 15. This is no accident, because the attributes and the parameters are both describing property descriptors' attributes.

For more information attributes and properties, see "About DOM Property Descriptors".

idAttribute: ID Attribute

This annotation is supported in the base `ParseAnnotations()` method. You can use this annotation to specify that the type has a unique ID. Its format is:

```
<idAttribute
  name="[id-name]"
/>
```

Its attribute is:

- `id-name`: the unique id name.

The ID attribute can also be specified with the attribute `<xs:id>` in an XML Schema. If you are using an XML Schema, you should use the `<xs:id>` attribute rather than this annotation. For more information, see "ID Attribute".

scea.dom.editors.attribute: Attribute Property Descriptor

This annotation creates an `AttributePropertyDescriptor` property descriptor for a type's attribute, and is supported by the `ParseXml()` method. It may optionally specify an editor for the property. For more information on `AttributePropertyDescriptor`, see "About DOM Property Descriptors".

Its format is:

```
<scea.dom.editors.attribute
  name="[attribute-name]"
  displayName="[display-name]"
  description="[description]"
  category="[category]"
  readOnly="[read-only]"
  editor="[UITypeEditor]:[editor-params]"
  converter="[TypeConverter]:[converter-params]"
/>
```

Its attributes are:

- `attribute-name`: the attribute's name, optionally qualified by an element path.
- `display-name`: the property name, displayed in the editor.
- `description`: an optional full description of the attribute.
- `category`: an optional category of the attribute, if any.
- `read-only`: an optional Boolean indicating whether or not the property is read-only.
- `UITypeEditor`: an optional fully qualified name (by namespace) of a `UITypeEditor` control for editing, followed by its assembly name.
- `editor-params`: an optional parameter string to configure the editor used to edit the property.
- `TypeConverter`: an optional fully qualified name (by namespace) of a `TypeConverter` class used for this property.
- `converter-params`: an optional parameter string to configure the type converter.

The following examples are from the Timeline Editor sample.

This annotation example defines a color property with a color editor control and type converter:

```
<scea.dom.editors.attribute
  name="color"
  displayName="Color"
  description="Display Color"
  editor="Sce.Atf.Controls.PropertyEditing.ColorEditor,Atf.Gui"
  converter="Sce.Atf.Controls.PropertyEditing.IntColorConverter" />
```

This example demonstrates an annotation providing an editor control for file names with editor parameters:

```
<scea.dom.editors.attribute
  name="ref"
  displayName="File Name"
  description="File name of timeline reference"
  editor="Sce.Atf.Controls.PropertyEditing.FileUriEditor,Atf.Gui.WinForms:
    Timeline files (*.timeline)|*.timeline" />
```

scea.dom.editors.child: Child Property Descriptor

This annotation creates a `ChildPropertyDescriptor` property descriptor for a child node type and is supported by `ParseXml()`. It may optionally specify an editor for the property. For more information on `ChildPropertyDescriptor`, see "About DOM Property Descriptors".

Its format is:

```
<scea.dom.editors.child
  name="[attribute-name]"
  displayName="[display-name]"
  description="[description]"
  category="[category]"
  readOnly="[read-only]"
  editor="[UITypeEditor]:[editor-params]"
  converter="[TypeConverter]:[converter-params]"
/>
```

These attributes are:

- `attribute-name`: the child attribute's name, optionally qualified by element path.
- `display-name`: the child name, displayed in the editor.
- `description`: an optional full description of the attribute.
- `category`: an optional category of the attribute, if any.
- `read-only`: an optional Boolean indicating whether or not the property is read-only.
- `UITypeEditor`: an optional fully qualified name (by namespace) of a `UITypeEditor` control for editing, followed by its assembly name.
- `editor-params`: an optional parameter string to configure the editor used to edit the property.
- `TypeConverter`: an optional fully qualified name (by namespace) of a `TypeConverter` class used for this property.
- `converter-params`: an optional parameter string to configure the type converter.

This example describes a "nameAttribute" attribute of a "trackChild" child:

```
<scea.dom.editors.child
  name="nameAttribute"
  displayName="trackChild"
  description="Track child name" />
```

scea.dom.editors.enumeration: Set Enumeration

This annotation creates an editor or a type converter for a set of enumerated values to use with a property descriptor. `ParseXml()`supports this annotation. It has the format:

```
<scea.dom.editors.enumeration
  name="[attribute-name]"
  displayName="[display-name]"
/>
```

Its attributes are:

- `attribute-name`: the attribute's name, optionally qualified by element path.
- `display-name`: the property name displayed in the editor.

Each annotation specifies one of the enumeration values.

This example shows an enumeration with three possible values:

```
<scea.dom.editors.enumeration name="Adequate"
  displayName="Adequate, good enough"/>
<scea.dom.editors.enumeration name="Capable"
```

```
    displayName="Capable, you betcha!"/>
  <scea.dom.editors.enumeration name="Enthusiastic" />
```

Other Annotations

If you have defined other annotations to do things like create palette items or tooltips, you have to handle them individually in your own `ParseAnnotations()` method. You must provide the support code for any features described by the annotation.

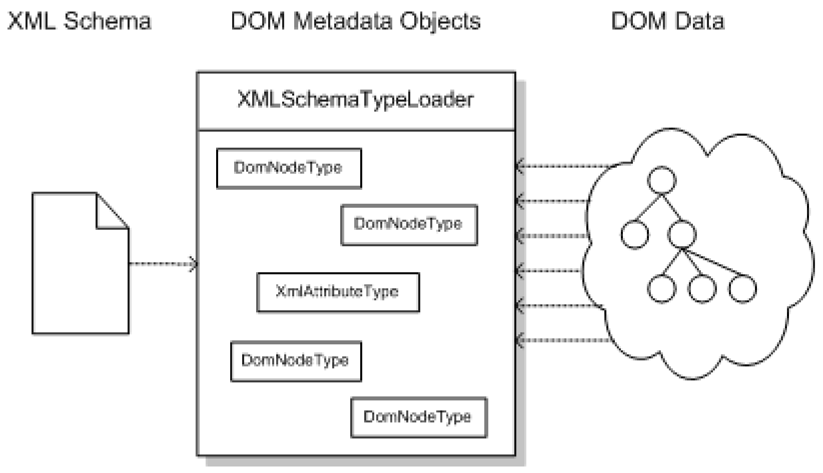
Using DomGen to Generate a Schema Stub Class

`DomGen` is an ATF DOM utility that generates a schema stub class from an XML schema file. This stub class contains a set of class and property definitions you can use in your schema type loader and in your DOM extension implementations to access DOM metadata directly. `DomGen` is in `components\wws_atf\DevTools\DomGen\bin`.

About DomGen

When the schema type loader reads the XML schema into the application, it converts all the types into internal DOM metadata objects. Those objects are stored internally in the type loader, as shown in Figure 14.

Figure 14 Loading XML Schema



When you create DOM adapters (during implementation time), and when you define adapters for DOM node types (at runtime), you need access to this DOM metadata. The `XmlSchemaTypeCollection` class accesses this metadata through its methods, such as `GetNodeType()`, `GetAttributeInfo()` and `GetChildInfo()`. Each of these methods is called with a string argument for the name of that type, attribute, or element—in the schema itself. For example, this statement gets the `DomNodeType` object corresponding to the type named “keyType”:

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

where `typeCollection` is an `XmlSchemaTypeCollection` object.

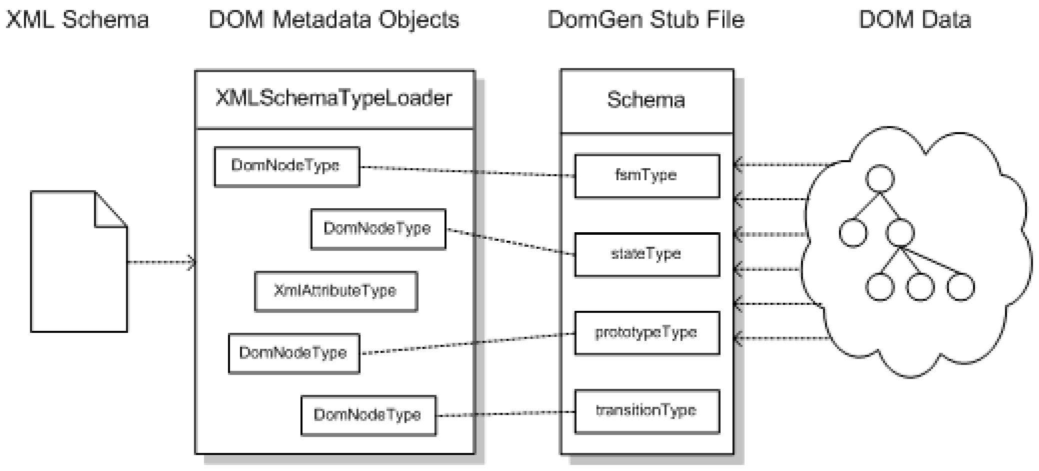
However, there are problems with accessing metadata directly through `XmlSchemaTypeCollection` methods:

- Direct access to schema metadata tightly couples your code to the schema, especially in terms of the type, attribute, and element names. As strings, those names are not checked at compile time, so mismatches or typos may not be apparent until the application runs. Changing a type name in the schema may require changes in multiple places throughout the code.

- If you have many types defined in your XML schema, you may find yourself creating tables of properties to wrap `XmlSchemaTypeCollection` methods to simplify their use. This can be tedious and difficult to maintain if types in your schema change.

`DomGen` solves both these problems by taking a schema as input and generating a class layer in between the schema and the rest of the code, as illustrated in Figure 15. `DomGen` defines a class, typically called `Schema`, that contains an inner class for every type in the type definition file. These classes and their properties, named for each of the schema types, attributes, and elements, are initialized to the appropriate metadata values in the `Schema.Initialize()` method. The class `Schema` is called the *schema stub class*. Instead of referencing schema names in your code, you use the class and field names generated by `DomGen`.

Figure 15 Schema Class Generated by DomGen



Here is an example of what `DomGen` does. First, here are the schema definitions of the "fsmType" and the root element in the Fsm Editor sample:

```
<!--
An FSM contains states, transitions among them, annotations, and a prototype
folder hierarchy.
-->
<xs:complexType name ="fsmType">
  <xs:sequence>
    <xs:element name="state" type="stateType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element name="transition" type="transitionType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element name="annotation" type="annotationType" minOccurs="0"
      maxOccurs="unbounded" />
    <xs:element name="prototypeFolder" type="prototypeFolderType" />
  </xs:sequence>
</xs:complexType>


<!--
The root element of an FSM document.
-->
<xs:element name="fsm" type ="fsmType"/>
```

Here is the corresponding code in the `Schema` class that `DomGen` creates:

```
public static void Initialize(XmlSchemaTypeCollection typeCollection)
{
    fsmType.Type = typeCollection.GetNodeType("fsmType");
    fsmType.stateChild = fsmType.Type.GetChildInfo("state");
```

```
        fsmType.transitionChild = fsmType.Type.GetChildInfo("transition");
        fsmType.annotationChild = fsmType.Type.GetChildInfo("annotation");
        fsmType.prototypeFolderChild =
           fsmType.Type.GetChildInfo("prototypeFolder");
        ...
     }
     ...
     public static class fsmType
     {
        public static DomNodeType Type;
        public static ChildInfo stateChild;
        public static ChildInfo transitionChild;
        public static ChildInfo annotationChild;
        public static ChildInfo prototypeFolderChild;
     }
     ...
     public static ChildInfo fsmRootElement;
```

The `fsmType` class represents "fsmType" defined in the schema. This class contains fields for the "fsmType" schema attributes that are DOM metadata types: `DomNodeType` and `ChildInfo`. The root element `fsmRootElement` is also defined as a `ChildInfo`. In the `Initialize()` function, `fsmType`'s fields are set to the attributes in the schema using DOM metadata types, such as `DomNodeType` and `ChildInfo`, using `XmlSchemaTypeCollection` and `DomNodeType` methods.

You application code can reference the `fsmType` class and its fields in `Schema`—rather than use names in the schema. If any names change in `Schema`, you get compilation errors in your code, so you know you need to make changes and where.

Running DomGen

Run `DomGen` as you build the application and each time the types, attributes, or elements in the schema change. You also need to re-run `DomGen` if you change the names of any parts of the schema. If the application's data model and schema are stable, you may only need to run `DomGen` a handful of times.

Note that the information `DomGen` generates does not include annotations or restriction facets. The schema type loader parses this information at runtime, and it can be changed at any time.

`DomGen` is a command-line tool included with ATF in the `components\wws_atf\DevTools\DomGen\bin` folder. You can run `DomGen` multiple ways:

- From the "Windows Command Prompt".
- From inside "Visual Studio".

Running DomGen from the Command Prompt

(1) Either use the `DomGen` utility prebuilt executable, or compile `DomGen` in `wws_atf\DevTools\DomGen` (either debug or release). `DomGen` can be used either in Debug or Release mode.

(2) Start a Windows Command Prompt and navigate to the location of the schema file, for example, `C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor\schemas`.

(3) At the command prompt, copy or type the full path location of the `DomGen` executable, for example: `C:\SceaDev\SCEA_WIP\ATF\DevTools\DomGen\bin\Debug\DomGen.exe`.

(4) Enter the `DomGen` options for the schema file. `DomGen` requires four options:

- The name of the schema file, for example, "timeline.xsd".
- The name of the schema stub class to generate. Conventionally this class is called "Schema.cs".
- The XML target namespace of the types within the schema, for example, "timeline" or "http://sce/timeline".

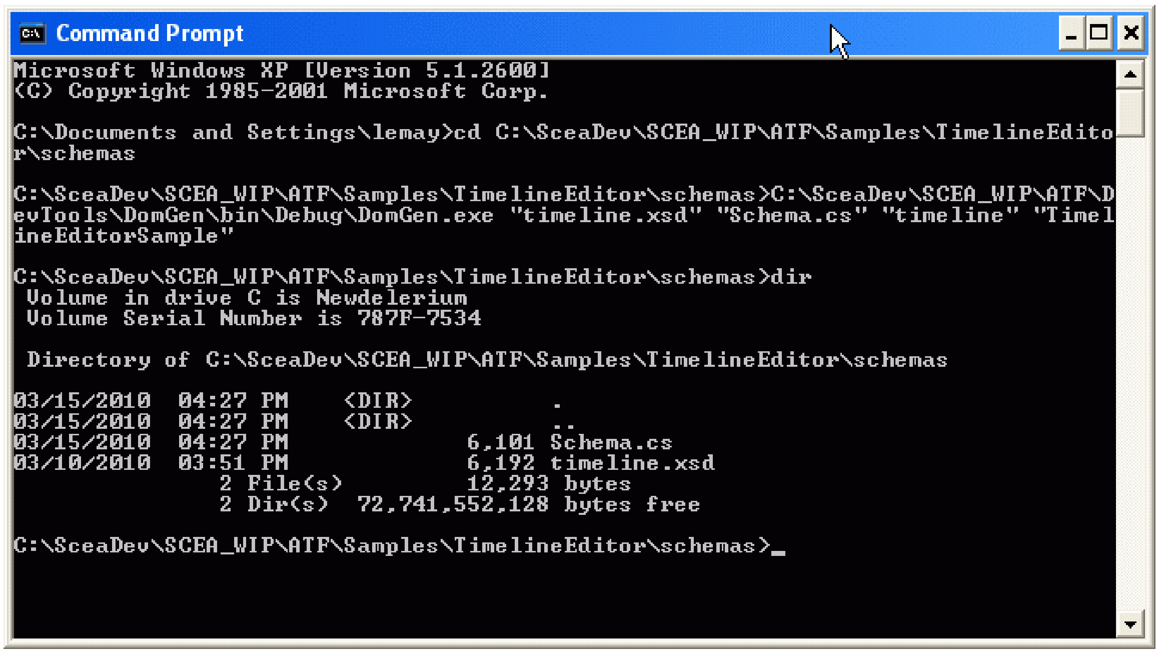
- The C# namespace you want for the generated schema stub class, for example, “TimelineEditorSample” or “Sce.Atf.Samples”.

(5) Press Enter to run `DomGen`.

The `Schema.cs` file (or whatever name you specified for the file name option) is generated in the schemas folder. You can leave the file in that folder or move it to a more convenient location.

Figure 16 shows using `DomGen` for the Timeline Editor sample:

Figure 16 Running DomGen from the Windows Command Prompt



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\lemay>cd C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEdito
r\schemas

C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor\schemas>C:\SceaDev\SCEA_WIP\ATF\D
evTools\DomGen\bin\Debug\DomGen.exe "timeline.xsd" "Schema.cs" "timeline" "Timel
ineEditorSample"

C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor\schemas>dir
 Volume in drive C is Newdelerium
 Volume Serial Number is 787F-7534

 Directory of C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor\schemas

03/15/2010  04:27 PM    <DIR>          .
03/15/2010  04:27 PM    <DIR>          ..
03/15/2010  04:27 PM             6,101 Schema.cs
03/10/2010  03:51 PM             6,192 timeline.xsd
               2 File(s)         12,293 bytes
               2 Dir(s)  72,741,552,128 bytes free

C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor\schemas>_
```

(6) Add the `Schema.cs` file to the ATF project with the **Project > Add Existing Item** command in Visual Studio.

Running DomGen from inside Visual Studio

You can also run `DomGen` directly from Visual Studio, which requires less typing.

(1) Open the `DomGen` solution in Visual Studio. Choose **Project > DomGen Properties** to edit the properties for the `DomGen` project. You can also right-click the `DomGen` project itself to edit its properties.

(2) Choose either the **Debug** or **Release** page of the properties, depending on the build type. `DomGen` can be used either in Debug or Release mode.

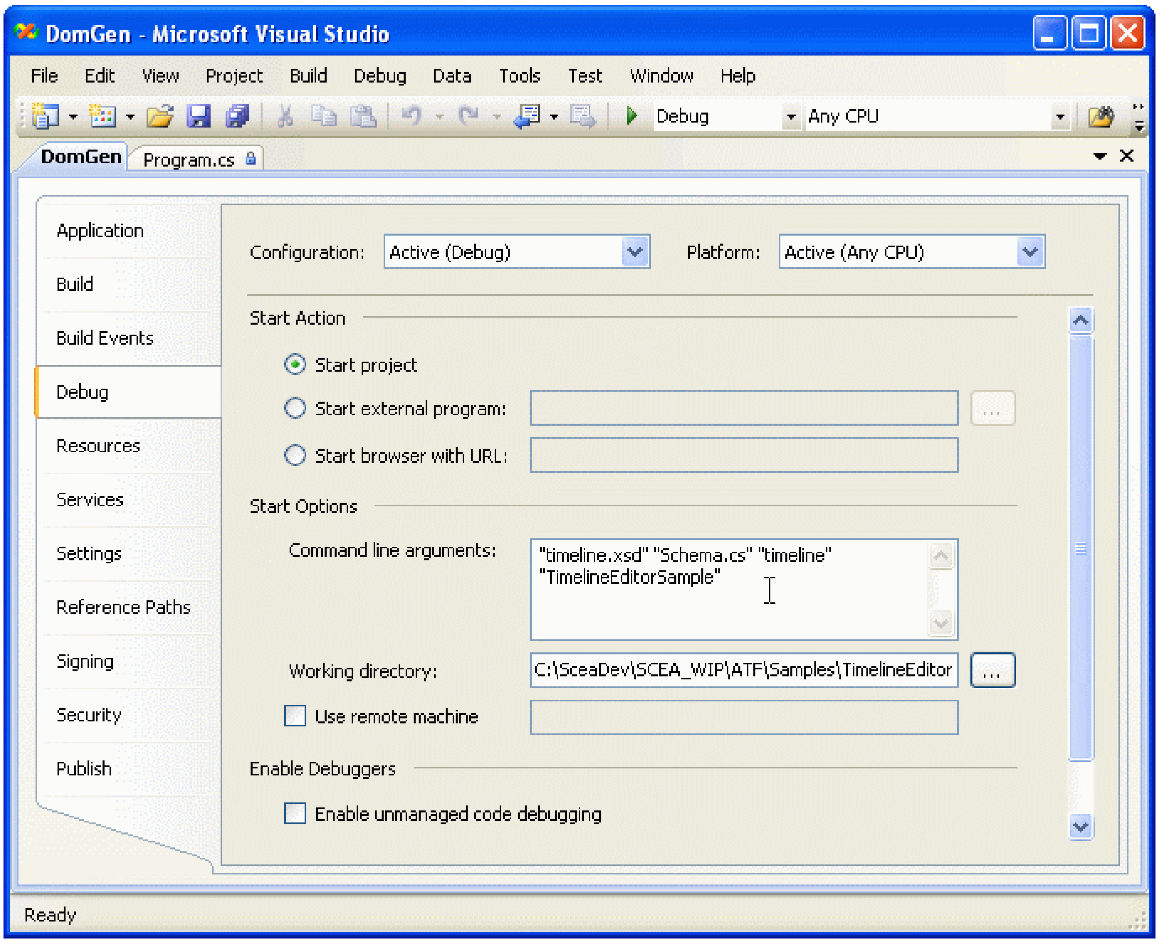
(3) Change the **Working directory** field to the folder that contains the schema file.

(4) Add the `DomGen` options to the **Command line arguments** field. Do not include the name of the `DomGen` program itself. `DomGen` requires four options:

- The name of the schema file, for example, “timeline.xsd”.
- The name of the schema stub class to generate. Conventionally this class is called “Schema.cs”.
- The XML target namespace of the types within the schema, for example, “timeline” or “http://sce/timeline”.
- The C# namespace you want for the generated schema stub class, for example, “TimelineEditorSample” or “Sce.Atf.Samples”.

Figure 17 shows the project **Properties** page with example input for Timeline Editor:

Figure 17 Running DomGen from within Visual Studio



Configuration: Active (Debug)
Platform: Active (Any CPU)

Start Action
(● Start project) ○ Start external program: ○ Start browser with URL:

Start Options
Command line arguments: "timeline.xsd" "Schema.cs" "timeline" "TimelineEditorSample"
Working directory: C:\SceaDev\SCEA_WIP\ATF\Samples\TimelineEditor
☐ Use remote machine

Enable Debuggers
☐ Enable unmanaged code debugging

(5) Save, build and run the `DomGen` project.

The `Schema.cs` file (or whatever name you specified for the file name option) is generated in the schemas folder. You can leave the file in that folder or move it to a more convenient location.

(6) Add the `Schema.cs` file to your ATF project with the **Project > Add Existing Item** command in Visual Studio.

Implementing a Schema Type Loader

The schema type loader brings the types defined by your schema into your application at runtime, creates metadata objects for those types, defines DOM extensions and adapters for types, and handles annotations, if any. To create a schema type loader class for your XML schema, you can derive from the `XmlSchemaTypeLoader` class and implement the `OnSchemaSetLoaded()` and `ParseAnnotations()` methods.

The XmlSchemaTypeLoader Base Class

Implement your schema type loader by deriving from the `XmlSchemaTypeLoader` base class. This class does much of the work of loading and managing the XML schema for you, including:

- Loading one or more schemas from local files, including resolving references to external resources, such as other schema documents.
- Creating internal DOM metadata objects (`DomNodeType`, `XmlAttributeType`, and so on) for the types, attributes, and elements in the schema, and storing those objects in an `XmlSchemaTypeCollection` object.

- Creating a collection (implementing `IDictionary`) of any annotations the schema defines.
- Defining an ID attribute for any type that defines `<xs:ID>`.
- Creating DOM validator rule objects for any restriction facets defined by the schema. For more information, see "Validating DOM Data".

Note: The `XmlSchemaTypeLoader` class uses the .NET Schema Object Model (SOM) to load and resolve XML schema. You need to include the `System.XML` reference in the project, as well as include the `System.Xml` and `System.Xml.Schema` namespaces with the `using` C# keyword. For more information about the .NET XML framework, see the XML Schema Object Model (SOM) section of the .NET Framework Developer's Guide.

Your schema type loader derived from `XmlSchemaTypeLoader` adds any additional behavior required for your schema. By convention, the schema type loader class is called `SchemaLoader`.

In your derived schema loader class, you should override two methods:

- "OnSchemaSetLoaded() Method" for behavior to execute after the schema itself has been loaded, an `XmlSchemaTypeCollection` is available, and DOM metadata objects have been created. In this method, you use metadata objects in the schema stub file generated by `DomGen` and define any DOM extensions and adapters for types.
- "ParseAnnotations() Method" to handle annotations, if any, defined in the schema by creating property descriptors, palette data, tooltips, or other objects from the annotation information.

The schema type loader class may also define these and other commonly-used properties:

- A `TypeCollection` property, to hold the `XmlSchemaTypeCollection` that contains the schema metadata.
- A `Namespace` property holding the namespace of the current schema, which you get from the `XmlSchemaTypeCollection`.

For examples of custom schema type loaders, see the Circuit Editor, DOM Tree Editor, Fsm Editor, Simple DOM Editor, State Chart Editor, Using Dom, Win Forms App, and Wpf App ATF samples in `wws_atf\Samples`.

OnSchemaSetLoaded() Method

Override the `OnSchemaSetLoaded()` method to add behavior for an XML schema. The `OnSchemaSetLoaded()` method is invoked from the `XmlSchemaTypeLoader` class after the schema has been loaded and resolved, and after the DOM metadata objects for each type have been created. There is no default behavior for this method.

Some tasks to implement for this method include:

- Set type collection and namespace for the `TypeCollection` and `Namespace` properties, if you have defined them.
- Invoke `Schema.Initialize()` to initialize the schema stub class definitions generated by `DomGen`.
- Define any DOM extensions for the types now available from `DomGen`'s stub class `Schema`.
- Define utility adapters for types, such as ID and reference validators.
- Add any other metadata needed by types, using the `NamedMetadata.SetTag()` method.

Here is part of `OnSchemaSetLoaded()` from the Fsm Editor sample:

```
protected override void OnSchemaSetLoaded(XmlSchemaSet schemaSet)
{
   foreach (XmlSchemaTypeCollection typeCollection in GetTypeCollections())
   {
      m_namespace = typeCollection.TargetNamespace;
      m_typeCollection = typeCollection;
      Schema.Initialize(typeCollection);
```

```
            // define extensions
            // extend FSM root node with FSM, document, editing context, ...
            Schema.fsmType.Type.Define(new ExtensionInfo<Fsm>());
            Schema.fsmType.Type.Define(new ExtensionInfo<EditingContext>());
            Schema.fsmType.Type.Define(new ExtensionInfo<ViewingContext>());
            Schema.fsmType.Type.Define(new ExtensionInfo<TransitionRouter>());
            Schema.fsmType.Type.Define(new ExtensionInfo<Document>());
            Schema.fsmType.Type.Define(new ExtensionInfo<PrintableDocument>());
    ...
            // extend with adapters to validate references and unique ids
            Schema.fsmType.Type.Define(new ExtensionInfo<ReferenceValidator>());
            Schema.fsmType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
    ...
            // define FSM object model
    ...
            Schema.transitionType.Type.Define(new ExtensionInfo<Transition>());
            Schema.annotationType.Type.Define(new ExtensionInfo<Annotation>());
    ...
            // annotate state and annotation types with display information for palette
            Schema.stateType.Type.SetTag(
                new NodeTypePaletteItem(
                    Schema.stateType.Type,
                    Localizer.Localize("State"),
                    Localizer.Localize("State in a finite state machine"),
                    Resources.StateImage));
    ...
            // the fsm schema defines only one type collection
            break;
        }
    }
```

Note: The *Annotation* mentioned in this example is a class for comments on a canvas and is different from annotations in XML schemas discussed in "Annotations".

Note the `foreach` loop that serves as the core of this method. There is a one-to-one correspondence between the schema and its `XmlSchemaTypeCollection`, so if you use only one schema for the application data model, this loop might seem unnecessary. The reason the loop is here is that type collections are registered by the `XmlSchemaTypeLoader` base class by namespace name. Calling the `GetTypeCollections()` method in a loop enables you to retrieve a type collection without hard-coding its namespace name in the code and creating a dependency on the schema. This particular sample does not take advantage of this generality, because its schema defines only one type collection, so it breaks out of the loop.

The first two lines initialize internal fields to set values used by the `NameSpace` and `TypeCollection` properties defined in the `SchemaLoader` class:

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

This statement initializes the schema stub class generated by `DomGen`:

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

The next set of lines all define DOM extensions for types in the schema stub class; the first group on the `fsmType` type, and the remaining two on the `transitionType` type. Note that names like "fsmType" that were generated by `DomGen` are used here—rather than identifier strings from the schema itself—to help insulate the code from schema changes. You get these class and field names from the `Schema.cs` file created by `DomGen`. For more information on creating and defining DOM adapters, see "DOM Nodes and DOM Adapters".

Note these two DOM adapter definitions in particular:

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

These DOM adapters are utility validators for references and identifiers, respectively. The reference validator validates internal and external references between DOM nodes. The ID validator ensures that DOM nodes with an ID have unique ID values. In this case, both validators are defined for the type `fsmType`, which is the type of the root node. This allows these validators to listen to events on *all* nodes in the tree, so all nodes are checked, which means all application data is checked. Validators may also require certain interfaces to be implemented for at least one DOM adapter initialized for the same `DomNode`, so defining such extensions on the root `DomNode` makes it easier to meet that requirement. You also need to call `InitializeExtensions()`on the root DOM node of your document, so that all nodes in the tree are checked. For more information, see "Validating DOM Data".

The method's final statements set tags on `DomNodeType` metadata objects to provide additional metadata (such as display information for a palette and for property descriptors) on a type using the `NamedMetadata.SetTag()` method:

```
Schema.stateType.Type.SetTag(
    new NodeTypePaletteItem(
        Schema.stateType.Type,
        Localizer.Localize("State"),
        Localizer.Localize("State in a finite state machine"),
        Resources.StateImage));
```

The `NodeTypePaletteItem` class describes how a particular `DomNodeType` appears in a palette of items that a user can drag and drop into a document.

You can add any kind of metadata to a `DomNodeType` by using the `SetTag()` method.

ParseAnnotations() Method

Override the `ParseAnnotations()` method to handle any annotations you have defined in the schema—for example, to create property descriptors, palette items, or tooltips—based on annotation metadata. The `ParseAnnotations()` method is invoked in the `XmlSchemaTypeLoader` base class just after `OnSchemaSetLoaded()`. `ParseAnnotations()`is called with a collection (implementing `IDictionary`) of any annotations the schema defines, created by `XmlSchemaTypeLoader`.

The base `ParseAnnotations()` method manages only the legacy `idAttribute` annotation.

Some of the legacy ATF 2 annotations, such as `scea.dom.editors.attribute`, `scea.dom.editors.child`, and `scea.dom.editors.enumeration`, are supported by ATF 3 in its `Sce.Atf.Dom.PropertyDescriptor` class. `PropertyDescriptor.ParseXml()`parses annotations for property descriptor information and creates new instances of the `PropertyDescriptor` class, returning them in a `PropertyDescriptorCollection`. However, `ParseXml()` is *not* called by the base `ParseAnnotations()` method, so you should override `ParseAnnotations()` in your schema loader class and call `ParseXml()` in your implementation. The Timeline Editor sample's schema loader illustrates this technique.

For a list of annotations supported in ATF 3, see "Annotations". For information on processing property descriptor annotations, see "Defining Property Descriptors in XML Schema Annotations".

This sample `ParseAnnotations()` method from the Timeline Editor sample parses annotations and then creates property descriptors and DOM palette items based on annotation information:

```
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);
                 }
              }
           }
        }
     }
```

Note that the base `ParseAnnotations()` method is called, which processes legacy annotations specifying the ID attribute:

```
base.ParseAnnotations(schemaSet, annotations);
```

The `foreach` statements iterate on each `DomNodeType` defined in the schema:

```
foreach (DomNodeType nodeType in m_typeCollection.GetNodeTypes())
```

The following code determines if any annotation refers to any of the `DomNodeTypes` and, if so, collects any annotation data for it from the XML schema with `ParseXml()`:

```
// 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);
```

In between parsing the XML annotations with `ParseXml()` and adding the `PropertyDescriptorCollection` to the metadata for a type with `SetTag()`, you can manually define any additional property descriptors that may not have been created with annotations. To learn how to do this, see "Defining Property Descriptors in the Schema Loader".

In general, if your annotation format differs from those currently supported, you need to write additional code to parse your custom annotations and perform whatever actions are required in your

`ParseAnnotations()` method. For instance, at the end of the loop in the previous example, annotation data is processed to create palette items. This code uses the `SetTag()` method to add annotation data from the schema file to the `DomNodeTypes`.

You also need to implement your own annotation handling code if you use a type definition file other than XML Schema for your data model.

Loading the XML Schema

Once you have an XML schema, a schema stub class, and a schema type loader class all implemented for the application, the final step is to invoke the schema type loader when the application starts. A straightforward way is to load the schema in the `SchemaLoader`'s constructor.

You can make the `SchemaLoader` class a Managed Extensibility Framework (MEF) component, so its instantiation is handled for you (as most of the samples do) or instantiate it directly (as the Timeline Editor sample does).

Here is the `SchemaLoader`'s constructor from the Timeline Editor sample. It calls `Load()` with the resolved path of the XML schema file to invoke the schema type loader:

```
public SchemaLoader()
{
   // set resolver to locate embedded .xsd file
   SchemaResolver =
      new ResourceStreamResolver(Assembly.GetExecutingAssembly(),
      "TimelineEditorSample.schemas");
   Load("timeline.xsd");
}
```

You may load as many schemas as the application needs, as long as each namespace is unique. Each schema you use is registered by the `XmlSchemaTypeLoader` base class by its target namespace, and a separate `XmlSchemaTypeCollection` is created for each schema.

4 DOM Nodes and DOM Adapters

An application's data is stored in a tree of DOM nodes: instances of the class `DomNode`. You can *adapt*, that is, dynamically cast a `DomNode` to another class's type, or track data related events, using a *DOM extension*. After this cast, that class's methods and properties become available to manipulate the data represented by the DOM node—just as if the `DomNode` were an instance of that class. In other words, extensions provide an API for an application to manipulate data in the DOM node tree. You can design the extension's API so it is convenient for you to work with and easier to change data in response to user editing. You can also use an extension to listen to events on a `DomNode` and its children, to perform data validation, for instance.

DOM adapters, extensions derived from the `DomNodeAdapter` class, are by far the most common DOM extensions. You don't need to handle the DOM node tree directly. Instead, you implement extensions and adapters and define them for types in the data model. The application accesses data in the DOM node tree through these DOM extensions.

This chapter discusses the following topics:

- "About DOM Nodes and Extensions": Learn how extensions and adapters work with DOM nodes and the DOM node tree, as well as how to implement and define extension adapters for data types.
- "Workflow": Steps to take at implementation time and at runtime to work with DOM nodes and extensions.
- "DOM Node and Extension Key Classes": A summary of the most important classes and interfaces described in this chapter.
- "Defining DOM Extensions and Adapters ": Using DOM extensions and adapters on DOM nodes by defining them for the node's types.
- "Using DOM Nodes and Adapters": Common tasks for working with DOM nodes and DOM adapters, including creating, adding, and removing nodes in the DOM node tree, and switching between DOM adapters.
- "DOM Events": Using `DomNode` events that occur when a `DomNode` or DOM tree changes.
- "Implementing DOM Adapters": Learn about creating DOM adapters derived from the `DomNodeAdapter` base class.
- "Validating DOM Data": Using validator and rule classes to check that DOM data meets specified requirements.

About DOM Nodes and Extensions

The ATF DOM consists of three layers:

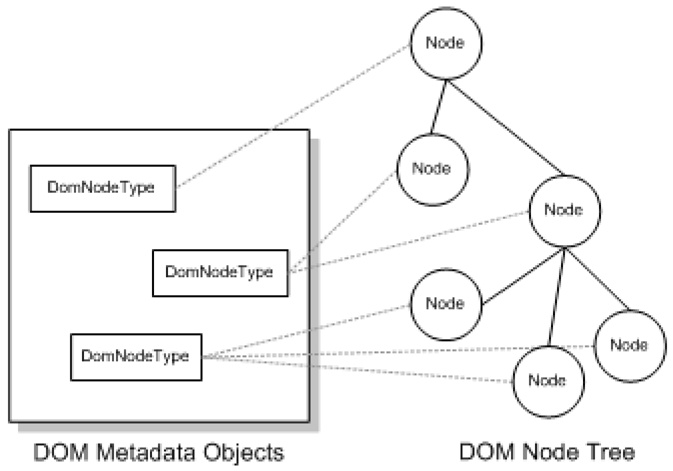
- The data model, which you define in a type definition file and load with the type loader, as previously discussed.
- The DOM node tree, which contains the runtime application data.
- A set of DOM extensions and adapters, which provide a C# API layer between the application code and the nodes (data) in the DOM node tree.

You learned about creating the data model for the DOM in the previous chapter. The remaining two layers, the DOM node tree and DOM extensions, work closely in conjunction, and so are both described in this chapter.

DOM Nodes and the DOM Node Tree

Your application's data is stored in the DOM as a tree of DOM nodes, instances of the `DomNode` class, as shown in Figure 18. Every DOM node has an underlying metadata type: an instance of `DomNodeType`. `DomNodeType` metadata objects are created by the type loader, based on information in the type definition file, as described in "Data Modeling, XML Schemas, and the DOM". `DomNodeType` metadata objects are shared among all DOM nodes of that type.

Figure 18 DOM Nodes and the DOM Node Tree



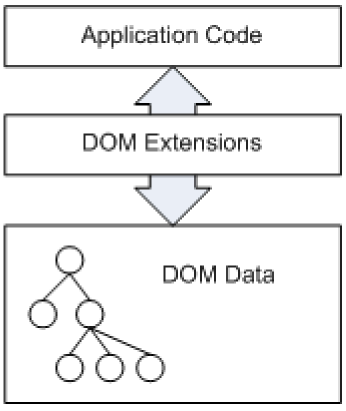
The `DomNode` and `DomNodeType` classes provide basic functionality for managing the tree and nodes within that tree, including event management, unique node naming, reference tracking, transactions, and synchronization with underlying documents that store application data persistently.

For more information on DOM nodes and the DOM node tree, see "Using DOM Nodes and Adapters".

DOM Extensions and DOM Adapters

Your application does not need to manipulate nodes in the DOM node tree directly to manage application data. Instead, DOM extensions provide a flexible, object-oriented adaptation layer between application code and DOM nodes' data. Through this adaptation layer, the application can create and remove nodes, modify node properties, and add and remove child nodes, as illustrated in Figure 19.

Figure 19 DOM Extensions, DOM Data, and the Application



You can create DOM extension classes to provide an object API for the types in the data model. A DOM extension can be an object of any type. Extensions that derive from the `DomNodeAdapter` class are called *DOM adapters*, which are the most widely used extension, because the `DomNodeAdapter` class provides useful capabilities for extensions. In practice, DOM extensions are DOM adapters.

Implementing DOM Adapters

Implement a DOM adapter by deriving from the `DomNodeAdapter` base class. This class provides the infrastructure for DOM extensions and many useful methods and properties especially useful for an extension. `DomNodeAdapters` allow `DomNodes` to be dynamically cast to more convenient forms. In your DOM adapter implementation, you can:

- Provide an API for data in the DOM node, wrapping elements and their attributes in properties from the class type you have adapted the node to.
- Provide specific initialization behavior when the adapter is created or when it is initialized for its underlying DOM node.
- Manage changes to data in the DOM node tree, both for this type and related types, including child node types, by listening to events on DOM nodes and their children.
- Obtain common behavior across DOM types by implementing context interfaces. These easy to implement interfaces provide basic common DOM node behavior. For example, the `INameable` interface enables DOM nodes to be named and thereby easily collected in a palette.

DOM adapters are DOM extensions. The two behave similarly in the DOM: they are defined for types and are initialized for `DomNodes`. The key difference is that DOM adapters derive from `DomNodeAdapter`, so they have all the capabilities this class offers. For this reason, this chapter concentrates on DOM adapters. In practice, you want to use a DOM adapter for your DOM extension.

For details, see "Implementing DOM Adapters".

Defining Extensions

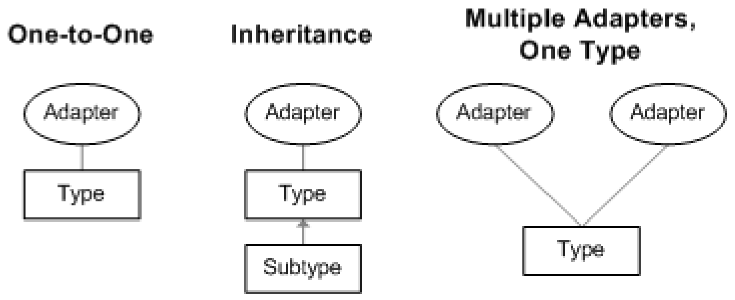
DOM extensions and adapters need to be defined at runtime for the types on which they operate. The type loader is a good place to define them for their types.

Typically you implement and define one DOM adapter for each type, providing a one-to-one correspondence between a type and its adapter. A defined adapter or extension automatically applies to both a specific type *and all of its subtypes*, enabling you to take advantage of type inheritance.

Extensions that listen to events are usually defined for the type of the `DomNode` tree root. These extensions can then listen to events on *all* nodes in the tree. Validators are examples of such extensions.

You can also define multiple extensions and adapters for the same type to provide a different interface or view on the same data in the DOM node tree. Figure 20 shows adapter definition.

Figure 20 Adapter Definition



For more information, see "Defining DOM Extensions and Adapters".

The DOM Extension and Adapter Lifecycle

When a DOM node is created with the `DomNode` constructor, DOM extension objects are automatically created for the node for all extensions defined for the node's type. These DOM extension objects are permanently associated with their `DomNode` object and cannot be removed. `DomNodeAdapter` is the most common kind of extension by far.

After creation, DOM adapters are initialized with the `DomNodeAdapter.OnNodeSet()` method when a DOM adapter reference is obtained. You can obtain a reference to a DOM adapter by using the `Is<T>()`, `As<T>()`, and `Cast<T>()` methods, where `T` is the adapter type. Although `Is<T>()` returns a Boolean indicating whether the `DomNode` can be adapted to `T`, it also gets (implicitly) a reference to the adapter. These casting methods behave similarly to their C# equivalents.

The other way to initialize DOM adapters is to call `DomNode.InitializeExtensions()` on a DOM node or on any ancestor of that DOM node. It is a good practice to call `InitializeExtensions()`on the root DOM node of your document, shortly after the document has been loaded.

Extension objects and their DOM nodes, like all C# objects, are destroyed when the garbage collector determines that they are not reachable by your application, which would occur after a node is deleted.

Workflow

Table 7 and Table 8 describe the steps to take at implementation time and at runtime to work with DOM nodes and extensions, as well as where to go in this document for more information on each of those steps.

At Implementation Time

This is a suggested order of tasks to perform during your development.

Table 7 DOM Adapter Workflow: Implementation Time

Step	Description	For more Information
1. Implement DOM extension and adapter classes for the types in the data model.	DOM extensions and adapters provide the API between the types and data in the DOM node tree and the application itself.	"Implementing DOM Adapters"
2. Implement DOM validators and rules for types you want to validate.	DOM validators are DOM extensions that allow you to validate data as it changes so it conforms to validation rules, which you specify.	"Validating DOM Data"
3. Subscribe to any DOM events you want to monitor.	Manage changes to nodes' attributes and the DOM node tree.	"DOM Events"
4. Use DOM extensions and validators.	Write code that uses the DOM extensions and adapters. In some cases, such as validators, you need to do very little.	"Using DOM Nodes and Adapters" and "Validating DOM Data"
5. Write code to define DOM extensions for types.	DOM extensions and adapters must be defined for the types on which they operate. This definition code is usually in the type loader.	"Defining DOM Extensions and Adapters"

At Run Time

Table 8 DOM Adapter Workflow: Run Time

Step	Description	For more Information
1. Define DOM extensions and adapters for types.	Invoke the type loader to define DOM extensions for their types.	"Defining DOM Extensions and Adapters"

Step	Description	For more Information
2. Create and initialize DOM nodes as needed.	Create new instances of `DomNode`, and use the `As<T>()` or `Cast<T>()` methods to initialize a DOM extension for its node. Use `InitializeExtensions()` to initialize extensions for large trees of DOM nodes at once, when loading a document, for example. If you persist DOM data in an XML file, the `XMLReader` ATF class can create a DOM tree from saved data.	"Creating and Initializing DOM Nodes" in "Using DOM Nodes and Adapters" "Reading XML" in "DOM Persistence"

DOM Node and Extension Key Classes

The `Sce.Atf.Dom` namespace includes a number of classes for creating and using DOM extensions and adapters, outlined in Table 9.

Table 9 Key Classes in Sce.Atf.Dom for DOM Extensions and Adapters

Class	Description
`DomNode`	Describes a node in the DOM data tree. Each DOM node has a metadata type, `DomNodeType`.
`DomNodeType`	Describes the underlying type of a DOM node. `DomNodeType` metadata objects are created by the type loader. Contains attributes, child node types, and DOM extensions. Extends `NamedMetadata`. For more details, see "DOM Metadata Objects".
`DomNodeAdapter`	Abstract base class for DOM adapters. Offers many useful capabilities for adapting DOM nodes. For information, see "Implementing DOM Adapters".
`DomNodeListAdapter<T>`	A wrapper class that contains a list of adapters of type `T` initialized for a list of DOM node child data. Derives from `IList<T>`. For more information, see "Initializing a List of Child DOM Adapters" in "Creating and Initializing DOM Nodes".
`TypeAdapterCreator`	Class for generating DOM node adapters from DOM node types. Implements `IAdapterCreator`.

Defining DOM Extensions and Adapters

Use the `DomNodeType.Define()` method to define DOM extensions and adapters for the types to which they apply with the `ExtensionInfo<T>` extension metadata constructor, where `T` specifies the extension or adapter type. These definitions are usually in the type loader for the application. If you're using a type loader that derives from `XmlSchemaTypeLoader`, place DOM extension definitions in `OnSchemaSetLoaded()`. For more information, see "Implementing a Schema Type Loader".

This example, from the Fsm Editor sample's type loader, defines several different kinds of DOM adapters on various types. All adapter definitions use the DOM metadata (such as `DomNodeType`) objects defined in the schema stub file—`Schema.fsmType`, `Schema.stateType`, and `Schema.prototypeType`—and so forth.

```
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>());
```

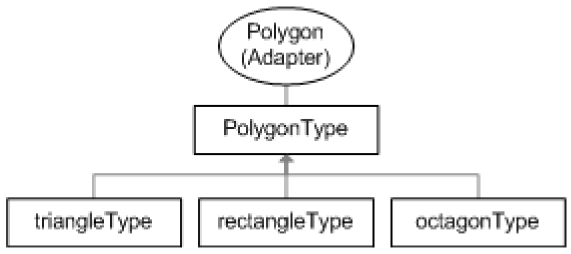
Note: The *Annotation* in this example is a class for comments on a canvas and is different from annotations in XML schemas discussed in "Annotations".

The first three lines in this example define DOM adapters for `fsmType`. They demonstrate how you can define multiple DOM adapters for the same DOM type—in this case, a DOM adapter for the Fsm Editor sample's `Fsm` class, as well as adapters for both an editing context (`EditingContext`) and viewing context (`ViewingContext`). Also defined for this type, but not shown here, are DOM adapters for validation, unique IDs, documents, printable documents, and other features. Note that these adapters that provide a context are defined on the root `DomNode` type.

The second set of adapter definitions in the example is for other types in the Fsm Editor sample data model. These types define a single adapter for a type, so there is a one-to-one correspondence between the type and the DOM adapter.

DOM adapters take advantage of type inheritance. If you define a DOM adapter for a type that has subtypes, you do not have to define that DOM adapter for the subtypes. You only need to define it for the base type, and it is automatically defined for all subtypes. For example, if you have a DOM adapter for the base type `polygonType` that has derived `triangleType`, `rectangleType`, and `octagonType` types, you only need to define the DOM adapter once, for `polygonType`, as illustrated in Figure 21.

Figure 21 DOM Adapters and Type Inheritance



The `DomNodeType.BaseOfAllTypes` static property is the root of *all* DOM metadata types defined in the data model. You can use the `BaseOfAllTypes` property and the `DomNodeType.Define()` method to define a DOM node adapter that applies to *all types*—meaning that every `DomNode` can be adapted to the type defined in the `ExtensionInfo<T>` passed to `Define()`.

You can also use the more powerful `DomNodeType.AddAdapterCreator()` method with `BaseOfAllTypes` to attach an `AdapterCreator<T>` object to all types. The `AdapterCreator<T>` object acts as a factory that can choose which `DomNodeType` can be adapted to a particular C# type. For example, your `AdapterCreator` can allow your `DomNodeTypes` "Shark" and "Whale" to be adapted to your C# `IWaterAnimal` interface, but not allow your "Giraffe" `DomNodeType` to be adapted to `IWaterAnimal`.

For example, this statement (used in several samples) adds a DOM adaptor to allow all `DomNodes` to adapt to an appropriate property descriptor, if any:

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

This statement enables metadata-driven property editing in the application. That is, the properties that appear in property editors are determined by the metadata provided about the properties (in property descriptors). If this statement were removed, the properties visible in a property editor for an object would be properties of the object and classes it derives from—not the properties you provide in your type definition. For more information, see "Defining and Using DOM Property Descriptors".

Using DOM Nodes and Adapters

Application data is stored in the DOM as a tree of individual DOM nodes—instances of the class `DomNode`. Each `DomNode` has a DOM metadata object `DomNodeType` that specifies its node type. Each type also has one or more defined DOM adapters, and the application can access data in the DOM node tree through those adapters, one at a time.

This section describes how to use DOM adapters to manipulate nodes in the DOM node tree. It assumes that DOM extensions or adapters have been defined for each type of DOM node in the tree. To learn how to implement and define DOM adapters for types, see "Implementing DOM Adapters" and "Defining DOM Extensions and Adapters".

Creating and Initializing DOM Nodes

To create and initialize a new node in the DOM node tree:

- Create a node or a list of nodes.
- Initialize an extension or adapter for that node or list of nodes.

If you are creating an entire tree of nodes at once (when opening a document and reading its data, for example), you need to explicitly initialize the extensions for those nodes.

Create a DOM Node

Create a new node with the `DomNode` constructor, specifying its type:

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

The default `DomNode` constructor takes a `DomNodeType` object as its argument. This example uses one of the metadata objects described in a schema stub class generated by `DomGen`.

In addition to the type, you can specify a `ChildInfo` metadata object in the `DomNode` constructor. The `ChildInfo` object contains metadata describing this node when it is a child of another node, including its `DomNodeType` and whether the child is a list of nodes. Note that even root nodes can have a `ChildInfo`:

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

Given a `DomNode` and its `ChildInfo`, the `DomNode`'s type must be the same or derived from the `ChildInfo`'s type. For example, in the Circuit Editor sample, a `CircuitGroup` node can have a list of child `DomNode`s of type `Module`. You can add a child of type `LightSwitch`, `Speaker`, or even another `CircuitGroup`, because these types all derive from `Module`.

If you have saved application data in an XML file, you can also use the class `XMLReader` to create a DOM tree from the XML data. For more information, see "Reading XML" in "DOM Persistence". In addition, the root `DomNode` of a document must have a `ChildInfo` to be persisted in an XML file.

Initialize Extensions for a Node

When a `DomNode` is created, an instance is created of each DOM extension that is defined for the node's type. These instances are all associated with this `DomNode`. In addition, if the `DomNode`'s type inherits from other types, the extensions defined for all these base types are also created.

If a DOM extension derives from other DOM extensions, an object is created *only* for the DOM extension defined for the type—not for any classes that extension derives from. In general, you do not want to define extensions for the type using the extension's base classes; otherwise duplicate events would be generated for the same node, which is probably not what you want. In any case, you do not need to explicitly define the base classes of an extension as extensions for a type for them to be available for that type. For example, `DataValidator` derives from `Validator`, which derives from `Observer`. If you have a `DomNode domNode` that is the root of a `DomNode` tree, and `DataValidator` is defined as an extension for `domNode`'s type, all three of these statements succeed:

```
domNode.As<DataValidator>();
domNode.As<Validator>();
domNode.As<Observer>();
```

A DOM extension is typically initialized for a `DomNode` by invoking one the methods `Is<T>()`, `As<T>()`, or `Cast<T>()` on the `DomNode`. For DOM adapters, the `OnNodeSet()` method for the adapter is called immediately after initialization occurs, which gives the opportunity for additional one time initialization. Note that `OnNodeSet()` is called just once, for each adapter instance, after the adapter is initialized for its node. An adapter object is never removed from its node.

DOM adapters are initialized lazily, that is, they are not initialized until needed.

Typically the `As<T>()` or `Cast<T>()` method is called to get the extension for a node. For example, this code creates a new DOM node for a prototype and then gets the `Prototype` DOM adapter for that node:

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

Often nodes are created and their corresponding adapters retrieved at the same time:

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

The `Pin` object returned by the call to `As<Pin>()` is fully initialized and ready to be used. You can use all the `Pin` class's methods and properties with `pin`, even though `pin` is actually a `DomNode` object, not a `Pin` object.

You can use code like the above examples to get objects of the various extensions defined for the type of a `DomNode`.

Implementing Interfaces for DOM Adapters

Some extensions require certain interfaces to be implemented to operate properly. For example, the `DataValidator` DOM adapter runs in `IValidationContext`. This means the type that `DataValidator` is defined for must also have an extension defined for that type that implements `IValidationContext`. The validator and the extension implementing the interface must also be initialized for the same `DomNode`, which can be done by calling `DomNode.InitializeExtensions()` on this node. (`DataValidator` derives from `Validator`, which obtains an `IValidationContext` by using `AsAll< IValidationContext>()` to enumerate the decorators on the node that can convert a reference to `IValidationContext`.)

Normally, only the root `DomNode` is adapted to validators and extensions providing context interfaces. This way, extensions can use any interface implemented by any extension initialized to the node. Also, extensions initialized on the root `DomNode` can listen to events on all `DomNode`s in the tree.

Initializing a Tree of Extensions

Some DOM extensions and adapters, such as validators, are never used directly; that is, you do not invoke the `Is<T>()`, `As<T>()`, or `Cast<T>()` methods on the DOM node. For DOM nodes that use these types of adapters, you need to initialize extensions for the entire DOM node tree or subtree explicitly, generally when you create a tree of DOM nodes for the document's data. To initialize the adapters for an entire tree of DOM nodes, use the `DomNode.InitializeExtensions()` method:

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

A common place to call this method is in your document client class's `Open()` method, because new trees of DOM nodes are often created when documents are opened or created.

Initializing a List of Child DOM Adapters

A DOM node is commonly the parent of child nodes. DOM node trees are structured so that all child nodes with the same ChildInfo are in a list. For details, see "Adding Child Nodes".

For example, a "folder" type may contain a number of file and subfolder types. Rather than creating a loop to iterate over those children and initialize all the DOM adapters for each child, ATF provides the DomNodeListAdapter<T> class to perform this multiple initialization. This class is a wrapper for a list of child DOM nodes, making them appear as an IList<T>, which is often more convenient to work with.

To create a DomNodeListAdapter, use a DomNode and the DOM metadata object ChildInfo that identifies the list of child elements of interest:

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

This statement initializes every child of the DomNode that has a ChildInfo type of "Schema.fsmType.stateChild" to the State adapter. The field m_states contains a list of the adapters for each of the child DomNodes that was initialized in the list. Each item in this list can be treated as an instance of State, to which it is adapted.

The DomNodeListAdapter<T> class derives from IList<T> and can be used like any list. Use the DomNode.GetChildList<T>() method to get a list of the child adapters.

Adding Child Nodes

To create a child DOM node and add it to a parent, use the DomNode.SetChild() method. The SetChild() method takes two arguments: a ChildInfo metadata type for the child's metadata, and the child DomNode. You need to create a new node for the child if it does not already exist.

For example, this code from the Fsm Editor sample creates the root node of an fsmType, and then creates a PrototypeFolder child of that root:

```
// create new document by creating a Dom node of the root type defined by schema
node = new DomNode(Schema.fsmType.Type, Schema.fsmRootElement);
// create an empty root prototype folder
node.SetChild(
    Schema.fsmType.prototypeFolderChild,
        new DomNode(Schema.prototypeFolderType.Type));
```

The ChildInfo objects used here are from the schema stub class created by DomGen.

A child of a node can be single DomNode or a list of DomNodes: IList<DomNode>. Children that have the same ChildInfo are all in a single list that is a child of the node. ChildInfo has an attribute indicating whether that child is a list or not. If a child node already exists and is a list of elements, you add new nodes to that list, just as for any collection. This example adds a Prototype instance prototype to a PrototypeFolder folder, which is a collection of Prototype objects:

```
Prototype prototype = node.As<Prototype>();
PrototypeFolder folder = Adapters.As<PrototypeFolder>(m_activeItem);
folder.Prototypes.Add(prototype);
```

For examples of how child nodes are handled as lists, see the Fsm Editor sample.

Getting Child Nodes

There are three DomNode methods that get child DomNodes, given a ChildInfo:

- GetChild(): to get a single child.
- GetChildList(): to get a list of children.
- GetChildren(): to get a single child or list of children.

The DomNode.Children property gets all children of a DomNode for all ChildInfos of that node.

Removing Nodes

To remove a node from its parent, use the `DomNode.RemoveFromParent()` method:

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

Note that you cannot remove the root node of the DOM node tree. Also note that removing a node from the tree does not destroy the node, so the node and any associated metadata objects, such as extensions, still exist.

Modifying Node Attributes

You can use properties in a DOM adapter to modify attributes of a DOM node:

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

When you create DOM adapters, you can implement properties in that adapter that use the `DomNode.GetAttribute()` and `DomNode.SetAttribute()` methods to get and modify attribute data within a DOM node. For more information, see "Wrapping Attributes and Elements in Properties" in "Implementing DOM Adapters".

Initializing DOM Node Adapters

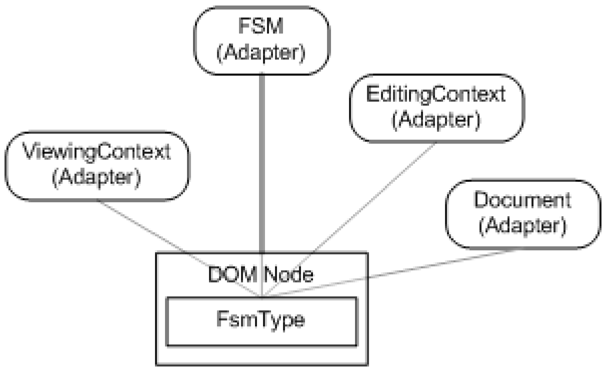
Type definitions often have many DOM extensions or adapters defined for a type. Multiple DOM extensions for a type enable you to provide different APIs for the same underlying DOM data and to listen to different events on nodes. You initialize and obtain an adapter for its node to adapt the node for that adapter's type. You can initialize a node for a different adapter to use the API provided by that adapter as well.

Use the `Is<T>()` method to initialize and test for the compatibility of DOM adapters to DOM nodes. Use the `As<T>()` and `Cast<T>()` methods to initialize DOM adapters to DOM nodes and get the adapter object. These methods operate similarly to their C# equivalents, providing type "conversions" for DOM nodes and DOM adapters. These casting methods are in the `Sce.Atf.Adaptation.Adapters` class as extension methods. Each method takes a generic type parameter `T`, indicating a DOM adapter type, which can be a context interface such as `ILayoutContext`.

As previously noted, you can also initialize extensions with the `DomNode.InitializeExtensions()` method. It is a good practice to call `InitializeExtensions()`on the root DOM node of your document; it initializes any extensions that have not already been initialized.

In Figure 22, each of the DOM adapters (rounded boxes) is defined for the `fsmType` and initialized for use as DOM nodes of that type. You can obtain a reference for each DOM adapter with the `As<T>()` or `Cast<T>()` method and use it as needed.

Figure 22 DOM Adapters and DOM Nodes



The following function determines whether an item can be adapted to a `Prototype` or `PrototypeFolder` type:

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

Use `As<T>()` to perform the same test as `Is<T>()` and also get the DOM adapter for the DOM node, if possible. The `As<T>()` method calls the `OnNodeSet()` method on the DOM adapter (if the DOM adapter was not already initialized) and returns the DOM adapter instance:

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

If the DOM adapter cannot be found, `As<T>()` returns `null`.

The `Cast<T>()` method operates nearly identically to `As<T>()`. However, if the DOM adapter cannot be found, `Cast<T>()` throws an `AdaptationException` exception instead of returning `null`.

The root of a tree of DOM nodes can often be adapted to a context interface, as shown in the following example from the Circuit Editor sample:

```
ITransactionContext transactionContext =
    grpPin.DomNode.GetRoot().As<ITransactionContext>();
```

Here is a more complicated example, from the `EditingContext` class in the Fsm Editor sample:

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

Acquiring an adapter object occurs in the line

```
ILayoutContext layoutContext = this.As<ILayoutContext>();
```

In this statement, `this` refers to an `EditingContext` instance, so at first sight, it does not appear that a `DomNode`’s adapter is being acquired. However, as the following statement from the type loader shows, `EditingContext` is defined for `fsmType`, which is the type of the DOM tree root:

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

When a document is opened, in the `Editor.Open()` method, the DOM tree root node is initialized for `EditingContext`:

```
EditingContext editingContext = node.Cast<EditingContext>();
```

As a result, the `EditingContext` instance `editingContext` used in this sample is actually the root `DomNode` adapted to `EditingContext`, which is then adapted to `ILayoutContext`.

It is worth noting that `EditingContext`, like many contexts, ultimately derives from `DomNodeAdapter`, so this context is a DOM adapter.

DOM Events

The `DomNode` class defines a set of events and event handlers for managing changes to nodes' attributes and child nodes. You can subscribe to the events on any DOM node in the DOM node tree and receive events for that node and all its children. For example, you can define event handlers and subscribe to events in your DOM adapter implementations, such as an editing context or data validator. DOM adapters, such as validators, can use these events to monitor DOM data.

DOM Node Events

Table 10 summarizes the events available for DOM nodes and the fields in the event argument object corresponding to each event. When you define an event handler to handle an event, the event arguments parameter is named `e` by convention.

Note that `DomNode` defines two groups of events: events raised *before* a change happens and events raised *after* the change.

Table 10 DOM Node Events

Event	Event Arguments (e)	Description
`AttributeChanged`	`AttributeEventArgs`	The value of a DOM node's attribute has changed. • `e.DomNode`: The node to which this attribute belongs. • `e.AttributeInfo`: A metadata object representing the attribute that changed. • `e.OldValue`: the old value of the attribute. • `e.NewValue`: the new value of the attribute.
`AttributeChanging`	`AttributeEventArgs`	The value of a DOM node's attribute is about to change. Event arguments are the same as for `AttributeChanged`.

Event	Event Arguments (e)	Description
`ChildInserted`	`ChildEventArgs`	A child of a DOM node has been inserted. The child may have been inserted anywhere in the tree of which this node is the root. • `e.Parent`: The immediate parent node of this child • `e.Child`: The inserted child node. • `e.ChildInfo`: A metadata object representing the child inserted. • `e.Index`: The list index at which the child was inserted
`ChildInserting`	`ChildEventArgs`	A child of a DOM node is about to be inserted. The event arguments are the same as for `ChildInserted`.
`ChildRemoved`	`ChildEventArgs`	A child of a DOM node has been removed. That child may be removed anywhere from the tree of which this node is the root. • `e.Parent`: The immediate parent node of this child. • `e.Child`: The removed child node. • `e.ChildInfo`: A metadata object representing the child removed. • `e.Index`: The list index from which the child was removed.
`ChildRemoving`	`ChildEventArgs`	A child of a DOM node is about to be removed. The event arguments are the same as for `ChildRemoved`.

Subscribing to Events

To use events in a DOM adapter class, subscribe to events in the adapter's `OnNodeSet()` method, and specify an event handler method name. In this example, the name of that handler method is `DomNode_AttributeChanged`.

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

   base.OnNodeSet();
}
```

Also define an event handler for any events you subscribe to, as a `private void` method that takes two arguments: the object that sends the event, and the event arguments parameter (`e`):

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

This example is from the `EditingContext` class in the Fsm Editor sample. In this class, the `AttributeChanged` event is subscribed to in `OnNodeSet()` as shown earlier. In the event handler method `DomNode_AttributeChanged`, if the parent of the node to which the attribute belongs is of the type `fsmType`, the `ItemChanged` event is raised. Item-related events are defined by the `IObservableContext` interface, part of the editing context framework.

Observer and Validator Events

ATF's base classes for observing and validating data add additional events to the DOM that help with managing transactions, selections, and validating data. These events typically operate on larger groups of changes, rather than those for individual attributes and child elements—items or transactions, for example. Depending on the task you want to perform, it may be easier to implement a validator that uses these larger scale events, rather than listen to DOM events on individual nodes.

In addition to the events themselves, the `Observer` and `Validator` base classes also add utility event methods to make listening to DOM events easy. You can derive from these classes and override the event methods to perform operations when an event occurs, without needing to subscribe to a specific event or create an event handler callback. For more information and examples of using such custom event methods, see "Implementing Custom Validators".

Implementing DOM Adapters

To create DOM adapters, extend the `DomNodeAdapter` base class. This class provides the infrastructure for adapting DOM nodes to extensions. In your DOM adapter implementation, you add:

- An override method for `OnNodeSet()` to provide initialization or configuration behavior when the adapter is initialized for its underlying node.
- Properties to get and set DOM nodes' attributes and element lists.
- Implementations for context interfaces for adapters that are contexts.
- Code to manage references between DOM nodes, as well as references to external resources.
- Any additional behavior needed, as methods in your adapter.

Some of the tasks described in this section also apply to DOM extensions that are not DOM adapters, but not tasks such as overriding `DomNodeAdapter` methods like `OnNodeSet()`.

The DomNodeAdapter Base Class

The `DomNodeAdapter` class provides many common methods and properties useful for implementing DOM adapters and working with DOM nodes through the adapter.

The `DomNodeAdapter` base class includes these features:

- It implements the `IAdapter`, `IAdaptable`, and `IDecorator` interfaces to provide adaptation for the DOM node.
- It provides the `DomNode` property to access the adapter's underlying DOM node.

- It provides an `OnNodeSet()` method, which is invoked when the adapter is initialized for the node. This method is called only once per node for each adapter. Override this method in the DOM adapter class. For details on using this method, see "OnNodeSet() Method".
- It provides methods for testing for the existence of a DOM node adapter, and for getting and setting different DOM adapters: `Is<T>()`, `As<T>()`, and `Cast<T>()`. These are all extension methods in the `Sce.Atf.Adaptation.Adapters` class. For more information on these methods, see "Initializing DOM Node Adapters".
- It defines utility methods for manipulating child and parent data of the DOM node, for getting and setting attributes of that node, and for managing references between DOM nodes.

OnNodeSet() Method

Use `OnNodeSet()` to perform initialization or synchronization tasks you want performed whenever a DOM adapter is initialized for a DOM node. The `OnNodeSet()` method is called the first time an adapter is initialized for a node. The `Is<T>()`, `As<T>()`, and `Cast<T>()` methods all call `OnNodeSet()`.

`OnNodeSet()` takes no arguments and returns void. You should always call `base.OnNodeSet()` to ensure that any code in the base class is invoked:

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

   base.OnNodeSet();
}
```

Common tasks for `OnNodeSet()` include:

- Updating any data in the adapter and the node that can get out of sync, for example, positions, bounding boxes, or internal state properties (hidden, locked, and so on).
- Calling `As<T>()` to initialize the node for some other adapter and then assigning the returned adapter to a field. Multiple DOM adapters frequently operate in conjunction with each other on the same DOM data. The `OnNodeSet()` method is the place to make initial connections between adapters.
- Subscribing to events to track changes to data in the DOM node.

For example, this `OnNodeSet()` method is from the `Fsm` class in the Fsm Editor ATF sample. The `Fsm` class is a DOM adapter that adapts DOM nodes of the type `fsmType`. The `Fsm` class contains lists of `State`, `Transition`, and `Annotation` objects. Its `OnNodeSet()` method uses the `DomNodeListAdapter` class to get a list of child adapters from a node and a schema type. For more information on this wrapper class, see "Initializing a List of Child DOM Adapters".

```
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);
}
```

Note: The *Annotation* mentioned here is a class for comments on a canvas and is different from annotations in XML schemas discussed in "Annotations".

The following `OnNodeSet()` method example, also from Fsm Editor, is from the `EditingContext` DOM adapter. The `EditingContext` class adapts the `fsmType` node to add editing capabilities including selections, validations, transactions, history, and so on. This `OnNodeSet()` method casts the fsmType DomNode to two other adapters (Fsm and ViewingContext) and saves those adapters in fields, and then subscribes to a set of event handlers for the DOM node to watch for changes in FSM data:

```
protected override void OnNodeSet()
{
   m_fsm = DomNode.Cast<Fsm>();
   // use the viewing context to access viewing control, for bounds calculations
   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();
}
```

Wrapping Attributes and Elements in Properties

A type's attributes and child node types map easily to properties in C#. Properties enable you to provide an object model for attribute and entity data in the underlying DOM node and shield application code from changes in type definitions.

To implement a property, use the DomNodeAdapter methods GetAttribute<T>() and SetAttribute()to get and set attribute values in the underlying DOM node:

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

Both GetAttribute<T>() and SetAttribute() take as an argument an AttributeInfo DOM metadata object, representing the specific attribute. The generic type parameter T is the type of the attribute, string in this example, which should match the type of the property. In this example, Schema.stateType.labelAttribute (generated by DomGen) is an AttributeInfo object representing the label attribute in "stateType". This code wraps that attribute in the Name property.

You can wrap lists of child DOM nodes representing child elements in the data model as generic IList properties. Use the DomNode.GetChildList<T>() method to get a list of children:

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

GetChildList<T>() takes a ChildInfo DOM metadata argument to indicate which child list to get. The generic type parameter T is the type each child is adapted to—the same type as in the IList—and is Prototype in this example. In this case, as with attributes, the ChildInfo object is obtained from the schema stub class generated by DomGen from the XML schema.

In this example, from the PrototypeFolder class in the Fsm Editor sample, a prototype folder type can contain one or more prototype types as child elements. A separate Prototypes DOM adapter class defines the prototypes themselves. The Prototypes property here is an IList object of all the child Prototype objects contained in the PrototypeFolder.

Note that when you wrap an element with `GetChildList<T>()`, the wrapped element is a read-only `get` property. To define adapter methods to add or remove a child element from the underlying DOM object, use methods available to the adapter class that add or remove child elements from the list.

You can initialize a list of DOM node children to adapters. For information on doing this, see "Initializing a List of Child DOM Adapters".

Using References

References define a relationship between two DOM nodes where neither node is necessarily a parent or a child of the other. These references can be:

- Internal, between two nodes of the same DOM tree.
- External, to resources outside the DOM tree, such as assets or subdocuments. External references are attributes that contain the URI, if any, of the external resource. The resource is only loaded when the application needs it, and you must implement the behavior for creating, loading, and managing these resources.

You can create DOM adapters to manage both internal and external references.

Internal References

Internal references between nodes in the same DOM tree are simple attributes that can be wrapped as properties in the DOM adapter.

Use the `DomNodeAdapter` methods `SetReference()` and `GetReference<T>()` methods to resolve or define an internal reference from one DOM node or another:

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

Both `GetReference<T>()` and `SetReference()` take as an argument an `AttributeInfo` DOM metadata object, representing an attribute. The generic type parameter `T` is the type of the reference, which should match the type of the property. (The `AttributeInfo` DOM metadata objects here are generated from an XML schema by the `DomGen` utility and listed in the schema stub class.) For `SetReference()`, you set the entity that is the target for this reference, such as a `DomNodeAdapter` or `DomNode`.

The previous example, from the Fsm Editor sample's `Transition` class, creates a `FromState` property that is a reference to a `State` object representing the source state of a transition. Here `Schema.transitionType.sourceAttribute` is an `AttributeInfo` object, indicating the source attribute in the transition type. There is a corresponding property for the `ToState` reference in this sample.

External References

External references are references to resources outside the DOM node tree, such as subdocuments. External references are defined as URIs. Your application is responsible for finding, loading, unloading, and managing data inside an external reference.

The Timeline Editor sample uses external references. In this sample, external timeline documents can be included inside a master timeline document. You can manipulate the external document reference as its own timeline reference, or load the document and edit tracks and markers inside the subdocument.

The timeline data model includes a "timelineRefType" type, which is a reference to an external subdocument containing timeline event data. The "ref" attribute of "timelineRefType" is a URI indicating the external document, `<xs:anyURI>` in the schema:

```
<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>
```

The `TimelineReference` DOM adapter represents timeline reference types. This DOM adapter wraps this "ref" attribute in a URI property:

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

You use the same `DomNodeAdapter` methods `GetReference<T>()` and `SetReference()` to get and set external references as you use for internal references.

In addition, this sample adds a `Target` property that, when read, loads the document at the `TimelineReference.Uri` property using ATF's document registry:

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

The `TimelineEditor` class uses `TimelineReference` objects and the `Uri` and `Target` properties to load and manage timeline subdocuments as needed.

Validating DOM Data

Validators are a type of DOM adapter that ensure consistency and integrity of data in the DOM. When DOM data changes, validators check that new data conforms to rules you defined for the data. Although simple data validation for attribute values and child elements is the most common form of validation, DOM validators can also be used for other kinds of validity checking, including checking internal and external references, or checking custom dependencies between different nodes in the tree.

This section contains these topics:

- "About DOM Validators": A summary of DOM validators and data validation rules.
- "Validator Key Classes": A summary of the classes and interfaces used by validators.
- "Using Validator Classes": A set of basic validators you can use in applications.
- "Using Data Validator Rule Classes": Rule classes that implement data validation rules.
- "Implementing Custom Validators": Creating your own validator classes.

About DOM Validators

Use validator DOM adapters to ensure that data in the DOM is correct, consistent, and meaningful for the application at all times. Each time DOM data changes—an attribute is changed, a DOM node is added or removed, or any other aspect of the DOM is modified—the DOM validators you defined for your DOM types check the changed data to make sure it is valid by their criteria.

DOM validation must take place inside the validation context `IValidationContext`, and the `EditingContext` class (in `Sce.Atf.Dom`) can provide this context (because `IValidationContext` is implemented by a class `EditingContext` derives from). In this editing context, validations occur within a transaction. When validations occur within transactions and the change is invalid, the change can be rolled back.

ATF includes classes for common validation operations you can use in applications. These classes include rule-based data validation, reference management, and unique DOM node ID management. To use these basic DOM validators, define the validator for a type in your schema type loader as you would any other DOM adapter. You should generally define validators on the type of the root `DomNode` in your DOM node tree and initialize extensions for the root DOM node using `DomNode.InitializeExtensions()`. This way, these extensions can use any interface implemented by any extension initialized to the root DOM node. One of these adapters, such as `EditingContext`, should implement `IValidationContext`. Also, extensions initialized on the root `DomNode` can listen to events on all `DomNode`s in the tree.

Data validation is the most common type of validation used by the DOM. The `DataValidator` class uses rules defined by the data model to ensure the validity of attributes and child elements of DOM nodes.

If you use XML schema as your data definition language, some validation rules are created for you automatically when your schema is loaded by `XmlSchemaTypeLoader`. As long as you define the DOM data validator for your root type, all the restrictions you use in your data model convert automatically to validators in the DOM. If you use a different type definition language and you've created your own type loader, you need to convert your data restrictions into DOM validation rules and add them to the attribute and child metadata for your types. New rules are easy to create if the existing rule set does not meet your needs.

You can also create new DOM validator classes to perform additional types of data validation. Derive from the `Validator` base class to create new validators, and implement methods that correspond to various events within the DOM node tree. For details, see "Implementing Custom Validators".

Validator Key Classes

The `Sce.Atf.Dom` namespace includes a number of classes related to validators, including classes for validators themselves, and classes of rules for data validation.

Classes for Dom Validation

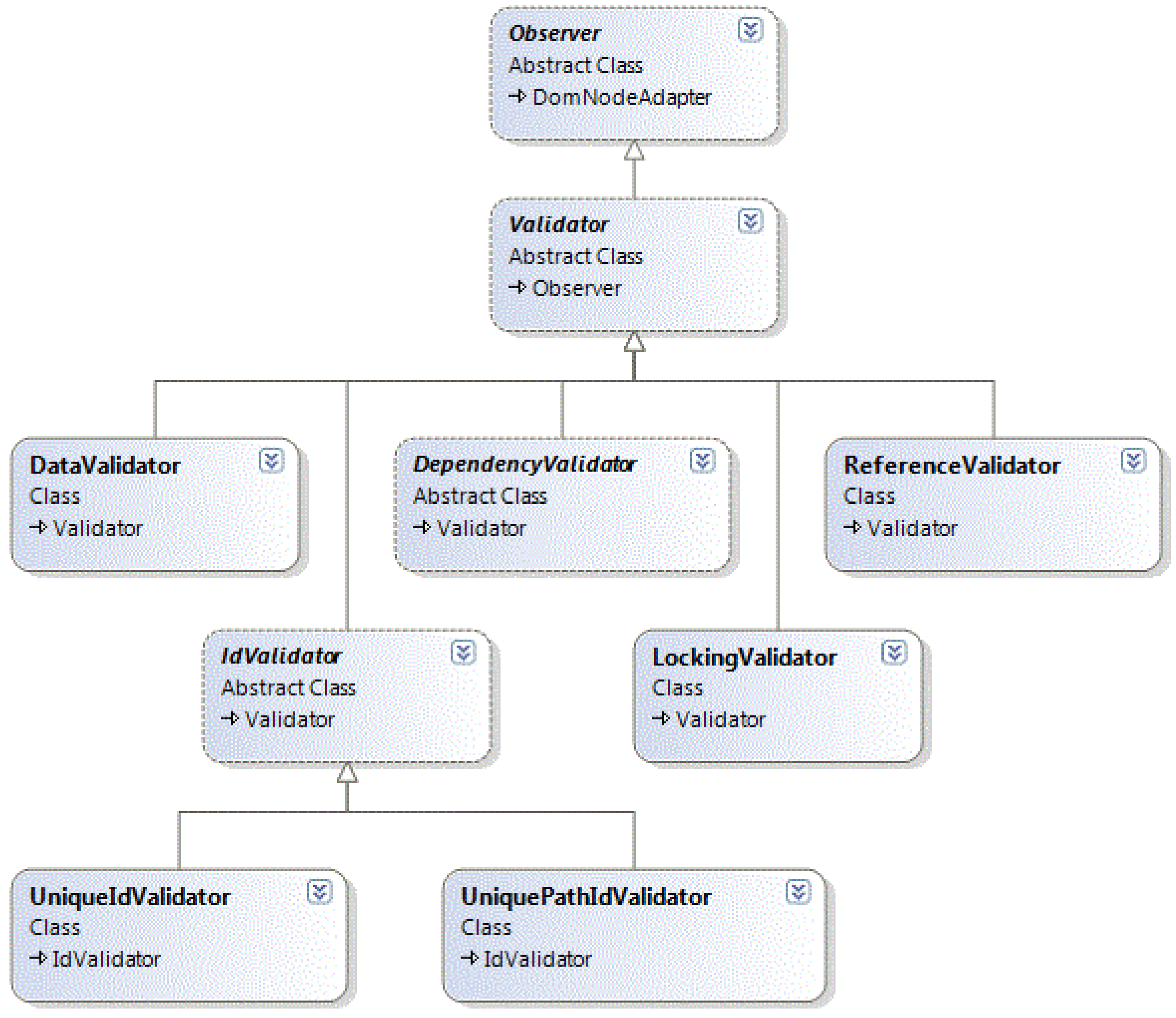
Table 11 outlines the classes and interface related to validation in the DOM. Figure 23 shows the class diagram for these types. `Validator` ultimately derives from `DomNodeAdapter`, so validators are also DOM adapters.

Table 11 Key Classes and Interface in Sce.Atf.Dom for Validators

Class	Description
`Validator`	Abstract base class for DOM validators. Derive from this class to create your own validators. Derives from `Observer`, which derives from `DomNodeAdapter`.

Class	Description
`DataValidator`	DOM validator for attribute and child rules defined by the data model. For more information, see "Using Data Validator Rule Classes". Extends `Validator`.
`IValidationContext`	Interface for contexts that perform validation. This interface should be implemented by at least one DOM adapter defined for a type a validator is also defined for. For more information, see "Implementing Interfaces for DOM Adapters" in "Creating and Initializing DOM Nodes".
`DependencyValidator`	DOM validator that tracks dependencies between DOM nodes. Extends `Validator`. Also see the `DependencySystem<T>` class.
`ReferenceValidator`	DOM validator for internal references between DOM nodes and references to external resources. Extends `Validator`.
`LockingValidator`	DOM validator for locked data in DOM nodes. Requires an adapter that implements `ILockingContext`, that is, `ILockingContext` should be implemented by at least one DOM adapter defined for a type the validator is also defined for. Extends `Validator`. For more information, see "Implementing Interfaces for DOM Adapters" in "Creating and Initializing DOM Nodes".
`IdValidator`	Abstract base class that validates IDs. Extends `Validator`.
`UniqueIdValidator`	DOM validator that ensures a DOM node has a unique ID. Extends `IdValidator`.
`UniquePathIdValidator`	DOM validator that ensures a DOM node has a unique path in the DOM node tree. Extends `IdValidator`.

Figure 23 DOM Validator Classes



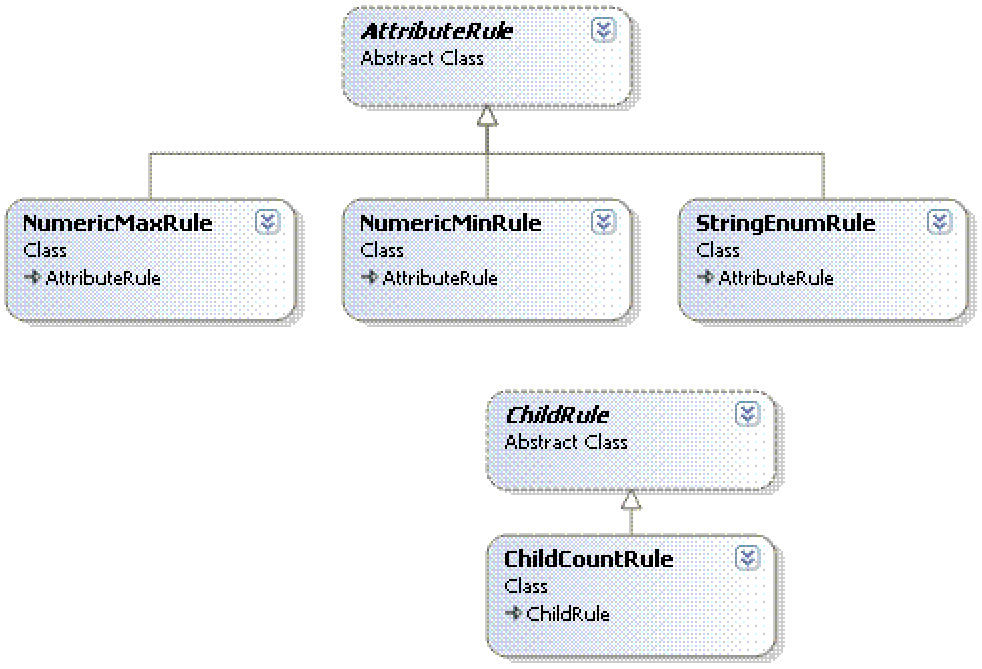
Classes for Data Validator Rules

Table 12 describes classes in `Sce.Atf.Dom` that define rules for data validation performed by the `DataValidator` class. Figure 24 shows the class diagram for these types. For more information about these classes, see "Using Data Validator Rule Classes".

Table 12 Key Classes in Sce.Atf.Dom for Data Validator Rules

Class	Description
`AttributeRule`	Abstract base class for validator rules of attribute restrictions.
`NumericMinRule`	Validator rule for attribute minimum value restriction. Extends `AttributeRule`.
`NumericMaxRule`	Validator rule for attribute maximum value restriction. Extends `AttributeRule`.
`StringEnumRule`	Validator rule for attribute enumeration restriction. Extends `AttributeRule`.
`ChildRule`	Abstract base class for validator rules of child restrictions.
`ChildCountRule`	Validator rule for number of children. Extends `ChildRule`.

Figure 24 Data Validator Rule Types



Using Validator Classes

ATF includes a set of basic validators you can use in applications. The most commonly used validators for DOM data are:

- `DataValidator`: Ensures that attribute and child data conforms to restrictions defined by the data model. It uses rule classes to check restrictions like maximum and minimum attribute values, limits on number of child elements, and string enumerations for attributes. Custom restrictions can be specified in new rule classes. For details, see "Using Data Validator Rule Classes".
- `UniqueIdValidator`: Ensures that every DOM node has a unique internal ID.
- `ReferenceValidator`: Manages internal and external references.

Table 11 lists all DOM validator classes.

To use any of these validators, define them as an extension on a type in the schema loader. To learn how to do this, see "Defining DOM Extensions and Adapters". As previously mentioned, you typically define validators on the data type of the root in your data model, as in these lines from the Fsm Editor sample:

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

DOM validators are also DOM adapters, and DOM adapters are only fully initialized the first time they are used. Because DOM validators are only used indirectly by the application, you must explicitly initialize them for new trees of DOM nodes: when you start the application or when you open a file or create a new one, for example. Use the `DomNode.InitializeExtensions()` method to initialize the extensions for an entire tree of DOM nodes. In this example, `node` is the root `DomNode` of the entire tree:

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

Using Data Validator Rule Classes

The `DataValidator` class validates attribute and child element restrictions defined in your data model. For example, you can define restrictions on a `Size` attribute to be less than 100, or limit a `City` attribute to an enumeration of strings.

Create restrictions in the type definition file for your data model. If you use XML schema, those restrictions are created using standard XSD restriction facets. If you use a different type definition language, you must define your own restrictions.

Restrictions are converted to data validation rules when the schema type loader loads the type definition file by `XmlSchemaTypeLoader`. Instances of the various rule classes listed in Table 12, such as `AttributeRule` and `ChildRule`, are added as part of the attribute or child metadata to the `AttributeInfo` and `ChildInfo` types. For example, a restriction on the maximum value is converted to a `NumericMaxRule`. If you add data validation to the application by defining `DataValidator` as an extension in the type loader and initializing this extension, these rules are automatically used to validate data for attributes and children when that data changes in the DOM.

`DataValidator` performs validation by calling the `Validate()` method of the `AttributeInfo` and `ChildInfo` classes.

Validation rules are simple classes, and it is easy to create your own if you use your own data model or type definition language. Extend the `AttributeRule` or `ChildRule` base classes and implement the `Validate()` method to create your own data validation rules. For examples, see the existing rule classes, such as `NumericMaxRule`, which derives from `AttributeRule`.

Add new rules with these steps:

(1) Create a new rule class by extending the `AttributeRule` or `ChildRule` base classes.

(2) Call the `AddRule()` method on the `AttributeInfo` and `ChildInfo` instances that you want the rule to apply to in your schema loader.

(3) Define a `DataValidator` on a type, so that `Validate()` is called on `AttributeInfo` or `ChildInfo` instances.

(4) Initialize the `DataValidator` extension for `DomNodes`, which may be done with the `DomNode.InitializeExtensions()` method, after data is loaded.

Here is an example of code that could be added to the Fsm Editor's `SchemaLoader` `OnSchemaSetLoaded()` method:

```
Schema.fsmType.Type.Define(new ExtensionInfo<DataValidator>());
Schema.stateType.sizeAttribute.AddRule(new NumericMaxRule(100, true));
```

The first line defines a `DataValidator` extension on "fsmType", which is the type of the root node in the data model, so the validator can listen to change events in all nodes, including nodes of "stateType". In addition, `DataValidator` requires that validation takes place inside the validation context `IValidationContext`, and "fsmType" has another extension defined that implements `IValidationContext`, so this requirement is met. The second line adds a rule to the "Schema.stateType.sizeAttribute" `AttributeInfo`, which represents the size of a "stateType" in the Fsm Editor. The rule class `NumericMaxRule` restricts the maximum value of an attribute, to less than or equal to 100, in this case. If you are running the editor and attempt to enter a size greater than 100 for a selected state object, an error message displays, warning you an invalid attribute value was entered. A new rule class could also have been defined that placed more complex restriction on size values.

For more information, see the `AddRule()` and `Validate()` methods in the `AttributeInfo` and `ChildInfo` classes.

Implementing Custom Validators

If existing ATF validator classes do not suit your needs, you can create your own validator classes by extending the abstract base `Validator` class or a class that extends `Validator`. This base class includes a set of common transaction methods that allow you to watch specific events in the DOM and, when validation transactions occur, to perform checks on the DOM data as it changes. Recall that DOM validation must take place inside the validation context `IValidationContext`. For details on using validators within an `IValidationContext`, see "Implementing Interfaces for DOM Adapters" in "Creating and Initializing DOM Nodes" and "About DOM Validators".

Derive from the `Validator` base class and override the event methods you are interested in following. Table 13 summarizes the available event methods in `Validator` and the methods' parameters. All event methods return `void`. Table 10 shows information on DOM events and their arguments like `AttributeEventArgs`.

Table 13 Validator Event Methods

Event Method	Description
`OnAttributeChanged()`	Called when an attribute in a DOM node in the tree has changed. Overriding this method is the same as listening to the `AttributeChanged` event. Arguments: • `object` (`DomNode` at root of tree) • `AttributeEventArgs`
`AddNode()`	Called when any node (a new node or a child node) was added to the tree. Argument: • `DomNode`: the added node.
`RemoveNode()`	Called when any node was removed from the tree. Argument: • `DomNode`: the removed node.
`OnChildInserted()`	Called when a child in a DOM node tree was inserted. Overriding this method is the same as listening to the `ChildInserted` event. Arguments: • `object` (`DomNode` at root of tree) • `ChildEventArgs`
`OnChildRemoved()`	Called when a child in a DOM node tree was removed. Overriding this method is the same as listening to the `ChildRemoved` event. Arguments: • `object` (`DomNode` at root of tree) • `ChildEventArgs`
`OnBeginning()`	Called at the start of a validation transaction. Arguments: • `object` (an `IValidationContext`) • `System.EventArgs`
`OnCancelled()`	Called if the validation was cancelled. Arguments: • `object` (`IValidationContext`) • `System.EventArgs`
`OnEnding()`	Called just before the validation transaction ends. Arguments: • `object` (an `IValidationContext`) • `System.EventArgs`
`OnEnded()`	Called after validation has ended. Arguments: • `object` (an `IValidationContext`) • `System.EventArgs`

The `Validator` class includes the `Validating` property, which is true when a validation context is active, indicating a validation is currently in process. `Validating` is true after the `OnBeginning()`

method is called, and false after `OnCancelled()` or `OnEnding()`. Use the `Validating` property in the DOM event methods to ensure that a validation transaction is taking place:

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

In addition, if data your validator is testing fails to meet your criteria, and a transaction is taking place, throw the `InvalidTransactionException` exception. This ensures that the transaction can be rolled back, and all the original DOM data preserved.

In this example, the Timeline Editor sample uses a validator to ensure that:

- Timeline event start times are greater than 0 and snap to the nearest integer.
- Interval lengths are greater than 1 and snap to the nearest integer.

The `TimelineValidator` class derives from the `Validator` base class. Both these tests are made in an `OnAttributeChanged()` method. For simplicity, only the test for the timeline event start time is shown here:

```
protected override void OnAttributeChanged(object sender,
    AttributeEventArgs e)
{
    BaseEvent _event = e.DomNode.As<BaseEvent>();
    if (_event != null)
    {
        if (e.AttributeInfo.Equivalent(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
            ...
        }
    }
}
```

Custom validators are demonstrated in the DOM Tree Editor, Circuit Editor, Fsm Editor, and Timeline Editor samples. For ideas on writing your own validators, see the code in these samples. For instance, `TransitionRouter` is a custom validator class (deriving from `Validator`) in Fsm Editor that tracks changes to transitions between states and updates their routing during validation.

5 Other DOM Features & Utilities

This chapter considers DOM features and functionality that do not fit into previous chapters, covering these topics:

- "DOM Persistence": Persisting application data using the DOM.
- "Defining and Using DOM Property Descriptors": Creating property descriptors for properties in the data model to enable property editing in the application.
- "Using DomExplorer": Using the DOM Explorer component to visualize the DOM node tree.

DOM Persistence

Persistent data is application data that lives on in some form after the application has quit, and can be read back in once the application starts again. Typically an application saves persistent data to a file, although that data could also be stored in a database or other service.

In ATF, data in the DOM is most commonly stored persistently as XML documents that use the same schema as the data model. ATF provides two classes, `DomXmlWriter` and `DomXmlReader`, to easily write and read XML data to any .NET stream, such as a file. If you do not want to use XML as your persistent data format, you must write your own classes to read and write data from a persistent data store and interact with the DOM.

This section discusses these topics:

- "DOM Persistence Key Classes": A summary of the most important classes for DOM persistence.
- "Reading XML": Using `DomXmlReader` to read XML data from a stream into the DOM.
- "Writing XML": Using `DomXmlWriter` to write XML data from the DOM to a stream.
- "Using Persistent Data without XML": Writing your own reader and writer classes if you prefer to use a persistent data format other than XML.

DOM Persistence Key Classes

The `Sce.Atf.Dom` namespace includes the classes described in Table 14 for using persistent data with the DOM.

Table 14 Key Classes in Sce.Atf.Dom for DOM persistence

Class	Description
`DomXmlReader`	Reads a stream of XML data, defined by an XML schema, into a DOM node tree.
`DomXmlWriter`	Writes a DOM node tree, defined by an XML schema, to an XML stream.
`XmlSchemaTypeCollection`	A collection of the DOM metadata objects representing the types defined by the XML schema, including namespaces, elements, attributes, and child elements. `DomXmlWriter` uses the `XmlSchemaTypeCollection` data to write an XML file from a DOM node tree.

Class	Description
`XmlSchemaTypeLoader`	Base class for schema type loaders. Derive from this class to create your own schema type loader for the application. `DomXmlReader` converts XML tags in the input to the appropriately-typed DOM nodes, using information from the schema loader, such as the `XmlSchemaTypeCollection` and node types. `DomXmlWriter` uses the schema metadata information, an `XmlSchemaTypeCollection` contained within the schema loader, to write an XML file from a DOM node tree.

Reading XML

Use the `DomXmlReader` class to read XML content from an input stream and create a DOM node tree. The stream you read from is often a file, but can be any input device. ATF uses the standard .NET stream classes from `System.IO`.

Reading and writing XML files for the application is usually part of your implementation of the ATF `IDocument` interface, say in an `Editor` class in your application. To read XML content from a stream, you need an instance of your schema loader, derived from `XmlSchemaTypeLoader`. Your `Editor` class can get an instance of your schema loader in its constructor.

The following code for the `IDocument.Open()` method comes from the Fsm Editor sample, but is nearly the same in all ATF samples:

```
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
        node = new DomNode(Schema.fsmType.Type, Schema.fsmRootElement);
        // create an empty root prototype folder
        node.SetChild(
            Schema.fsmType.prototypeFolderChild,
                new DomNode(Schema.prototypeFolderType.Type));
    }

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

This example opens an XML document specified by a local file using the `System.IO.Path`, `File`, and `FileStream` classes. A new instance of `DomXmlReader` is created with the schema loader object `m_schemaLoader`. Then `DomXmlReader.Read()` reads the data in that file stream and creates a tree of `DomNode` instances from the XML data in that file. `Read()` returns a new instance of `DomNode`, which is the root node of the new DOM node tree.

The second part of this `if` statement is used when the requested file does not exist; the `IDocument.Open()` method doubles as the way to create a new document. This part creates a new root DOM node for an empty DOM node tree.

Note that after you read an XML file, there are usually other steps you need to take to initialize the DOM data so that it performs as a document. For example, you need to create a document, create viewable and/or editable contexts for the DOM data, and to initialize the extensions for data types with the `Is<T>()`, `As<T>()`, or `Cast<T>()` methods. Your new document and contexts may use controls and adapters that need to be added and initialized. You may need to call the `DomNode.InitializeExtensions()` method to ensure that all the other adapters, such as validators, have been initialized.

For simplicity, some code was omitted from the previous example, but you can view it in the Fsm Editor sample or other ATF samples.

Writing XML

Use the `DomXMLWriter` class to write a DOM node tree to an output stream. The stream you write to is often a file, but can be any output device. ATF uses the standard .NET stream classes from `System.IO`.

As with `DomXmlReader`, you can use `DomXmlWriter` as part of your implementation of the `IDocument` interface, say in an `Editor` class in your application. For example, the following code for the `IDocument.Save()` method comes from the Fsm Editor sample, but is nearly the same in all the ATF samples:

```
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);
    }
}
```

First, this example obtains the path of a new or existing file from the `uri` argument. It then creates a new instance of `DomXmlWriter` with the type collection from the schema loader, so that the writer knows which XML tags to write. It casts the incoming `IDocument` parameter to the application's document type (`Document`), a DOM adapter, so the `DomNodeAdapter.DomNode` property is available. Finally, the `Write()` method, given the root of the DOM tree, writes the document data to the open file stream.

Using Persistent Data without XML

If you don't want to use XML as the persistence format for your DOM node trees, you need to create your own reader and writer classes. You can use `DomXmlReader` and `DomXmlWriter` as starting points for your own code. The Simple DOM No XML Editor sample provides an example of an application that uses the DOM, but does not use XML for its data model or for its persistence format. `Read()` and `Write()` methods in its `EventSequenceDocument` class read and write persistent data. For details, see this sample's code.

Defining and Using DOM Property Descriptors

In .NET, *property descriptors* add additional metadata to class properties for controls such as property editors. Similarly, *DOM property descriptors* are custom descriptors that describe properties of DOM nodes—the nodes' attributes and child nodes. Create descriptors for properties to enable DOM node data property editing in your application.

This section addresses these topics:

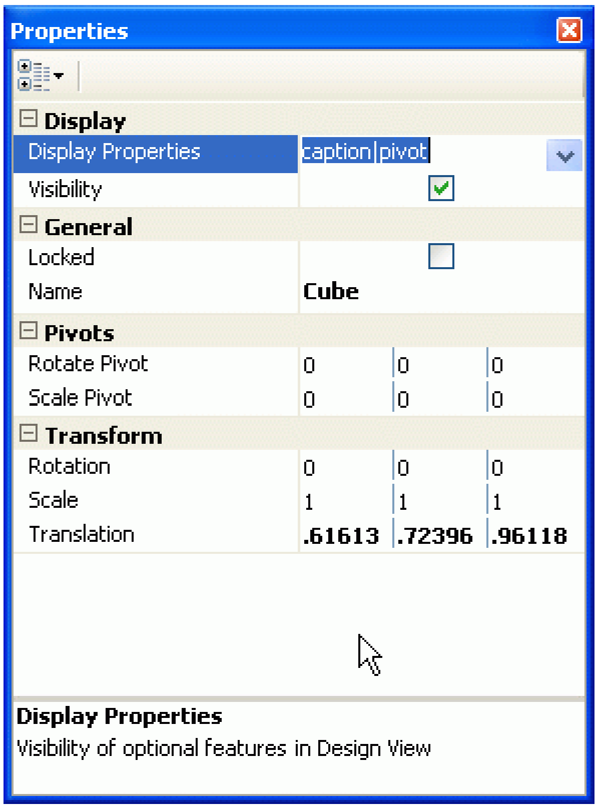
- "About DOM Property Descriptors": An overview of how property descriptors are used in the DOM and the kind of information they contain.
- "Property Descriptor Key Classes": A summary of the classes and interfaces used by property descriptors.
- "Defining Property Descriptors in the Schema Loader": How to define property descriptors directly in code using the schema loader.
- "Defining Property Descriptors in XML Schema Annotations": An alternate method of defining property descriptors external to the code, in which property descriptors are defined using annotations to XML schema. This method was used in ATF 2 and can be used in later ATF versions.

About DOM Property Descriptors

A property descriptor is an abstraction for a class property. It provides information about a property that can be used by external components and controls, such as property editors. Information provided by a property descriptor can include such items as the display name of a property, descriptive text for tooltips, or the editor control to be used to edit that property. Creating a property descriptor means that a property can be viewed and edited in a property editor control with very little coding effort. If information is not provided about a property in a property descriptor, the property cannot be used in a property editor.

The .NET `System.ComponentModel` namespace defines the core `PropertyDescriptor` and `PropertyDescriptorCollection` classes. ATF extends `System.ComponentModel.PropertyDescriptor` to provide an abstract `Sce.Atf.Dom.PropertyDescriptor` class that can itself be extended to describe attributes and children of `DomNode` objects. ATF's `PropertyEditor`, shown in Figure 25, and `GridPropertyEditor` components both use property descriptors.

Figure 25 Property Editor



Properties

Display			
Display Properties	caption|pivot		
Visibility	✓		
General			
Locked	☐		
Name	**Cube**		
Pivots			
Rotate Pivot	0	0	0
Scale Pivot	0	0	0
Transform			
Rotation	0	0	0
Scale	1	1	1
Translation	**.61613**	**.72396**	**.96118**

Display Properties
Visibility of optional features in Design View

ATF extends its abstract class `Sce.Atf.Dom.PropertyDescriptor` to define property descriptors for the following entities:

- Attributes: Because `DomNode` attributes map directly to properties, these are the most common type of property descriptors. The `AttributePropertyDescriptor` class defines these descriptors.
- Child attributes: Child attributes are the same as regular attributes, but are defined on a child node and include child metadata `ChildInfo`. The `ChildAttributePropertyDescriptor` class defines these descriptors.
- Child nodes. Child nodes can be associated with property descriptors, defined by the `ChildPropertyDescriptor` class.

Property descriptors are created as additions to an application's data model and are added to DOM metadata. The easiest place to create these additions is in the schema loader class of the application. For information on defining property descriptors using this method, see "Defining Property Descriptors in the Schema Loader".

Alternately, ATF 2 used annotations in the XML schema for the data model to define property descriptors, and those annotations were automatically converted into code when the XML schema was parsed at runtime. ATF 3 supports some of those legacy XML annotations. You can use this code as the basis for supporting your own annotations of property descriptors. For more information, see "Defining Property Descriptors in XML Schema Annotations".

The constructors for the DOM's custom property descriptors use a set of parameters listed in Table 15. Only the `name` and `attribute/attributeInfo` parameters are required, so the other parameters can be `null`. Each property descriptor has several constructors, so you can use the constructor with the parameters you need.

Table 15 DOM Property Descriptor Parameters

Parameter	Type	Description
`name`	`string`	Name of the property.
`attribute, attributeInfo`	`AttributeInfo`	`AttributeInfo` that defines the property.
`childInfo`	`ChildInfo`	`ChildInfo` identifying the child.
`category`	`String`	Category of the property, for properties that are organized in groups in property editors.
`description`	`String`	Description of the property, as displayed in the property grid or property editor as help text or tooltip.
`isReadOnly`	`Boolean`	Whether the property can be changed. Affects only the ability to edit the property in the property editor; the underlying DOM attribute may not be read-only. By default, all properties are editable.
`editor`	`object`	The property editor control that should be used to change this property's value. Property editors are available in the `Atf.Gui` assembly, in the `Controls/PropertyEditing` folder. For examples, see the DOM Tree Editor sample.
`typeConverter`	`TypeConverter`	Class for converting from one type to another. DOM property-related type converters are available in the `Atf.Gui` assembly, in the `Controls/PropertyEditing` folder. For examples, see the DOM Tree Editor sample.
`attributes`	`System.Attribute` array	Array of `System.Attributes` describing custom attributes for this property.

Property Descriptor Key Classes

Table 16 lists the classes in `Sce.Atf.Dom` that define property descriptors.

Table 16 Key Classes in Sce.Atf.Dom for Property Descriptors

Class	Description
`PropertyDescriptor`	Abstract base class for DOM property descriptors. Extends `System.ComponentModel.PropertyDescriptor`.
`AttributePropertyDescriptor`	Property descriptor for attribute of DOM node. Extends `Sce.Atf.Dom.PropertyDescriptor`.
`ChildPropertyDescriptor`	Property descriptor for child DOM node. Extends `Sce.Atf.Dom.PropertyDescriptor`.
`ChildAttributePropertyDescriptor`	Property descriptor for attribute of child node. Extends `AttributePropertyDescriptor`.

Defining Property Descriptors in the Schema Loader

You can define property descriptors directly in the type loader class for the data model, typically in the schema loader. This allows you to keep schema modifications and additions (such as extensions) and property descriptor definitions together in a single class, but if you change the property descriptors, you need to recompile your code.

Each property descriptor belongs to a specific `DomNodeType` DOM metadata object, as defined in the `Schema` class. Use the `NamedMetadata.SetTag()` method to define additional metadata—including property descriptors—for a type. This example from the Fsm Editor sample defines property descriptors (not all are shown) for the `stateType` type's attributes. It determines which properties appear in the property editor for a selected state object in the application editor:

```
Schema.stateType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                // 'nameAttribute' is unique id, label is user visible name
                Schema.stateType.labelAttribute,
                null,
                Localizer.Localize("State name"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Size"),
                Schema.stateType.sizeAttribute,
                null,
                Localizer.Localize("State size"),
                false),
    ...
}));
```

For property descriptors, pass the `SetTag()` method a new `System.Componentmodel.PropertyDescriptorCollection` object. The constructor for the `PropertyDescriptorCollection` class, in turn, takes an array of `PropertyDescriptor` objects. Each individual property descriptor is an instance of `PropertyDescriptor` or one of its derived classes, listed in Table 16. Table 15 describes the parameters for the various `PropertyDescriptor` constructors.

This `PropertyDescriptor` information must be provided if you want to be able to edit the property in a property editor. For instance, if the code creating the `AttributePropertyDescriptor` for the `Schema.stateType.labelAttribute` were removed, the property editor for a state in the Fsm Editor would have no "Name" field, so that property could not be viewed or edited. The constructor parameters also determine how the property is edited. If the `isReadOnly` parameter (last one) in the `AttributePropertyDescriptor` were changed to `true`, for instance, the value of "Name" could be viewed, but not changed.

In the previous example, the `stateType` type has a property descriptor for the "Name" attribute. In this case, the constructor for the `AttributePropertyDescriptor` has five arguments:

- The display name of the property (a localized string, "Name" in this example).
- The metadata that defines this property (in this case the `AttributeInfo` object defined by `Schema.stateType.labelAttribute` from the schema stub class).
- The category of the property, if any (`null` in this example).
- The description of the property, for tooltips in property grids (the localized string "State name").
- The "read-only" Boolean (`false` in this example, because the name is editable).

Here is another example, from the DOM Tree Editor sample, of a `ChildAttributePropertyDescriptor`. This property descriptor is defined on the `UIControlType` type:

```
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()),
                    ...
        }));
```

The constructor for this example adds several arguments to those from the previous `AttributePropertyDescriptor` example:

- The metadata that defines the child information for which this attribute applies, in this case the `ChildInfo` object defined by `UISchema.UIControlType.TransformChild`.
- The editor control with which to edit this property in a property editor, and any parameters to that editor control. In this example, the editor is a `NumericTupleEditor` from `Sce.Atf.Controls.PropertyEditing`.
- The type converter for data in this property, in this example a `FloatArrayConverter` from `Sce.Atf.Controls.PropertyEditing` to convert back and forth between strings and floating point numbers.

Defining Property Descriptors in XML Schema Annotations

You can also define property descriptors using annotations in the XML schema for the application's data model. The advantage of storing property descriptor definitions in the XML schema—as opposed to defining them in the schema loader—is that all the information that defines the data model is then located in a single place. In addition, because the annotations for property descriptors are loaded with the XML schema at application runtime, non-programmers or designers can make changes to those properties without needing access to the code. Changing annotations doesn't require rebuilding the application either.

For information about annotation format and supported annotations in the XML Schema, see "Annotations". The two annotations defining property descriptors are:

- `scea.dom.editors.attribute`: associate a property descriptor with an attribute.
- `scea.dom.editors.child`: associate a property descriptor with children of this type.

Annotations are processed by the `ParseAnnotations()` method called in the schema loader, but it provides only minimal support for annotations unless it is augmented. To learn how to support these annotations that create property descriptors, see "ParseAnnotations() Method".

Note: If your annotation format differs from those currently supported, or if you want to support additional property descriptor types, you need to write additional code to parse those annotations and create instances of `PropertyDescriptor` in your `ParseAnnotations()` method. You also need to do this if you use a type definition file other than XML schema for your data model.

Annotations for Property Descriptors

The XML schema annotations supported by the `PropertyDescriptor` class are in the same format used by legacy ATF 2 applications. The `PropertyDescriptor.ParseXml()` method supports annotations for attributes (`scea.dom.editors.attribute` tag) and child elements (`scea.dom.editors.child` tag).

For example, this annotation for "intervalType" (from the Timeline Editor sample) contains three property descriptor annotations for attributes: name, length, and color:

```
<xs:complexType name="intervalType">
  <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,Atf.Gui"
          converter="Sce.Atf.Controls.PropertyEditing.IntColorConverter" />
      </xs:appinfo>
    </xs:annotation>
    ...
  </xs:complexType>
```

The `ParseXml()` method processes these annotation and creates property descriptors based on their information.

An additional annotation here, `scea.dom.editors`, describes the type itself and is used to display this type in a palette. The `PropertyDescriptor` class does not process this annotation. To learn how this annotation is processed in the Timeline Editor sample, see the `ParseAnnotations()` method in its `SchemaLoader` class.

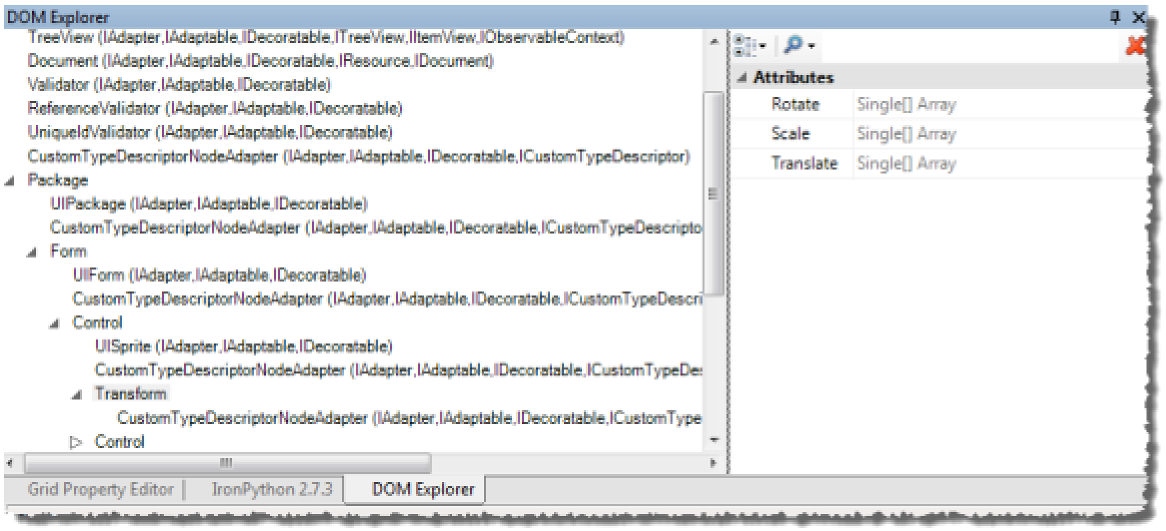
Processing Property Descriptor Annotations

Property descriptor annotations are processed in the `ParseAnnotations()` method in your schema loader class. Call `PropertyDescriptor.ParseXml()` and `NamedMetadata.SetTag()` to parse the XML schema annotations, create an instance of `PropertyDescriptorCollection`, and add that collection as metadata to a DOM metadata object. Note that you can add any kind of metadata to a `DomNodeType` with `SetTag()`. For more information on `ParseAnnotations()` and an example of using it, see “ParseAnnotations() Method”.

Using DomExplorer

`DomExplorer`, in the `Sce.Atf.Dom` namespace, is a MEF component you can use to visualize the contents of a DOM node tree. The Dom Tree Editor sample shown in Figure 26 uses the `DomExplorer` class.

Figure 26 Dom Explorer



```
DOM Explorer
    TreeView (IAdapter,IAdaptable,IDecoratable,ITreeView,IItemView,IObservableContext)
    Document (IAdapter,IAdaptable,IDecoratable,IResource,IDocument)
    Validator (IAdapter,IAdaptable,IDecoratable)
    ReferenceValidator (IAdapter,IAdaptable,IDecoratable)
    UniqueIdValidator (IAdapter,IAdaptable,IDecoratable)
    CustomTypeDescriptorNodeAdapter (IAdapter,IAdaptable,IDecoratable,ICustomTypeDescriptor)
 ◢  Package
      UIPackage (IAdapter,IAdaptable,IDecoratable)
      CustomTypeDescriptorNodeAdapter (IAdapter,IAdaptable,IDecoratable,ICustomTypeDescripto
   ◢  Form
        UIForm (IAdapter,IAdaptable,IDecoratable)
        CustomTypeDescriptorNodeAdapter (IAdapter,IAdaptable,IDecoratable,ICustomTypeDescri
     ◢  Control
          UISprite (IAdapter,IAdaptable,IDecoratable)
          CustomTypeDescriptorNodeAdapter (IAdapter,IAdaptable,IDecoratable,ICustomTypeDe
       ◢  Transform
            CustomTypeDescriptorNodeAdapter (IAdapter,IAdaptable,IDecoratable,ICustomType
       ▷  Control
```

◢ Attributes	
Rotate	Single[] Array
Scale	Single[] Array
Translate	Single[] Array

Grid Property Editor | IronPython 2.7.3 | DOM Explorer

The left pane of the editor shows a tree view of all the nodes in a DOM node tree. Nodes are listed by name, from the `DomNodeType.Name` property. All of the node's adapters are listed as its children, with all the interfaces that node implements in parentheses after the adapter name. For example, in Figure 26 the "Package" node has two adapters:

- `UIPackage`, the primary DOM adapter for this node.
- `CustomTypeDescriptorNodeAdapter`, an adapter added to all nodes to manage property descriptors.

In addition to these adapters, the "Package" node contains a "Form" child node (with its corresponding `UIForm` and `CustomTypeDescriptorNodeAdapter` adapters), which in turn contains a "Control" node (with `UISprite` and `CustomTypeDescriptorNodeAdapter` adapters).

The right pane of the DOM explorer component contains a property editor that enables you to view the types of attributes in the node, if any. The property grid is for information only; you cannot edit attribute values. In this example, the highlighted "Transform" node has three attributes: "Rotate", "Scale", and "Translate", all arrays of type `Single`.

You can add DOM Explorer to an application by adding the `DomExplorer` component to the type catalog of the program, as you would any other MEF component, usually in the `Program.cs` file in the `Main()` method:

```
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
    ...
);
```