

ATF Code Samples

(日本語で表示 

The following sample applications are shipped with the ATF in the ATF\Samples directory. The sample applications show how to use ATF components to build applications with particular elements and functionality. To try them, open and build one of the Visual Studio solutions in \Samples. The sample applications' .exe's are placed in each sample application's bin folder.

The sample applications build on each other, from the simplest functionality of File Explorer's file hierarchy display to the more complex Diagram Editor, which combines several sample applications to show how multiple applications can share one application shell.

Click each sample's link to view its documentation. Also, it may be helpful to [view a chart](#) showing which key technologies are in which sample applications.

Code Sample	Purpose
Circuit Editor	<p>This sample is an editor for circuits, consisting of modules with input and output pins and connections between them. It uses an XML schema to define the data file format, reads and writes XML circuit files, and allows circuits to be edited using a graphical representation of modules and connections. It uses the AdaptableControl to display and edit the circuit. Multiple documents can be edited simultaneously. CircuitEditor uses several ATF Editor components to implement standard editing commands. CircuitEditor also demonstrates:</p> <ul style="list-style-type: none">• Prototyping: you can create a custom set of circuit fragments that can be inserted into documents.• Layering: you can assign diagram components to layers which can be turned on and off (visible and invisible).
Code Editor	This sample is a code editor that uses the ActiproSoftare SyntaxEditor to provide an editing Control. It provides language syntax sensitive editing for plain text, C#, Lua, Squirrel, Python, XML, COLLADA and Cg files.
Diagram Editor	This sample editor combines the Circuit, Finite State Machine, and State Chart editors into a single application to show how multiple editors can share an application shell and editor components. Note that not all of the related features of each of the other editors are included, so if you are mainly interested in circuits, for example, please take a look at the Circuit Editor sample app.
DOM Tree Editor	This sample is an editor that operates on simple User interface definition files. The UI data is hierarchical and DomTreeEditor displays is in a TreeControl. You can select and edit UI elements in the tree, and you can edit the properties on the selected elements.
File Explorer	This sample shows how to build a simple Windows Explorer-like application to view the contents of your hard drive. It demonstrates ATF features including the use of MEF to put applications together and develop a file view, use of the application shell framework, use of controls such as TreeControl, use of ATF list components to list files in the selected file folder, use of the UserFeedbackService for bug reporting, and use of the VersionUpdateService for upgrading to the latest SHIP version of the application.
Fsm Editor	This sample is a finite state machine editor. You can drag states from a palette onto a canvas and connect the states with transitions. A list or grid property editor allows you to edit states and transitions. You can create prototypes from selected states and transitions.
Model Viewer	This model viewer shows how to use the ATF's ATGI and Collada file loaders and OpenGL rendering.
Property Editing	This simple property editor sample shows how to use both standard and grid property editors.
Simple DOM Editor	This sample editor demonstrates the use of the DOM (Document Object Model), including defining a data model. It shows how to implement IDocumentClient and use the document framework to manage multiple documents, how to implement File menu items and more. It shows how to implement a UI parts palette and how to display editable lists of events and resources, how to adapt data to a list, and how to use ContextRegistry to track the active editing context. It also demonstrates how to adapt data so that ATF command components can be used to get undo/redo, cut/paste, and selection commands, how to enable property editing on selected UI elements, and how to implement a standard Help/About dialog.

Simple DOM No XML Editor	This sample editor is very similar to the Simple DOM Editor sample, but does not use XML. It demonstrates the use of the DOM (Document Object Model), but does not use an XML schema for its data model. It shows how to implement IDocumentClient and use the document framework to manage multiple documents, how to implement File menu items and more. It shows how to implement a UI parts palette and how to display editable lists of events and resources, how to adapt data to a list, and how to use ContextRegistry to track the active editing context. It also demonstrates how to adapt data so that ATF command components can be used to get undo/redo, cut/paste, and selection commands, how to enable property editing on selected UI elements, and how to implement a standard Help/About dialog.
State Chart Editor	This sample is an editor for statecharts. It uses an XML schema to define the data file format, reads and writes XML statechart files, and allows them to be edited using a graphical representation of states and transitions. It uses the AdaptableControl to display and edit the statechart. Annotations, which are comments pasted on to the document canvas, can be added. Multiple documents can be edited simultaneously. Many ATF Editor components are used to implement standard editing commands. StatechartEditor also demonstrates prototyping: how the user can create a custom set of statechart fragments that can be inserted into documents.
Target Manager	This sample shows how to use the TargetEnumerationService to discover, add, configure and select targets, which are network endpoints, such as TCP/IP addresses, PS3 DevKits (to be added) or Vita DevKits. It demonstrates ATF features, such as using the application shell framework and target plugins.
Timeline Editor	This sample is a relatively full-featured timeline editor whose components have been used in real production tools. It shows how to use MEF to put an application together using optional components, use of the application shell framework, how to use the timeline manipulators and a manipulator architecture that allows adding or removing a TimelineControl's functionality without modifying the TimelineControl, how to use the Palette as a parts depot for timeline objects, use of the property and grid property editors, sub-document support, how to enable opening multiple documents simultaneously, and copy and paste within and between documents.
Tree List Editor	This sample editor shows how to create and add entries to various kinds of Tree lists, including a hierarchical list to display selected folders' underlying folders and files. It demonstrates ATF features such as using the application shell framework, use of controls such as TreeListView and use of the SettingsService to persist list column widths. The sample shows adding and removing list items and notifying the user of these events.
Using Direct2D	This sample application demonstrates how to use Direct2D and ATF classes that support Direct2D. It does not use MEF.
Using Dom	This sample application is a simple demo of basic DOM use. It has no UI, running in a command prompt window.
Win Forms App	This is a basic WinForms sample application. It illustrates how to compose a WinForms application with ATF components using MEF.
Wpf App	This is a basic WPF sample application. It illustrates how to compose a WPF application with ATF components using MEF.

ATF Code Samples_j

(View in English)

次のサンプルアプリケーションは、ATF の ATF\Samples

ディレクトリに同梱されています。サンプルアプリケーションは、特定の要素や機能を持つアプリケーションを作成するための ATF コンポーネントの使用方法を示します。これらを試用するには、Visual Studio ソリューションのいずれかで \Samples を開きビルドします。サンプルアプリケーションの .exe ファイルは、各サンプルアプリケーションの bin フォルダにあります。

File Explorer のファイル階層表示の簡単な機能から、複数のサンプルアプリケーションを組み合わせて、複数のアプリケーションが 1 つのアプリケーションシェルを共有する方法を示す、より複雑な Diagram Editor まで、サンプルアプリケーションはお互いをベースにして作成されています。

ドキュメントを表示するには、各サンプルのリンクをクリックします。また、どの主要な技術がどのサンプルアプリケーションで使用されているかを示す一覧表も役立ちます。

Timeline Editor	このサンプルは、比較的機能豊富なタイムラインエディターで、そのコンポーネントは製品用ツールに実際に使用されています。次の例が示されています。 <ul style="list-style-type: none"> オプションのコンポーネントを使用したアプリケーションをまとめるための MEF (Managed Extensibility Framework) 使用法 アプリケーションシェルフレームワークの使用 TimelineControl を変更せずに新機能の追加および削除を可能にする、タイムラインマニピュレータおよびマニピュレーターアーキテクチャ タイムラインオブジェクトのバーツ格納用にパレットを使用する方法 プロパティエディターおよびグリッドプロパティエディターの使用 サブドキュメントのサポート 複数のドキュメントを開き、ドキュメント内およびドキュメント間でコピー、貼り付けをする方法
Tree List Editor	このサンプルエディターは、選択されたフォルダの下層フォルダおよびファイルの階層構造リストなど、さまざまな種類のツリー構造リストの作成を実現するためのツリー構造リストの使用例を示しています。また、リスト項目の追加と削除、およびユーザに対するそれらイベントの処理方法も示されています。
Using Direct2D	このサンプルアプリケーションは、Direct2D および Direct2D をサポートする ATF クラスの使用例を示しますが、MEF は使用しません。
Using Dom	このサンプルアプリケーションは、DOM の基本的な使用法を簡単に示します。UI ではなく、コマンドプロンプトウィンドウ内で実行します。
Win Forms App	基本的な WinForms のサンプルアプリケーションです。MEF を使用して、ATF コンポーネントを備えた WinForms アプリケーションを作成する方法を示します。
Wpf App	基本的な WPF のサンプルアプリケーションです。MEF を使用して、ATF コンポーネントを備えた WPF アプリケーションを作成する方法を示します。

ATF Circuit Editor Sample

(日本語で表示 )

Description

CircuitEditor is a sample editor for circuits, consisting of modules with input and output pins and connections between them. It uses an XML Schema to define the data file format, reads and writes XML circuit files, and allows circuits to be edited using a graphical representation of modules and connections. It uses the AdaptableControl to display and edit the circuit. Multiple documents can be edited simultaneously. CircuitEditor uses several ATF Editor components to implement standard editing commands. CircuitEditor also demonstrates:

- Prototyping: you can create a custom set of circuit fragments that can be inserted into documents.
- Layering: you can assign diagram components to layers that can be turned on and off (visible and invisible).
- [Circuit Groups and Circuit Templates](#).

For details of programming this sample, see [Circuit Editor Programming Discussion](#). For information about graph handling generally in ATF, see [Graphs in ATF](#).

ATF Features Demonstrated by CircuitEditor

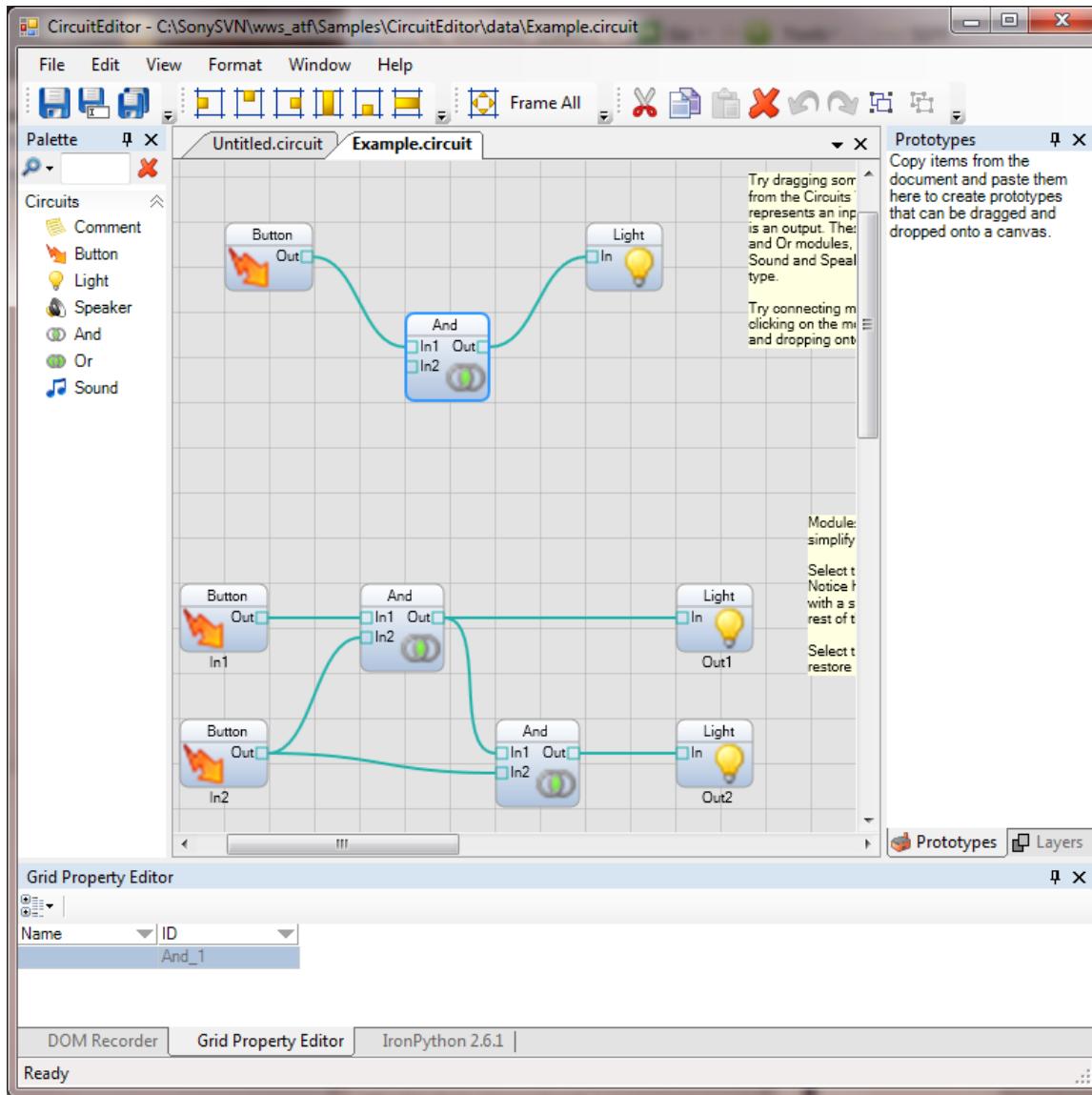
- Define a data model using a Circuit.xsd XML schema.
- Use of DOM to store a data model in memory.
- Use of adapters to decorate the DOM to create a Circuit data model.
- Use of ContextRegistry to track the active editing context, so application components always apply where the user is editing.
- Use of AdaptableControl to display and edit circuits using graph abstractions.
- Use of TransformAdapter, CanvasAdapter, and ViewingAdapter to implement the Circuit canvas.
- Use of ScrollbarAdapter, AutoTranslateAdapter, MouseTransformManipulator, and MouseWheelManipulator to allow the user to pan and zoom the Circuit canvas.
- Use of D2dGraphAdapter, D2dGraphNodeEditAdapter, and D2dGraphEdgeEditAdapter to display and allow editing of circuit modules and connections.
- Use of HoverAdapter to display information when the mouse hovers over circuit module items.
- Use of D2dAnnotationAdapter to display annotations and edit their text on the canvas.
- How to use the global (Windows) clipboard, so that users can copy/paste into other ATF circuit diagram editor samples.
- Use of PropertyEditor and GridPropertyEditor components to edit properties in list and grid property controls.
- [Circuit Groups and Circuit Templates](#), which are described [here](#).

Modules

- Editor.cs implements IDocumentClient and uses document framework to manage multiple documents. It implements File menu commands, auto-new and open documents on startup.
- ModulePlugin.cs shows how to implement IPaletteClient and uses IPaletteService to create the Circuit Module parts palette.
- PrototypingContext.cs shows how to implement IPrototypingContext and uses the PrototypeLister component to allow prototyping.
- LayeringContext.cs shows how to implement ILayeringContext and uses the LayerLister component to allow editing and showing/hiding layers.
- GroupingCommands.cs shows how to implement group/ungroup commands: replace selected modules with a single module that contains the original modules and preserves all connections and the reverse process.
- MasteringCommands.cs shows how to implement master/un-master commands: create a new module type from the selection, replace the selection with an instance of the new module type, and the reverse process.

Run CircuitEditor

1. Double-click the CircuitEditor.exe in ATF\Samples\CircuitEditor\bin\Release.
2. The CircuitEditor window appears.
3. You can open a sample file data\Example.circuit in the Circuit Editor project files.



CircuitEditor has the following panes:

- Palette (Circuit Modules): the Circuit parts palette: Comment, Button, Light, Speaker, And, Or, and Sound.
- Canvas: define, view, and edit circuits.
- Property Editor: edit the selected module's property in a list control.
- Grid Property Editor: edit the selected module's property in a grid control.
- Prototypes: lists the custom Circuit fragments that you define for use in your circuits.
- Layers: list layers and their circuit modules.

The toolbar contains buttons for file management: Save, Save As, and Save All; for aligning: Lefts, Tops, Rights, Centers, Bottoms, and Middles; for framing: Selected and All; and for editing: Cut, Copy, Paste, Delete, Undo/Redo, and Group/Ungroup.

The menu bar contains:

- File: create a new circuit, open an existing circuit, Save, Save As, Save All, Close, Recent Files, and Exit CircuitEditor.
- Edit: in addition to the standard editing functions (Cut, Copy, Paste, Undo/Redo, and so on), Edit provides:
 - Selection: Select All, Deselect All, Invert Selection.
 - Master/Unmaster: create/disassemble a master circuit.
 - Group/Ungroup: create/disassemble a circuit module group.
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current CircuitEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.
- View: choose Frame Selection or Frame All.
- Format: determine the alignment and size of state elements.
- Window: set layouts, choose pane tiling, and show and hide panes.
- Help: display About dialog describing the application.

How to Use CircuitEditor

CircuitEditor opens with an empty canvas grid.

Create a circuit:

- Drag and drop circuit module elements from the Palette onto the grid.
- Select and drag modules to position them on the grid.

Pan and Zoom:

- Pan around the canvas by holding down the Alt key and left mouse button and drag on the canvas.
- Zoom in and out by pressing Alt key while rotating the mouse wheel.

To connect circuit modules, draw a connection from one module to another:

1. Position the cursor at the edge of one module near an Out pin until the cursor becomes an up-arrow.
2. Drag to another module until the cursor again becomes an up-arrow near an In pin and then release the left mouse button. Note that only appropriate destination elements are displayed while you are dragging; the others are "whited out".

Another way to create a connection:

1. Click the left mouse button on the Out pin.
2. Click again on the destination In pin.

This way has the advantage of letting the user pan and/or zoom the canvas around in between the first click and second click.

Select elements several ways:

- Click an item to select it.
- Shift+click adds an element to the selection.
- Add/remove items to/from the selection by Ctrl+click.
- Drag a rectangle around a set of elements to select them.
- Change the selection with the arrow keys. This also scrolls the selection into view.

Change element properties:

- Select an element to view its properties.
- Change properties in either the list view or grid property editor.

To create a layer:

1. Select the circuit module(s) to include in the layer.
2. Copy the selected circuit module(s) to the clipboard.
3. Paste the circuit module(s) in the Layers pane.
4. Click the "New Layer" field and enter the name you want for the layer.

The Layers pane shows the circuit layers in a tree view. Click the + or - to expand and collapse the tree view. Check/uncheck the boxes next to the layer or its components to show/hide the layer or its elements.

To create a master circuit:

1. Select the circuit modules to be part of the new master circuit module type.
2. Select Edit > Master. The circuit collapses into a "MasterInstance".

You can copy the master and paste it to create new, identical instances of the circuit, paste it into the Prototype pane to use as a circuit prototype, or in the Layers pane to use as a layer. Hovering the mouse over the master displays a subwindow showing the master's elements. Double-clicking a master opens a sub-document that shows what that master contains and allows the contents to be edited.

To undo the master circuit module type:

1. Select the master circuit.
2. Select Edit > Unmaster.

To create a group:

1. Select the circuit modules to be grouped.
2. Select either Edit > Group, or click the Group button on the toolbar.

The circuit collapses into a group. You can copy the group and paste it in other circuit documents, in the Prototype pane to use as a circuit prototype, or in the Layers pane to use as a layer.

To ungroup elements:

1. Select the group.
2. Click the Ungroup button on the toolbar.

To create a prototype:

1. Select circuit elements including group(s).
2. Paste into the prototype pane.
3. Click the "Prototypes" label and enter the name of the prototype.
4. Drag prototypes onto the canvas to instantiate new elements from the prototype.

ATF Circuit Editor Sample_j

(View in English 

説明

CircuitEditor

は、入力ピンと出力ピンを持つモジュールとモジュール間の接続で構成される回路用のサンプルエディターです。データファイル形式の定義に XML スキーマを使用し、XML 回路ファイルの読み込みと書き込みを行います。また、モジュールと接続を視覚的に表示して回路を編集できるようにします。AdaptableControl を使用して、回路を表示および編集します。同時に複数のドキュメントを編集できます。CircuitEditor では標準の編集コマンドの実装に、ATF Editor コンポーネントがいくつか使用されています。CircuitEditor には次の機能も含まれています。

- プロトタイピング: ドキュメントに挿入可能な回路フラグメントのカスタムセットを作成できます。
- レイヤー化: オン/オフ (表示/非表示) にできるレイヤーに図コンポーネントを割り当てることができます。
- 回路グループおよび回路テンプレート。

CircuitEditor が示す ATF の機能

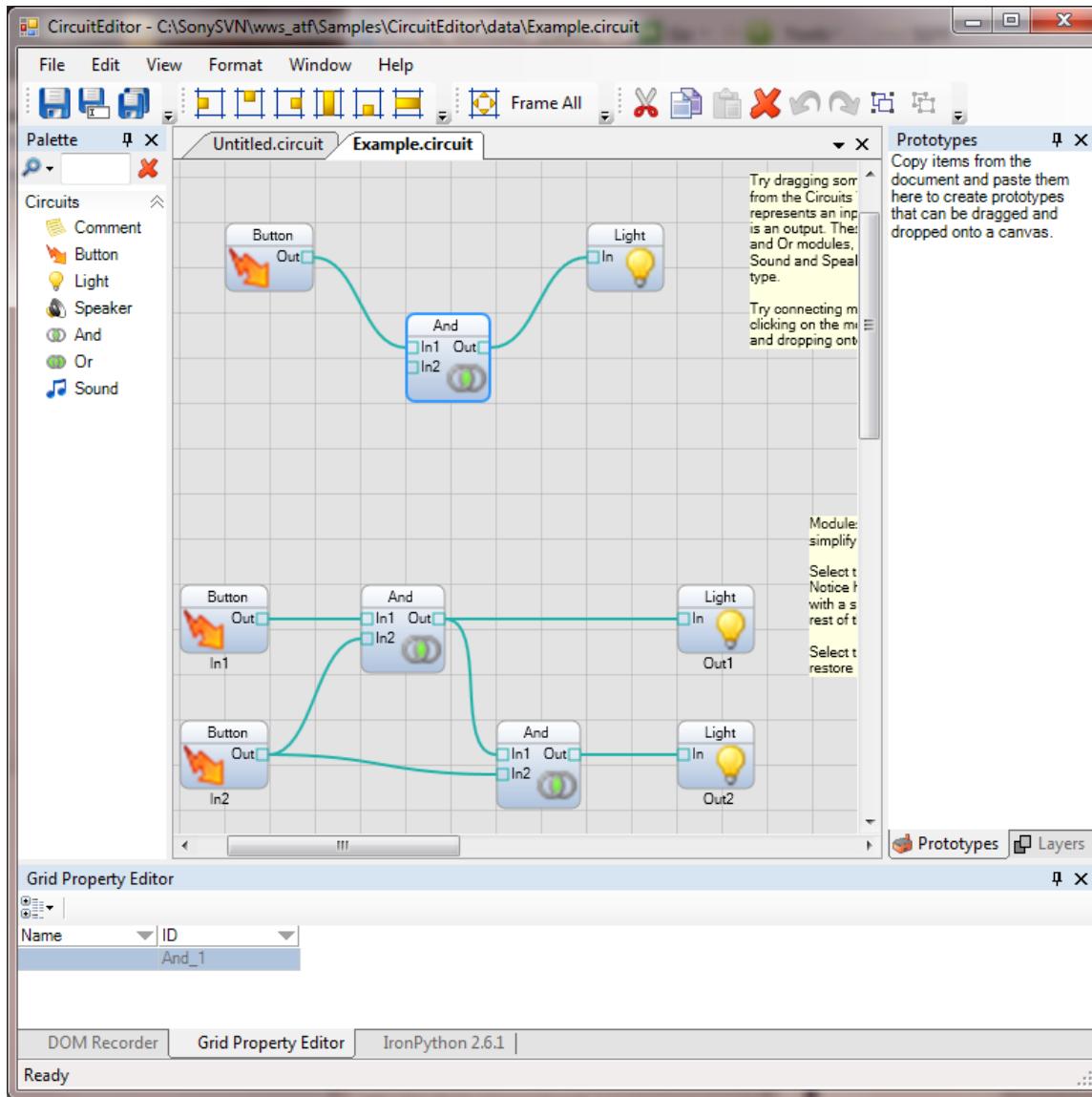
- Circuit.xsd XML スキーマを使用するデータモデルを定義する。
- DOM を使用してメモリ内にデータモデルを格納。
- アダプタを使用して、Circuit データモデルを作成するために DOM を装飾する。
- ContextRegistry を使用して、アクティブな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにする。
- AdaptableControl を使用して、グラフ抽象化を使い回路を表示および編集する。
- TransformAdapter、CanvasAdapter、および ViewingAdapter を使用して、Circuit キャンバスを実装する。
- ScrollbarAdapter、AutoTranslateAdapter、MouseTransformManipulator、および MouseWheelManipulator を使用して、Circuit キャンバスをパンおよびズームできるようにする。
- D2dGraphAdapter、D2dGraphNodeEditAdapter、および D2dGraphEdgeEditAdapter を使用して、回路モジュールと接続を表示し編集可能にする。
- HoverAdapter を使用して、回路モジュールアイテム上にマウスポインタを移動したときに情報を表示する。
- D2dAnnotationAdapter を使用して、キャンバス上に注釈を表示しそのテキストを編集する。
- 別の ATF 回路図エディターのサンプルにコピー/貼り付けをするための、グローバル (Windows) クリップボードの使用法を示す。
- PropertyEditor コンポーネントおよび GridPropertyEditor コンポーネントを使用して、リストおよびグリッドプロパティコントロールのプロパティを編集可能にする。
- 回路グループおよび回路テンプレート。詳細は、[こちら](#)を参照してください。

モジュール

- Editor.cs が IDocumentClient を実装し、ドキュメントフレームワークを使用して複数のドキュメントを管理する。起動時に、File (ファイル) メニュー命令の、複数ドキュメント自動作成および複数ドキュメントの自動オープンを実装する。
- ModulePlugin.cs は、IPaletteClient の実装方法を示し、回路モジュールパーティパレットの作成に IPaletteService を使用する。
- PrototypingContext.cs は、IPrototypingContext の実装方法を示し、プロトタイピングを可能にするために PrototypeLister コンポーネントを使用する。
- LayeringContext.cs は、ILayeringContext の実装方法を示し、レイヤーの編集や表示/非表示を可能するために LayerLister コンポーネントを使用する。
- GroupingCommands.cs は、グループ化/グループ化解除コマンドの実装方法を示す。これは、選択したモジュールを元のモジュールを含みすべての接続を維持する 1 つのモジュールに置き換える、またはその逆のプロセス。
- MasteringCommands.cs は、マスター/マスター解除コマンドの実装方法を示す。これは、選択したモジュールから新しいモジュールタイプを作成し、元のモジュールを作成したイ

CircuitEditor の実行

1. ATF\Samples\CircuitEditor\bin\Release にある CircuitEditor.exe をダブルクリックします。
2. [CircuitEditor] ウィンドウが表示されます。
3. Circuit Editor プロジェクトファイルのサンプルファイル data\Example.circuit を開きます。



CircuitEditor には以下のペインがあります。

- [Palette]: Circuit パーツパレット: [Comment]、[Button]、[Light]、[Speaker]、[And]、[Or]、および [Sound]。
- キャンバス: 回路を定義、表示、および編集する場所です。
- [Property Editor]: 選択したモジュールのプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択したモジュールのプロパティをグリッドコントロールで編集します。
- [Prototypes]: 回路での使用を定義するカスタム Circuit フラグメントを一覧表示します。
- [Layers]: レイヤーとその回路モジュールを一覧表示します。

ツールバーに含まれるボタンは、ファイル管理用が、保存、名前を付けて保存、すべて保存。位置合わせ用が、左揃え、上揃え、右揃え、中央揃え、下揃え、上下メニューバーには次の項目があります。

- [File]: [New Circuit] (回路の新規作成)、[Open Circuit] (既存の回路を開く)、[Save] (保存)、[Save as] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (閉じる)、[Recent Files] (最近使ったファイル)、および [Exit] (終了)。
- [Edit]: 標準の編集機能 ([Undo]/[Redo] (元に戻す/やり直し)、[Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)) のほかに、次の項目があります。
 - 選択: [Select All] (すべて選択)、[Deselect All] (すべての選択を解除)、[Invert Selection] (選択の切り替え)。
 - [Master]/[Unmaster] (マスター/マスター解除): マスター回路を作成/分解します。
 - [Group]/[Ungroup] (グループ化/グループ化解除): 回路モジュールグループを作成/分解します。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load or Save Settings] (設定の読み込み/保存): 現在の CircuitEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定):
 - コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [View] (表示): [Frame Selection] または [Frame All] を選択します。
- [Format] (フォーマット): ステート要素の配置とサイズを指定します。
- [Window] (ウィンドウ): レイアウト、ペインの配置、ペインの表示と非表示を設定します。

- [Help] (ヘルプ): [About] ダイアログにアプリケーションの情報を表示します。

CircuitEditor の使用法

CircuitEditor を開くと空のキャンバスグリッドが表示されます。

回路の作成

- [Palette] からグリッドに回路モジュール要素をドラッグアンドドロップします。
- モジュールを選択してドラッグし、グリッド上に配置します。

パンとズーム

- Alt キーとマウスの左ボタンを押したまま、キャンバス上をドラッグしてパンします。
- Alt キーを押したままマウスホイールを回転させて、表示を縮小または拡大します。

回路モジュールを接続するには、次の手順でモジュール間に接続を描きます。

1. 出力ピンの近くのモジュールの端に、上矢印が表示されるようにカーソルを置き、マウスの左ボタンを押したままにします。
2. 他方のモジュールにドラッグし、入力ピンの近くでカーソルが再び上矢印になつたらドロップします。ドラッグしている時は、接続先に適した要素のみが表示されます。

以下の方法での接続も可能です。

1. 出力ピン上でマウスの左ボタンをクリックします。
2. 接続先の入力ピン上でもう一度クリックします。

この方法では、ユーザは 1 回目と 2 回目のクリックの間にキャンバスをパンまたはズームすることができます。

要素は次のようにいくつかの方法で選択できます。

- アイテムをクリックして選択する。
- Shift を押しながら要素をクリックして選択に加える。
- Ctrl を押しながら要素をクリックして、選択に追加または選択から削除する。
- 一組の要素の周囲に矩形をドラッグして選択する。
- 矢印キーを使って選択を変更する。この方法では、選択した要素が表示されるように画面がスクロールします。

要素プロパティを変更する方法を次に示します。

1. 要素を選択して、そのプロパティを表示します。
2. リストビューまたはグリッドプロパティエディターでプロパティを変更します。

レイヤーを作成する手順を次に示します。

1. レイヤーに含める回路モジュールを選択します。
2. 選択した回路モジュールをクリップボードにコピーします。
3. [Layer] ペインに回路モジュールを貼り付けます。
4. [New Layer] フィールドをクリックし、レイヤー名を入力します。

[Layers]

ペインのツリービューに回路レイヤーが表示されます。「+」または「-」をクリックしてツリービューを展開または折りたたみます。レイヤーまたはそのコンポーネントを複数選択できます。

マスター回路を作成する手順を次に示します。

1. 新しいマスター回路モジュールタイプに含める回路モジュールを選択します。
2. [Edit] > [Master] をクリックします。回路が [MasterInstance] にまとめられます。

マスターをコピーし貼り付けて、同一の新しい回路のインスタンスを作成できます。回路のプロトタイプとして使用するには [Prototype]

ペインに、レイヤーとして使用するには [Layers]

ペインに貼り付けます。マスター上にマウスを置くと、マスター要素を表示するサブウインドウが表示されます。マスターをダブルクリックするとマスターの内容

マスター回路モジュールタイプを元に戻す手順を次に示します。

1. マスター回路を選択します。
2. [Edit] > [Unmaster] をクリックします。

グループを作成する手順を次に示します。

1. グループ化する回路モジュールを選択します。
2. [Edit] > [Group] をクリックするか、またはツールバーの [Group] ボタンをクリックします。

回路がグループにまとめられます。グループをコピーしてほかの回路ドキュメントに貼り付けることができます。回路のプロトタイプとして使用するには [Prototype] ペインに、レイヤーとして使用するには [Layers] ペインに貼り付けます。

要素のグループを解除する手順を次に示します。

1. グループを選択します。
2. ツールバーの [Ungroup] ボタンをクリックします。

プロトタイプを作成する手順を次に示します。

1. グループを含む回路要素を選択します。
2. 選択した要素を [Prototypes] ペインに貼り付けます。
3. [Prototype] ラベルをクリックし、プロトタイプ名を入力します。
4. プロトタイプをキャンバスにドラッグして、プロトタイプから新しい要素をインスタンス化します。

ATF Diagram Editor Sample

(日本語で表示 

Description

DiagramEditor combines the Circuit, FSM, and Statechart editors in a single application to show how multiple editors can share an application shell.

For a description of this editor's internals, see [Diagram Editor Programming Discussion](#).



The newest features of each of the other editors may not be included, so if you are mainly interested in circuits, for example, please take a look at the [Circuit Editor Sample](#).

ATF Features Demonstrated by DiagramEditor

- Multiple editors sharing the same ATF application shell and editor components.

Run DiagramEditor

1. Double-click the DiagramEditor.exe in ATF\Samples\DiagramEditor\bin\Release.
2. The DiagramEditor window appears.

DiagramEditor has the following panes:

- Palette (Circuit Modules, FSM, Statecharts): the parts palette: this is the combined parts palette containing parts for designing circuit modules, finite state machines, and statecharts for your diagrams.
- Property Editor: edit the selected element's property in a list control.
- Grid Property Editor: edit the selected element's property in a grid control.
- Prototypes: lists the custom fragments that you define for use in your diagrams.
- Layers: lists layers and their elements.
- Canvas: define, view, and edit diagrams.

The toolbar contains buttons for file management: save, save as, and save all, print, page set-up, and print preview, and for editing: cut, copy, paste, delete, undo/redo, select all, lock/unlock, group/ungroup. The menu bar contains:

- File: create a new circuit, finite state machine, or statechart, open an existing circuit, finite state machine, or statechart, Save, Save as, Save all, Close, Print, Page Set-up, Print Preview, and Exit DiagramEditor.
- Edit: in addition to the standard editing functions (cut, copy, paste, undo/redo, and so on), Edit provides:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current StatechartEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.
- View: choose Frame Selection or Frame All
- Format: determine the alignment and size of state elements.
- Window: select and deselect panes.

How to Use DiagramEditor

DiagramEditor opens with an empty canvas grid. Drag elements from the Palette onto the canvas to work with them. Click the tabs for the various panes to view and edit element properties.

To create a group within the circuit canvas:

1. Select the circuit elements to be grouped.
2. Select either Edit > Group, or click the Group button on the toolbar.

The circuit elements collapse into a group. You can copy the group and paste it in other circuit documents, in the Prototype pane to use as a diagram prototype, or in the Layers pane to use as a layer. To ungroup:

1. Select the group.
2. Click the Ungroup button on the toolbar.

To create a layer:

1. Select the diagram element(s) to include in the layer.
2. Copy the selected diagram element (s) to the clipboard.
3. Paste the diagram element(s) in the Layers pane.

The Layers pane shows the diagram layers in a tree view. Click the + or - to expand and collapse the tree view.

ATF Diagram Editor Sample_j

(View in English 

説明

DiagramEditor は、回路、FSM、およびステートチャートの各エディターを 1 つのアプリケーションに組み合わせて、複数のエディターがアプリケーションシェルを共有する方法を示します。

 各エディターの最新機能が必ずしも含まれているわけではないため、主に特定のエディターについて知りたい場合は、その工

DiagramEditor が示す ATF の機能

- 複数のエディターによる、エディターコンポーネントと同一の ATF アプリケーションシェルの共有。

DiagramEditor の実行

- ATF\Samples\DiagramEditor\bin\Release にある DiagramEditor.exe をダブルクリックします。
- [DiagramEditor] ウィンドウが表示されます。

DiagramEditor には以下のペインがあります。

- [Palette]: パーツパレット (Circuit, FSM, Statecharts):
回路モジュール、有限ステートマシン、およびステートチャートを設計するためにパーツパレット組み合わせたもので、すべてのパーツを含んでいます。
- [Property Editor]: 選択した要素のプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択した要素のプロパティをグリッドコントロールで編集します。
- [Prototypes]: 図での使用を定義するカスタムフラグメントを一覧表示します。
- [Layers]: レイヤーとその要素を一覧表示します。
- キャンバス: 図を定義、表示、および編集する場所です。

ツールバーに含まれるボタンは、ファイル管理用が、保存、名前を付けて保存、すべて保存、印刷、ページ設定、印刷プレビューで、編集用が、切り取り、コピー

- [File]: [New] > [Statechart], [Circuit], [Finite State Machine] (ステートチャート、回路、または FSM の新規作成)、[Open] > [Statechart], [Circuit], [Finite State Machine] (既存のステートチャート、回路、または FSM を開く)、[Save] (保存)、[Save As] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (閉じる)、[Page Setup] (ページ設定)、[Print Preview] (印刷プレビュー)、[Print] (印刷)、および [Exit] (終了)。
- [Edit]: 標準の編集機能 ([Undo]/[Redo] (元に戻す/やり直し)、[Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)、[Select All]/[Deselect All] (すべて選択/選択解除)、[Invert Selection] (選択対象を反転する) など) のほかに、次の項目があります。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load or Save Settings] (設定の読み込み/保存): [Load and Save Settings] ウィンドウを使用して、現在の StatechartEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [View]: [Frame Selection] または [Frame All] を選択します。
- [Format]: ステート要素の配置とサイズを指定します。
- [Window]: ペインを選択/選択解除します。

DiagramEditor の使用法

DiagramEditor を開くと空のキャンバスグリッドが表示されます。[Palette]

から使用する要素をキャンバスにドラッグします。各ペインのタブをクリックして、要素プロパティを表示し編集します。

回路キャンバス内でグループを作成する手順を次に示します。

- グループ化する回路要素を選択します。
- [Edit] > [Group] をクリックするか、またはツールバーの [Group] ボタンをクリックします。

回路要素がグループにまとめられます。

グループをコピーしてほかの回路ドキュメントに貼り付けることができます。プロトタイプとして使用するには [Prototype] ペインに、レイヤーとして使用するには [Layers] ペインに貼り付けます。グループを解除する手順を次に示します。

1. グループを選択します。
2. ツールバーの [Ungroup] ボタンをクリックします。

レイヤーを作成する手順を次に示します。

1. レイヤーに含める図要素を選択します。
2. 選択した図要素をクリップボードにコピーします。
3. [Layer] ペインに図要素を貼り付けます。

[Layers] ペインのツリービューに図レイヤーが表示されます。「+」または「-」をクリックしてツリービューを展開または折りたたみます。

ATF DOM Tree Editor Sample

(日本語で表示 

Description

DomTreeEditor is a sample editor that operates on simple User Interface definition files. The UI data is hierarchical and DomTreeEditor displays it in a TreeControl. You can select and edit UI elements in the tree, and you can edit the properties on the selected elements.

To learn about the programming of this sample, see [DOM Tree Editor Programming Discussion](#).

ATF Features Demonstrated by DomTreeEditor

- Define a data model using a UISchema.xsd XML schema.
- Use of DomGen to autogenerate DOM metadata for use by adapters.
- Use of adapters to decorate the DOM to create a UI data model.
- Use of IDocumentService / IDocumentClient framework to manage documents.
- Use of IPaletteService / IPaletteClient to create a UI parts palette.
- Use of TreeControl, TreeControlAdapter, TreeControlEditor to display an editable tree view of the UI definition.
- Use of ITreeView / IItemView interfaces to adapt data to a tree.
- Use of ContextRegistry to track the active editing context, so application components always apply where the user is editing.
- Use of the ATF Editor framework interfaces IInstancingContext, ISelectionContext, IHistoryContext and IObservableContext to adapt data so that ATF command components can be used to get undo/redo, cut/paste, and selection commands.
- Use of ATF PropertyEditor / GridPropertyEditor components to allow property editing on selected UI elements.
- Use of the DomExplorer component to show how it can be used to view the raw DOM data and extensions as a diagnostic tool.
- Use of the CurveEditor component to create and edit curves or splines, including working with control points (keys) and tangents, and to create different kinds of curves (linear, stepped, clamped, smooth).

Run DomTreeEditor

1. Double-click the DomTreeEditor.exe in \bin\wws_atf\DomTreeEditor\Release.vs2008.
2. The DomTreeEditor window appears.

DomTreeEditor has the following panes:

- UI Tree Lister: editable tree view of the UI definition
- Palette (UI Objects): the UI parts palette: Package, Form, Shader, Texture, Font, Sprite, Text, Animation
- Property Editor: edit the selected element's property in a list control
- Grid Property Editor: edit the selected element's property in a grid control
- DomExplorer: view the raw DOM data
- Canvas: view and edit the UI file contents.
- Curve Editor: view and edit splines.

The toolbar contains buttons for file management: save, save as, and save all, and editing: cut, copy, paste, delete, undo/redo, and select all.

The menu bar contains:

- File: create a new UI, open an existing UI, Save, Save as, Save all, Close, and Exit DomTreeEditor.
- Edit: in addition to the standard editing functions (cut, copy, paste, and so on), Edit provides:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current DomTreeEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.
- Window: select and deselect panes.

How to Use DomTreeEditor

DomTreeEditor opens with an empty UI definition in the UI Tree Lister and an empty UI canvas. Drag and drop UI objects from the Palette onto the UI definition in the UI Tree Lister to define and work with a data model. The first element must be a Package UI object. Drag and drop other elements onto the Package object to define your data model. Click an element to select it, then click the tabs to display the various panes, view and edit properties, and so on. Click the DomExplorer tab to view the raw DOM data as you define the UI. Select an Animation object to see curves in the Curve Editor.

ATF DOM Tree Editor Sample_j

(View in English 

説明

DomTreeEditor は、単純なユーザインターフェース定義ファイルを操作するサンプルエディタです。UI データは階層構造で、DomTreeEditor では TreeControl に表示されます。ツリー内の UI 要素を選択して編集できます。選択した要素のプロパティも編集できます。

DomTreeEditor が示す ATF の機能

- UISchema.xsd XML スキーマを使用するデータモデルを定義する。
- DomGen を使用して、アダプタが使用する DOM メタデータを自動生成する。
- アダプタを使用して、UI データモデルを作成するために DOM を装飾する。
- IDocumentService/IDocumentClient フレームワークを使用して、ドキュメントを管理する。
- IPaletteService/IPaletteClient を使用して、UI パーツパレットを作成する。
- TreeControl、TreeControlAdapter、TreeControlEditor を使用して、UI 定義の編集可能なツリービューを表示する。
- ITreeView/IItemView インタフェースを使用して、データをツリーに適合させる。
- ContextRegistry を使用して、アクティプな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにする。
- ATF Editor フレームワークインターフェースである IInstancingContext、ISelectionContext、IHistoryContext、および IObservableContext を使用してデータを適合させ、元に戻す/やり直し、切り取り/貼り付け、および選択コマンドの取得に ATF コマンドコンポーネントを使用できるようにする。
- ATF PropertyEditor / GridPropertyEditor コンポーネントを使用して、選択した UI 要素のプロパティを編集可能にする。
- DomExplorer コンポーネントを使用して、診断ツールとして未処理の DOM データと拡張子を表示するための使用法を示す。
- CurveEditor コンポーネントを使用して、カーブやスプラインを作成および編集する (コントロールポイント (キー) および接線の操作を含む)。また、異なる種類のカーブ (linear, stepped, clamped, smooth) を作成する。

DomTreeEditor の実行

1. \bin\www_atf\DomTreeEditor\Release.vs2008 にある DomTreeEditor.exe をダブルクリックします。
2. [DomTreeEditor] ウィンドウが表示されます。

DomTreeEditor には以下のペインがあります。

- [UI Tree Lister]: UI 定義の編集可能なツリービュー。
- [Palette]: [UI Objects] パーツパレット: [Package], [Form], [Shader], [Texture], [Font], [Sprite], [Text], [Animation]。
- [Property Editor]: 選択した要素のプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択した要素のプロパティをグリッドコントロールで編集します。
- [DomExplorer]: 未処理の DOM データを表示します。
- キャンバス: UI ファイルの内容を表示および編集します。
- [Curve Editor]: スプラインを表示および編集します。

ツールバーに含まれるボタンは、ファイル管理用が、保存、名前を付けて保存、すべて保存で、編集用が、切り取り、コピー、貼り付け、削除、元に戻す/やり直しメニューには次の項目があります。

- [File]: [New UI] (UI の新規作成)、[Open UI] (既存の UI を開く)、[Save] (保存)、[Save as] (名前を付けて保存)、[Save all] (すべて保存)、[Close] (閉じる) および [Exit] (終了)。
- [Edit] (編集): 標準の編集機能 (切り取り、コピー、貼り付け、元に戻す/やり直しなど) のほかに、次の項目があります。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts]
 ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load and Save Settings] (設定の読み込み/保存): このウィンドウを使用して、現在の DomTreeEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定):
 コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [Window] (ウィンドウ): ペインを選択/選択解除します。

DomTreeEditor の使用法

DomTreeEditor を開くと、[UI Tree Lister] に空の UI 定義および空の UI キャンバスが表示されます。[Palette] から [UI Tree Lister] の UI 定義に UI オブジェクトをドラッグアンドドロップして、データモデルを定義し操作します。最初の要素は、Package UI オブジェクトである必要があります。ほかの要素を Package オブジェクトにドラッグアンドドロップして、データモデルを定義します。要素をクリックして選択してから、タブをクリックし、各ペインを表示したり、プロパ

の定義時に [DomExplorer] タブをクリックすると、未処理の DOM データが表示されます。Animation オブジェクトを選択すると、[Curve Editor] にカーブが表示されます。

ATF File Explorer Sample

([日本語で表示](#) 

Description

FileExplorer is a sample viewer that displays the user's file hierarchy and files. For information on this sample's programming, see [File Explorer Programming Discussion](#).

ATF Features Demonstrated by FileExplorer

- Use of Managed Extensibility Framework (MEF) to put applications together and to extend a file view.
- Use of the application shell framework, including CommandService, SettingsService, and StatusService. FileExplorer does not use ControlHostService.
- Use of the TreeControl, TreeControlAdapter, and the ITreeView and IItemView interfaces to display the file folder tree structure.
- Use of ListView, ListViewAdapter, and IListView and IItemView interfaces to display the list of files in the selected file folder.
- Use of the UserFeedbackService for bug reporting.
- Use of the VersionUpdateService to test upgrading to the latest SHIP version of the application. This is just a test. There is not really a newer version of FileExplorer on SHIP.

Run FileExplorer

1. Double-click the FileExplorer.exe in ATF\Samples\FileExplorer\bin\Release.
2. An Explorer-like window appears, showing the user's file hierarchy and files, and an Update dialog box pops up, that tests the version update feature.
3. Click "No" on the Update dialog box.

Menu Options

- File: choose Exit to exit FileExplorer.
- Edit:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current FileExplorer application settings, or to load application settings from a file.
 - Preferences: set application preferences such as whether to check for updates at startup (default is TRUE).
- Help:
 - Send Feedback: use the Send Feedback window to report and submit bugs to SHIP.
 - Check for update: use to check for updates to FileExplorer.

How to Use FileExplorer

Expand and collapse file hierarchies by clicking the "+" or "-" sign next to folder names.

ATF File Explorer Sample_j

(View in English 

説明

FileExplorer は、ユーザのファイル階層とファイルを表示するサンプルビューワーです。

FileExplorer が示す ATF の機能

- Managed Extensibility Framework (MEF) を使用した、アプリケーションのまとめとファイル表示の展開。
- CommandService、SettingsService、および StatusService を含むアプリケーションシェルフレームワークの使用。FileExplorer は、ControlHostService を使用しません。
- TreeControl、TreeControlAdapter、および ITreeView と IItemView インタフェースを使用して、ファイルフォルダのツリー構造を表示する。
- ListView、ListViewAdapter、および IListView と IItemView インタフェースを使用して、選択したファイルフォルダにあるファイルを一覧表示する。
- UserFeedbackService を使用したバグ報告。
- VersionUpdateService を使用して、アプリケーションの SHIP の最新バージョンへのアップグレードをテストする。
これはテストのみで、FileExplorer の新しいバージョンは SHIP には存在しません。

FileExplorer の実行

1. ATF\Samples\FileExplorer\bin\Release にある FileExplorer.exe をダブルクリックします。
2. Explorer
に似たウィンドウが表示され、ユーザのファイル階層とファイルが表示されます。また、バージョンアップデート機能をテストする [Update] ダイアログボックスが表示されます。
3. [No] をクリックしてください。

メニュー操作

- [File]: [Exit] をクリックすると、FileExplorer が終了します。
- [Edit]
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts]
ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load or Save Settings] (設定の読み込み/保存): [Load and Save Settings] ウィンドウを使用して、現在の FileExplorer の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): 起動時にアップデートのチェックを行うかどうか (デフォルトは [True]) などのアプリケーションの設定をします。
- [Help]
 - [Send Feedback]: [Send Feedback] ウィンドウを使用して、バグを SHIP に報告します。
 - [Check for update]: FileExplorer のアップデートをチェックするために使用します。

FileExplorer の使用法

フォルダ名の隣にある「+」または「-」記号をクリックして、ファイル階層を展開または折りたたみます。

ATF FSM Editor Sample

([日本語で表示](#) 

Description

FsmEditor is a sample editor for Finite State Machines (FSM's). It uses an XML Schema to define the data file format, reads and writes XML FSM files, and allows them to be edited using a graphical representation of states and transitions. It uses the AdaptableControl to display and edit the FSM. Multiple documents can be edited simultaneously. Many ATF Editor components are used to implement standard editing commands. FsmEditor uses the PrototypeLister component to allow the user to create a custom clipboard of FSM fragments that can be inserted into documents. Annotations, which are post it-like comments on the document canvas, can be added.

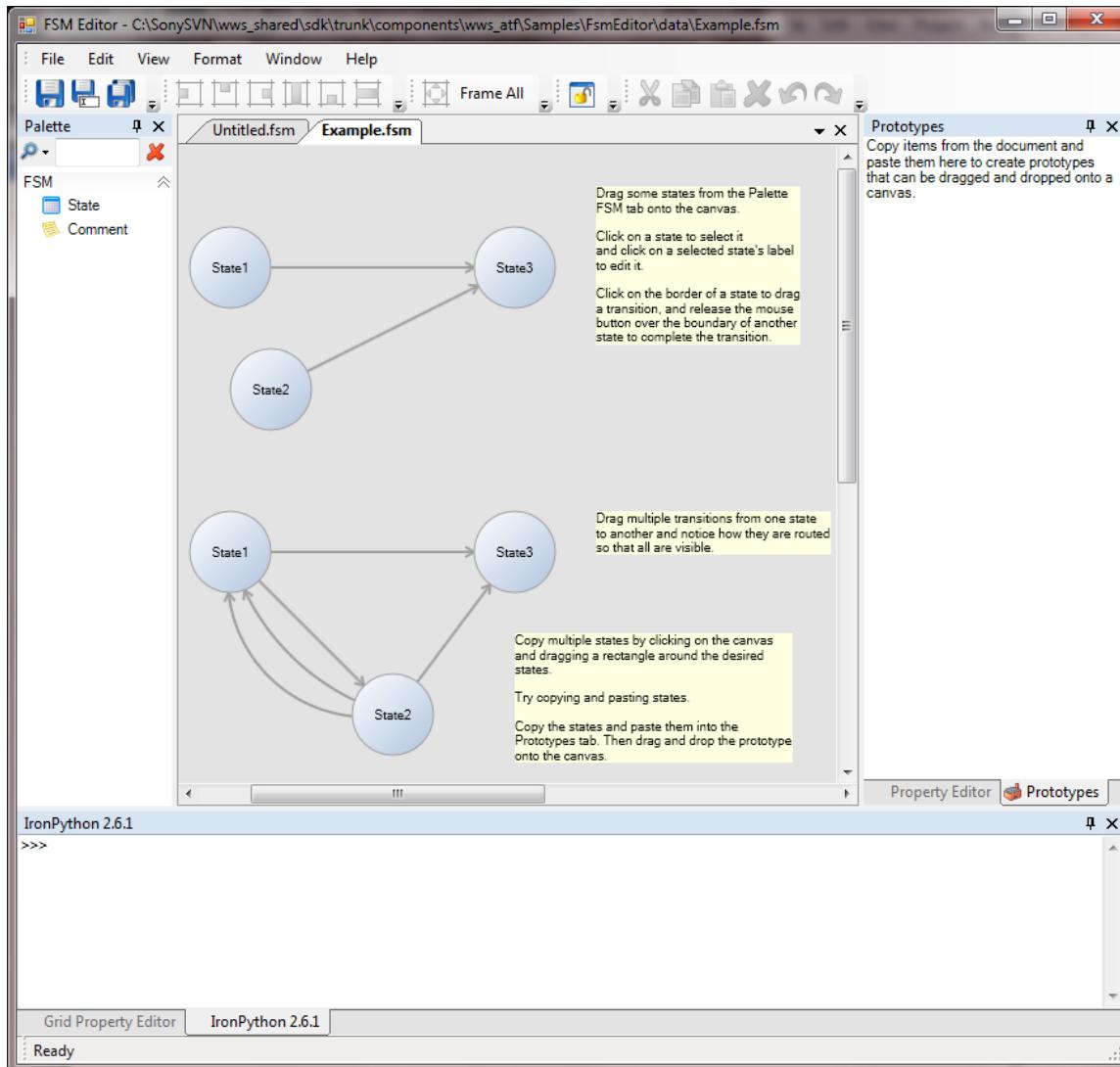
To learn about the internals of this sample, see [FSM Editor Programming Discussion](#).

ATF Features Demonstrated by FsmEditor

- Define a data model using an `FSM.xsd` XML schema.
- Use DOM to store a data model in memory.
- Use `ContextRegistry` to track the active editing context, so that application components always apply where the user is editing.
- Use `AdaptableControl` to display and edit a finite state machine using graph abstractions.

Run FsmEditor Sample

1. Double-click the `FsmEditor.exe` in `ATF\Samples\FsmEditor\bin\Release`.
2. The `FsmEditor` window appears.



FsmEditor has the following panes:

- Palette (FSM): the FSM parts palette: State and Comment
- Property Editor: edit the selected element's property in a list control
- Grid Property Editor: edit the selected element's property in a grid control
- Prototypes: lists custom FSM fragments that you define and use in your FSM's
- Canvas: define, view, and edit finite state machines

The toolbar contains these buttons:

- File management: save, save as, and save all.
- Alignment: left, top, right, center, bottom and middle.
- Frame: selected and all
- Lock/unlock UI layout
- Editing: cut, copy, paste, delete, undo/redo.

The menu bar contains:

- File: create a new or open an existing finite state machine, Save, Save as, Save all, Close, Recent Files, and Exit FsmEditor.
- Edit: in addition to the standard editing functions (undo/redo, cut, copy, paste, delete, and selection actions), Edit provides:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current FsmEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.
- View: choose Frame Selection or Frame All
- Format: determine the alignment and size of state elements.
- Window:
 - Tile Horizontal: tile window panes horizontally.
 - Tile Vertical: tile window panes vertically.
 - Tile Overlapping: overlap window panes horizontally.
 - Lock/Unlock UI Layout: lock or unlock the window layout.

- List of checked menu items; check to display the corresponding control.
- Help: its About menu item displays a dialog describing FsmEditor.

How to Use FsmEditor

FsmEditor opens with an empty canvas grid. Drag and drop State elements from the palette FSM tab onto the grid, select and drag states to position them on the grid. Click either the Property Editor or the Grid Property Editor to view and edit state properties of the selected item.

To connect states, draw a transition line from one state to another: position the cursor at the edge of one state until the cursor becomes an up-arrow, drag to another state until the cursor again becomes an up-arrow then release the mouse button to complete the transition.

You can then click on a state to select it and click on a selected state's label to edit it. You can copy and paste states and their transitions. You can also paste selected states and their transitions in the prototypes pane to create prototypes you can name and reuse.

Drag Comment elements onto State elements to annotate the states.

FsmEditor Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF and internal classes used.
- Editors.cs: Implements IDocumentClient to open, make visible, save and close documents.
- Fsm.cs: Adapts the DOM root node to a finite state machine (FSM), and a routed graph, for display in an AdaptableControl.
- Document.cs: Adapts the FSM to IDocument.
- TransitionRouter.cs: Adapter that tracks changes to transitions and updates their routing during validation.
- ViewingContext.cs: Adapter that provides viewing functions for a finite state machine.
- SchemaLoader.cs: Loads the FSM schema, registers data extensions on the DOM types, annotates the types with display information and PropertyDescriptors.
- EditingContext.cs: Adapts FSM for editing. Implements ISelectionContext, IValidationContext, ITransactionContext, IH歷史Context, IInstancingContext, and IEditableGraph, for editing by AdaptableControl.

Note that FsmEditor makes heavy use of adaptation to take advantage of ATF modules to display and handle states and transitions.

ATF FSM Editor Sample_j

(View in English 

説明

FsmEditor は、有限ステートマシン (FSM) のサンプルエディタです。データファイル形式の定義に XML スキーマを使用し、XML FSM ファイルの読み込みと書き込みを行います。また、ステートと遷移を視覚的に表示して編集できるようにします。AdaptableControl を使用して、FSM を表示および編集します。同時に複数のドキュメントを編集できます。標準の編集コマンドの実装には多くの ATF Editor コンポーネントが使用されています。FsmEditor では、ドキュメントに挿入可能な FSM フラグメントのカスタムクリップボードの作成に PrototypeLister コンポーネントが使用されています。ドキュメントキャンバスには、付箋紙のような役割を果たす注釈を追加できます。

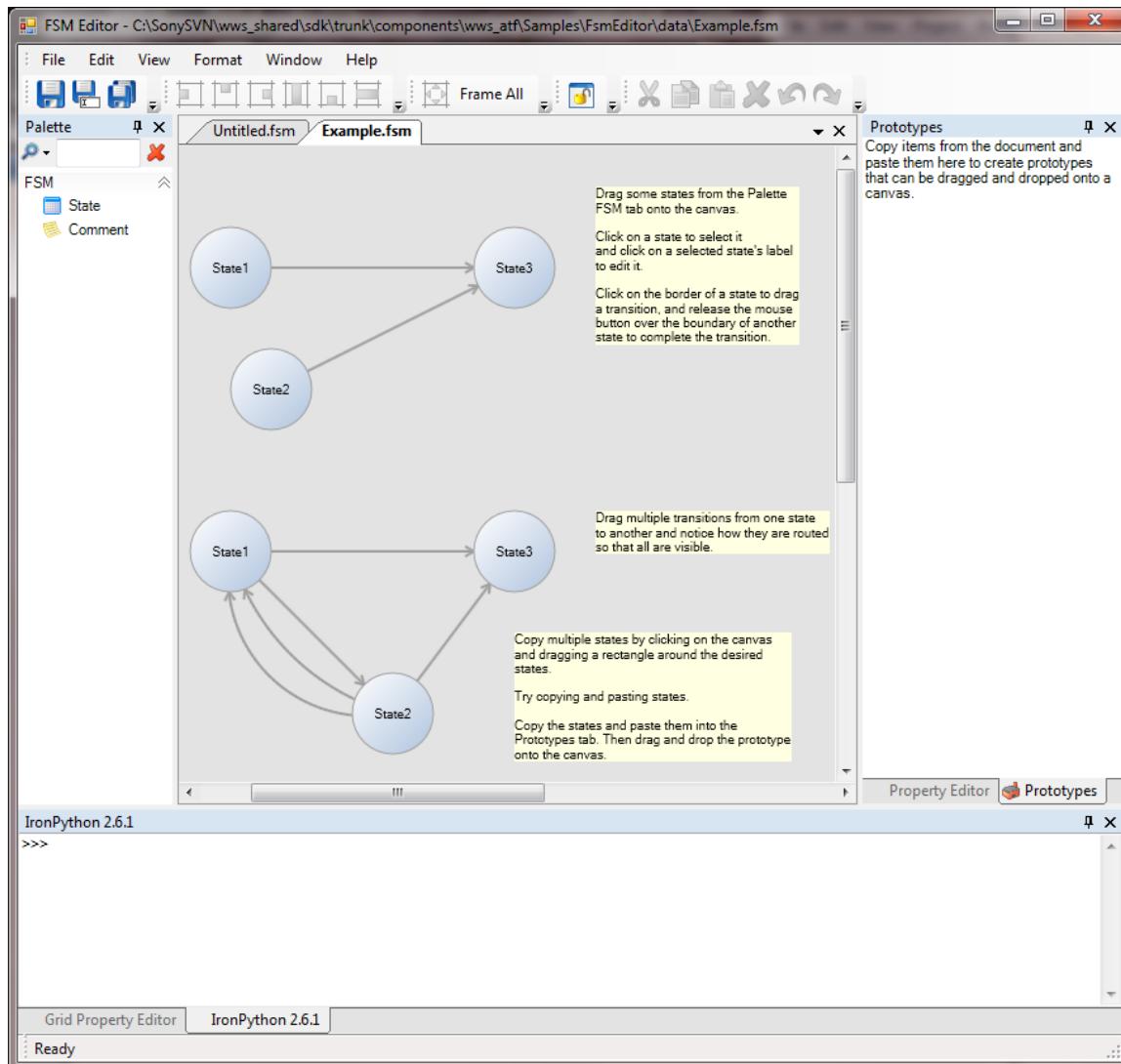
このサンプルの内容は、[Fsm Editor Programming Discussion](#) を参照してください。

FsmEditor が示す ATF の機能

- FSM.xsd XML スキーマを使用するデータモデルを定義します。
- DOM を使用してメモリ内にデータモデルを格納します。
- ContextRegistry を使用して、アクティブな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにします。
- AdaptableControl を使用して、グラフ抽象化を使い FSM を表示および編集します。

FsmEditor サンプルの実行

1. ATF\Samples\FsmEditor\bin\Release にある FsmEditor.exe をダブルクリックします。
2. [Fsm Editor] ウィンドウが表示されます。



FsmEditor には以下のペインがあります。

- [Palette] ([FSM]): FSM のパーツパレットで、[State] と [Comment] があります。
- [Property Editor]: 選択した要素のプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択した要素のプロパティをグリッドコントロールで編集します。
- [Prototypes]: FSM で定義し使用するカスタム FSM フラグメントを一覧表示します。
- キャンバス: FSM を定義、表示、および編集する場所です。

ツールバーには、次のボタンが含まれます。

- ファイル管理: [Save] (保存)、[Save As] (名前を付けて保存)、および [Save All] (すべて保存)。
- 配置: [Lefts] (左揃え)、[Tops] (上揃え)、[Rights] (右揃え)、[Centers] (中央揃え)、[Bottoms] (下揃え)、および [Middles] (上下中央揃え)。
- 最大表示: [Frame Selection] (選択範囲) および [Frame All] (全体)
- ウィンドウレイアウトのロックとロック解除の切り替え。
- 編集: [Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)、[Undo] (元に戻す)、[Redo] (やり直し)。

メニューバーには次の項目があります。

- [File] (ファイル): [New Finite State Machine] (FSM の新規作成)、[Open Finite State Machine] (既存の FSM を開く)、[Save] (保存)、[Save As] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (閉じる)、[Recent Files] (最近使用したファイル) および [Exit] (終了)。
- [Edit] (編集): 標準の編集機能 ([Undo]/[Redo] (元に戻す/やり直し)、[Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)、[Select All]/[Deselect All] (すべて選択/選択解除)、[Invert Selection] (選択対象を反転する)) のほかに、次の項目があります。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load and Save Settings] (設定の読み込み/保存): このウィンドウを使用して、現在の FsmEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [View] (表示): [Frame Selection] (選択範囲) または [Frame All] (全体) を選択します。
- [Format] (フォーマット): ステート要素の配置とサイズを指定します。
- [Window] (ウィンドウ):
 - [Tile Horizontal] (左右に並べて表示): ウィンドウペインを水平に並べて表示します。
 - [Tile Vertical] (上下に並べて表示): ウィンドウペインを上下に並べて表示します。
 - [Tile Overlapping] (重ねて表示): ウィンドウペインを重ねて表示します。
 - [Lock UI Layout] および [Unlock UI Layout]: ウィンドウレイアウトのロックとロックの解除を切り替えます。
 - チェックボックス付きのメニューアイテムのリスト:
コントロールをクリックするとチェックマークが付き、そのコントロールが表示されます。
- [Help] (ヘルプ): [About] ダイアログに FsmEditor の情報を表示します。

FsmEditor の使用法

FsmEditor を開くと空のキャンバスグリッドが表示されます。[Palette] の [FSM] タブから [State]

要素を選択してグリッド上にドラッグアンドドロップし、配置位置までドラッグします。[Property Editor] または [Grid Property Editor] をクリックして、選択したアイテムのステートプロパティを表示、編集します。

ステートを接続するには、ステート間に遷移ラインを描きます。ひとつのステートの端に、上矢印が表示されるようにカーソルを置いて他方のステートに向けてド

ステートのラベルは、ステートを選択してラベルをクリックすれば編集できます。ステートおよびステート間の遷移はコピー、貼り付けできます。また、選択した [Prototypes] ペインに貼り付けるとプロトタイプが作成されます。これは名前を付けて再利用できます。

[Comment] 要素をステートにドラッグして、ステートに注釈を付けます。

FsmEditor のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- Editor.cs: IDocumentClient を実装して、ドキュメントを開き、視覚化し、保存し、閉じることができます。
- Fsm.cs: DOM ルートノードを、AdaptableControl を使用して表示するために、有限ステートマシン (FSM) および有向グラフに適合させます。
- Document.cs: FSM を IDocument に適合させます。
- TransitionRouter.cs: 遷移の変更を追跡し、検証中に経路をアップデートするアダプタです。
- ViewingContext.cs: FSM に表示機能を提供するアダプタです。
- SchemaLoader.cs: FSM スキーマをロードし、データ拡張を DOM 型に登録し、型に表示情報と PropertyDescriptors の注釈を付けます。

- EditingContext.cs: FSM を編集用に適合させます。AdaptableControl による編集のために、IselectionContext、IvalidationContext、ItransactionContext、IhistoryContext、IinstancingContext、および IEditableGraph を実装します。

FsmEditor は、ステートおよび遷移の表示と処理に ATF モジュールを活用するために、適合を多用します。

ATF Simple DOM Editor Sample

(日本語で表示 

Description

SimpleDomEditor is a sample editor that demonstrates the use of the DOM (Document Object Model). SimpleDomEditor operates on event sequence files, *.xml or *.esq files, that contain a sequence of events. The events can contain resources: animations or geometries. Each event sequence file displays in a ListView control, which shows all events and resources that can be selected and edited, as well as the properties on the selected items to be edited. The editor can load multiple event sequence files. The Resources editor tracks the last selected event and displays its resources in another ListView control.

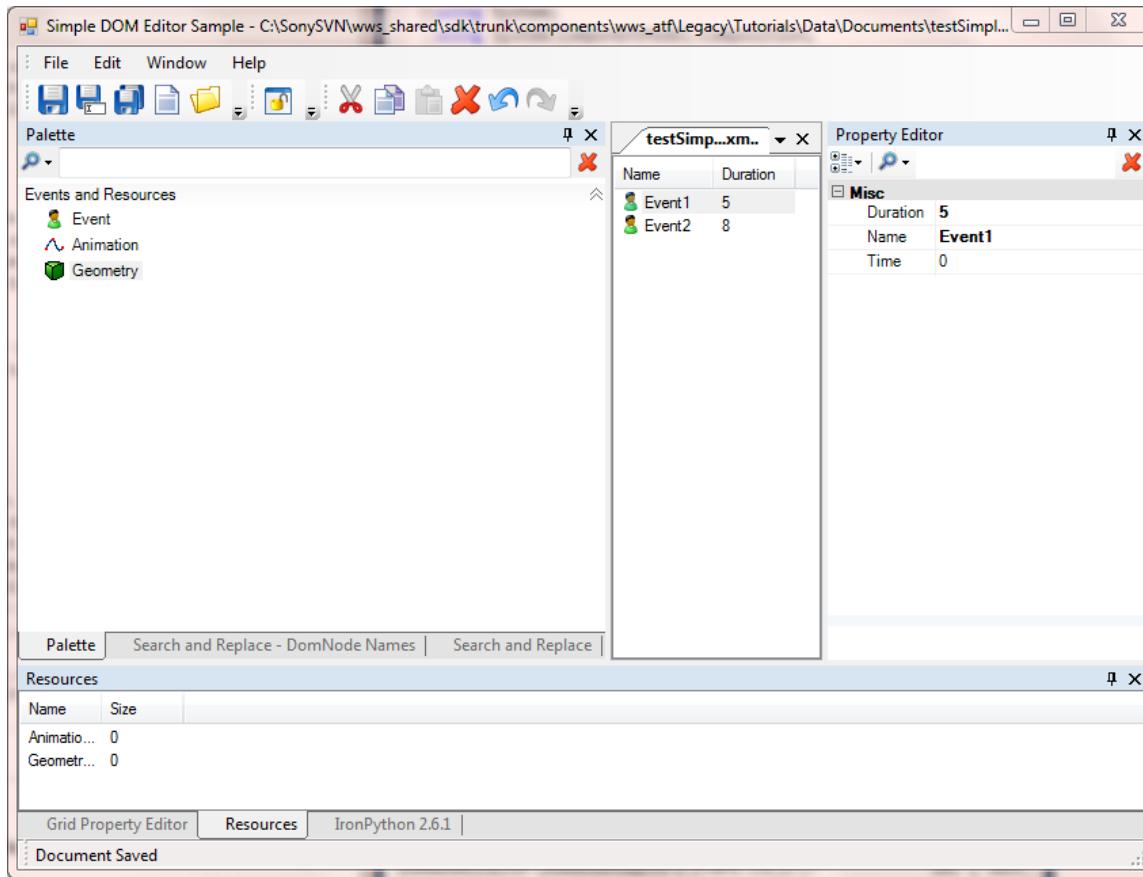
For information about programming in this sample, see [Simple DOM Editor Programming Discussion](#).

ATF Features Demonstrated by SimpleDomEditor

- Define a data model using an EventSequence.xsd XML schema. XML is also used for application data.
- Use of DomGen to autogenerate DOM metadata for use by adapters.
- Use of adapters to decorate the DOM to create event sequence data models.
- Use of IDocumentClient and the document framework to manage multiple documents.
- Show implementing IPaletteClient to create a UI parts palette.
- Using ListView and ListViewAdapter to display editable lists of events and resources.
- Use of IListView, IItemView, and IObservableContext interfaces to adapt data to a list.
- Use of ContextRegistry to track the active editing context, so application components always apply where the user is editing.
- Use of the interfaces IInstancingContext, ISelectionContext, and IHISTORYContext to adapt data for editing commands.
- Use of ATF PropertyEditor and GridPropertyEditor components to allow property editing on selected UI elements.

Run SimpleDomEditor

1. Double-click the SimpleDomEditor.exe in ATF\Samples\SimpleDomEditor\bin\Release.
2. If the Event sequence file window that shows the edited file was closed without opening another file when SimpleDomEditor ran the last time, a Save As window appears. In this case, choose the location and new file name for your event sequence file, then click Save. You can also create or open a file using toolbar buttons or the File menu item.
3. The SimpleDomEditor window appears.



SimpleDomEditor has the following windows:

- Palette (Events and Resources): choose Event, Animation, or Geometry.
- Event sequence file: the current event sequence file being edited.
- Property Editor: edit a selected event or resource property in a list control.
- Grid Property Editor: edit a selected event or resource property in a grid control.
- Resources: list the resources for the active event sequence file.
- Canvas: display the event sequence file contents.
- Search and Replace: two panes to search and replace or search using a regular expression for DomNodes.

The toolbar and menu bar contain standard buttons for file management: save, create new, open existing, and so on, and editing: cut, copy, paste, delete, undo/redo, lock/unlock. The Edit menu item also provides the standard editing functions as well as keyboard shortcuts, load or save settings, and preferences. Use the Window menu item to select and deselect panes. Help provides information about SimpleDomEditor.

How to Use SimpleDomEditor

To start:

1. Create an event sequence file. Use the initial "Untitled.xml", or choose File > New Event Sequence in the SimpleDomEditor toolbar. A Save As window appears. Choose location and new file name for your event sequence file, then click OK. An empty sequence file opens in the editing canvas.
2. From the Palette, drag and drop an Event onto the event sequence file in the editing canvas.
3. Click the tabs for the various panes to view and edit event properties.

To add resources:

1. Drag and drop resources (Animation or Geometry) onto the Resource pane.
2. Click the tabs for the various panes to view and edit resource properties.

SimpleDomEditor Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF and internal classes used.
- Editors.cs: Implements IDocumentClient and uses the document framework to manage multiple documents, implement File menu commands, auto-new and open documents on startup.
- PaletteClient.cs: Implements IPaletteClient and uses IPaletteService to create a UI parts palette.

- EventListEditor.cs: Uses ListView and ListViewAdapter to display editable lists of events and resources.
- ResourceListEditor.cs: Display and edit the resources that belong to the most recently selected event. It handles drag and drop, and right-click context menus for events and resources.
- EventSequenceContext.cs, EventContext.cs: Use IListView, IItemView, and IObservableContext interfaces to adapt data to a list.
- EventSequenceContext.cs, EventContext.cs: Implement the ATF interfaces IInstancingContext, ISelectionContext, and IH歷史Context to adapt data so that ATF command components can be used to get undo/redo, cut/paste, and selection commands.
- DomNodeNameSearchControl.cs: Defines a simple GUI for searching and replacement of DomNode names on the currently active document.
- HelpAboutCommand.cs: Implements a standard Help/About dialog.

ATF Simple DOM Editor Sample_j

(View in English 

説明

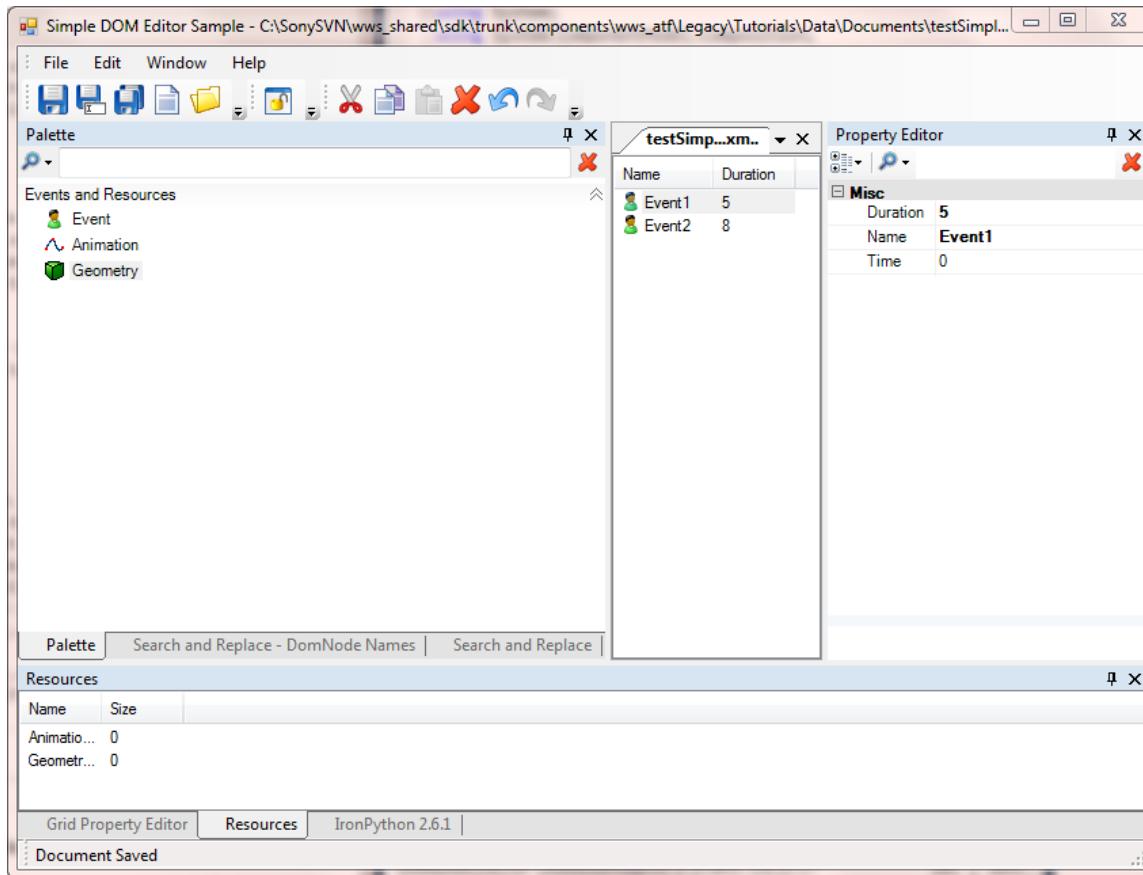
SimpleDomEditor は ドキュメントオブジェクトモデル (DOM) の使用を示すサンプルエディタです。SimpleDomEditor は一連のイベントを含むイベントシーケンスファイル (.xml または .esq ファイル) を操作します。イベントには、リソース (アニメーションおよびジオメトリ) が含まれます。各イベントシーケンスファイルは ListView コントロールに表示されます。すべてのイベントとリソースが表示されますが、これらは選択して編集することができます。また編集するために選択されたアイテムは最後に選択されたイベントを追跡して、そのリソースを別の ListView コントロールに表示します。

SimpleDomEditor が示す ATF の機能

- EventSequence.xsd XML スキーマを使用するデータモデルを定義します。XML はアプリケーションデータにも使用されます。
- DomGen を使用して、アダプタが使用する DOM メタデータを自動生成します。
- アダプタを使用して、イベントシーケンスデータモデルを作成するために DOM を装飾します。
- IDocumentClient およびドキュメントフレームワークを使用して複数のドキュメントを管理します。
- IPaletteClient を実装して、UI パーツパレットを作成します。
- ListView と ListViewAdapter を使用して、イベントおよびリソースの編集可能なリストを表示します。
- IListView、IItemView、および IObservableContext インタフェースを使用して、データをリストに適合させます。
- ContextRegistry を使用して、アクティブな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにします。
- IInstancingContext、ISelectionContext および IHistoricContext のインターフェースを使用して、データを編集コマンドに適合させます。
- ATF PropertyEditor コンポーネントおよび GridPropertyEditor コンポーネントを使用して、選択した UI 要素のプロパティを編集可能にします。

SimpleDomEditor の実行

1. ATF\Samples\SimpleDomEditor\bin\Release にある SimpleDomEditor.exe をダブルクリックします。
2. 前回の SimpleDomEditor の実行を、イベントシーケンスファイルが何も開かれていない状態で終了した場合は、[名前を付けて保存] ウィンドウが表示されます。この場合は、イベントシーケンスファイルの保存場所と新しい名前を指定して、[保存] をクリックします。ツールバーの [File] メニューアイテムを使ってファイルを作成したり開いたりすることもできます。
3. [SimpleDomEditor] ウィンドウが表示されます。



SimpleDomEditor には以下のウィンドウがあります。

- [Palette] の [Events and Resources]: [Event] (イベント)、[Animation] (アニメーション)、または [Geometry] (ジオメトリ) を選択します。
- イベントシーケンスファイル: 編集中のイベントシーケンスファイルです。
- [Property Editor]: リストコントロール内で、選択したイベントまたはリソースのプロパティを編集します。
- [Grid Property Editor]: グリッドコントロール内で、選択したイベントまたはリソースのプロパティを編集します。
- [Resources]: 選択されたイベントシーケンスファイルのリソースを一覧表示します。
- キャンバス: イベントシーケンスファイルの内容を表示します。
- [Search and Replace]: 2 つのペインで、正規表現を使用して DomNode を検索、または検索して置換できます。

ツールバーおよびメニューバーには標準的な機能が含まれており、ファイル管理用として、保存、新規作成、既存のファイルを開くなど、編集用として、切り取りメニューアイテムを使用して、ペインを選択/選択解除します。[Help] では、SimpleDomEditor に関する情報が提供されます。

SimpleDomEditor の使用法

起動する手順を次に示します。

1. イベントシーケンスファイルを作成します。起動時に表示される Untitled.xml ファイルを使用するか、またはツールバーで [File] > [New Event Sequence] をクリックします。[名前を付けて保存] ダイアログボックスが表示されます。イベントシーケンスファイルの保存場所と新しい名前を指定して、[保存] をクリックします。編集キャンバスに空のシーケンスファイルが開かれます。
2. [Palette] から、[Event] を編集キャンバスのイベントシーケンスファイルにドラッグアンドドロップします。
3. 各ペインのタブをクリックして、イベントプロパティを表示し編集します。

リソースを追加する手順を次に示します。

1. リソース (アニメーションまたはジオメトリ) を [Resource] ペインにドラッグアンドドロップします。
2. 各ペインのタブをクリックして、リソースプロパティを表示し編集します。

SimpleDomEditor のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- Editor.cs: IDocumentClient を実装し、ドキュメントフレームワークを使用して、複数ドキュメントを管理し、[File] メニュー命令を実装し、起動時にドキュメントを自動生成して開きます。

- PaletteClient.cs: IPaletteClient を実装し、IPaletteService を使用して UI パーツパレットを作成します。
- EventListEditor.cs: ListView と ListViewAdapter を使用して、イベントおよびリソースの編集可能なリストを表示します。
- ResourceListEditor.cs:
直近に選択されたイベントのリソースを表示し編集します。イベントとリソースのドラッグドロップおよび右クリックコンテキストメニューを処理します。
- EventSequenceContext.cs および EventContext.cs: IListView、IItemView、および IObservableContext
インターフェースを使用して、データをリストに適合させます。
- EventSequenceContext.cs および EventContext.cs: ATF インタフェースである IInstancingContext、ISelectionContext、および
IHistoryContext を実装し、ATF
コマンドコンポーネントを使用して、元に戻す/やり直し、切り取り/貼り付け、および選択のコマンドを取得できるようにデータを適合させます。
- DomNodeNameSearchControl.cs: 現在アクティブなドキュメント上で DOMNode 名を検索および置換するための簡単な GUI
を定義します。
- HelpAboutCommand.cs: 標準のヘルプ//バージョン情報ダイアログを実装します。

ATF State Chart Editor Sample

(日本語で表示 

Description

StatechartEditor is a sample editor for statecharts. It uses an XML Schema to define the data file format, reads and writes XML statechart files, and allows them to be edited using a graphical representation of states and transitions. It uses the AdaptableControl to display and edit the statechart. Annotations, which are post it-like comments on the document canvas, can be added. Multiple documents can be edited simultaneously. Many ATF Editor components are used to implement standard editing commands. StatechartEditor also demonstrates prototyping: how the user can create a custom set of statechart fragments that can be inserted into documents.

For details on the internals of this sample, see [State Chart Editor Programming Discussion](#).

ATF Features Demonstrated by StatechartEditor

- How to define a data model in Statechart.xsd.
- Use of DOM to store a data model in memory.
- Use of adapters to decorate the DOM to create a statechart data model.
- Editor.cs shows how to implement IDocumentClient and use a document framework to manage multiple documents, implement File menu commands, auto-new and open documents on startup.
- PaletteClient.cs shows how to implement IPaletteClient and uses IPaletteService to create the Statechart parts palette.
- Use of ContextRegistry to track the active editing context so that application components always apply where the user is editing.
- Use of AdaptableControl to display and edit a statechart using graph abstractions.
- Use TransformAdapter, CanvasAdapter, and ViewingAdapter to implement the statechart canvas.
- Use ScrollbarAdapter, AutoTranslateAdapter, MouseTransformManipulator, and MouseWheelManipulator to allow the user to pan and zoom the statechart canvas.
- Use of GraphAdapter, GraphNodeEditAdapter, and GraphEdgeEditAdapter to display and allow editing of statechart states and transitions.
- Use of IHierarchicalNode interface to allow states within states.
- Use of HoverAdapter to display information when the mouse hovers over statechart items.
- Use of AnnotationAdapter to display annotations and edit their text on the statechart canvas.
- PrototypingContext.cs shows how to implement IPrototypingContext, and uses PrototypeLister to allow prototyping.

Run StatechartEditor

1. Double-click the StatechartEditor.exe in ATF\Samples\StatechartEditor\bin\Release.
2. The StatechartEditor window appears.

StatechartEditor has the following panes:

- Palette (Statecharts): the Statechart parts palette: Comment, State, Start State, Final State, History State, Conditional State
- Property Editor: edit the selected statechart element's property in a list control
- Grid Property Editor: edit the selected statechart element's property in a grid control
- Prototypes: lists the custom statechart fragments that you define for use in your statecharts
- Canvas: define, view, and edit statechartsThe toolbar contains buttons for file management: save, save as, and save all, print, page set-up, and print preview, and for editing: cut, copy, paste, delete, undo/redo, select all, lock/unlock, group/ungroup.

The menu bar contains:

- File: create a new statechart, open an existing statechart, Save, Save as, Save all, Close, Print, Page Set-up, Print Preview, and Exit.
- Edit: in addition to the standard editing functions (cut, copy, paste, undo/redo, and so on), Edit provides:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current StatechartEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.
- View: choose Frame Selection or Frame All
- Format: determine the alignment and size of state elements.
- Window: toggle the panes on and off; organize the document panes.

How to Use StatechartEditor

StatechartEditor opens with an empty canvas grid. Drag statechart elements from the Palette onto the canvas to work with them. Click the tabs for the various panes to view and edit statechart and statechart element properties.

ATF State Chart Editor Sample_j

(View in English 

説明

StatechartEditor はステートチャート用のサンプルエディタです。データファイル形式の定義に XML スキーマを使用し、XML ステートチャートファイルの読み込みと書き込みを行います。また、ステートと遷移を視覚的に表示して編集できるようにします。AdaptableControl を使用して、ステートチャートを表示および編集します。ドキュメントキャンバス上の付箋紙のような役割を果たす注釈を追加できます。同時に複数のドキュメントが使用されています。ATF Editor コンポーネントが使用されています。StatechartEditor には、プロトタイピングの実例も含まれており、ドキュメントに挿入可能なステートチャートフラグメントのカスタムセットを作成する方法が示されています。

StatechartEditor が示す ATF の機能

- Statechart.xsd 内でデータモデルを定義する方法。
- DOM を使用してメモリ内にデータモデルを格納。
- アダプタを使用して、ステートチャートデータモデルを作成するために DOM を装飾する。
- Editor.cs は IDocumentClient を実装してドキュメントフレームワークを使用する方法を示す。これにより、複数のドキュメントの管理、[File] メニュー命令の実装、起動時にドキュメントを自動新規作成したり開いたりすることが可能。
- PaletteClient.cs は、IPaletteClient の実装方法を示し、Statechart パーツパレットの作成に IPaletteService を使用する。
- ContextRegistry を使用して、アクティブな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにする。
- AdaptableControl を使用して、グラフ抽象化を使いステートチャートを表示および編集する。
- TransformAdapter、CanvasAdapter、および ViewingAdapter を使用して、ステートチャートキャンバスを実装する。
- ScrollbarAdapter、AutoTranslateAdapter、MouseTransformManipulator、および MouseWheelManipulator を使用して、ステートチャートキャンバスをパンおよびズームできるようにする。
- GraphAdapter、GraphNodeEditAdapter、および GraphEdgeEditAdapter を使用して、ステートチャートのステートと遷移を表示し編集可能にする。
- IHierarchicalNode インタフェースを使用して、ステート内のステートを許可する。
- HoverAdapter を使用して、ステートチャートアイテム上にマウスポインタを移動したときに情報を表示する。
- Use of AnnotationAdapter to display annotations and edit their text on the statechart canvas.
- PrototypingContext.cs は、IPrototypingContext の実装方法を示し、プロトタイピングを可能にするために PrototypeLister を使用する。

StatechartEditor の実行

1. ATF\Samples\StatechartEditor\bin\Release にある StatechartEditor.exe をダブルクリックします。
2. [StatechartEditor] ウィンドウが表示されます。

StatechartEditor には以下のペインがあります。

- [Palette]: [Statecharts] パーツパレット: [Comment]、[State]、[Start State]、[Final State]、[History State]、[Conditional State]
- [Property Editor]: 選択したステートチャート要素のプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択したステートチャート要素のプロパティをグリッドコントロールで編集します。
- [Prototypes]: ステートチャートで使用するように定義したカスタムステートチャートフラグメントを一覧表示します。
- キャンバス: ステートチャートを定義、表示、および編集します。ツールバーに含まれるボタンは、ファイル管理用が、保存、名前を付けて保存、すべて保存、印刷、..

メニューには次の項目があります。

- [File]: [New Statechart] (ステートチャートの新規作成)、[Open Statechart] (既存のステートチャートを開く)、[Save] (保存)、[Save as] (名前を付けて保存)、[Save all] (すべて保存)、[Close] (閉じる)、[Page Setup] (ページ設定)、[Print Preview] (印刷プレビュー)、[Print] (印刷) および [Exit] (Statechart の終了)。
- [Edit]: 標準の編集機能 ([Undo]/[Redo] (元に戻す/やり直し)、[Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)、[Select All]/[Deselect All] (すべて選択/選択解除)、[Invert Selection] (選択対象を反転する) など) のほかに、次の項目があります。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load or Save Settings] (設定の読み込み/保存): [Load and Save Settings] ウィンドウを使用して、現在の StatechartEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [View]: [Frame Selection] または [Frame All] を選択します。

- [Format]: ステート要素の配置とサイズを指定します。
- [Window]: ベインの表示/非表示を切り替え、ドキュメントペインを整理します。

StatechartEditor の使用法

StatechartEditor を開くと空のキャンバスグリッドが表示されます。[Palette]

から使用するステートチャート要素をキャンバスにドラッグします。各ペインのタブをクリックして、ステートチャートとステートチャート要素プロパティを表示

ATF Timeline Editor Sample

([日本語で表示](#) 

Description

TimelineEditor is a relatively full-featured timeline editor whose components have been used in real production tools.

To learn about the programming of this sample, see [Timeline Editor Programming Discussion](#).

ATF Features Demonstrated by TimelineEditor

- Use of MEF (Managed Extensibility Framework) to put an application together using optional components.
- Use of the application shell framework, including CommandService, SettingsService, StatusService, and ControlHostService.
- Use of the TimelineControl, TimelineRenderer, and all the timeline manipulators - MoveManipulator, ScaleManipulator, ScrubberManipulator, SelectionManipulator, SnapManipulator, and SplitManipulator.
- A manipulator architecture that allows for new functionality to be added to or removed from a TimelineControl without having to modify the TimelineControl.
- Use of IPaletteService for showing a list of timeline object types that can be dragged from the palette into the document in order to instantiate new timeline objects.
- Use of the two property editors, PropertyEditor and GridPropertyEditor.
- An implementation of sub-document support where referenced documents appear as child documents within the tab of a main document.
- Multiple documents being opened simultaneously.
- Copy and paste within a document and between documents.

Run TimelineEditor

1. Double-click the TimelineEditor.exe in ATF\Samples\TimelineEditor\bin\Release.
2. The TimelineEditor window appears.

TimelineEditor has the following panes:

- Palette: the timeline parts palette with these parts:
 - Marker: marks a global event, for example, 6 minutes into an animation, your world blows up. Markers apply globally and are owned by the timeline document: they do not belong to animation tracks or groups. Markers are drawn across all tracks.
 - Group: provides a way to organize animation tracks. Groups contain one or more tracks. You can hide a group of tracks or delete it, expand or collapse groups, and move tracks from one group to another. If a group is collapsed, you can't select anything in it. You can't nest groups.
 - Track: an animation track. Tracks are cinematic tools that ensure that animations play in the correct order. Each track contains intervals and keys. You can drag and drop tracks into a group and rename the track.
 - Interval: represents a duration of time that has a definite start and definite end.
 - Key: represents a point in time that has a definite start and a duration of 0. Keys belong to a track.
 - Reference: contains a reference to another timeline document along with a start time that offsets the referenced timeline.
- Property Editor: edit the selected element's property in a list control.
- Grid Property Editor: edit the selected element's property in a grid control.
- Output: show diagnostics; optional.
- Canvas: define, view, and edit timelines.

The toolbar contains buttons for file management: save, save as, save all, create new timeline document, open existing timeline document, print, page set-up, and print preview; for canvas viewing: panning and zooming to center a selection on the canvas; for editing: cut, copy, paste, delete, undo/redo.

The menu bar contains:

- File: create a new timeline, open an existing timeline, save, save as, and save all, close the current timeline, page setup, print preview, recently used files, and exit the Timeline Editor.
- Edit: in addition to the standard editing functions (cut, copy, paste, undo/redo, select/deselect, invert selection), Edit provides:
 - Interval Splitting Mode: split the any interval with the next mouse click.
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current TimelineEditor application settings, or to load application settings from a file.
 - Preferences: set application and document preferences, such as command icon size and auto-load the last active documents.

- View: Fit in Active View: ensure that the current selection is visible on the canvas.
- Window: select and deselect panes, organize multiple document panes

How to Use TimelineEditor

TimelineEditor opens with an empty canvas. Choose File > New Timeline or click the button to create a new timeline document, or choose File > Open Timeline or click the button to open an existing timeline document. Drag elements from the Palette onto the canvas to work with them. Click the tabs for the various panes to view and edit element properties.

Working with Timeline Reference Documents

You can include other timeline documents in the current timeline by dragging a Reference object from the Palette onto the canvas. Timeline reference documents are also called timeline sub-documents. The timeline sub-document can contain additional timeline sub-documents. When you reference a timeline document, it is loaded into the current timeline document.

You cannot edit the timeline sub-document file within the current timeline document. You can expand and collapse the display of the timeline sub-document in the current timeline document, just as you expand and collapse groups. You can have the timeline sub-document and current timeline open at the same time, show them side by side on the canvas, and make changes. You can see the changes in both documents as you work with them: they're kept in sync. The timeline sub-document has a definite start point, which is like a key or interval. There is a diamond icon that indicates where the "0" time is on the timeline sub-document. When you drag a timeline reference from the Palette onto the canvas, you'll see a salmon-colored line. To set the timeline reference file:

1. Click the reference to select it. The timeline reference properties appear in the Property Editor and Grid Property Editor panes.
2. Click the FileName property, and then browse to the timeline file.

Working with the Scrubber

The scrubber is a vertical black line that is visible when you work with your timeline. The scrubber is not part of the document. There are events that are raised when you move the scrubber, and your application can take some action in response to the events. For example, if the timeline editor is attached to a game, moving the scrubber changes the time in the game. You can have the application take whatever action you want; you're basically saying, "I'm setting the current time to be something new". You could use the scrubber in conjunction with a play and pause button, for example.

Working with the Tools

Pan and Zoom

Use the Alt-left mouse and alt-right mouse buttons to pan and zoom the view of the timeline on the canvas.

Interval Splitting Mode

To use Interval Splitting Mode:

1. Select an interval.
2. Enter "S" (just plain "S"). The interval splits in half. You can split multiple intervals.

You can also choose Edit > Interval Splitting Mode

Scaling

There are two types of scaling:

- Drag the left or right border of a selected interval. If you have several intervals that represent the same event, you can drag the left or right edge of any selected intervals. The selected intervals scale by the same amount. Note that the scaling is not proportional: the selected intervals scale by the same distance along the timeline.
- Use the black scaling bar that appears on the horizontal timeline scale when you select any number of timeline objects (intervals, keys, or markers). The scaling bar scales the selected portion of the timeline. All timeline objects within the selected interval scale relative to each other; intervals that were adjacent before scaling remain adjacent with no gaps or overlaps.

ATF Timeline Editor Sample_j

(View in English 

説明

TimelineEditor は、比較的機能豊富なタイムラインエディタで、そのコンポーネントは製品用ツールに実際に使用されています。

TimelineEditor が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用して、オプションのコンポーネントを使用するアプリケーションをまとめる。
- CommandService、SettingsService、StatusService、および ControlHostService を含むアプリケーションシェルフレームワークの使用。
- TimelineControl、TimelineRenderer、およびすべてのタイムラインマニピュレータ (MoveManipulator、ScaleManipulator、ScrubberManipulator、SelectionManipulator、SnapManipulator、および SplitManipulator) の使用。
- TimelineControl を変更せずに新機能を TimelineControl に追加または削除することを可能にするマニピュレータのアーキテクチャ。
- IPaletteService を使用して、タイムラインオブジェクトタイプのリストを表示する。オブジェクトは、新しいタイムラインオブジェクトをインスタンス化するために、パラメータを渡します。
- 2 つのプロパティエディタ、PropertyEditor と GridPropertyEditor を使用。
- リファレンスドキュメントがメインドキュメントのタブ内に子ドキュメントとして表示されるサブドキュメントのサポートの実装。
- 同時に開いている複数のドキュメント。
- ドキュメント内およびドキュメント間でのコピーと貼り付け。

TimelineEditor の実行

1. ATF\Samples\TimelineEditor\bin\Release にある TimelineEditor.exe をダブルクリックします。
2. [TimelineEditor] ウィンドウが表示されます。

TimelineEditor には以下のペインがあります。

- [Palette]: [Timelines] パーツパレットには次の要素があります。
 - [Marker]: アニメーションが 6 分経過後に世界が破滅するなどのグローバルイベントをマークします。マーカーはグローバルに適用され、タイムラインドキュメントに属します。
 - [Group]: アニメーションのトラックを整理できるようにします。グループには 1 つ以上のトラックが含まれます。トラックのグループの非表示や削除、グループの展開や折りたたみ、グループ間でのトラックの移動が可能です。
 - [Track]: アニメーショントラックです。トラックは、アニメーションが正しい順番で再生されるようにする映像ツールです。各トラックには、間隔とキーが含まれます。
 - [Interval]: 明確な開始と終了が決まった時間的な間隔を示します。
 - [Key]: 明確な開始と継続時間が 0 である時間上の一点を示します。キーはトラックに属します。
 - [Reference]: 別のタイムラインドキュメントへの参照と、そのリファレンスマップをオフセットする開始時間を含みます。
- [Property Editor]: 選択した要素のプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択した要素のプロパティをグリッドコントロールで編集します。
- [Output]: 診断を表示します (オプション)。
- キャンバス: タイムラインを定義、表示、および編集する場所です。

ツールバーに含まれるボタンは、ファイル管理用が、保存、名前を付けて保存、すべて保存、タイムラインドキュメントの新規作成、既存のタイムラインドキュメントメニューには次の項目があります。

- [File]: [New Timeline] (タイムラインの新規作成)、[Open Timeline] (既存のタイムラインを開く)、[Save] (保存)、[Save As] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (現在のタイムラインを閉じる)、[Page Setup] (ページ設定)、[Print Preview] (印刷プレビュー)、[Print] (印刷)、[Recent Files] (最近使用したファイル) および [Exit] (Timeline Editor の終了)。
- [Edit]: 標準の編集機能 ([Undo]/[Redo] (元に戻す/やり直し)、[Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、[Delete] (削除)、[Select All]/[Deselect All] (すべて選択/選択解除)、[Invert Selection] (選択対象を反転する)) のほかに、次の項目があります。
 - [Interval Splitting Mode]: 選択済みの間隔が、次にクリックした箇所で分割されます。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load or Save Settings] (設定の読み込み/保存): [Load and Save Settings] ウィンドウを使用して、現在の TimelineEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズや最後にアクティブになったドキュメントの自動読み込みなど、アプリケーションやドキュメントを設定します。
- [View] > [Fit in Active View]: 現在の選択がキャンバス上に表示されるようにします。

- [Window]: ベインを選択/選択解除したり、複数のドキュメントベインを整理します。

TimelineEditor の使用法

TimelineEditor を開くと空のキャンバスが表示されます。[File] > [New Timeline]

を選択するか、ボタンをクリックして新しいタイムラインドキュメントを作成します。または、[File] > [Open Timeline]

を選択するか、ボタンをクリックして既存のタイムラインドキュメントを開きます。[Palette]

から使用する要素をキャンバスにドラッグします。各ペインのタブをクリックして、要素プロパティを表示し編集します。

タイムラインリファレンスドキュメントの使用

Reference オブジェクトを [Palette]

からキャンバスにドラッグして、現在のタイムラインにほかのタイムラインドキュメントを含めることができます。タイムラインリファレンスドキュメントは、タイムラインサブドキュメントには、追加のタイムラインサブドキュメントを含めることができます。

タイムラインドキュメントを参照するときに、現在のタイムラインドキュメントに読み込まれます。

現在のタイムラインドキュメント内にあるタイムラインサブドキュメントファイルは編集できません。

グループの展開と折りたたみ同様に、現在のタイムラインドキュメントにあるタイムラインサブドキュメントの表示を展開または折りたたむことができます。

タイムラインサブドキュメントと現在のタイムラインを同時に開き、キャンバスに並べて表示して、変更できます。使用しながら両ドキュメントの変更を確認でき

タイムラインサブドキュメントには、キーや間隔のように明確な開始点があります。

タイムラインサブドキュメント上の「0」時間の位置がひし形のアイコンにより示されます。タイムラインリファレンスを [Palette]

からキャンバスにドラッグすると、サーモン色のラインが表示されます。タイムラインリファレンスファイルを設定するには、次の手順を実行します。

1. リファレンスをクリックして選択します。タイムラインリファレンスのプロパティが [Property Editor] と [Grid Property Editor] ペインに表示されます。
2. [File Name] プロパティをクリックして、タイムラインファイルにブラウズします。

スクラバーの使用

スクラバーは垂直の黒い線で、タイムラインを使用しているときに表示されます。スクラバーはドキュメントの一部ではありません。スクラバーを移動すると発生したとえば、再生ボタンおよび一時停止ボタンと併用してスクラバーを使用できます。

ツールの使用

パンとズーム

Alt キーと左マウスボタンおよび Alt キーと右マウスボタンを使って、キャンバス上のタイムライン表示をパンまたはズームできます。

Interval Splitting (間隔分割) モード

間隔分割モードの使用法を次に示します。

1. 間隔を選択します。
2. 「S」(「S」のみ)を入力します。間隔が分割されます。複数の間隔を分割できます。

[Edit] > [Interval Splitting Mode] でもこの動作を実行できます。

拡大縮小

拡大縮小には、次の 2 つのタイプがあります。

- 選択された間隔の左または右の枠線をドラッグします。同じイベントを示す間隔が複数ある場合、選択した間隔の左端または右端をドラッグできます。選択する場合は、水平のタイムラインスケール上に表示される黒い拡大縮小バーを使用します。拡大縮小バーは、タイムラインの選択部分を拡大縮小します。選択した間隔内に他のタイムラインオブジェクトはすべて互いに相対的な位置を保って拡大縮小されます。拡大縮小前に隣接していた間隔は、間が空いた
- 任意の数のタイムラインオブジェクト(間隔、キー、またはマーカー)

ATF Tree List Editor Sample

(日本語で表示 )

Description

TreeListEditor is a sample editor that shows how to use various tree lists.

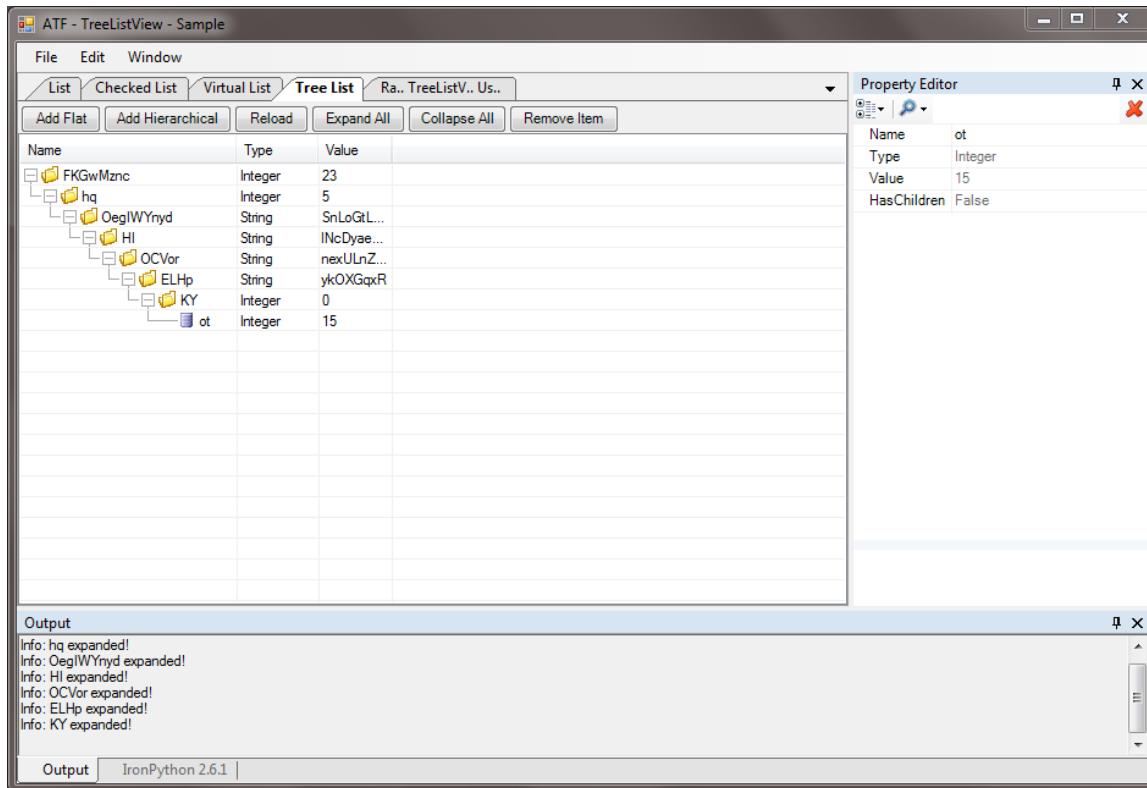
To learn about the programming of this sample, see [Tree List Editor Programming Discussion](#).

ATF Features Demonstrated by TreeListEditor

- Use of Managed Extensibility Framework (MEF) to put applications together and to extend a tree list view.
- Use of the application shell framework, including CommandService, SettingsService, ControlHostService and WindowLayoutService.
- Use of TreeListView and TreeListViewEditor to display a list, a checked list, a virtual list and a tree list on the main dialog's tabs.
- Use of the ITreeListView, IItemView, and ISelectionContext interfaces to contain and select generated list items.
- Use of the IComparer interface to sort column lists.

Run TreeListEditor

1. Double-click the TreeListEditor.exe in ATF\Samples\TreeListEditor\bin\Release.
2. A dialog appears with a tab for every list type demonstrated.



Menu Options

- File: choose Exit to exit TreeListEditor.
- Edit:
 - Copy: Automatically added by ATF; does nothing in this sample.
 - Select All: Automatically added by ATF; does nothing in this sample
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current TreeListEditor application settings or load application settings from a file.
 - Preferences: set application preferences, such as command icon size.
- Window:

- Layouts:
 - Save Layout As...: associate the current layout with a name.
 - Manage Layouts...: show a list of layouts and manage the list.
- Tile Horizontal: tile the list type tabs horizontally.
- Tile Vertical: tile the list type tabs vertically.
- Tile Overlapping: overlap the list type tabs horizontally.
- List of checked menu items; check to display the corresponding control.

How to Use TreeListEditor

Click on the tab of the list you want to try out. The four list tabs each have a set of buttons to randomly generate flat lists of items. In addition, the Tree List tab has a button to add a hierarchical list. The Raw TreeListView Usage tab allows you to select folders to display their file hierarchy. You can remove a selected item from the Tree List and Raw TreeListView Usage tabs.

TreeListEditor Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF and internal classes used.
- Editors.cs: Sets up the user interface elements on the List, Checked List, Virtual List and Tree List tabs.
- RawUsage.cs: Sets up the user interface elements on the Raw TreeListView Usage tab.
- DataGenerator.cs: Generates data for the tabs. Handles all button click events. Sets up sorting for column entries. Sets up the Property Editor.

ATF Tree List Editor Sample_j

(View in English 

説明

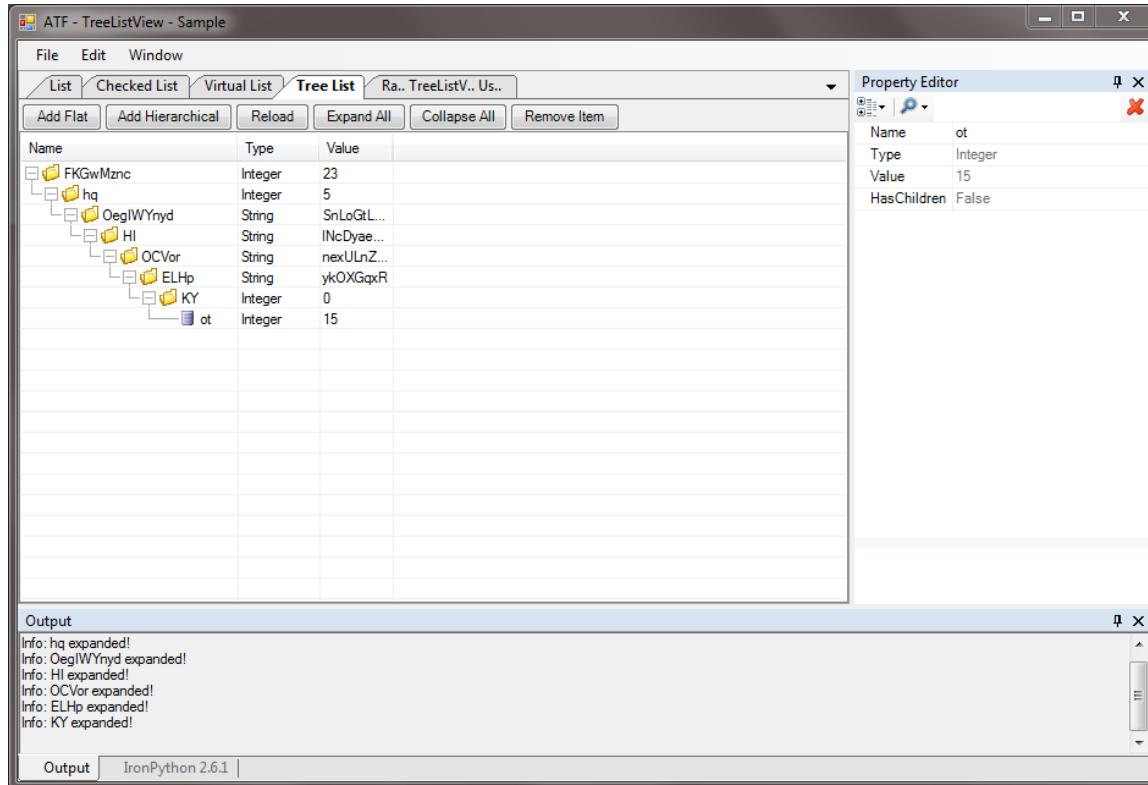
TreeListEditor はさまざまなツリーリストの使用法を示すサンプルエディタです。

TreeListEditor が示す ATF の機能

- Managed Extensibility Framework (MEF) を使用した、アプリケーションのまとめとツリーリスト表示の展開。
- CommandService、SettingsService、ControlHostService、および WindowsLayoutService を含むアプリケーションシェルフレームワークの使用。
- TreeListView および TreeListViewEditor を使用して、メインダイアログのタブ上にリスト、チェックボックス付きリスト、仮想リスト、およびツリー構造リストを表示する。
- ITreeListView、IItemView および ISelectionContext インタフェースを使用して、生成されたリストアイテムを含めて選択する。
- IComparer インタフェースを使用して、列リストを並べ替える。

TreeListEditor の実行

1. ATF\Samples\TreeListEditor\bin\Release にある TreeListEditor.exe をダブルクリックします。
2. 全種類のリストのタブを含むダイアログが表示されます。



メニューとオプション

- [File] (ファイル): [Exit] (終了) をクリックすると、TreeListEditor が終了します。
- [Edit] (編集)
 - [Copy] (コピー): ATF によって自動的に追加されていますが、このサンプルでは何もしません。
 - [Select All] (すべて選択): ATF によって自動的に追加されていますが、このサンプルでは何もしません。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
- [Load and Save Settings]: このウィンドウを使用して、現在の TreeListEditor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズなど、アプリケーションを設定します。
- [Window] (ウィンドウ):
 - [Layouts] (レイアウト):

- [Save Layout As...] (名前を付けてレイアウトを保存): 現在のレイアウトに名前を付けて保存します。
- [Manage Layouts...] (レイアウトの管理): レイアウトのリストを表示し、リストを管理します。
- [Tile Horizontal] (左右に並べて表示): リストの種類タブを水平に並べて表示します。
- [Tile Vertical] (上下に並べて表示): リストの種類タブを上下に並べて表示します。
- [Tile Overlapping] (重ねて表示): リストの種類タブを重ねて表示します。
- チェックボックス付きのメニューアイテムのリスト: 表示するコントロールのチェックボックスをオンにします。

TreeListEditor の使用法

試してみたいリストのタブをクリックします。4

つのリストタブそれぞれにボタンがあり、アイテムの階層のないリストをランダムに生成します。[Tree List]

タブには、階層構造のリストを追加するためのボタンも付いています。[Raw TreeListView Usage]

タブでは、ファイル階層を表示するフォルダを選択できます。[TreeList] タブと [Raw TreeListView Usage]

タブでは選択した項目を削除できます。

TreeListEditor のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- Editors.cs: [List]、[Checked List]、[Virtual List]、および [Tree List] の各タブにユーザインタフェース要素を設定します。
- RawUsage.cs: [Raw TreeListView Usage] タブにユーザインタフェース要素を設定します。
- DataGenerator.cs: タブにデータを生成し、すべてのボタンクリックイベントを処理し、列の並べ替えを設定し、Property Editor を設定します。

ATF Target Manager Sample

(日本語で表示 

Description

TargetManager is a sample that shows how to use the TargetEnumerationService to discover, add, configure and select targets. Targets are network endpoints, such as TCP/IP addresses, PS3 DevKits (to be added) or Vita DevKits. TargetEnumerationService is implemented as a MEF plugin that supports the ITargetConsumer interface for querying and enumerating targets. It consumes target providers created by the application.

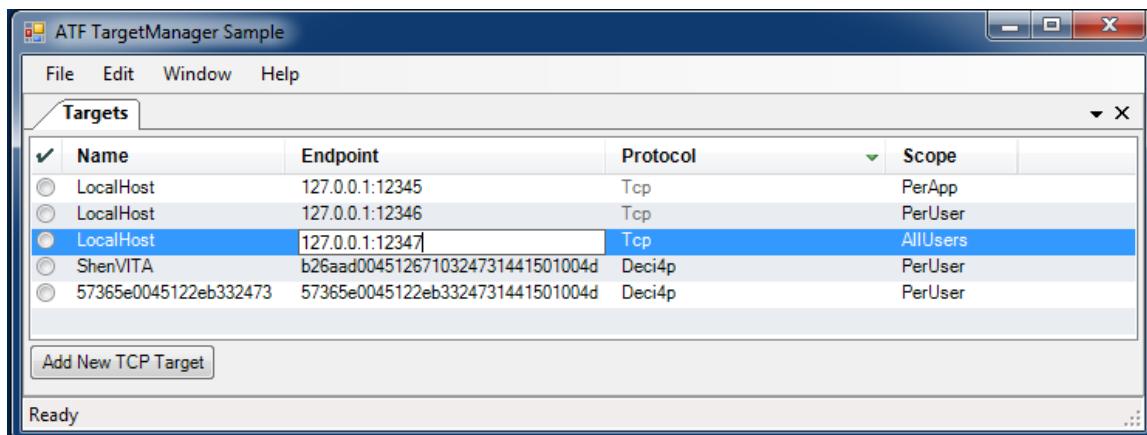
To learn about TargetManager's implementation, see [Target Manager Programming Discussion](#).

ATF Features Demonstrated by TargetManager

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including CommandService, SettingsService and ControlHostService.
- Use of TcplpTargetProvider and Deci4pTargetProvider to add, display and edit TCP/IP and Deci4p target providers. Each target provider is responsible for discovering and reporting targets of its specific type and their parameters. Each target provider is implemented as a MEF plugin that supports the ITargetProvider interface.
- Use of TargetEnumerationService to enumerate available target providers. TargetEnumerationService combines all targets' information into a heterogeneous list view for displaying and editing. Currently it supports targets of type Deci4p and TCP/IP, although it is designed to easily support other target types in the future.

Run TargetManager

1. Double-click the TargetManager.exe in ATF\Samples\TargetManager\bin\Release.
2. A dialog appears listing all targets.



Menu Options

- File: choose Exit to exit TargetManager.
- Edit:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current TargetManager application settings or load application settings from a file.
 - Preferences: set application preferences, such as command icon size.
- Window:
 - Tile Horizontal: automatically added by ATF; does nothing in this sample.
 - Tile Vertical: automatically added by ATF; does nothing in this sample.
 - Tile Overlapping: automatically added by ATF; does nothing in this sample.
 - Targets: check to display the Targets list control.
- Help
 - About: display an information dialog.

How to Use TargetManager

Click on one of the Add New... buttons at the bottom of the dialog to add a target of that type. The buttons available depend on the computer's configuration, although you can always add a TCP/IP target. Once added, you can edit the Endpoint and Scope items for each connection.

TargetManager Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF classes used, which are instantiated using MEF. It uses TargetEnumerationService along with TcplpTargetProvider and Deci4pTargetProvider, so no other modules are needed.

ATF Target Manager Sample_j

(View in English 

説明

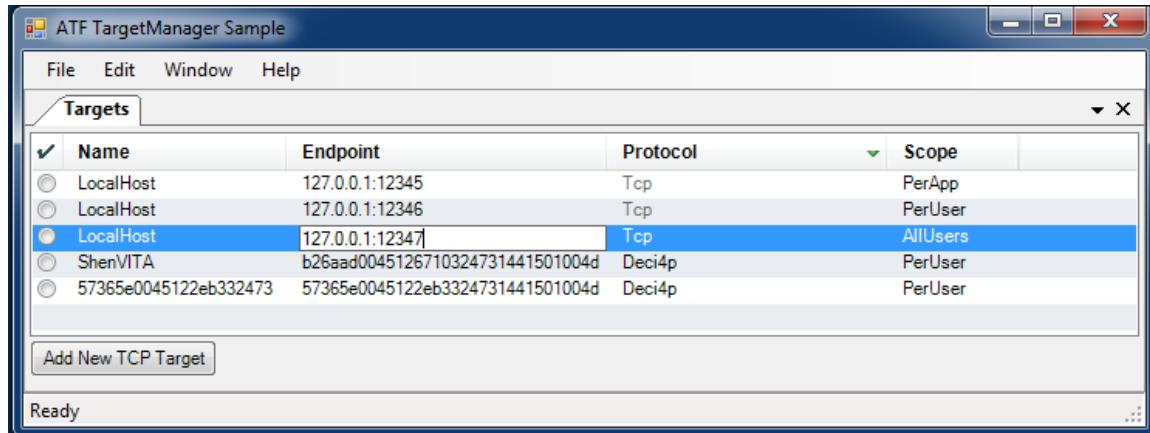
TargetManager は、ターゲットを検出、追加、構成、および選択するための TargetEnumerationService の使用方法を示すサンプルです。ターゲットは、TCP/IP アドレス、PS3 開発ツール（追加予定）、または Vita 開発ツールなどのネットワークエンドポイントです。TargetEnumerationService は、ターゲットのクエリと列挙を実行するために ITargetConsumer インタフェースをサポートする、MEF プラグインとして実装され、アプリケーションによって作成されたターゲットプロバイダを消費します。

TargetManager が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用して、アプリケーションをまとめる。
- CommandService、SettingsService、および ControlHostService を含むアプリケーションシェルフレームワークの使用。
- TcpIpTargetProvider および Deci4pTargetProvider を使用して、TCP/IP および Deci4p ターゲットプロバイダを追加、表示、編集する。
各ターゲットプロバイダは、それぞれの特定のタイプのターゲットとパラメータを検出および報告する必要があります。
各ターゲットプロバイダは、ITargetProvider インタフェースをサポートする MEF プラグインとして実装されます。
- TargetEnumerationService を使用して利用可能なターゲットプロバイダを列挙する。TargetEnumerationService は、異なるターゲットすべての情報をリストにまとめて、表示および編集できるようにします。現在は Deci4p と TCP/IP のタイプのターゲットをサポートしていますが、将来他のターゲットタイプも容易にサポートできるように設計されています。

TargetManager の実行

1. ATF\Samples\TargetManager\bin\Release にある TargetManager.exe をダブルクリックします。
2. ダイアログが表示され、すべてのターゲットがリスト表示されます。



メニューとオプション

- [File] (ファイル): [Exit] (終了) をクリックすると、TargetManager が終了します。
- [Edit] (編集)
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts]
ウインドウを使用して、キーボードショートカットを設定します。
- [Load and Save Settings]: このウインドウを使用して、現在の TargetManager の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズなど、アプリケーションを設定します。
- [Window] (ウインドウ):
 - [Tile Horizontal] (左右に並べて表示): ATF によって自動的に追加されていますが、このサンプルでは何もしません。
 - [Tile Vertical] (上下に並べて表示): ATF によって自動的に追加されていますが、このサンプルでは何もしません。
 - [Tile Overlapping] (重ねて表示): ATF によって自動的に追加されていますが、このサンプルでは何もしません。
 - [Targets]: チェックボックスをオンにすると、[Targets] リストコントロールが表示されます。
- [Help] (ヘルプ):
 - [About] (バージョン情報): バージョン情報ダイアログを表示します。

TargetManager の使用法

ダイアログの下に表示されている [Add New...] ボタンをクリックして、そのボタンのタイプのターゲットを追加します。利用できるボタンはコンピュータの構成によって異なりますが、TCP/IP ターゲットは常に追加できます。追加した後は、各接続の [Endpoint] と [Scope] のアイテムを編集できます。

TargetManager のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。MEF を使用してインスタンス化された、使用されている ATF クラスをリストした TypeCatalog を作成します。TcplpTargetProvider およびDeci4pTargetProvider とともにTargetEnumerationService を使用するので、他のモジュールは必要ありません。

ATF Code Editor Sample

(日本語で表示 

Description

CodeEditor is a code editor that uses the ActiproSoftware SyntaxEditor to provide an editing Control. It provides language syntax sensitive editing for plain text, C#, Lua, Squirrel, Python, XML, COLLADA and Cg files.

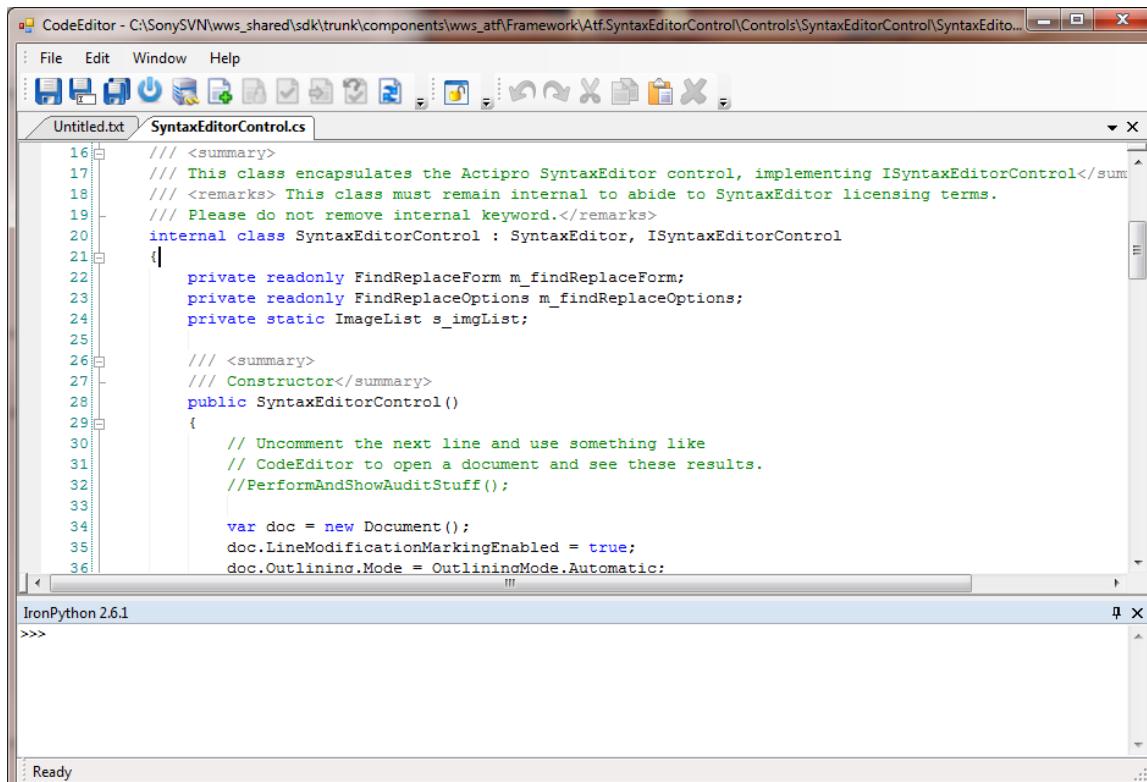
For a description of the sample's programming, see [Code Editor Programming Discussion](#).

ATF Features Demonstrated by CodeEditor

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including CommandService, SettingsService, ControlHostService and WindowLayoutService.
- Use of FileDialogService, StandardFileCommands and StandardFileExitCommand to provide standard menus.
- Use of PerforceService, SourceControlCommands and SourceControlContext to provide source control.
- Use of a third party control (ActiproSoftware SyntaxEditor) that conforms to the ISyntaxEditorControl to provide most of the editor functions. See the files in the Atf.SyntaxEditorControl project for more details.

Run CodeEditor

1. Double-click the CodeEditor.exe in ATF\Samples\CodeEditor\bin\Release.
2. A dialog appears with an empty text window.



Menu Options

- File:
 - New: create a new file of various types using submenus for the language options.
 - Open: open an existing file of various types using submenus for the language options.
 - Save: save the active file.
 - Save As ...: save the active file to a selected path.
 - Save All: save all the open files.
 - Close: close the active file.

- Source Control: select its various submenu items to perform standard source control operations, such as adding, checking in or out, refreshing files.
- Recent Files: open a recently opened file.
- Exit: exit CodeEditor.
- Edit:
 - Undo and Redo: undo or redo the last editing operation.
 - Cut, Copy, Paste and Delete: performs these editing operations.
 - Find and Replace...: display a Find/Replace dialog with the usual options.
 - Go to: display a dialog to enter a line number to go to.
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current CodeEditor application settings or load application settings from a file.
 - Preferences: set application preferences, such as command icon size.
- Window:
 - Tile Horizontal: tile window panes horizontally.
 - Tile Vertical: tile window panes vertically.
 - Tile Overlapping: overlap window panes horizontally.
 - Layouts:
 - Save Layout As...: associate the current layout with a name.
 - Manage Layouts...: show a list of layouts and manage the list.
 - Lock/Unlock UI Layout: lock or unlock the window layout.
 - List of checked menu items; check to display the corresponding control.
- Help: its About menu item displays a dialog describing Code Editor.

Toolbar Options

The toolbar contains buttons for most of the commands in the menus. For instance, all of the various Save commands are available, as well as the Edit commands.

How to Use CodeEditor

Create a new file by clicking File > New > language. Or open an existing file by clicking File > Open > language. Enter text and perform the usual editing operations.

After creating content, you can save it with the various save options under the File menu or toolbar.

CodeEditor Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF components used.
- Editors.cs: Sets up the ActiproSoftware SyntaxEditor control that does the editing. Sets up document clients for the different language types the editor supports.
- CodeDocument.cs: Defines the CodeDocument class, which implements the IDocument interface. Editors.cs creates a CodeDocument instance for every open document. CodeDocument performs the general document actions such as saving and reading text to and from files.
- SourceControlContext.cs: Defines the SourceControlContext class, which implements ISourceControlContext. The PerforceService, SourceControlCommands and SourceControlContext components actually do the source control management work.

ATF Code Editor Sample_j

(View in English 

説明

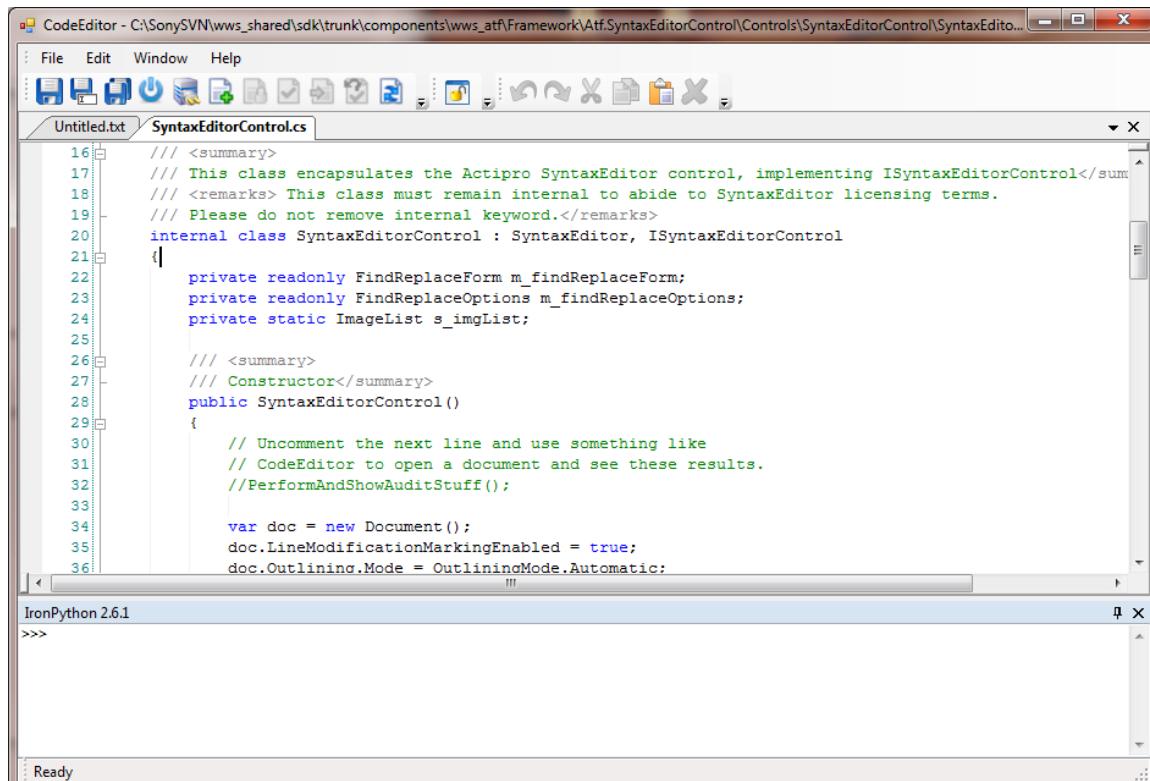
CodeEditor は、Actipro Software 社の SyntaxEditor を使用して編集コントロールを提供するコードエディタです。書式なしテキスト、C#、Lua、Squirrel、Python、XML、COLLADA、および Cg ファイルのための言語構文依存型編集機能を提供します。

CodeEditor が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用した、アプリケーションのまとめ。
- CommandService、SettingsService、ControlHostService、および WindowsLayoutService を含むアプリケーションシェルフレームワークの使用。
- FileDialogService、StandardFileCommands、および StandardFileExitCommand を使用した標準メニューの提供。
- PerforceService、SourceControlCommands、および SourceControlContext を使用したソース管理機能の提供。
- ISyntaxEditorControl に適合したサードパーティ製コントロール (Actipro Software 社のSyntaxEditor) を使用した、エディタ機能の大部分の提供。詳細は、Atf.SyntaxEditorControl プロジェクト内のファイルを参照してください。

CodeEditor の実行

1. ATF\Samples\CodeEditor\bin\Release にある CodeEditor.exe をダブルクリックします。
2. 空のテキストウィンドウが開いた diáloが表示されます。



メニュー オプション

- [File] (ファイル):
 - [New] (新規作成): サブメニューの言語オプションを使用して、さまざまな種類のファイルを新規作成します。
 - [Open] (開く): サブメニューの言語オプションを使用して、さまざまな種類の既存のファイルを開きます。
 - [Save] (保存): アクティブなファイルを保存します。
 - [Save As] (名前を付けて保存): アクティブなファイルを、保存先およびファイル名を指定して保存します。
 - Save All (すべて保存): 開いているファイルをすべて保存します。
 - [Close] (閉じる): アクティブなファイルを閉じます。
 - [Source Control] (ソース管理): サブメニュー項目を選択し、追加、チェックアウト、チェックイン、最新バージョンの取得などの標準のソース管理の操作をします。

- [Recent Files] (最近使用したファイル): 最近開いたファイルを開きます。最近開いたファイルがない場合、このメニュー項目は表示されません。
- [Exit] (終了): CodeEditor を終了します。
- [Edit] (編集)
 - [Undo] および [Redo] (元に戻す/やり直し): 最後の編集操作を元に戻すかやり直します。
 - [Cut] (切り取り)、[Copy] (コピー)、[Paste] (貼り付け)、および [Delete] (削除): 各編集操作を実行します。
 - [Find and Replace] (検索と置換): 通常のオプションを含む [Find/Replace] ダイアログを表示します。
 - [Go to] (指定行へ移動): 表示されるダイアログに、移動先の行番号を入力します。
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
 - [Load and Save Settings] (設定の読み込みおよび保存): このウィンドウを使用して、現在の Code Editor の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズなど、アプリケーションの詳細を設定します。
- [Window] (ウィンドウ):
 - [Tile Horizontal] (左右に並べて表示): ウィンドウペインを水平に並べて表示します。
 - [Tile Vertical] (上下に並べて表示): ウィンドウペインを上下に並べて表示します。
 - [Tile Overlapping] (重ねて表示): ウィンドウペインを重ねて表示します。
 - [Layouts] (レイアウト):
 - [Save Layout As] (名前を付けてレイアウトを保存): 現在のレイアウトに名前を付けて保存します。
 - [Manage Layouts] (レイアウトの管理): レイアウトのリストを表示し、リストを管理します。
 - [Lock UI Layout] および [Unlock UI Layout]: ウィンドウレイアウトのロックとロックの解除を切り替えます。
 - チェックボックス付きのメニュー項目のリスト: コントロールをクリックするとチェックマークが付き、そのコントロールが表示されます。
- [Help] (ヘルプ): [About] ダイアログに CodeEditor の情報を表示します。

ツールバー オプション

メニューにあるコマンドのほとんどが、ツールバーにボタンとして表示されます。たとえば、保存と編集に関するコマンドはツールバーにすべて含まれています。

CodeEditor の使用法

[File] > [New] > 言語をクリックして新規ファイルを作成します。または、[File] > [Open] > 言語をクリックして既存のファイルを開きます。テキストを入力し、通常どおりに編集します。

コンテンツを作成し終えたら、[File] メニューまたはツールバーにある、さまざまな保存オプションを使用して保存できます。

CodeEditor のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF コンポーネントをリストした TypeCatalog を作成します。
- Editors.cs: 編集を実行する Actipro Software 社製 SyntaxEditor コントロールを設定します。同エディタが対応する各言語タイプに対してドキュメントクライアントを設定します。
- CodeDocument.cs: IDocument インタフェースを実装する CodeDocument クラスを定義します。Editors.cs は、開いているドキュメントごとに CodeDocument インスタンスを作成します。CodeDocument は、テキストのファイルからの読み取り、ファイルへの保存などの、一般的なドキュメント操作を実行します。
- SourceControlContext.cs: ISourceControlContext を実装する SourceControlContext クラスを定義します。実際のソース管理操作は、PerforceService、SourceControlCommands、および SourceControlContext コンポーネントにより実行されます。

ATF Property Editing Sample



(日本語で表示)

Description

PropertyEditing is a sample that shows how to use and customize both standard and grid property editors.

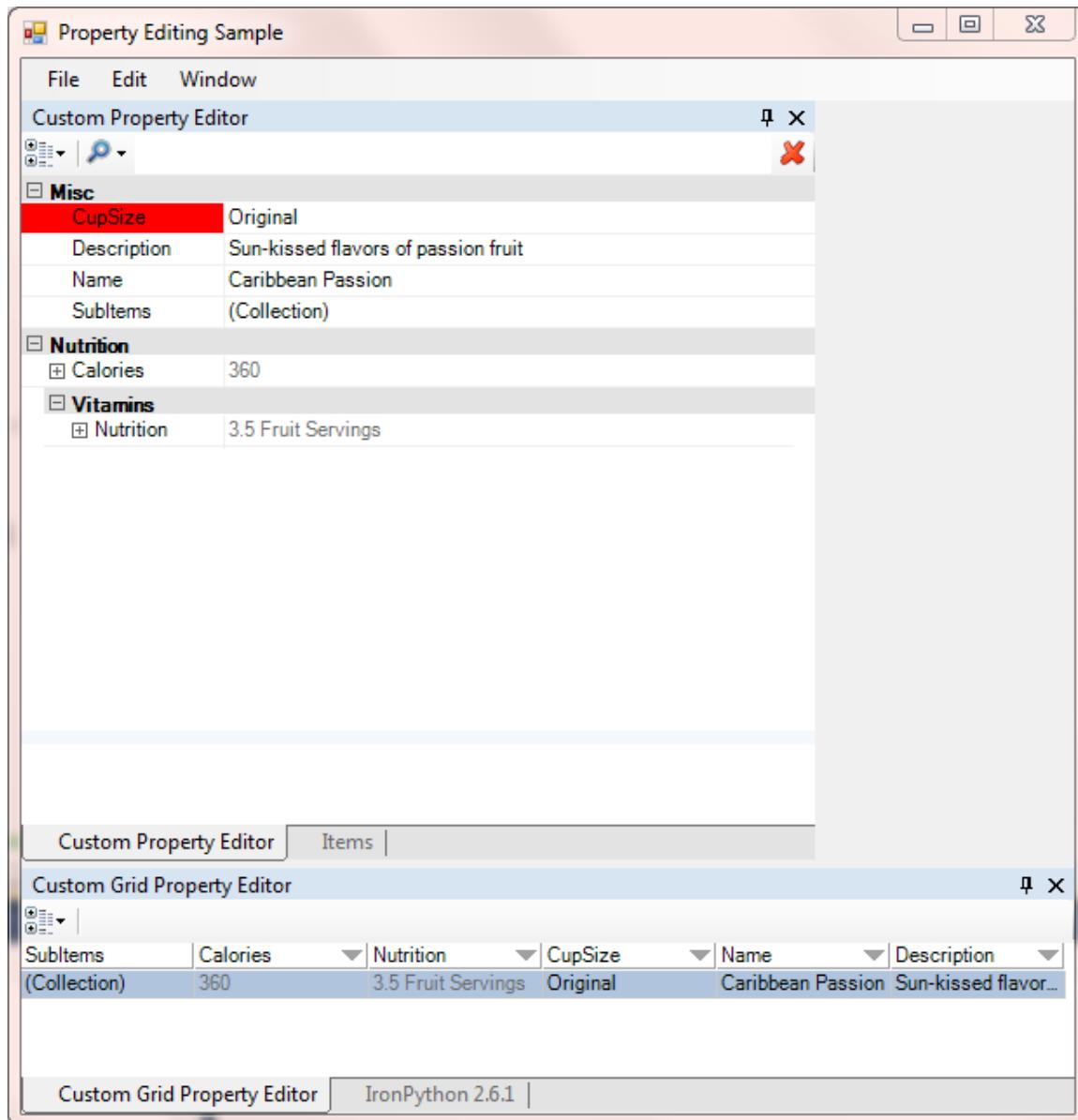
For information about the programming in this sample, see [Property Editing Programming Discussion](#).

ATF Features Demonstrated by PropertyEditing

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including CommandService, SettingsService and ControlHostService.
- Use of CustomPropertyEditor, CustomGridPropertyEditor and PropertyEditingCommands to implement property editors.

Run PropertyEditing

1. Double-click the PropertyEditing.exe in ATF\Samples\PropertyEditing\bin\Release.
2. A dialog appears showing property editor controls.



Menu Options

- File: choose Exit to exit PropertyEditing.
- Edit:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current PropertyEditing application settings or load application settings from a file.
 - Preferences: set application preferences, such as command icon size.
- Window:
 - Tile Horizontal: tile window panes horizontally.
 - Tile Vertical: tile window panes vertically.
 - Tile Overlapping: overlap window panes horizontally.
 - Lock/Unlock UI Layout: lock or unlock the window layout.
 - List of checked menu items; check to display the corresponding control.

How to Use PropertyEditing

Click the Items tab and select one of the built in items. Then click the Custom Property Editor tab to show the list property editor. Click the Custom Grid Property Editor tab to show the grid property editor. You can edit items in either of these property editors.

PropertyEditing Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF classes used.
- CustomPropertyEditor.cs: Customized version of two-column property editor component.
- CustomGridPropertyEditor.cs: Component to edit item values and attributes using the GridControl.
- ListEditingContext.cs: Context for list item handling and selection. Initializes list values.
- ListItem.cs: An attribute that can be placed on a field within a class to expose that field as an editable property. This editable property is a child property in the two-column property editor.

ATF Property Editing Sample_j

(View in English 

説明

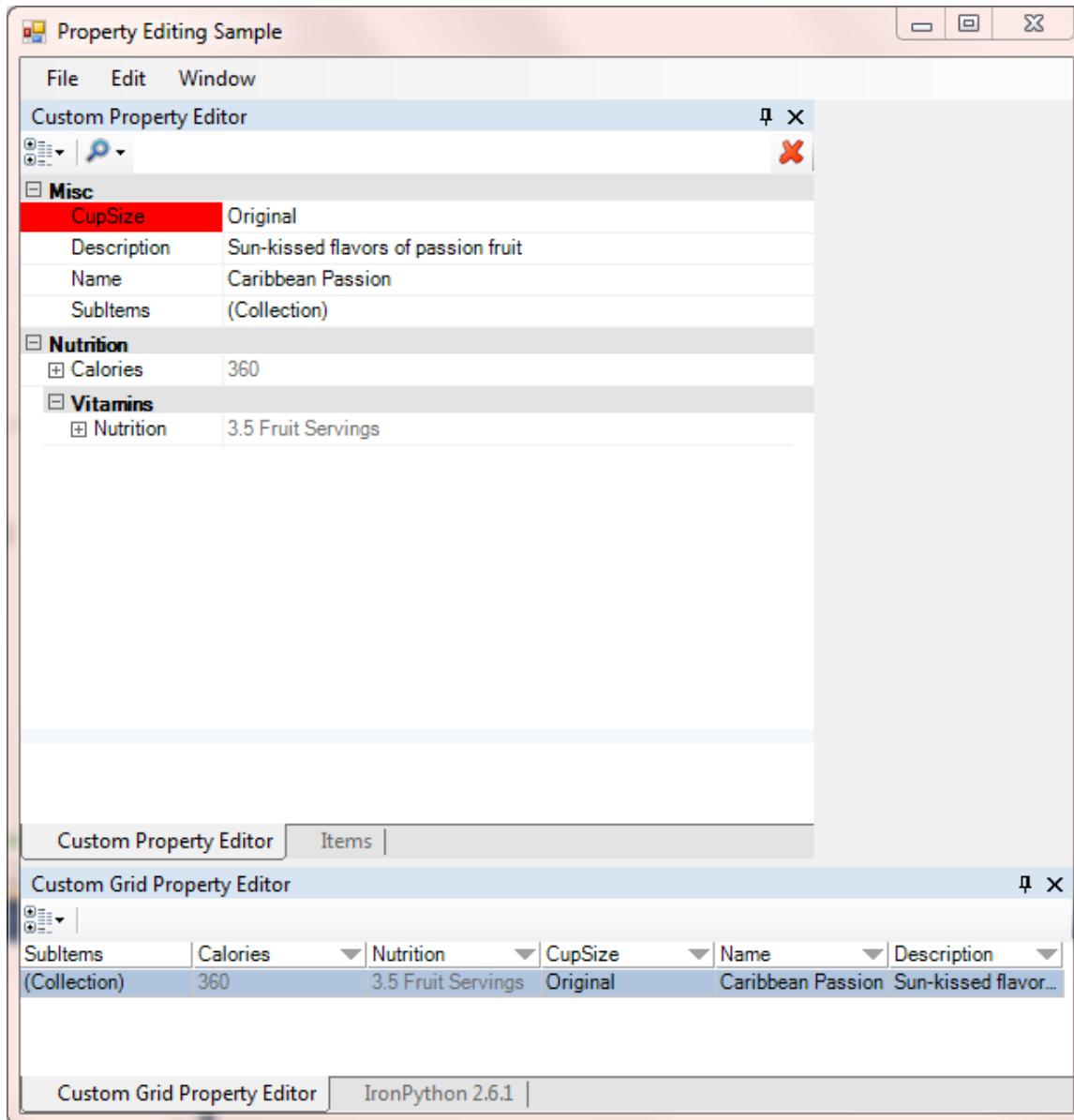
PropertyEditing は標準プロパティエディタおよびグリッドプロパティエディタ両方の使用法およびカスタム化手順を示すサンプルです。

PropertyEditing が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用して、アプリケーションをまとめる。
- CommandService、SettingsService、および ControlHostService を含むアプリケーションシェルフレームワークの使用。
- CustomPropertyEditor、CustomGridPropertyEditor および PropertyEditingCommands を使用して、プロパティエディタを実装する。

PropertyEditing の実行

1. ATF\Samples\PropertyEditing\bin\Release にある PropertyEditing.exe をダブルクリックします。
2. ダイアログが表示され、プロパティエディタのコントロールが表示されます。



メニューとオプション

- [File] (ファイル): [Exit] (終了) をクリックすると、PropertyEditing が終了します。

- [Edit] (編集)
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts]
ウインドウを使用して、キーボードショートカットを設定します。
- [Load or Save Settings]: このウインドウを使用して、現在の PropertyEditing のアプリケーション設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズなど、アプリケーションの詳細を設定します。
- [Window] (ウィンドウ):
 - [Tile Horizontal] (左右に並べて表示): ウィンドウペインを水平に並べて表示します。
 - [Tile Vertical] (上下に並べて表示): ウィンドウペインを上下に並べて表示します。
 - [Tile Overlapping] (重ねて表示): ウィンドウペインを重ねて表示します。
 - [Lock UI Layout] および [Unlock UI Layout]: ウィンドウレイアウトのロックとロックの解除を切り替えます。
 - チェックボックス付きのメニューアイテムのリスト:
コントロールをクリックするとチェックマークが付き、そのコントロールが表示されます。

PropertyEditing の使用法

[Items] タブをクリックし、組み込まれている項目をひとつ選択します。選択したら、[Custom Property Editor]

タブをクリックして、プロパティエディタのリストを表示します。 [Custom Grid Property Editor]

タブをクリックして、グリッドプロパティエディタを表示します。 いずれのプロパティエディタでも項目を編集できます。

PropertyEditing のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。 このプログラムが、使用されている ATF クラスをリストした TypeCatalog を作成します。
- CustomPropertyEditor.cs: 列が 2 つであるプロパティエディタコンポーネントのカスタム化版です。
- CustomGridPropertyEditor.cs: GridControl を使用して、項目の値と属性を編集するためのコンポーネントです。
- ListEditingContext.cs: リスト項目の処理や選択のためのコンテキストです。 リスト値を初期化します。
- ListItem.cs: クラス内のフィールド上に配置が可能な属性で、フィールドを編集可能なプロパティとして表示します。
この編集可能なプロパティは、列が 2 つあるプロパティエディタ内の子プロパティです。

ATF Simple DOM No XML Editor Sample

(日本語で表示 

Description

SimpleDomNoXmlEditor is a sample editor that demonstrates the use of the DOM (Document Object Model). SimpleDomNoXmlEditor operates on event sequence files, *.SimpleDomTxt files, that contain a sequence of events. The events can contain resources: animations or geometries. Each event sequence file displays in a ListView control, which shows all events and resources that can be selected and edited, as well as the properties on the selected items to be edited. The editor can load multiple event sequence files. The Resources editor tracks the last selected event and displays its resources in another ListView control.

SimpleDomNoXmlEditor is very similar to the SimpleDOMEditor sample, but does not use XML.

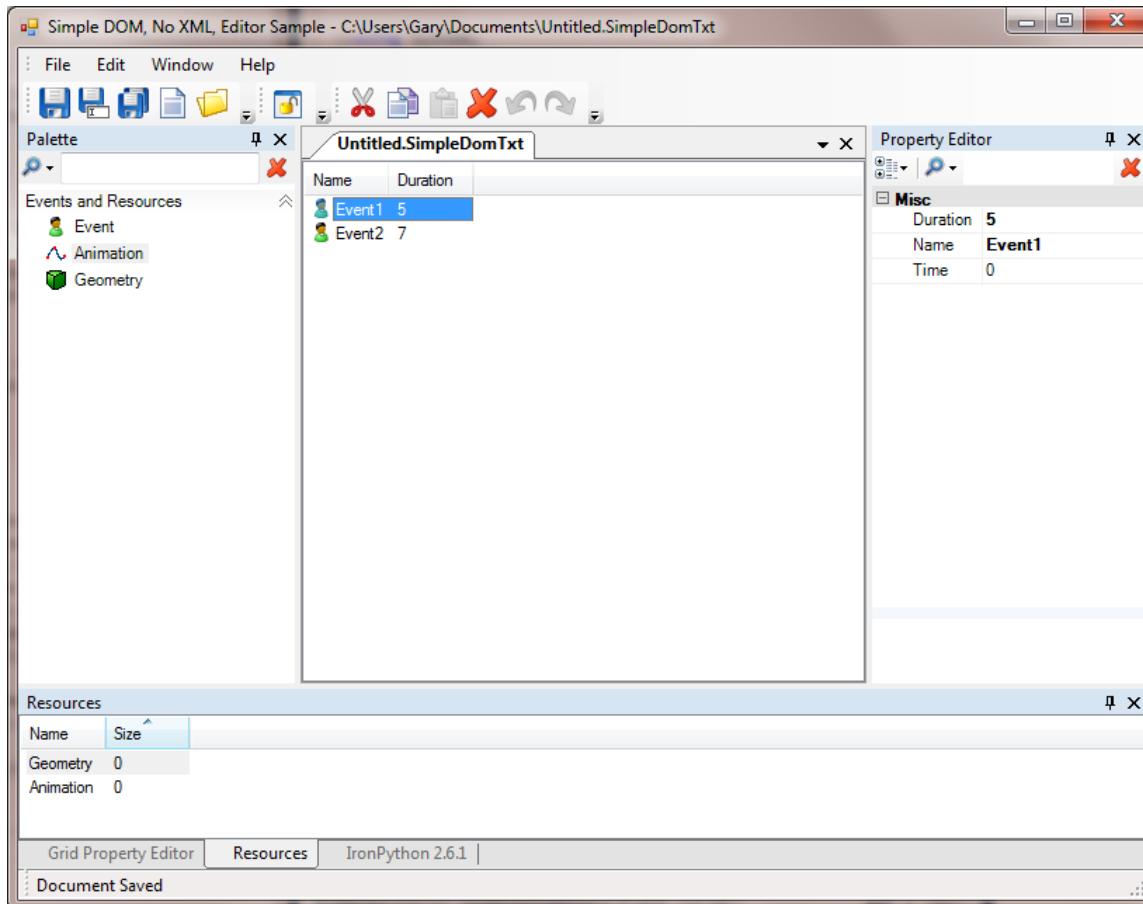
For information about programming in this sample, see [Simple DOM No XML Editor Programming Discussion](#).

ATF Features Demonstrated by SimpleDomNoXmlEditor

- Define a custom data model, not using an XML schema. It does not use XML for application data either, which SimpleDOMEditor does.
- Use of adapters to decorate the DOM to create event sequence data models.
- Use of IDocumentClient and the document framework to manage multiple documents.
- Show implementing IPaletteClient to create a UI parts palette.
- Using ListView and ListViewAdapter to display editable lists of events and resources.
- Use of IListView, IItemView, and IObservableContext interfaces to adapt data to a list.
- Use of ContextRegistry to track the active editing context, so application components always apply where the user is editing.
- Use of the interfaces IInstancingContext, ISelectionContext, and IHISTORYContext to adapt data for editing commands.
- Use of ATF PropertyEditor and GridPropertyEditor components to allow property editing on selected UI elements.

Run SimpleDomNoXmlEditor

1. Double-click the SimpleDomNoXmlEditor.exe in ATF\Samples\SimpleDomNoXmlEditor\bin\Release.
2. The SimpleDomNoXmlEditor window appears.



SimpleDomNoXmlEditor has the following window panes:

- Palette (Events and Resources): choose Event, Animation, or Geometry.
- Property Editor: edit a selected event or resource property in a list control.
- Grid Property Editor: edit a selected event or resource property in a grid control.
- Resources: list the resources for the active event sequence file.
- Canvas: displays the event sequence file contents.

The toolbar and menu bar contain standard items for file management: save, create new, open existing files, and so on; for locking/unlocking the UI window panes; and for editing: cut, copy, paste, delete, undo/redo. The Edit menu provides the standard editing functions as well as keyboard shortcuts, load or save UI window settings, and preferences. You can use Window menu items to activate and rearrange window panes. Help provides information about SimpleDomNoXmlEditor.

How to Use SimpleDomNoXmlEditor

To start:

1. Create an event sequence file. Use the initial "Untitled.SimpleDomText" window, or choose File > New Event Sequence in the SimpleDomNoXmlEditor toolbar. An empty sequence file opens in the editing canvas.
2. From the Palette, drag and drop an Event onto the event sequence file in the editing canvas.
3. Click the tabs for the various panes to view and edit event properties.

To add resources:

1. Drag and drop resources (Animation or Geometry) onto the Resource pane.
2. Click the tabs for the various panes to view and edit resource properties.

TreeListEditor Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF and internal classes used.
- DomTypes.cs: Defines DOM data types, not using an XML schema.
- Editors.cs: Implements IDocumentClient and uses the document framework to manage multiple documents, implement File menu commands, auto-new and open documents on startup.
- PaletteClient.cs: Implements IPaletteClient and uses IPaletteService to create a UI parts palette.

- EventListEditor.cs: Uses ListView and ListViewAdapter to display editable lists of events and resources.
- ResourceListEditor.cs: Display and edit the resources that belong to the most recently selected event. It handles drag and drop, and right-click context menus for events and resources.
- EventSequenceContext.cs, EventContext.cs: Use IListView, IItemView, and IObservableContext interfaces to adapt data to a list.
- EventSequenceContext.cs, EventContext.cs: Implement the ATF interfaces IInstancingContext, ISelectionContext, and IH歷史Context to adapt data so that ATF command components can be used to get undo/redo, cut/paste, and selection commands.
- HelpAboutCommand.cs: Implements a standard Help/About dialog.

ATF Simple DOM No Xml Editor Sample_j

(View in English 

説明

SimpleDomNoXmlEditor は、ドキュメントオブジェクトモデル (DOM) の使用法を示すサンプルエディタです。SimpleDomNoXmlEditor は一連のイベントを含むイベントシーケンスファイルである *.SimpleDomTxt を操作します。イベントには、リソース (アニメーションおよびジオメトリ) が含まれます。各イベントシーケンスファイルは ListView コントロールに表示されます。すべてのイベントとリソースが表示されますが、これらは選択して編集することができます。また編集するために選択されたアイテムエディタは最後に選択されたイベントを追跡して、そのリソースを別の ListView コントロールに表示します。

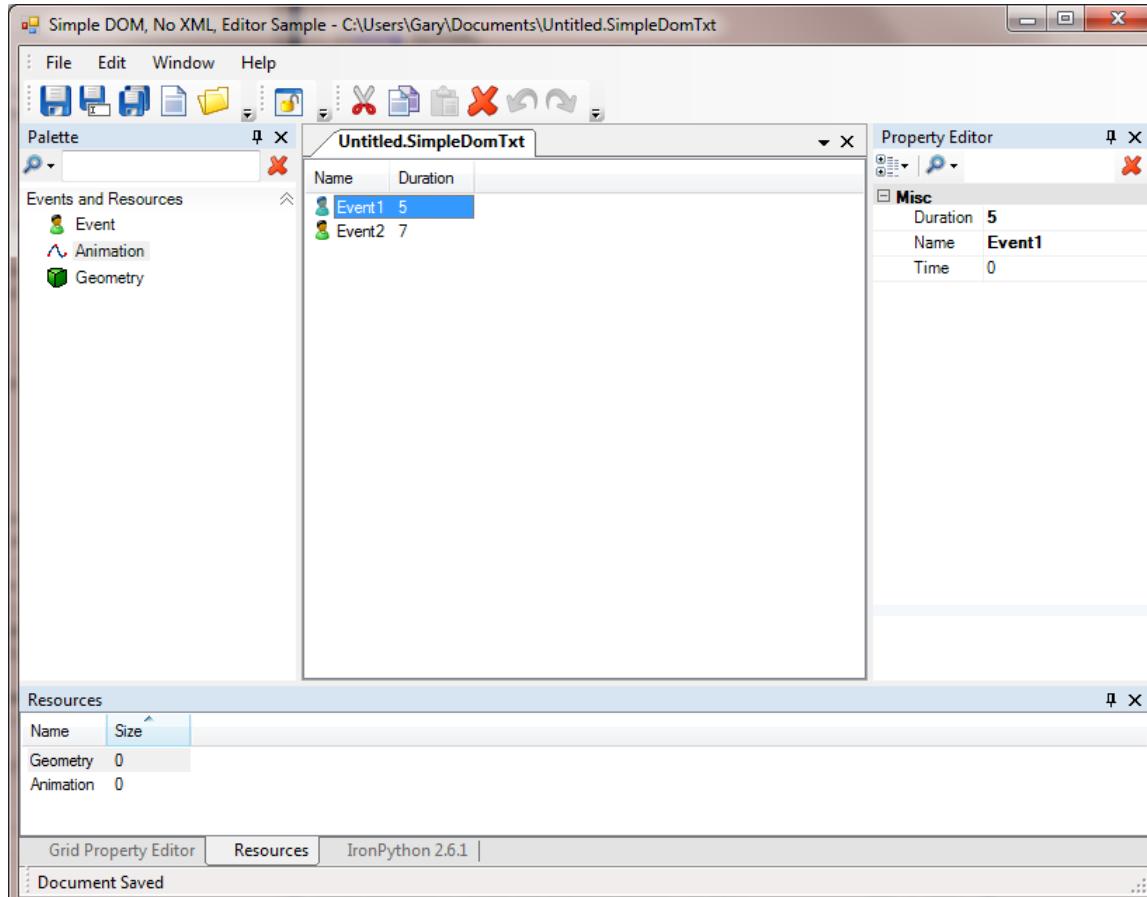
SimpleDomNoXmlEditor は SimpleDOMEdition サンプルにとてもよく似ていますが、XMLを使用しません。

SimpleDomNoXmlEditor が示す ATF の機能

- XML スキーマを使用しないカスタムデータモデルを定義する。SimpleDOMEdition と異なり、SimpleDomNoXmlEditor ではアプリケーションデータにも XML を使用しません。
- アダプタを使用して、イベントシーケンスデータモデルを作成するために DOM を装飾する。
- IDocumentClient およびドキュメントフレームワークを使用して複数のドキュメントを管理する。
- IPaletteClient を実装して、UI パーツパレットを作成する。
- ListView と ListViewAdapter を使用して、イベントおよびリソースの編集可能なリストを表示する。
- IListview、IItemView、および IObservableContext インタフェースを使用して、データをリストに適合させる。
- ContextRegistry を使用して、アクティペイブな編集コンテキストを追跡し、編集している場所にアプリケーションコンポーネントが常に適用されるようにする。
- IInstancingContext、ISelectionContext および IHistoricContext のインターフェースを使用して、データを編集コマンドに適合させる。
- ATF PropertyEditor コンポーネントおよび GridPropertyEditor コンポーネントを使用して、選択した UI 要素のプロパティを編集可能にする。

SimpleDomNoXmlEditor の実行

1. ATF\Samples\SimpleDomNoXmlEditor\bin\Release にある SimpleDomNoXmlEditor.exe をダブルクリックします。
2. [SimpleDomNoXmlEditor] ウィンドウが表示されます。



SimpleDomNoXmlEditor には次のペインがあります。

- [Palette]: [Event] (イベント)、[Animation] (アニメーション)、または [Geometry] (ジオメトリ) を選択します。
- [Property Editor]: 選択したイベントまたはリソースのプロパティをリストコントロールで編集します。
- [Grid Property Editor]: 選択したイベントまたはリソースのプロパティをグリッドコントロールで編集します。
- [Resources]: 選択されたイベントシーケンスファイルのリソースを一覧表示します。
- キャンバス: イベントシーケンスファイルの内容を表示します。

ツールバーおよびメニュー バーには、ファイル管理用に保存、イベントシーケンスを作成および開くなど、UI ウィンドウ ベインのロック/アンロック、編集用に切り取り、コピー、貼り付け、削除、元に戻す/やり直しなどの、標準的な機能が含まれています。編集メニュー - ウィンドウ 設定の読み込みまたは保存、詳細設定が含まれます。[Window]

メニュー アイテムを使用して、ウィンドウ ベインを表示したり並べ替えたりできます。[Help] では、SimpleDomNoXmlEditor に関する情報が提供されます。

SimpleDomNoXmlEditor の使用法

起動する手順を次に示します。

1. イベントシーケンスファイルを作成します。起動時に表示される [Untitled.SimpleDomText] タブを使用するか、またはツールバーで [File] > [New Event Sequence] をクリックします。編集キャンバスに空のシーケンスファイルが開かれます。
2. [Palette] から、[Event] を編集キャンバスのイベントシーケンスファイルにドラッグアンドドロップします。
3. 各ペインのタブをクリックして、イベントプロパティを表示し編集します。

リソースを追加する手順を次に示します。

1. リソース (アニメーションまたはジオメトリ) を [Resource] ペインにドラッグアンドドロップします。
2. 各ペインのタブをクリックして、リソースプロパティを表示し編集します。

TreeListEditor のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- DomTypes.cs: XML スキーマを使用せずに DOM データ型を定義します。
- Editor.cs: IDocumentClient を実装し、ドキュメントフレームワークを使用して、複数ドキュメントを管理し、[File] メニュー コマンドを実装し、起動時にドキュメントを自動作成して開きます。
- PaletteClient.cs: IPaletteClient を実装し、IPaletteService を使用して UI パーツ パレットを作成します。
- EventListEditor.cs: ListView と ListViewAdapter を使用して、イベントおよびリソースの編集可能なリストを表示します。
- ResourceListEditor.cs:
直近に選択されたイベントのリソースを表示し編集します。イベントとリソースのドラッグドロップおよび右クリックコンテキストメニューを処理します。
- EventSequenceContext.cs および EventContext.cs: IListView、IItemView、および IObservableContext インタフェースを使用して、データをリストに適合させます。
- EventSequenceContext.cs および EventContext.cs: ATF インタフェースである IInstancingContext、ISelectionContext、および IHISTORYContext を実装し、ATF コマンドコンポーネントを使用して、元に戻す/やり直し、切り取り/貼り付け、および選択のコマンドを取得できるようにデータを適合させます。
- HelpAboutCommand.cs: 標準のヘルプ/バージョン情報ダイアログを実装します。

ATF Using Direct2D Sample

(日本語で表示 )

Description

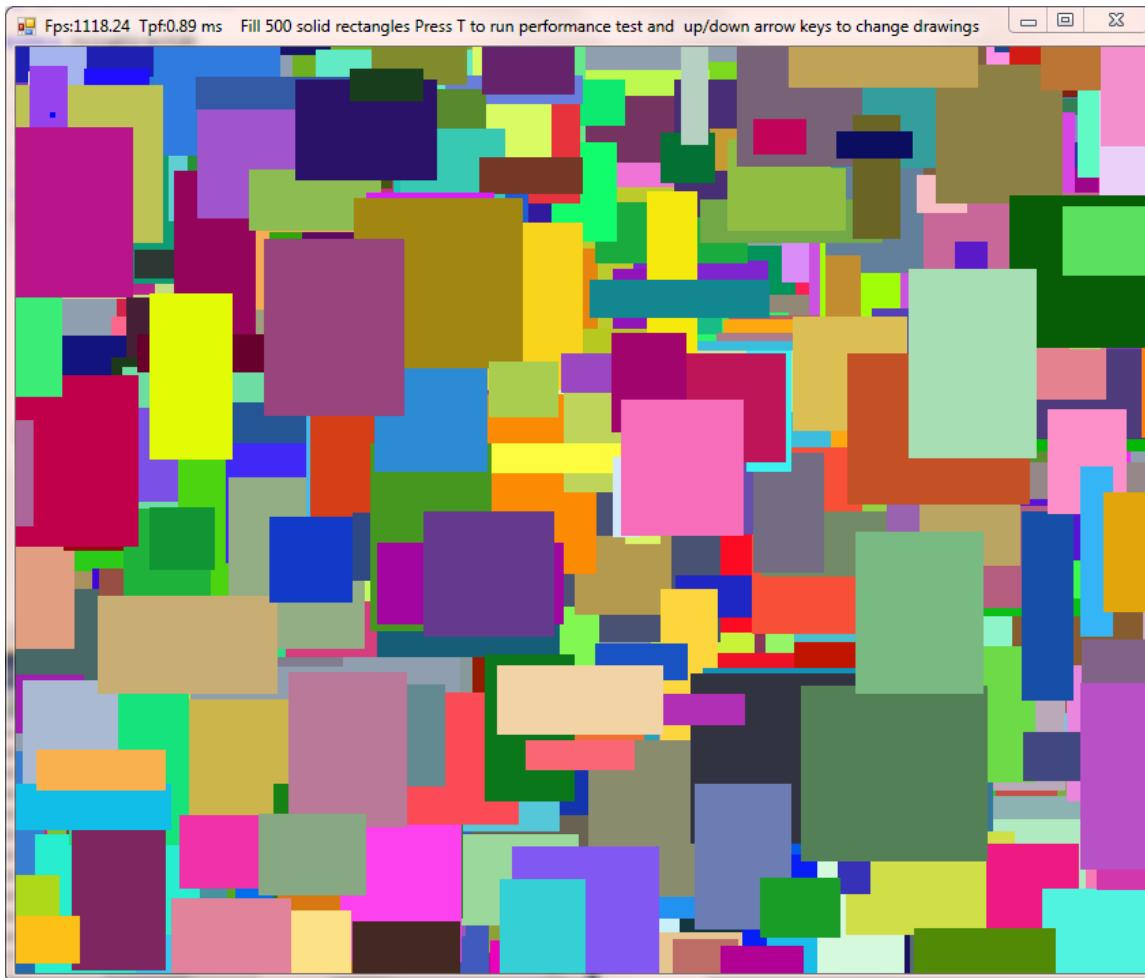
UsingDirect2D is a sample application that demonstrates how to use Direct2D and ATF classes that support Direct2D. It draws a sequence of graphics objects in a window. It does not use MEF.

ATF Features Demonstrated by UsingDirect2D

- Use ATF Direct2D classes, such as D2dHwndGraphics, D2dTextFormat and D2dBitmapBrush.
- Demonstrate ATF vector math classes, such as Matrix3x2F.

Run UsingDirect2D

1. Double-click the UsingDirect2D.exe in ATF\Samples\UsingDirect2D\bin\Release.
2. A dialog appears showing the first in a sequence of graphics drawings.



How to Use UsingDirect2D

UsingDirect2D draws a sequence of graphics in a window to demonstrate various Direct2D capabilities. Each drawing shows a particular type of graphics object. For instance, the first drawing in the sequence contains 500 rectangles with a gradient. The window title indicates what type of graphics objects were drawn. Press the down or up cursor keys to go to the next or previous in the sequence. Press the "t" key to run a performance test for each drawing.

Using Direct2D Modules

Modules perform these functions:

- Program.cs: Contains the Main program.
- Form1.cs: Draws a sequence of drawings using Direct2D and ATF helper classes.
- GdiCanvas.cs: Draws various graphics using GDI. This is currently unused.

ATF Using Direct2D Sample_j

(View in English 

説明

UsingDirect2D は Direct2D および Direct2D をサポートする ATF クラスの使用方法を示すサンプルアプリケーションで、ウィンドウに一連のグラフィックオブジェクトを描きます。UsingDirect2D は MEF を使用しません。

UsingDirect2D が示す ATF の機能

- D2dHwndGraphics、D2dTextFormat、および D2dBitmapBrush のような ATF Direct2D クラスの使用。
- Matrix3x2F のような ATF ベクトル演算クラスの使用。

UsingDirect2D の実行

1. ATF\Samples\UsingDirect2D\bin\Release にある UsingDirect2D.exe をダブルクリックします。
2. 一連のグラフィック描画の最初の描画を示すダイアログが表示されます。



UsingDirect2D の使用法

UsingDirect2D は ウィンドウに一連のグラフィックを描いて、Direct2D の様々な機能を具体的に示します。各描画には特定のタイプのグラフィックオブジェクトが描かれます。たとえばシーケンスの最初の描画には 500 の長方形が描かれています。ウィンドウのタイトルに、描画されているグラフィックオブジェクトの種類が示されます。↓キーまたは↑キーを押すと、次または前キーを押すと各描画のパフォーマンステストを実行します。

UsingDirect2D のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。
- Form1.cs: Direct2D および ATF ヘルパークラスを使用して一連のグラフィックを描きます。
- GdiCanvas.cs: GDI を使用して様々なグラフィックを描きますが、現在、この機能は使用されていません。

ATF Using Dom Sample

(日本語で表示 )

Description

The sample application UsingDom is a simple demo of basic DOM use. It illustrates loading a schema with a schema loader, creating a game using DomNodes then saving the game data using the schema loader. It has no UI, running in a command prompt window.

To see how this sample is programmed, see [Using Dom Programming Discussion](#).

ATF Features Demonstrated by UsingDom

- Deriving from XmlSchemaTypeLoader to customize a schema loader.
- Creating game data using either DomNodes or DomNodeAdapters.
- Saving application data with DomXmlWriter.

Run UsingDom

1. Double-click the UsingDom.exe in ATF\Samples\UsingDom\bin\Release.
2. A command prompt window appears briefly while the application is running.

How to Use UsingDom

There is no user interface. UsingDom automatically creates and saves the file game.xml XML file of game data.

UsingDom Modules

Modules perform these functions:

- Program.cs: Contains the Main program. This loads the schema and creates and saves the game data.
- GameSchemaLoader.cs: Derive from XmlSchemaTypeLoader to customize a schema loader.
- Various files in the Schemas folder: Game object accessors.

ATF Using Dom Sample_j

(View in English 

説明

サンプルアプリケーションの UsingDom は、DOM の基本的な使用法を簡単に示します。スキーマローダーを使用したスキーマのロード、DomNotes を使用したゲームの作成、そしてスキーマローダーを使用したゲームデータの保存を実行します。UI ではなく、コマンドプロンプトウィンドウ内で実行します。

UsingDom が示す ATF の機能

- XmlSchemaTypeLoader から派生し、スキーマローダーをカスタム化します。
- DomNodes または DomNodeAdapters を使用して、ゲームデータを作成します。
- DomXmlWriter を使用して、アプリケーションデータを保存します。

UsingDom の実行

1. ATF\Samples\UsingDom\bin\Release にある UsingDom.exe をダブルクリックします。
2. アプリケーションの実行中にコマンドプロンプトウィンドウが表示されます。アプリケーションはすぐに終了し、ウィンドウもすぐに閉じます。

UsingDom の使用法

ユーザインターフェースはありません。UsingDom は XML 形式のゲームデータファイルである game.xml ファイルを自動的に作成、保存します。

UsingDom のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。スキーマをロードし、ゲームデータを作成、保存します。
- GameSchemaLoader.cs: XmlSchemaTypeLoader から派生し、スキーマローダーをカスタム化します。
- Schemas フォルダ内の様々なファイル: ゲームオブジェクトのアクセサーです。

ATF Win Forms App Sample

(日本語で表示 

Description

WinFormsApp is a basic WinForms sample application. It illustrates how to compose a WinForms application with ATF components using MEF. It is a starting point for an application, such as an editor, though WinFormsApp does not offer any editing capabilities.

The WinFormsApp sample shares much code with the WpfApp sample, so the two provide a similar set of capabilities. In fact, the two applications differ by only a few files; the bulk of the application code is in common. This demonstrates the ease of developing an ATF application and converting it from WinForms to WPF or vice versa.

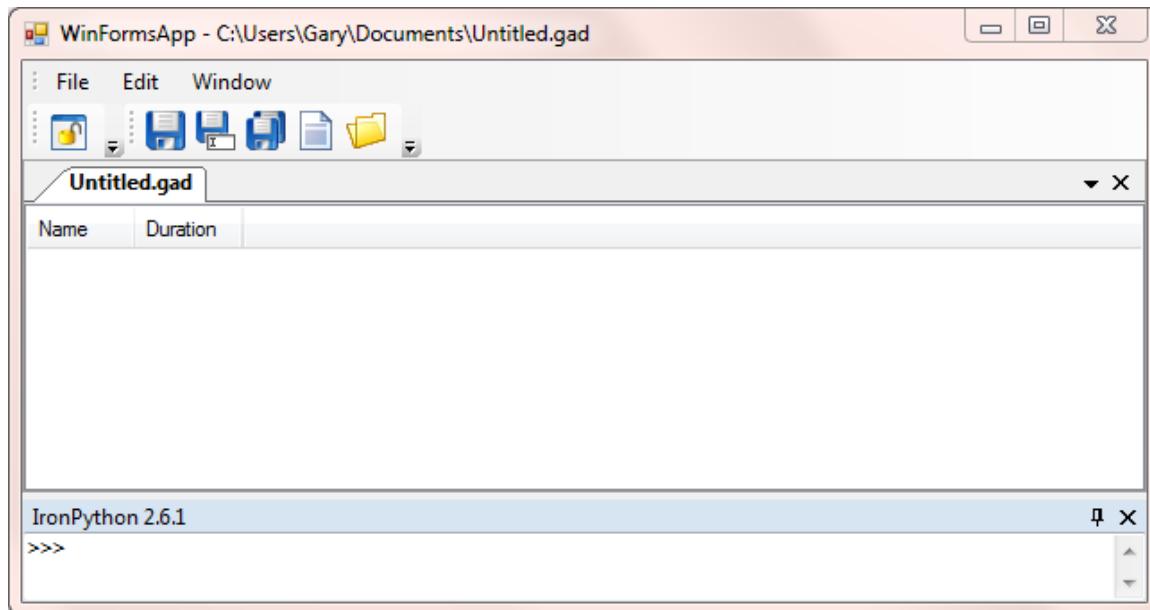
For details on programming in the sample, see [WinForms and WPF Apps Programming Discussion](#).

ATF Features Demonstrated by WinFormsApp

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including CommandService, SettingsService, ControlHostService and WindowLayoutService.
- Show loading a schema with SchemaLoader.

Run WinFormsApp

1. Double-click the WinFormsApp.exe in ATF\Samples\WinFormsApp\bin\Release.
2. A dialog appears with a list box for application data.



Menu and Toolbar Options

- File: create a new or open an existing Gui App Data (.gad) file, Save, Save as, Save all, Close and Exit WinFormsApp.
- Edit:
 - Keyboard Shortcuts: use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: use the Load and Save Settings window to save current WinFormsApp application settings or load application settings from a file.
 - Preferences: set application preferences, such as command icon size.
- Window:
 - Tile Horizontal: tile window panes horizontally.
 - Tile Vertical: tile window panes vertically.
 - Tile Overlapping: overlap window panes horizontally.
 - Layouts:
 - Save Layout As...: associate the current layout with a name.

- Manage Layouts...: show a list of layouts and manage the list.
- Lock/Unlock UI Layout: lock/unlock window pane layout.
- List of checked menu items; check to activate the corresponding control.

The toolbar offers a subset of these capabilities: Lock/Unlock UI Layout and File commands to save, save as, save all, create a new, or open existing Gui App Data (.gad) file.

How to Use WinFormsApp

WinFormsApp provides a schema with event, animation and other types; it also has the capability of loading a schema. WinFormsApp also has code to handle events and resources involving DomNodes. However, it does not provide editing capabilities for creating or changing application data.

WinFormsApp Modules

Modules perform these functions:

- Program.cs: Contains the Main program. It creates a TypeCatalog listing the ATF and internal classes used.
- Editors.cs: Implements IDocumentClient to open, show, save and close documents.
- SchemaLoader.cs: Loads the event schema, registers data extensions on the DOM types, annotates the types with display information and PropertyDescriptors.
- EventContext.cs and WinGuiCommonDataContext.cs: Provide a context for data.

ATF Win Forms App Sample_j

(View in English 

説明

WinFormsApp は、基本的な WinForms サンプルのアプリケーションです。MEF を使用して、ATF コンポーネントを備えた WinForms アプリケーションを作成する方法を示します。これはエディタのようなアプリケーションの開始点ですが、WinFormsApp では編集機能を提供していません。

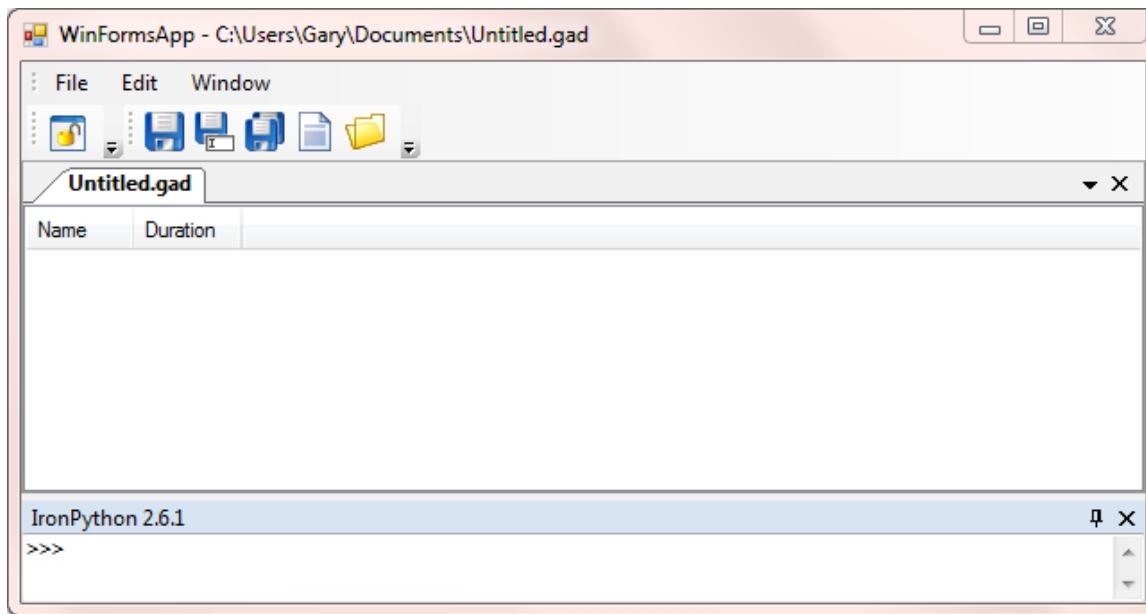
WinFormsApp サンプルのコードの多くは WpfApp サンプルと共にありますため、この 2 つが提供する機能のセットは似ています。実際、この 2 つのアプリケーションは 2、3 のファイルが異なるだけで、アプリケーションコードの大部分は共通です。このため、ATF アプリケーションを開発して WinForms と WPF 間で容易に変換できます。

WinFormsApp が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用して、アプリケーションをまとめる。
- CommandService、SettingsService、ControlHostService、および WindowsLayoutService を含むアプリケーションシェルフレームワークの使用。
- SchemaLoader を使用したスキーマのロード。

WinFormsApp の実行

1. ATF\Samples\WinFormsApp\bin\Release にある WinFormsApp.exe をダブルクリックします。
2. アプリケーションデータのリストボックスを含むダイアログが表示されます。



メニューおよびツールバー操作

- [File] (ファイル): [New Gui App Data] (.gad ファイルの新規作成)、[Open Gui App Data] (既存の .gad ファイルを開く)、[Save] (保存)、[Save As] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (閉じる) および [Exit] (終了)。
- [Edit] (編集)
 - [Keyboard Shortcuts] (キーボードショートカット): [Customize Keyboard Shortcuts] ウィンドウを使用して、キーボードショートカットを設定します。
- [Load or Save Settings]: このウィンドウを使用して、現在の WinFormsApp の設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
 - [Preferences] (設定): コマンドアイコンのサイズなど、アプリケーションを設定します。
- [Window] (ウィンドウ):
 - [Tile Horizontal] (左右に並べて表示): ウィンドウペインを水平に並べて表示します。
 - [Tile Vertical] (上下に並べて表示): ウィンドウペインを上下に並べて表示します。
 - [Tile Overlapping] (重ねて表示): ウィンドウペインを重ねて表示します。
 - [Layouts] (レイアウト):
 - [Save Layout As...] (名前を付けてレイアウトを保存): 現在のレイアウトに名前を付けて保存します。

- [Manage Layouts...] (レイアウトの管理): レイアウトのリストを表示し、リストを管理します。
- [Lock UI Layout] および [Unlock UI Layout]: ウィンドウペインのレイアウトのロックとロックの解除を切り替えます。
- チェックボックス付きのメニューアイテムのリスト:
コントロールをクリックするとチェックマークが付き、アクティブ化します。

ツールバーでは、レイアウトのロックおよびロック解除、[File]

コマンドの保存、名前を付けて保存、すべて保存、新規作成、および開くのボタンを選択的に使用できます。

WinFormsApp の使用法

WinFormsApp はイベント、アニメーション、その他の型を含むスキーマを提供します。また、スキーマをロードする機能もあります。

WinFormsApp には、DomNode に関連したイベントおよびリソースを処理するコードもありますが、

アプリケーションデータの作成および変更のための編集機能はありません。

WinFormsApp のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- Editor.cs: IDocumentClient を実装して、ドキュメントを開き、表示し、保存し、閉じることができます。
- SchemaLoader.cs: イベントスキーマをロードし、データ拡張を DOM 型に登録し、型に表示情報と PropertyDescriptors の注釈を付けます。
- EventContext.cs および WinGuiCommonDataContext.cs: データのコンテキストを提供します。

ATF Wpf App Sample

(日本語で表示 )

Description

WpfApp is a basic WPF sample application. It illustrates how to compose a WPF application with ATF components using MEF. It is a starting point for an application, such as an editor, though WpfApp does not offer any editing capabilities.

The WpfApp sample shares much code with the WinFormsApp sample, so the two provide a similar set of capabilities. In fact, the two applications differ by only a few files; the bulk of the application code is in common. This demonstrates the ease of developing an ATF application and converting it from WinForms to WPF or vice versa.

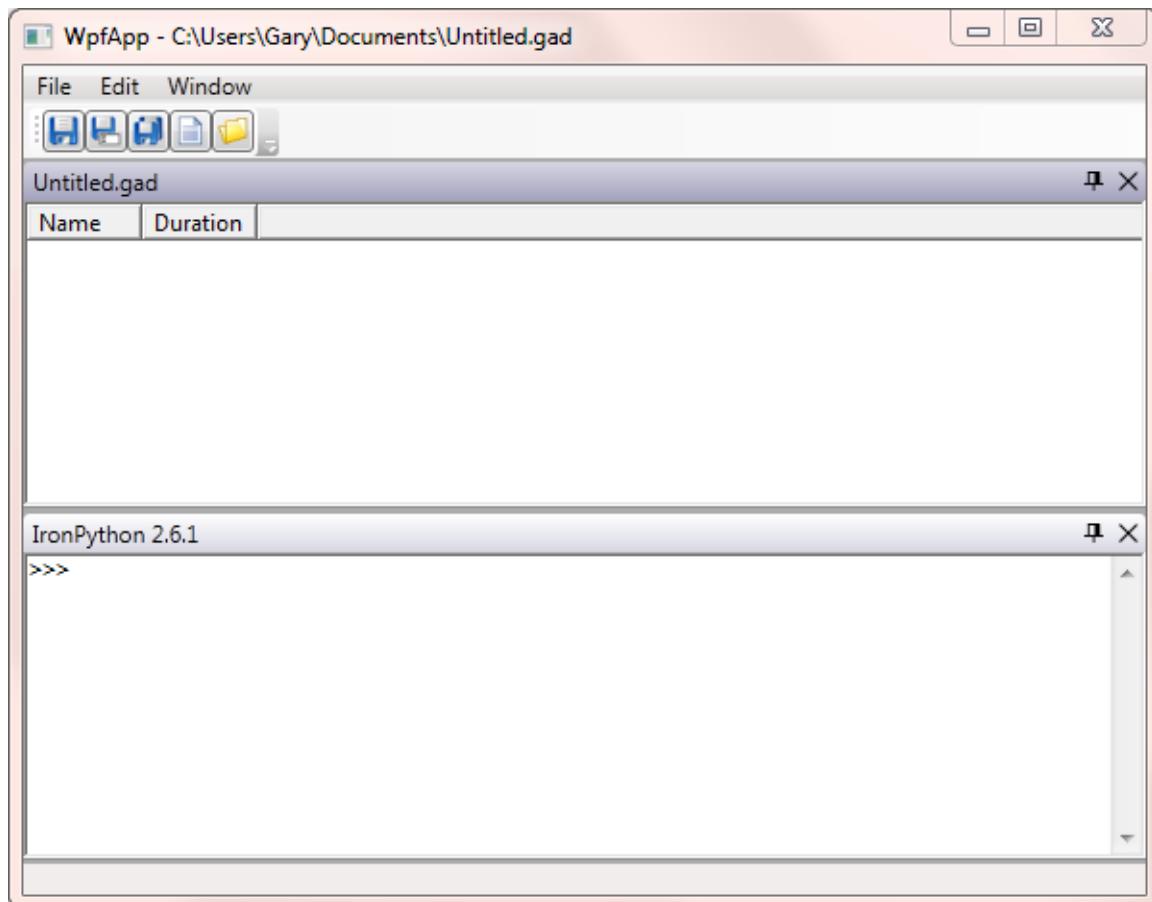
For details on programming in the sample, see [WinForms and WPF Apps Programming Discussion](#).

ATF Features Demonstrated by WpfApp

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including `CommandService`, `SettingsService`, `ControlHostService` and `WindowLayoutService`.
- Show loading a schema with `SchemaLoader`.

Run WpfApp

1. Double-click the `WpfApp.exe` in `ATF\Samples\WpfApp\bin\Release`.
2. A dialog appears with a list box for application data.



Menu and Toolbar Options

- File: create a new or open an existing Gui App Data (.gad) file, Save, Save as, Save all, Close and Exit WpfApp.

- Edit:
 - Preferences: set application preferences, such as auto-loading a new document.
 - Load or Save Settings: use the Load and Save Settings window to save current WpfApp application settings or load application settings from a file.
- Window:
 - Layouts:
 - Save Layout As...: associate the current layout with a name.
 - Manage Layouts...: show a list of layouts and manage the list.
 - List of checked menu items; check to activate the corresponding control.

The toolbar offers a subset of these capabilities: File commands to save, save as, save all, create a new, or open existing Gui App Data (.gad) file.

How to Use WpfApp

WpfApp provides a schema with event, animation and other types; it also has the capability of loading a schema. WpfApp also has code to handle events and resources involving `DomNode` objects. However, it does not provide editing capabilities for creating or changing application data.

WpfApp Modules

Modules perform these functions:

- `App.xaml.cs`: Contains `App` class deriving from `AtfApp`. It creates a `TypeCatalog` listing the ATF and internal classes used.
- `Editors.cs`: Implements `IDocumentClient` to open, show, save and close documents.
- `SchemaLoader.cs`: Loads the event schema, registers data extensions on the DOM types, annotates the types with display information and `PropertyDescriptor` objects.
- `EventContext.cs` and `WinGuiCommonDataContext.cs`: Provide a context for data.

ATF Wpf App Sample_j

(View in English 

説明

WpfApp は、基本的な WPF サンプルのアプリケーションです。MEF を使用して、ATF コンポーネントを備えた WPF アプリケーションを作成する方法を示します。これはエディタのようなアプリケーションの開始点ですが、WpfApp では編集機能を提供していません。

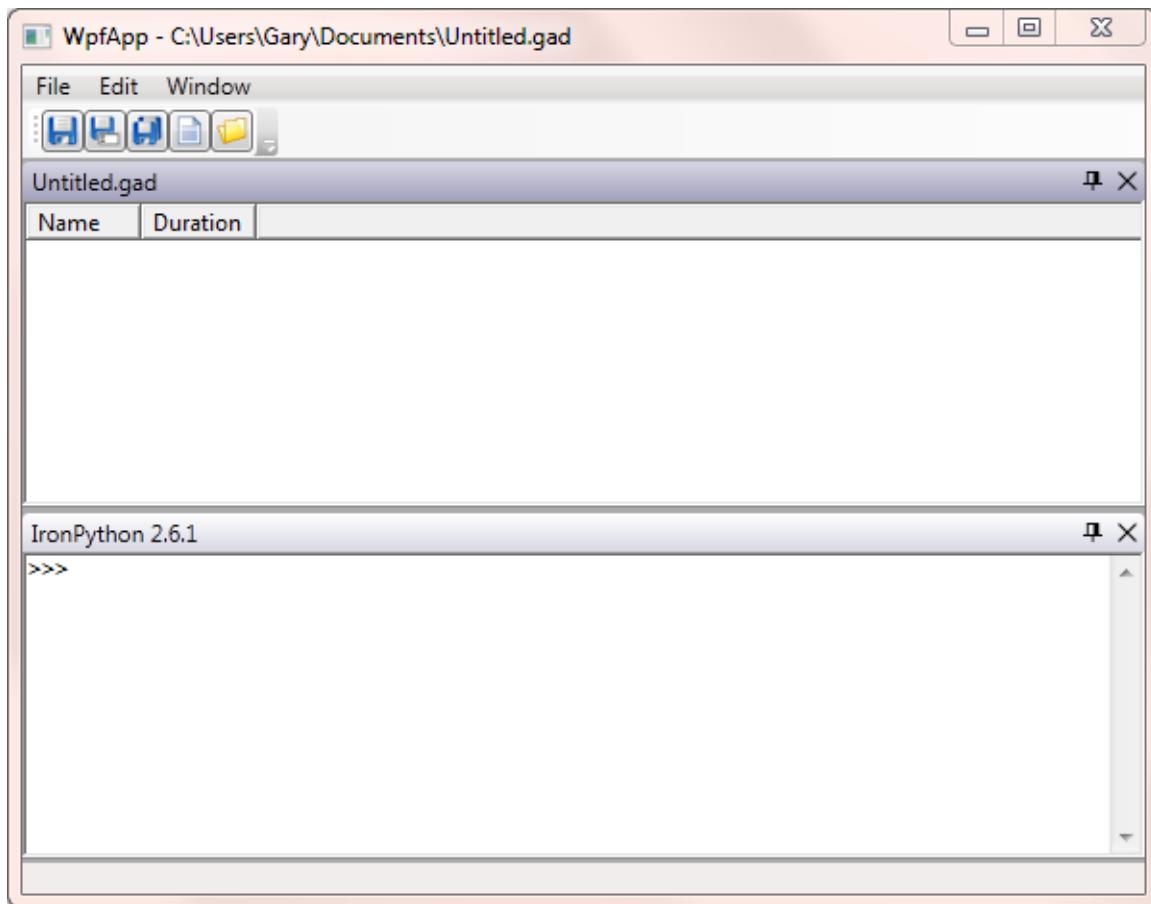
WpfApp サンプルのコードの多くは WinFormsApp サンプルと共にありますため、この 2 つが提供する機能のセットは似ています。実際、この 2 つのアプリケーションは 2、3 のファイルが異なるだけで、アプリケーションコードの大部分は共通です。このため、ATF アプリケーションを開発して WinForms と WPF 間で容易に変換できます。

WpfApp が示す ATF の機能

- MEF (Managed Extensibility Framework) を使用して、アプリケーションをまとめる。
- CommandService、SettingsService、ControlHostService、および WindowsLayoutService を含むアプリケーションシェルフレームワークの使用。
- SchemaLoader を使用したスキーマのロード。

WpfApp の実行

1. ATF\Samples\WpfApp\bin\Release にある WpfApp.exe をダブルクリックします。
2. アプリケーションデータのリストボックスを含むダイアログが表示されます。



メニューおよびツールバー オプション

- [File] (ファイル): [New Gui App Data] (.gad ファイルの新規作成)、[Open Gui App Data] (既存の .gad ファイルを開く)、[Save] (保存)、[Save As] (名前を付けて保存)、[Save All] (すべて保存)、[Close] (閉じる) および [Exit] (終了)。
- [Edit] (編集)
 - [Preferences] (設定): 新規ドキュメントの自動読み込みなど、アプリケーションの詳細を設定します。

- [Load or Save Settings]: このウインドウを使用して、現在の WpfApp のアプリケーション設定を保存するか、またはファイルからアプリケーション設定を読み込みます。
- [Window] (ウィンドウ):
 - [Layouts] (レイアウト):
 - [Save Layout As...] (名前を付けてレイアウトを保存): 現在のレイアウトに名前を付けて保存します。
 - [Manage Layouts...] (レイアウトの管理): レイアウトのリストを表示し、リストを管理します。
 - チェックボックス付きのメニューアイテムのリスト:
コントロールをクリックするとチェックマークが付き、アクティブ化します。

ツールバーでは、[File] コマンドの保存、名前を付けて保存、すべて保存、新規作成、および開くのボタンを選択的に使用できます。

WpfApp の使用法

WpfApp はイベント、アニメーション、その他の型を含むスキーマを提供します。また、スキーマをロードする機能もあります。 WpfApp には、DomNode に関連したイベントおよびリソースを処理するコードもありますが、アプリケーションデータの作成および変更のための編集機能はありません。

WpfApp のモジュール

モジュールには次のような機能があります。

- Program.cs: Main プログラムを含みます。このプログラムが、使用されている ATF クラスおよび内部クラスをリストした TypeCatalog を作成します。
- Editor.cs: IDocumentClient を実装して、ドキュメントを開き、表示し、保存し、閉じることができるようになります。
- SchemaLoader.cs: イベントスキーマをロードし、データ拡張を DOM 型に登録し、型に表示情報と PropertyDescriptors の注釈を付けます。
- EventContext.cs および WinGuiCommonDataContext.cs: データのコンテキストを提供します。

ATF Model Viewer Sample

(日本語で表示 )

Description

ModelViewer shows how to use ATF classes to load ATGI and Collada models and to render them using OpenGL.

For details on how this sample is programmed, see [Model Viewer Programming Discussion](#).

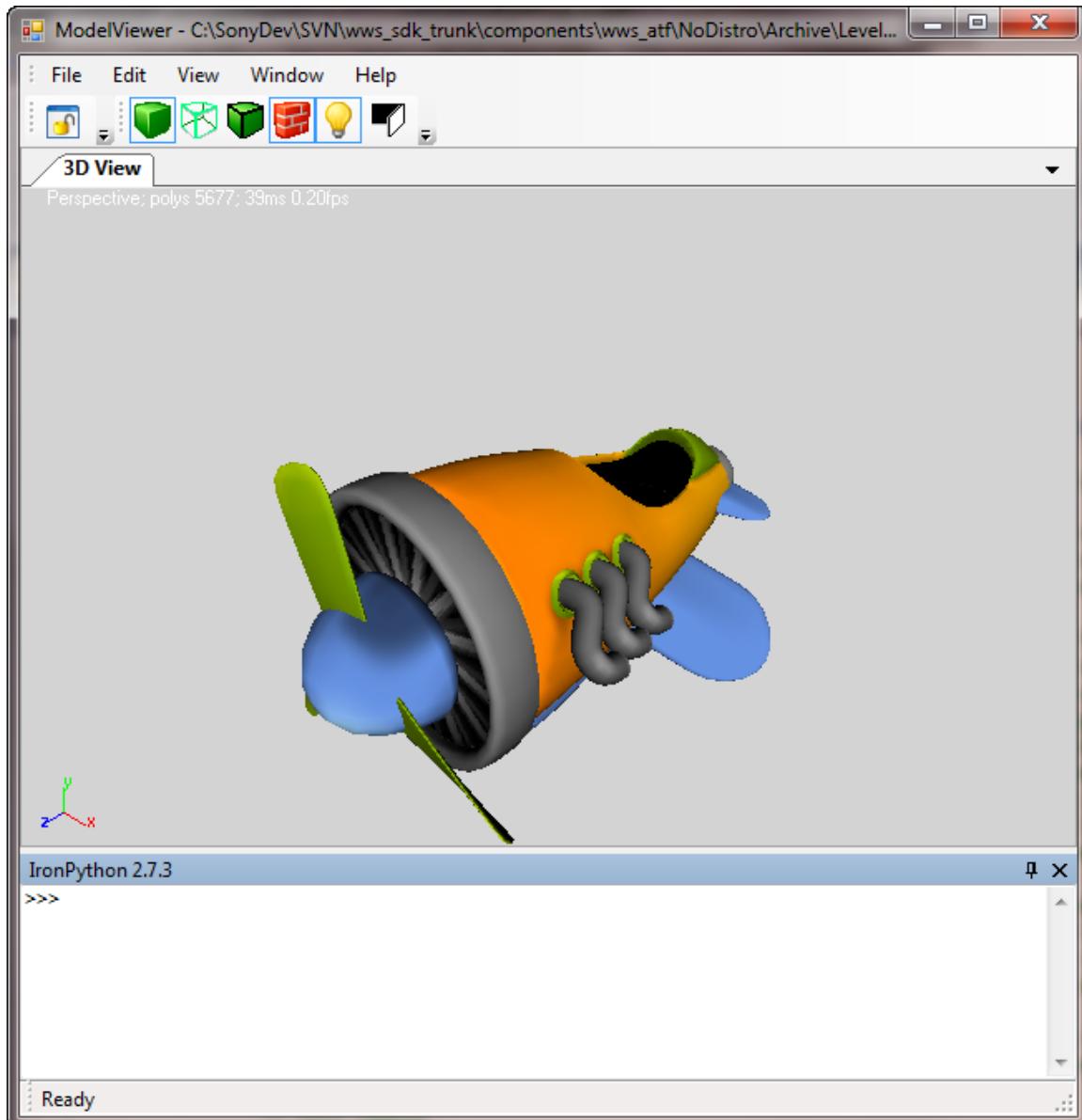
ATF Features Demonstrated by ModelViewer

- Use of Managed Extensibility Framework (MEF) to put applications together.
- Use of the application shell framework, including `CommandService`, `SettingsService`, and `ControlHostService`.
- Use of components to handle files, such as `FileDialogService`, `RecentDocumentCommands`, and `StandardFileExitCommand`.
- Using resource resolvers to load model files.
- Using a document and a client to handle documents of a certain type.
- Using adaptation and DOM adapters to handle DOM nodes in a variety of ways.
- Using a `DesignControl` canvas to display a graphic model.
- Render ATGI and Collada documents using OpenGL.

Run ModelViewer

To run the sample, double-click `ModelViewer.exe` in `ATF\Samples\ModelViewer\bin\Release`.

A dialog appears with an empty canvas. You can open a model file to view it in the canvas.



Menu Options

- File
 - Open 3D Model...: Open a 3D model file.
 - Recent Files: Submenu with recent files you can open.
 - Exit: Exit ModelViewer.
- Edit:
 - Keyboard Shortcuts: Use the Customize Keyboard Shortcuts window to set up keyboard shortcuts.
 - Load or Save Settings: Use the Load and Save Settings window to save current ModelViewer application settings or load application settings from a file.
 - Preferences: Set application preferences, such as command icon size.
- View
 - Fit: Fit the model object in the window. Keyboard shortcut: F key.
 - Smooth: View the model fully rendered.
 - Wireframe: View a wireframe of the model.
 - Outlined: View the model fully rendered with a wireframe.
 - Textured: Toggle viewing the model with textures. Keyboard shortcut: T key.
 - Lighting: Toggle viewing the model with shading. Keyboard shortcut: L key.
 - Backface: Toggle rendering backfaces in the model. Keyboard shortcut: B key.
 - CycleRenderModes: Cycle between the Smooth, Wireframe, and Outlined modes. Keyboard shortcut: space bar.
- Window:
 - Tile Horizontal: Tile the windows horizontally; does nothing.
 - Tile Vertical: Tile the windows vertically; does nothing.
 - Tile Overlapping: Overlap the windows horizontally; does nothing.
 - List of checked menu items for windows; check to display the corresponding window.
 - Lock UI Layout: Lock the UI so that windows can't be moved.

- Help
 - About: Display dialog with information about ModelViewer.

Tool bar

The tool buttons invoke functions also available from menu items:

- Lock UI Windows
- Smooth shading: Same as View > Smooth.
- Wireframe rendering: Same as View > Wireframe.
- Smooth shading with wireframe outlining: Same as View > Outlined.
- Textured rendering: Same as View > Textured.
- Lighting: Same as View > Lighting.
- Render backfaces: Same as View > Backface. This mode is only apparent in Wireframe mode.

How to Use ModelViewer

Open an ATGI (.atgi extension) or Collada (.dae) model file and it displays in the "3D View" window tab. Only one model displays at a time.

Click the tool buttons or menu items or use the keyboard shortcuts to change the manner of rendering.

You can choose only one of these renderings:

- Smooth: View the model fully rendered.
- Wireframe: View a wireframe of the model.
- Outlined: View the model fully rendered with a wireframe.

Pressing the space bar cycles between these modes.

You can toggle these rendering modes:

- Textured. Keyboard shortcut: T key.
- Lighting. Keyboard shortcut: L key.
- Backface. Keyboard shortcut: B key.

Rotating the mouse wheel zooms the object. Selecting View > Fit or pressing "F" changes the zoom back to fit the model in the window.

ModelViewer Modules

Modules perform these functions:

- Program.cs: Contains the Main() function. It creates a MEF TypeCatalog listing the ATF and internal components used.
 - ModelDocument.cs: Defines the ModelDocument class for a document of a model file.
 - ModelViewer.cs: Specifies the ModelViewer component, which is the document client for a ModelDocument.
 - RenderView.cs: RenderView component that registers a DesignControl control to display a 3D scene from a model document.
 - RenderCommands.cs: RenderCommands component provides user commands related to the RenderView component to change rendering mode.
 - RenderPrimitives.cs: RenderPrimitives class DOM adapter that does the work of rendering the model, using OpenGL.
 - RenderTransform.cs: RenderTransform class DOM adapter that applies the appropriate transform for rendering objects so they appear in the proper place in the model.
-

ATF Model Viewer Sample_j

([View in English](#) 

説明

ModelViewer は、ATGI および Collada モデルを、ATF クラスを使用して読み込み、OpenGL を使用してレンダリングする方法を示します。

ATF Documentation

(日本語で表示 )



New

The newly revised and updated [ATF Curve Editor User and Programming Guide](#) provides information about using the Curve Editor, including programming this component to add curves.



New

All PDF documents now have a search index that allows you to easily search them. For details, see [PDF Searching](#).

You can download all ATF documentation here, and it is also available as part of both ATF release packages in the ATF/Docs directory. See the [Getting Started with ATF](#) wiki page for details and steps to get to know ATF.

Document	Updated in Release	Version Date	Description
Getting Started with ATF	3.6	31July13	Provides an introduction and a description of the most important components of the Authoring Tools Framework
ATF Tech Talk PowerPoint		14May13	A presentation that provides an introduction to Authoring Tools Framework
ATF Programmer's Guide	3.6	In progress	Guides programmers in writing ATF-based applications
ATF Programmer's Guide: Document Object Model (DOM)	3.6	31July13	A comprehensive guide to the ATF Document Object Model (DOM)
ATF Curve Editor User and Programming Guide	3.6	18Oct13	A guide to using the ATF Curve Editor component and programming it
ATF Refactor User Guide	3.3	30Jul12	Using the ATF Refactor auto-update tool, for migrating your application from one version of ATF to the next

PDF Searching

All PDF documents have a search index that allows you to search all of them at once. The search hits list all documents found and each instance of the search string, allowing you to go immediately to the page with the found text highlighted.

First, download the [search index](#) and unzip it into the folder with the PDF documents. To start searching, select Edit > Advanced Search in Reader or Acrobat. For more information, see [Advanced Searching PDF Documents](#).

ATF Documentation_j

(View in English 



変更点

このページは、バージョン番号および日付を表示するようにフォーマットが変更されました。

ドキュメント

ATF ドキュメントはすべてここからダウンロード可能です。また ATF リリースパッケージにも含まれており、ATF/Docs ディレクトリに保管されています。詳細は、[ATF ご使用の前に](#)を参照してください。

以下のドキュメントが、ATF/Docs ディレクトリに含まれています。

ドキュメント	日本語版 リリース番号	日本語版 リリース日	英語版 リリース番号	英語版 リリース日	説明
ATF Getting Started Guide	-	-	3.5	2013/05/14	オーサリングツールフレームワークの紹介と、重要なコンポーネン
ATF Tech Talk PowerPoint	-	-		2013/05/14	オーサリングツールフレームワークを紹介するプレゼン
ATF の概要	3.0	2012/07/08	3.3	2012/04/11	Authoring Tools Framework の最も重要なコンポーネントの説明
ATF 3.0 プログラマーズガイド: ドキュメントオブジェクトモデル (DOM)	3.1	2011/12/14	3.3	2012/02/29	ATF Document Object Model (DOM) の総合ガイド
ATF Refactor ユーザーズガイド	3.0	2012/05/08	3.3	2012/07/30	ATF のあるバージョンから次のバージョンへの、ATF Refactor 自動更新ツールを使用したアプリケーションの移行

ATF Getting Started

(日本語で表示 )

Start by reading the [ATF 3 Overview](#) to get a high-level understanding of the major parts of the Authoring Tools Framework (ATF) and how they fit together. Additionally, it may be helpful to look at:

- Our available documentation PDF files: [ATF Documentation](#).
- A description of our sample projects: [ATF Code Samples](#).
- The list of key technologies and the sample apps that demonstrate them: [ATF Technology and Sample App Matrix](#).

Source Code

The current ATF release is on our Alfresco server as a ZIP package that includes documentation and sample application code.

 [Download Latest ATF Packages](#)

Directory Contents

The ATF release package contains the following subdirectories:

- DevTools: Tools to help our clients set-up, port, and manage their ATF projects.
- Docs: [ATF Documentation](#) including programmer and reference documentation.
- Framework: Reusable .Net components, written in C#, for building your applications. Clients should avoid modifying this code.
- Legacy: ATF 2.0 framework and tutorials. As much as possible, ATF 2 types derive from or make use of ATF 3 types, to avoid duplicate code.
- Samples: Code for sample applications that demonstrate various aspects of the ATF and its features. Clients are encouraged to start with a sample app and then modify it.
- Test: Unit tests and Everything.sln, which is our one solution file that references all source code in an ATF distribution.
- ThirdParty: pre-compiled managed and unmanaged dlls that are licensed to be used by ATF.

Building ATF

The ATF source is organized into a set of Visual Studio solution and project files. The main solution files are:

- \Samples\Samples.sln – includes all of the ATF 3 samples and framework.
- \Test\Everything.sln – includes ATF 3 and ATF 2 framework and samples, as well as unit tests and development tools.

Joining the ATF Community

Join the ATF team on our [Forum](#).

ATF Getting Started_j

(View in English 

Authoring Tools Framework (ATF) ご使用の前に

まずははじめに [ATF 3 の概要](#)を読み、Authoring Tools Framework (ATF) の主要部分と、各部分がどのように整合するのかの概要を理解してください。また、次のページも参考になります。

- PDF 形式のドキュメント
- サンプルプロジェクトの説明
- 主要なテクノロジーと、その例を示すサンプルアプリケーションの一覧表

ソースコード

現行の ATF リリースは、ドキュメントとサンプルアプリケーションのコードを含む ZIP パッケージとして Alfresco サーバ上で提供されています。

➡ [最新 ATF パッケージのダウンロード](#)

同じ ATF リリースが、[WWS SDK SHIP](#) ページからもダウンロードできます。

ディレクトリの内容

ATF リリースパッケージには、以下のサブディレクトリが含まれます。

- DevTools: セットアップ、移植、ATF プロジェクトの管理に役立つツール。
- Docs: プログラマのドキュメントやリファレンス資料を含む [ATF ドキュメント](#)。
- Framework: アプリケーション作成のための、C# で書かれ再利用可能な .Net コンポーネント。
このコードの変更は避けてください。
- Legacy: ATF 2.0 フレームワークとチュートリアル。ATF 2 タイプは可能な限り ATF 3 タイプから派生させるか、ATF 3 タイプを利用し、重複コードを避けています。
- Samples: ATF の多様な面とその機能を示すサンプルアプリケーションのコード。
サンプルアプリケーションからはじめてそれを修正することを推奨します。
- Test: ユニットテストと Everything.sln。ATF 配布に含まれるすべてのソースコードを参照するソリューションファイルです。
- ThirdParty: ATF で使用できるようにライセンスを受けているコンパイル済みの管理 DLL と非管理 DLL。

ATF のビルド

ATF ソースは、Visual Studio のソリューションファイルとプロジェクトファイルのセットで構成されています。

主要なソリューションファイルは、以下のとおりです。

- \Samples\Samples.sln – ATF 3 のサンプルとフレームワークをすべて含む。
- \Test\Everything.sln – ATF 3 と ATF 2 のフレームワークとサンプル、およびユニットテストと開発ツールを含む。

ATF コミュニティへの参加

フォーラムで、ATF チームに参加してください。

ATF Technology and Sample App Matrix

(日本語で表示)

Live Connect													
Skinning Service													
Target Service													
Open Sound Control													

* Note that [Diagram Editor](#) contains most of [Circuit Editor](#), [FSM Editor](#), and [Statechart Editor](#), so the latter two are not listed. [Win Forms App](#) is not listed either, because it is a basic WinForms application, which is illustrated by all the other samples except [Using Dom](#).

ATF Technology and Sample App Matrix_j

(View in English 

テクノロジー\サンプル	Circuit Editor	Code Editor	Diagram Editor*	DOM Tree Editor	File Explorer	ModelViewer	Property Editing	Simple DOM Editor	Simple DOM No XML	Target Manager
MEF (Managed Extensibility Framework)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DOM (Document Object Model)	✓		✓	✓		✓		✓	✓	✓
XML, スキーマ	✓		✓	✓				✓		
GUI、ドッキング、コマンド	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
WPF (Windows Presentation Foundation)										
ドキュメント	✓	✓	✓	✓		✓		✓	✓	✓
プロパティ編集	✓		✓	✓			✓	✓	✓	✓
子プロパティ、サブカテゴリ、、プロパティコレクション							✓			
回路、FSM、ステートチャート	✓		✓							
ツリーコントロール	✓		✓	✓	✓			✓	✓	
ツリーリストビュー										
タイムライン										
曲線編集					✓					
Direct2D	✓		✓							
3D、OpenGL							✓			
検索および置換								✓		
Python スクリプティング	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Window Layout Service	✓	✓	✓	✓						
フィルターされたツリービュー							✓			
Live Connect										
Skinning Service										
Target Service										✓
Open Sound Control										

* FSM Editor、および Statechart Editor は、Circuit Editor 同様、その大部分が Diagram Editor に

含まれているため、リストには含まれていません。Win Forms App もリストに含まれていませんが、これは、Win Forms App が、Using Dom 以外のすべてのサンプルに含まれる基本的な WinForms アプリケーションであるためです。

ATF Programmer's Guide



The ATF Programmer's Guide assists programmers in writing ATF applications. Its topics include:

- How to create an ATF application from an ATF sample.
- Using and creating Managed Extensibility Framework (MEF) components in ATF to quickly add capabilities to your application.
- Learning about key ATF concepts, such as adaptation and contexts.
- Seeing how basic tasks, such as adding commands and controls, are done in ATF.

Start learning how to use ATF at [Getting Started with ATF](#).

This Guide does not directly discuss the ATF Document Object Model (DOM), although some of the concepts discussed here, such as adaptation, apply to the DOM. For details on using the ATF DOM, download the new revision of the ATF Programmer's Guide: Document Object Model (DOM) from [ATF Documentation](#).

Feedback

The ATF Programmer's Guide is a work in progress. Are there ATF topics you'd like to see included or topics you'd like more information on? Send your feedback to the Owner on [Authoring Tools Framework](#). You can also raise issues to the entire ATF team with the [ATF Forum](#).

Preface

- [ATF Programmer's Guide Copyright](#)
- [What the ATF Programmer's Guide Contains](#)
- [ATF Documentation](#)
- [ATF Resources](#)
- [ATF Programmer's Guide Typographical Conventions](#)

Getting Started with ATF

- [ATF 3 Overview](#)
- [Getting Started with ATF](#), downloadable at [ATF Documentation](#)
- [ATF Code Samples](#)
- [ATF Technology and Sample App Matrix](#)
- [ATF Programmer's Guide: Document Object Model \(DOM\)](#), downloadable at [ATF Documentation](#)
- [MEF with ATF](#)
- [ATF Application Basics and Services](#)
- [ATF Glossary](#)
- [Download latest ATF packages](#)
- [Creating an Application from an ATF Sample](#)
- [ATF Code Samples Discussions](#)
- [Join the ATF Forum](#)

ATF Structure

- [ATF Frameworks](#)

- ATF Namespaces
- ATF Assemblies
- ATF Functional Areas

Creating an Application from an ATF Sample

- Choose ATF Sample to Base Application On
- Copy ATF Sample and Run It
- Start Customizing Your Application Software
- Define Your Application's New Data Model
- Modify Your Application's Existing Software
- Add New Code and Parts to Your Application
- Build, Run, and Debug Your Application

MEF with ATF

- What is MEF?
- How MEF is Used in ATF
- Initializing Components
- Using ATF Components
- Creating MEF Components
- Important ATF Components
- Finding ATF Components

ATF Application Basics and Services

- WinForms Application
- WPF Application
- ControlHostService Component
- CommandService Component
- SettingsService Component
- StatusService Component
- Other ATF Services

Controls in ATF

- Using Controls in ATF
- ControllInfo and ControlDef Classes
- ATF Control Groups
- Registering Controls
- Creating Control Clients
- Using WinForms Controls in WPF
- Adaptable Controls
- ATF Custom Controls
- ATF Custom Dialogs

Commands in ATF

- Using Commands in ATF
- CommandInfo and CommandDef Classes
- ATF Command Groups
- Using Standard Command Components
- Registering Menus and Commands
- Creating Command Clients
- Using WinForms Commands in WPF
- Using Context Menus

Adaptation in ATF

- What is Adaptation?
- General Adaptation Interfaces
- General Adaptation Classes
- Control Adapters
- Other Adaptation Classes
- Adapting to All Available Interfaces

ATF Contexts

- What is a Context?
- Context Registry
- Context Interfaces
- Context Classes
- Context Related Classes
- Implementing a Context Interface

Instancing In ATF

- What is Instancing?
- IInstancingContext and IHierarchicalInsertionContext Interfaces
- StandardEditCommands and Instancing
- Drag and Drop and Instancing
- Implementing Instancing

Documents in ATF

- What is a Document in ATF?
- Document Registry and Services
- Document Handling Components
- Implementing a Document and Its Client

Property Editing in ATF

- Using Properties in ATF
- Selection Property Editing Context
- Property Descriptors
- Value Editors
- Value Editors and Value Editing Controls
- Value Converters
- Property Editor Components
- Implementing a Property Editor

Graphs in ATF

- What is a Graph in ATF?
- Graph Data Model
- Types of Graphs
- ATF Graph Interfaces
- General Graph Support
- Circuit Graph Support
- Statechart Graph Support

Timelines in ATF

- What is a Timeline in ATF?
- Timeline Interfaces

- WinForms Timeline Renderers, Controls, and Manipulators
- General Timeline Classes

Development, Debugging, and Testing

- DOM Debugging in Visual Studio

ATF Code Samples Discussions

- Circuit Editor Programming Discussion
- Code Editor Programming Discussion
- Diagram Editor Programming Discussion
- DOM Tree Editor Programming Discussion
- File Explorer Programming Discussion
- FSM Editor Programming Discussion
- Model Viewer Programming Discussion
- Property Editing Programming Discussion
- Simple DOM Editor Programming Discussion
- Simple DOM No XML Editor Programming Discussion
- State Chart Editor Programming Discussion
- Target Manager Programming Discussion
- Timeline Editor Programming Discussion
- Tree List Editor Programming Discussion
- Using Dom Programming Discussion
- WinForms and WPF Apps Programming Discussion

ATF Glossary

ATF Glossary

ATF Programmer's Guide Preface

This preface describes this Guide and related documents in the following sections:

- [ATF Programmer's Guide Copyright](#): Sony copyrights and trademarks.
 - [What the ATF Programmer's Guide Contains](#): Description of each section on main page.
 - [ATF Documentation](#): Download all ATF documentation described here.
 - [ATF Resources](#): Description of the many resources available for using ATF.
 - [ATF Programmer's Guide Typographical Conventions](#): Typographical conventions of the pages.
-

ATF Programmer's Guide Copyright

Copyright

© Copyright 2013, Sony Computer Entertainment America LLC

Material contained in this document may not be copied, reproduced, reduced to any electronic medium or machine readable form or otherwise duplicated and the information herein may not be used, disseminated or otherwise disclosed, except with the prior written consent of an authorized representative of Sony Computer Entertainment America, Inc.

Sony is a registered trademark of Sony Corporation.

All other trademarks are the properties of their respective owners.

Topics in this section

Links on this page to other topics

No links

What the ATF Programmer's Guide Contains

This guide shows you how to use the Authoring Tools Framework (ATF) to develop rich Windows client applications and game-related tools.

- [Preface](#). Information about this document, such as contents, documentation, available resources, and typographical conventions.
- [ATF Structure](#). Structure of ATF, guiding you to find what you need to develop specific items in ATF.
- [Creating an Application from an ATF Sample](#). Starting with an ATF sample, shows how to develop a new application.
- [MEF with ATF](#). Introduction to the Managed Extensibility Framework (MEF), and how to use existing MEF components in ATF and develop your own components.
- [ATF Application Basics and Services](#). Information on basic structure of an ATF-based application, for both WinForms and ATF, including discussing the Application Shell Framework.
- [Controls in ATF](#). How to use controls in ATF, including the `ControlHostService` component, and a survey of ATF custom controls and dialogs.
- [Commands in ATF](#). Adding custom commands for application menus, buttons, and other controls, including context menus.
- [Adaptation in ATF](#). How to support different kinds of data models and managed data stores, such as a DOM or other CLR objects, by converting objects to other types.
- [ATF Contexts](#). Using the variety of ATF contexts, which are logical views of application data presented to a user for viewing or editing.
- [Instancing In ATF](#). Using the Instancing Framework, which works with object instances that can be edited, that is, copied, inserted, or deleted.
- [Documents in ATF](#). Using components to track documents that a user is working on.
- [Property Editing in ATF](#). How to edit properties of application data, setting up property descriptors, value editors, editing controls, and converters, and using property editing components.
- [ATF Code Samples Discussions](#). Overview of ATF samples, including their programming.
- [ATF Glossary](#). Glossary defining ATF technical terms and utilities.

Topics in this section

Links on this page to other topics

[Adaptation in ATF](#), [ATF Application Basics and Services](#), [ATF Code Samples Discussions](#), [ATF Contexts](#), [ATF Glossary](#), [ATF Programmer's Guide Preface](#), [ATF Structure](#), [Commands in ATF](#), [Controls in ATF](#), [Creating an Application from an ATF Sample](#), [Documents in ATF](#), [Instancing In ATF](#), [MEF with ATF](#), [Property Editing in ATF](#)

ATF Resources

ATF Resources

Many resources are available for ATF.

Samples

The ATF distribution includes a full set of complete, running sample applications that demonstrate various aspects of the ATF architecture. These applications reside in the ATF\Samples folder. Documentation for these tutorials is available from the [ATF Code Samples](#) on SHIP. Build the samples using the Samples solution in \components\wws_atf\Samples.

SHIP Resources

SHIP contains several pages about ATF:

- [Authoring Tools Framework](#)
This ATF page has team contact information and links to video tutorials, news, forums, downloads, the bug tracker, and various wikis.
- [ATF Documentation](#)
Download documentation and presentation files here.
- [ATF Code Samples](#)
Samples' description.
- [ATF Technology and Sample App Matrix](#)
Table of what ATF technology is used in samples.
- [ATF Programming Guidelines](#)
Programming and code style guidelines followed by the Authoring Tools Framework team. Even if you are not developing ATF itself, these are useful development guidelines.

If you have any trouble viewing these pages, send an email to support@ship.scea.com to request access.

SCE Developer Community Resources

In addition to providing useful resources, SHIP is home to a vibrant SCE developer community that includes the ATF team. Visit SHIP to contact the ATF team, read blogs, request features, report bugs, and discuss topics of mutual interest with other SCE developers.

Forums

Have a question? Found a bug? Need information that isn't in the documentation? On the [Authoring Tools Framework](#), you can follow the discussions or post a new thread about your hot topic. The ATF team frequently monitors the forum, so a team member may be the first responder to your query.

Reporting Bugs and Requesting Features

If you want to report a bug or request a feature, visit [Trackers for bugs and feature requests](#).

If you don't see your issue there, post a new thread on the [ATF Forum](#).

Contacting the ATF Team

To contact the ATF development team, see the Owner on [Authoring Tools Framework](#). You can raise issues to the entire team with the [ATF Forum](#).

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.

Links on this page to other topics

[ATF Code Samples](#), [ATF Documentation](#), [ATF Programming Guidelines](#), [ATF Technology and Sample App Matrix](#), [Authoring Tools Framework](#)

ATF Programmer's Guide Typographical Conventions

Typographical Conventions

This document follows the typographical conventions in the following table.

Font	Used for	Examples
monospace	paths and file names	wws_atf\Samples\CircuitEditor
	scripts	<pre>function ShouldRevive() return m_shouldRevive end</pre>
	code elements and code	<pre>public Command(string description) { m_description = description ; }</pre>
	formatted text, such as XML	<pre><?xml version="1.0" encoding="utf-8" standalone="yes"?> <StringLocalizationTable> <StringItem id="Cut" context="" translation= "Cut" /></pre>
bold	user interface elements	Add button
	menu items and other paths	Build > Build Solution
italic	new technical terms	Document Object Model
	emphasis	Do not do this.
	document titles	Getting Started with ATF
blue	URLs (external)	ATF Forum
	links inside the document or wiki	ATF Documentation
quoted	exact text	"ActiproSoftware SyntaxEditor"
	names	"Add Layer"

Topics in this section

Links on this page to other topics

[ATF Documentation](#)

Getting Started with ATF

Here are some suggested steps for getting to know and work with ATF:

Introduction

- Read the [ATF 3 Overview](#) to get a high-level understanding of the major parts of the Authoring Tools Framework (ATF) and how they fit together.
- Read Getting Started with ATF, especially the first chapter, "Introduction to ATF". Download this document at [ATF Documentation](#).
- Watch this short three and a half minute video describing ATF:



More Advanced Documentation

- Look at [ATF Code Samples](#) to get an overview of the samples.
- See the [ATF Technology and Sample App Matrix](#) to see what pieces of ATF technology are exercised in various samples.
- If your application might use a DOM to handle application data, read the ATF Programmer's Guide: Document Object Model (DOM), especially the first chapter, "Overview of the Document Object Model (DOM)", downloadable at [ATF Documentation](#).
- Look at [MEF with ATF](#), especially if you are not familiar with MEF, Microsoft's Managed Extensibility Framework.
- Read [ATF Application Basics and Services](#) to see basic ATF application structure.
- Read the [ATF Glossary](#).

Hands On

- Install the latest version of ATF with the Package Manager, as explained in the "Installing ATF" chapter of Getting Started with ATF, including a description of the contents. You can also [download the latest ATF packages](#) from our Alfresco server as a ZIP package that includes the documentation and sample application code. The ATF release package contains the following folders:
 - DevTools: Tools to help clients set-up, port, and manage their ATF projects.
 - Docs: ATF documentation, including programmer and reference documentation. The documentation is also available at [ATF Documentation](#).
 - Framework: Reusable .NET components, written in C#, for building your applications. Clients should avoid modifying this code.
 - Legacy: ATF 2.0 framework and tutorials. As much as possible, ATF 2 types derive from or make use of ATF 3 types to avoid duplicating code.
 - Samples: Code for sample applications that demonstrate various aspects of ATF and its features. Clients are encouraged to start with a sample application and then modify it. For details, see [Creating an Application from an ATF Sample](#).
 - Test: Unit tests and Everything.sln, which is our one solution file that references all source code in an ATF distribution.
 - ThirdParty: Pre-compiled managed and unmanaged DLLs that are licensed to be used by ATF.
- Build ATF. ATF source is organized into a set of Visual Studio solution and project files. The main solution files are:
 - \Samples\Samples.sln: Includes all the ATF 3 samples and framework.
 - \Test\Everything.sln: Includes ATF 3 and ATF 2 framework and samples, as well as unit tests and development tools
- Read [Creating an Application from an ATF Sample](#), which includes source for the created application.
- Study the programming description for any samples of interest under [ATF Code Samples Discussions](#).

Joining the ATF Community

- Join the ATF team on our [Forum](#).

ATF Structure

ATF's structure can be visualized in a variety of ways:

- **ATF Frameworks:** Frameworks in ATF.
 - **ATF Namespaces:** ATF is organized under a single namespace `Sce.Atf` with numerous nested namespaces.
 - **ATF Assemblies:** ATF is divided into several assemblies.
 - **ATF Functional Areas:** Look here to find what you need to work in various functional areas.
-

Topics in this section

Links on this page to other topics

[ATF Assemblies](#), [ATF Frameworks](#), [ATF Functional Areas](#), [ATF Namespaces](#)

ATF Frameworks

ATF is a framework that contains frameworks:

- Adaptation Framework: convert objects to other types to support different kinds of data models and managed data stores, such as a DOM. For details on using adaptation, see [Adaptation in ATF](#).
- Application Shell Framework: add core application services needed for applications with a GUI. For more information, see [ATF Application Basics and Services](#).
- Commands: create custom commands for application menus, buttons, and other controls. For information on creating and using commands, see [Using Commands in ATF](#).
- Context Framework: track and work with application contexts. For details on context usage, see [ATF Contexts](#).
- Document Framework: track documents a user is working on. To learn about this, see [Documents in ATF](#).
- Document Object Model (DOM) Framework: load, store, validate, and manage changes to application data, independently of application code. To find out how to use the DOM, see the ATF Programmer's Guide: Document Object Model (DOM), which you can download from [ATF Documentation](#).
- Instancing Framework: work with object instances being edited. For information on instancing, see [Instancing In ATF](#).
- Property Editing Framework: edit properties with controls. For information, see [Property Editing in ATF](#).

Topics in this section

Links on this page to other topics

[Adaptation in ATF](#), [ATF Application Basics and Services](#), [ATF Contexts](#), [ATF Documentation](#), [Documents in ATF](#), [Instancing In ATF](#), [Property Editing in ATF](#), [Using Commands in ATF](#)

ATF Namespaces

ATF is organized under a single namespace, `Sce.Atf`, which contains numerous nested namespaces.

Each namespace has an emphasis, although this is not rigid. Key namespaces are described below. Those marked * are especially important.

Some namespaces, such as `Sce.Atf` and `Sce.Atf.Applications`, are in more than one assembly. Others, such as `Sce.Atf.Wpf`, are in a single assembly (`Atf.Gui.Wpf`).

- *`Sce.Atf`: Provide a variety of components, utilities, and interfaces. For example, it contains general purpose file dialogs, loggers, file utilities, GDI utilities, Windows interoperability functions, LINQ (Language Integrated Query) classes, and math utilities. It contains major interfaces, such as `IDocument`, `IInitializable`, and the context interfaces `IValidationContext`, `IObservableContext`, `ITransactionContext`, and `ILockingContext`.
- *`Sce.Atf.Adaptation`: Classes that specialize in adapting objects to other types. It contains the interfaces `IAdaptable` and the key extension methods `As<T>()` and `Cast<T>()`. The Adaptation framework resides in this and the `Sce.Atf.Controls.Adaptable` namespace, so these namespaces provide adaptation support in ATF.
- *`Sce.Atf.Applications`: This is the key namespace. Many key ATF components and interfaces reside here. This namespace provides a great variety of services, including many MEF components you can easily incorporate into applications.
- `Sce.Atf.Controls`: Many of ATF's controls are here, including special purpose dialogs and various tree related controls. This namespace provides many controls not in standard .NET Windows Forms, including controls for editing numbers and file paths, a canvas, and tree controls. `Sce.Atf.Controls.Adaptable` and `Sce.Atf.Controls.Adaptable.Graphs` also contain many controls.
 - *`Sce.Atf.Controls.Adaptable`: Controls with adapters, including the fundamental `AdaptableControl`. The Adaptation framework resides in this and the `Sce.Atf.Adaptation` namespace, so these namespaces provide adaptation support in ATF.
 - `Sce.Atf.Controls.Adaptable.Graphs`: Many classes that work with graphs, various controls using Direct2D. For more details, see [Graphs in ATF](#).
 - `Sce.Atf.Controls.PropertyEditing`: Property editor controls and associated classes to edit many kinds of data.
 - `Sce.Atf.Controls.Timelines`: Timeline classes and interfaces that are UI-platform-agnostic (such as Windows Forms or WPF).
 - `Sce.Atf.Controls.Timelines.Direct2D`: Classes for working with timeline elements, including a renderer, control, and manipulators, that are dependent on the Windows Forms UI platform and use Direct2D to render timeline objects.
- *`Sce.Atf.Dom`: Classes that allow you to use a DOM for application data, including the very important `DomNode`, `DomNodeAdapter`, and `DomNodeType`. The Document Object Model (DOM) framework is defined here.
- `Sce.Atf.Rendering`: Facilities for rendering 3D graphics. These are used by the Level Editor tool in the WWS SDK.
 - `Sce.Atf.Rendering.Dom`: Classes that render data typically stored in a DOM, such as a 3D Scene. These are used by the Level Editor tool in the WWS SDK.
- *`Sce.Atf.Wpf`: This and its nested namespaces provide WPF facilities.
 - `Sce.Atf.Wpf.Applications`: Variety of classes, components, and utilities for WPF. This namespace provides the Application Shell framework components used for WPF.
 - `Sce.Atf.Wpf.Behaviors`: Classes to adapt WPF behavior to ATF.
 - `Sce.Atf.Wpf.Models`: Provide services to manage controls that use the Model-View-Controller (MVC) design pattern in WPF. With the MVC control framework, you attach data controls to an underlying data model (usually the DOM) through adapter client classes that you define. The adapter client exposes, filters, and presents the data model in a way the control understands.

Topics in this section

Links on this page to other topics

[Graphs in ATF](#)

ATF Assemblies

ATF is divided into the following assemblies:

- `Atf.Atgi`: support for ATGI asset description files.
- `Atf.Collada`: support for Collada asset description files.
- `Atf.Core`: core application services, including components.
- `Atf.Gui.OpenGL`: OpenGL support.
- `Atf.Gui`: additional GUI support.
- `Atf.Gui.WinForms`: primary WinForms application support.
- `Atf.Gui.Wpf`: WPF application support.
- `Atf.IronPython`: Python services.
- `Atf.Perforce`: Perforce services.
- `Atf.SyntaxEditorControl`: syntax editor controls

These assemblies vary considerably in their number of constituent classes. The largest are `Atf.Core`, `Atf.Gui`, `Atf.Gui.WinForms`, and `Atf.Gui.Wpf`.

Each assembly contains namespaces. Some namespaces are in more than one assembly.

Each assembly can be built into a single DLL, which can be included with applications, if needed.

Topics in this section

Links on this page to other topics

No links

ATF Functional Areas

To find what you need in various functional areas, see the namespaces and frameworks in this table.

Functional Area	Namespaces	Framework	See
Adaptation	Sce.Atf.Adadaptation	Adadaptation Framework	Adadaptation in ATF
Adaptable controls	Sce.Atf.Controls.Adaptable Sce.Atf.Controls.Adaptable.Graphs	Adadaptation Framework	Adadaptation in ATF Controls in ATF
Commands	Sce.Atf.Applications	Commands	Commands in ATF
Components	Sce.Atf.Applications	Application Shell Framework, for core application services	ATF Application Basics and Services MEF with ATF
Controls	Sce.Atf.Controls		Controls in ATF
Documents	Sce.Atf Sce.Atf.Applications	Document Framework	Documents in ATF
DOM	Sce.Atf.Dom	Document Object Model (DOM) Framework	ATF Programmer's Guide: Document Object Model (DOM) , downloadable from ATF Documentation
Property editing	Sce.Atf.Controls.PropertyEditing	Property Editing Framework	Property Editing in ATF
3D rendering	Sce.Atf.Rendering Sce.Atf.Rendering.Dom		Model Viewer Programming Discussion

Topics in this section

Links on this page to other topics

[Adadaptation in ATF](#), [ATF Application Basics and Services](#), [ATF Documentation](#), [Commands in ATF](#), [Controls in ATF](#), [Documents in ATF](#), [MEF with ATF](#), [Model Viewer Programming Discussion](#), [Property Editing in ATF](#)

Creating an Application from an ATF Sample

This section shows how to build an application based on an existing ATF sample, step by step. An actual application is developed, based on the [ATF Simple DOM Editor Sample](#).

- Choose ATF Sample to Base Application On: Pick a sample that most closely matches your new application's features.
- Copy ATF Sample and Run It: Copy the indicated folders and files in ATF to start your application development.
- Start Customizing Your Application Software: Begin the process of transforming the sample into your application.
- Define Your Application's New Data Model: Define the types in your application's data in a file and create a schema.
- Modify Your Application's Existing Software: Modify the sample's existing code to work for your application.
- Add New Code and Parts to Your Application: Add the application's unique features, using the appropriate ATF classes and components.
- Build, Run, and Debug Your Application: With all code changed and features added, finish development.

Download zipped source for the created application [here](#).

Choose ATF Sample to Base Application On

Start with the specification (in some form) that you have for your new application.

Pick a sample that most closely matches your new application's features. Look at the [ATF Code Samples](#) and their descriptions and the [ATF Technology and Sample App Matrix](#) showing which samples use various pieces of ATF technology to help you pick the most suitable sample.

You might find that several samples have features close to your new application, and you can incorporate parts from several samples in your application. It's easiest to build upon one sample though.

This topic shows the development of an application to list landscaping materials: plants and ornamentation items for a garden. The Landscape Guide application has these features:

- A palette with plant and ornamentation icons.
- Palette items can be dragged onto a list view.
- The selected list view item can have its properties set in property editors.

All landscaping items have this information:

- Name.
- Text notes on the item.
- Rating, an integer from 1 to 10 indicating level of interest.

In addition, a plant has a botanical name; an ornamental item has a color.

The [ATF Simple DOM Editor Sample](#) fits this type of application pretty well: it features a palette of items that can be dragged onto a list view. Items can be selected and have their properties set and changed in two property editors.

Topics in this section

Links on this page to other topics

[ATF Code Samples](#), [ATF Simple DOM Editor Sample](#), [ATF Technology and Sample App Matrix](#)

Copy ATF Sample and Run It

ATF Folder Structure

You need several folders and a file in the `wws_atf` folder of the ATF distribution to build samples or an ATF application:

- `Framework`: folders of source files for each assembly in ATF.
- `Samples`: the samples, one folder for each sample containing all the sample's source and other files.
- `ThirdParty`: third party DLLs and other files ATF uses.
- `wws_atf.component`: a file needed to build an ATF application.
- `DevTools\DomGen`: a utility for generating schema stub classes.

It is simplest to maintain this folder structure for your application. You do not need all the files in each folder; what you need depends on your application's requirements.

Topics

- [ATF Folder Structure](#)
- [Copy Sample](#)
- [Copy Needed Files](#)
- [Build and Run Sample](#)

Copy Sample

Make a copy of the project folder for the sample you are going to be working from, `SimpleDomEditor`, in this case. There are several ways to do this.

One way is to simply copy the folder for the sample and leave the copy in the `Samples` folder. All the other folders and files are then available.

You can also replicate the folder structure of the ATF distribution for your new application project. Create a new folder for your project(s), say `Projects`; create `Framework` and `ThirdParty` folders in `Projects`; copy the `wws_atf.component` file into `Projects`, too. Create a folder in `Projects` for your projects, analogous to the `Samples` folder, say `ATF Applications`. Make a copy of the `Samples/SimpleDomEditor` folder, rename it "LandscapeGuide", and place it in `ATF Applications`. This is the approach taken here.

Copy Needed Files

Determine what assemblies are used by the chosen sample project. For `SimpleDomEditor`, these are:

- `Atf.Core`
- `Atf.Gui`
- `Atf.Gui.WinForms`
- `Atf.IronPython`

You can determine the needed assemblies by opening the Samples' Solution file, `Samples.vs2010.sln`, and going to the Solution's Property Pages. In Common Properties > Project Dependencies, select the desired project. The Depends on panel has the assemblies checked that you need. There is one folder in `Framework` for each assembly; copy the assembly folders needed to your `Framework` folder.

You may change these assemblies later as needed, based on what code you add or remove.

Next, determine the third party files needed. For `SimpleDomEditor`, these are:

- `Bespoke`
- `DockPanelSuite`
- `IronPython`
- `MEF`
- `SharpDX`
- `Wws.LiveConnect`

The simplest way to find what's needed is to try building. Error messages indicate what's missing.

Copy the `Samples.vs2010.sln` file to your `ATF Applications` folder.

Build and Run Sample

To verify that you have copied everything you need, open the `.sln` file and build `SimpleDomEditor`. If it fails to build, examine the error messages and make the appropriate fixes. For instance, you might need some third party software that you have not copied into the `ThirdParty` folder. Make the changes needed until you successfully build. Next, run `SimpleDomEditor` and verify it's working, making changes until it operates properly.

Topics in this section

Links on this page to other topics

No links

Start Customizing Your Application Software

Begin the process of transforming the sample into your application.

Topics

- Fix Up .sln File
- Remove Code
- Remove MEF Components
- Modify Resources

Fix Up .sln File

Rename the solution file to something more appropriate for your development, such as `ATFProjects.vs2010.sln`.

Open the solution in Visual Studio and edit the project appropriately in the Solution Explorer pane:

- Remove the ATF assembly projects you don't need.
- Remove the ATF sample projects, except for the sample you are copying, "SimpleDomEditor" in this case.
- Rename the "SimpleDomEditor" project "LandscapeGuide.vs2010".

Change all the namespace statements in the source `.cs` files to something appropriate. In this case, they are changed to:

```
namespace LandscapeGuide
```

When you change the namespace, also change it in the Project settings > Application page's Default namespace field. Change Assembly name on this page appropriately, too.

In `SchemaLoader.cs`, you must also change the name of the namespace:

```
SchemaResolver = new ResourceStreamResolver(Assembly.GetExecutingAssembly(),
    "LandscapeGuide/schemas");
Load("eventSequence.xsd");
```

If you don't change Default namespace for the project, when you run, you get an unhandled exception of type `System.ComponentModel.Composition.CompositionException` in `Atf.Core.dll`. This results from a problem with the above lines in `SchemaLoader.cs`. Because the default namespace is not set properly, the schema address is not resolved, so `Load()` fails. Note that "LandscapeGuide/schemas" is the namespace in the code above.

Build the slightly modified project again and verify that it still runs properly, making changes as needed.

Remove Code

If you have not already done so, you should become familiar with the code in the sample you're renovating. After doing this, you can figure out what code you don't need.

You can start by removing files from the project that aren't relevant. In `SimpleDomEditor`, these files are not required for `LandscapeGuide`:

- `ResourceListEditor.cs`: component that adds the Resources pane, which `LandscapeGuide` doesn't have.
- `DomNodeNameSearchControl.cs`: provides the Search and Replace - DomNode Names pane.
- `DomNodeNameSearchService.cs`: provides the Search and Replace pane.
- `DomNodeAdapters/Resource.cs`: DomNode adapter for the "Animation" and "Geometry" resources, which are not used in `LandscapeGuide`.
- `EventContext.cs`: Editing context that provides a context for updating the views of the Resources pane.

Remove MEF Components

In `Program.cs`, you can remove the following components from the MEF catalog, because their capabilities are not used in `LandscapeGuide` and their corresponding files have already been removed:

- `DomNodeNameSearchService`

- DomNodePropertySearchService
- ResourceListEditor

Rebuild the project again and verify that it still runs, making changes as needed. (You need to remove a reference to `EventContext` in `SchemaLoader.cs` to build.) The panes created by the removed components are now gone, but the rest of the application should still function.

Modify Resources

Remove resources you don't need, such as graphics files. Remove references to these resources in the `Resources` class in the `Resources.cs` file also.

Modify the `About.rtf` file to describe your new application. Change `HelpAboutCommand.cs` to suit the new application, as in:

```
using (AboutDialog dialog = new AboutDialog()
    Localizer.Localize("Landscape Guide"), appURL, richTextBox, null, null, true))
```

Once you make these changes, though, the original application won't run properly, because it is missing resources. This gets fixed in subsequent steps.

Topics in this section

Links on this page to other topics

No links

Define Your Application's New Data Model

Overview

Most applications have a data model, because a typical application allows a user to create, modify, and save some kind of data, whether it's plain text or graphic objects in a scene. If your application works with data, you need to define its data model, that is, specify the types of the data it uses. The new `LandscapeGuide` application allows a user to enter data, so this is a necessary step.

ATF provides a Document Object Model (DOM) framework for managing application data. For details of using the ATF DOM, see the ATF Programmer's Guide: Document Object Model (DOM), downloadable from [ATF Documentation](#).

In ATF, the simplest way to define the data model is to create a type definition file in the XML Schema Definition (XSD) language (or XML schema for short), because ATF provides built-in support for XSD. `SimpleDomEditor` uses XSD and provides its type definition in the `eventSequence.xsd` schema file, and `LandscapeGuide` also takes this approach.

If you are not familiar with XSD, see the W3 Schools tutorial at <http://www.w3schools.com/schema/>.

Topics

- Overview
- `LandscapeGuide`'s Basic Types
- Define the Data Types in XSD
- Modify `SchemaLoader`
- Generate a Schema Stub Class File

`LandscapeGuide`'s Basic Types

Define `LandscapeGuide`'s data model by specifying the data types it uses. `LandscapeGuide` uses mainly two kinds of data objects: plant and ornamentation items. Both these items have attributes associated with them: a name, notes, and rating, and each type has distinct attributes as well. These items are similar enough that they can be considered variants of a single type. Because these types have attributes, they are complex types.

Applications often have a data type that represents a collection of data items in the application. In this case, a second data type represents a list of plant and ornamentation items, which display in a list view. This type is also complex, since it has child items.

Define the Data Types in XSD

The next step is to formally define the types in XSD in a type definition file, `items.xsd`. The following describes each section of this file.

First, set the namespace for the schema:

```
<xs:schema
  elementFormDefault="qualified"
  targetNamespace="itemList_1_0"
  xmlns="itemList_1_0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

The following simple type defines an integer between 1 and 10, which describes the rating attribute of an item:

```
<!--Simple type for number in a range-->
<xs:simpleType name="ratingType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
```

The "ratingType" type places restrictions on the minimum and maximum bounds for an integer of this type.

Both plant and ornamentation items are items, so define an abstract "itemType" that fits both, containing their common attributes. This subtype approach simplifies development.

```
<!--Abstract base type for items-->
<xss:complexType name="itemType" abstract="true">
  <xss:attribute name="name" type="xs:string" use="required"/>
  <xss:attribute name="notes" type="xs:string"/>
  <xss:attribute name="rating" type="ratingType" default="5"/>
</xss:complexType>
```

The three common item attributes are defined here. Note that the type of the "rating" attribute is "ratingType", defined above. The "rating" attribute also has a default value of 5.

Now that the general "itemType" for an item has been defined, specify its subtypes, first plant:

```
<!--Plant, with botanical name and other attributes-->
<xss:complexType name ="plantItemType">
  <xss:complexContent>
    <xss:extension base="itemType">
      <xss:attribute name="botanicalName" type="xs:string"/>
    </xss:extension>
  </xss:complexContent>
</xss:complexType>
```

The type "plantItemType" extends "itemType", adding the "botanicalName" attribute. Next, ornamentation:

```
<!--Ornamentation, with color and other attributes-->
<xss:complexType name ="ornamentationItemType">
  <xss:complexContent>
    <xss:extension base="itemType">
      <xss:attribute name="color" type="xs:string"/>
    </xss:extension>
  </xss:complexContent>
</xss:complexType>
```

The "ornamentationItemType" type extends "itemType", adding the "color" attribute.

Finally, the item list:

```
<!--Landscape item list, a list of items-->
<xss:complexType name ="itemListType">
  <xss:sequence>
    <xss:element name="item" type="itemType" maxOccurs="unbounded"/>
  </xss:sequence>
</xss:complexType>
```

Note that this type contains "itemType" children — not "plantItemType" or "ornamentationItemType". It allows an unlimited number of children.

The ATF DOM stores application data in a tree of nodes, and each node has a type that is one of the data model's defined types. You can specify the root node of this tree, which is of "itemListType" type:

```
<!--Declare the root element of the document-->
<xss:element name="itemList" type="itemListType"/>
```

As you are defining types, you can either edit the existing file `eventSequences.xsd` and rename it `items.xsd`, or remove `eventSequence.xsd` and add the new one, `items.xsd`.

Make sure that `items.xsd` has these properties set, especially if you add a new file:

- Build Action: Embedded Resource
- Copy to Output Directory: Copy if newer

These are not the default values for an added file, so you must set them. The schema file must be embedded in the application. Otherwise the application gets an exception when it tries to load the schema file.

Modify SchemaLoader

Now that you have created a new schema file `items.xsd`, modify the `SchemaLoader` to load it:

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

Generate a Schema Stub Class File

Given a type definition file in XML schema, the ATF DomGen utility generates a stub class that defines the metadata classes for the data types. Using this stub class, conventionally called Schema, allows you to avoid using string names for types and attributes from the type definition file and instead use the DomGen generated names. This allows the compiler to check for name correctness.

The easiest way to run DomGen is from the Schema/GenSchemaDef.bat file. Edit this file to something like this, and then run it:

```
<DomGen path>\DomGen.exe items.xsd Schema.cs "itemList_1_0" LandscapeGuide
```

These parameters provide the input file `items.xsd`, the output file `Schema.cs`, the XML target namespace of the types within the schema "`itemList_1_0`", and the C# namespace "`LandscapeGuide`".

The type and attribute names in `Schema.cs` determine many of the changes you need to make in your application, because the type and attribute names are frequently used.

Topics in this section

Links on this page to other topics

[ATF Documentation](#)

Modify Your Application's Existing Software

This section discusses how to modify the sample's existing code to work for the new application.

What's Different?

Examining the differences between `SimpleDomEditor` and `LandscapeGuide` helps you see what changes you need to make to get to the goal application. In particular, look at the data model differences, because these two applications are very similar otherwise.

`SimpleDomEditor` has a main type "eventType" that describes events, and the main list view shows a sequence of events. `LandscapeGuide` does something similar, but has two types of items in the list view: plants and ornaments. On the other hand, `SimpleDomEditor` has two resource types, "animationResourceType" and "geometryResourceType", which are both subtypes of "resourceType", and are both listed in the Resources pane. Following this pattern, "plantItemType" and "ornamentationItemType" are subtypes of "itemType" in `LandscapeGuide`. In `SimpleDomEditor`, the two resource types only appeared in the Resources pane's list view; in `LandscapeGuide` we want these two types to appear in the application's main list view, a list box, instead.

Topics

- What's Different?
- Changing Types
- DOM Adapter Modifications
- SchemaLoader Changes
 - Define Extensions
 - Property Editing and Descriptors
 - Palettes
- Palette Changes
- Context Changes
- Document Modifications
- Editor Changes
- Build Application

Changing Types

You can start modifying source by mapping old type names to new ones. In general, "event" transforms to "item" and "eventSequence" to "itemList". This means making global text changes in the project files like these:

- "eventType" to "itemType"
- "eventSequenceType" to "itemListType"
- "eventList" to "itemList"

Keep an eye on the use of these types from `SimpleDomEditor`:

- "resourceType"
- "animationResourceType"
- "geometryResourceType"

Although these types aren't used in `LandscapeGuide`, their use suggests ways to use types in your new application, such as code that differentiates between the animation and geometry resource subtypes.

DOM Adapter Modifications

Two of the DOM adapters in `SimpleDomEditor` carry over pretty well to `LandscapeGuide`:

- `DomNodeAdapters/Event.cs`: change its name to `Item.cs` and use it for "itemType".
- `DomNodeAdapters/EventSequence.cs`: change its name to `ItemList.cs` and use it for "itemListType".

The original files can be easily modified to work for the new DOM adapters. `SimpleDomEditor`'s adapter `DomNodeAdapters/Resource.cs` has already been removed.

`Item.cs` provides the `Item` class, a simple DOM adapter with properties for the item type's attributes:

```

/// <summary>
/// DomNode adapter for item data</summary>
public class Item : DomNodeAdapter
{
    /// <summary>
    /// Gets or sets name associated with item, such as a label</summary>
    public string Name
    {
        get { return GetAttribute<string>(Schema.itemType.nameAttribute); }
        set { SetAttribute(Schema.itemType.nameAttribute, value); }
    }

    /// <summary>
    /// Gets or sets notes associated with item</summary>
    public string Notes
    {
        get { return GetAttribute<string>(Schema.itemType.notesAttribute); }
        set { SetAttribute(Schema.itemType.notesAttribute, value); }
    }

    /// <summary>
    /// Gets or sets rating associated with item</summary>
    public int Rating
    {
        get { return GetAttribute<int>(Schema.itemType.ratingAttribute); }
        set { SetAttribute(Schema.itemType.ratingAttribute, value); }
    }
}

```

Be careful to specify the correct type for properties. If you edit an existing file, it's easy to forget this and produce a property like this:

```

/// <summary>
/// Gets or sets notes associated with item</summary>
public int Notes
{
    get { return GetAttribute<int>(Schema.itemType.notesAttribute); }
    set { SetAttribute(Schema.itemType.notesAttribute, value); }
}

```

This causes an exception, warning that a value could not be cast, because the Notes attribute is a `string`, not an `int`.

Here is `ItemList.cs`, whose `Items` property returns a list of the items:

```

/// <summary>
/// DomNode adapter for item list data</summary>
public class ItemList : DomNodeAdapter
{
    /// <summary>
    /// Gets list of items in item list</summary>
    public IList<Item> Items
    {
        get { return GetChildList<Item>(Schema.itemListType.itemChild); }
    }
}

```

Note that these adapters use the class and field names defined in the `Schema.cs` file's stub class `Schema`. Again, this is similar to `SimpleDomEditor`, which uses names from its `Schema` class for its DOM adapters.

SchemaLoader Changes

The schema loader class, in `SchemaLoader.cs`, loads the `items.xsd` file to get information from the schema, such as restrictions.

The `SchemaLoader.cs` file from `SimpleDomEditor` can be used with some straightforward changes. Many of these have already been made with the global string substitutions for type names described above.

The constructor `SchemaLoader()` has already been changed, so the remaining changes are in `OnSchemaSetLoaded()`. These changes are to define extensions, set up property editing and descriptors, and set up palette items. The code to do this is already in the sample's `OnSchemaSetLoaded()` method; it simply needs to be edited for the `LandscapeGuide` types.

Define Extensions

Extensions are classes that allow you to cast data in DOM nodes to other, more convenient types, and perform a variety of useful behaviors, such as listening to events on DOM nodes. Extensions are defined for data types, and this is usually done in the schema loader.

This extension definition from `SimpleDomEditor` goes away, because the editing context `EventContext` no longer exists:

```
Schema.eventType.Type.Define(new ExtensionInfo<EventContext>());
```

This extension definition from `SimpleDomEditor` goes away, because the `Resource` DOM adapter no longer exists:

```
Schema.resourceType.Type.Define(new ExtensionInfo<Resource>());
```

The new DOM adapters, `Item` and `ItemList` discussed above, have these definitions for their associated types:

```
Schema.itemListType.Type.Define(new ExtensionInfo<ItemList>());  
...  
Schema.itemType.Type.Define(new ExtensionInfo<Item>());
```

These extension definitions in `SimpleDomEditor`:

```
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceDocument>());  
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceContext>());
```

change in `LandscapeGuide` by simply changing type names as indicated above:

```
Schema.itemListType.Type.Define(new ExtensionInfo<ItemListDocument>());  
Schema.itemListType.Type.Define(new ExtensionInfo<ItemListContext>());
```

This definition is new for `LandscapeGuide`:

```
Schema.itemListType.Type.Define(new ExtensionInfo<DataValidator>()); //for restriction facets
```

The `ratingAttribute` attribute is used for ratings and is restricted to an integer between 1 and 10, as specified in "ratingType" in the XML Schema. Rules are automatically added to validate this attribute when the schema is loaded if you are using the default ATF schema loader. However, for the rules to operate, the `DataValidator` validator must be defined for the root type, as in this statement above.

Property Editing and Descriptors

Property descriptors are metadata for class properties that controls such as property editors can use. You can set up property descriptors for each type, which determines which properties — that is attributes — can be edited in property editing controls. The following block (very similar to the descriptors set up in `SimpleDomEditor`) specifies the editable properties for a plant item in a property descriptor:

```

// Register property descriptors for plant type
Schema.plantItemType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                Schema.plantItemType.nameAttribute,
                null,
                Localizer.Localize("Plant name"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Botanical name"),
                Schema.plantItemType.botanicalNameAttribute,
                null,
                Localizer.Localize("Plant botanical name"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Notes"),
                Schema.plantItemType.notesAttribute,
                null,
                Localizer.Localize("Notes on plant"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Rating"),
                Schema.plantItemType.ratingAttribute,
                null,
                Localizer.Localize("Rating reflecting interest"),
                false),
        }));

```

The `AttributePropertyDescriptor` constructor's parameters for each attribute/property specify the localized name, attribute, category, description, and whether the property is read only.

All the plant item's attributes are specified here, including the ones provided in "itemType", not just the "botanicalNameAttribute" attribute unique to "plantItemType". If any attributes are omitted, they don't appear in any property editor for a plant item. There is one `AttributePropertyDescriptor` constructor for each attribute that can be edited.

A similar property descriptor should also be set up for the ornamentation item type.

The class `CustomTypeDescriptorNodeAdapter` enables metadata-driven property editing in an application. This means that properties that appear in property editors are determined by the metadata provided about the properties in property descriptors, as in the above example. The following statement ensures that when items, either plants or ornaments, are edited in property editor controls, the attributes that are editable come from the property descriptors provided for each item type:

```

// Enable metadata driven property editing for events and resources
AdapterCreator<CustomTypeDescriptorNodeAdapter> creator =
    new AdapterCreator<CustomTypeDescriptorNodeAdapter>();
Schema.itemType.Type.AddAdapterCreator(creator);

```

Palettes

The `NodeTypePaletteItem` class indicates how a type should appear in a palette of items that a user can drag and drop into a document. The following statements set the properties for the palette objects:

```
// Annotate types with display information for palette
Schema.plantItemType.Type.SetTag(
    new NodeTypePaletteItem(
        Schema.plantItemType.Type,
        Localizer.Localize("Plant"),
        Localizer.Localize("Plant item"),
        Resources.PlantImage));

Schema.ornamentationItemType.Type.SetTag(
    new NodeTypePaletteItem(
        Schema.ornamentationItemType.Type,
        Localizer.Localize("Ornamentation"),
        Localizer.Localize("Ornamentation item"),
        Resources.OrnamentationImage));
```

Among other things, the `NodeTypePaletteItem` sets the icon that appears in the palette, specified as an image resource like `Resources.PlantImage` here.

Palette Changes

The palette for `LandscapeGuide` contains two items: plants and ornamentation items. The schema loader has already changed to accommodate the new palette.

To create the new palette, make a few changes to `PaletteClient.cs`. The first change, in `Initialize()`, determines the items in the palette:

```
string category = "Landscaping items";
//m_paletteService.AddItem(Schema.plantItemType.Type, category, this);
//m_paletteService.AddItem(Schema.ornamentationItemType.Type, category, this);
foreach (DomNodeType itemType in m_schemaLoader.GetNodeTypes(Schema.itemType.Type))
{
    NodeTypePaletteItem paletteItem = itemType.GetTag<NodeTypePaletteItem>();
    if (paletteItem != null)
        m_paletteService.AddItem(itemType, category, this);
}
```

This loop finds all "itemType" subtypes and gets information about the `NodeTypePaletteItem` object for that type, which was set in `OnSchemaSetLoaded()`, as described above in [Palettes](#). In particular, it gets the icon for the item's image that appears in the palette. The commented out lines also do the job, but the loop is preferable, because it automatically accommodates adding new item types to the schema.

The next change is to a code segment in `IPaletteClient.Convert()`. It provides information on each palette client item, in this case the item to be used for an object when it is dragged onto a control, such as the list view in this application:

```
if (nodeType == Schema.plantItemType.Type)
    node.SetAttribute(Schema.plantItemType.nameAttribute, paletteItem.Name);
else
    if (nodeType == Schema.ornamentationItemType.Type)
        node.SetAttribute(Schema.ornamentationItemType.nameAttribute, paletteItem.Name);
```

These statements discriminate between the two item types, plant and ornament, and set the attribute accordingly.

Context Changes

Only one context file needs to change. `SimpleDomEditor`'s `EventContext.cs` is not needed in `LandscapeGuide` and has already been removed. `EventSequenceContext.cs` on the other hand, is useful, and its name is changed to `ItemListContext.cs` in `LandscapeGuide`, as well as changing names in the file.

Most of the modifications are trivial text changes:

- "eventSequenceType" to "itemListType".
- "Event" to "Item". (Be sure to do this as a "Match whole word" replacement, or terrible things will happen to your event handlers.)
- "_event" to "_item".

The following `ColumnNames` property in the `IListView` interface specifies what the column names are in the list view control, and lists strings for the attributes common to all items:

```

/// <summary>
/// Gets names for table columns</summary>
public string[] ColumnNames
{
    get { return new string[] { Localizer.Localize("Name"), Localizer.Localize("Notes"),
        Localizer.Localize("Rating") }; }
}

```

The change for the `GetInfo()` method is a little more involved. This code is actually copied from the `SimpleDomEditor EventContext.cs`'s `GetInfo()` method and then slightly modified to determine information for an item type:

```

/// <summary>
/// Fills in or modifies the given display info for the item</summary>
/// <param name="item">Item</param>
/// <param name="info">Display info to update</param>
public void GetInfo(object item, ItemInfo info)
{
    Item _item = Adapters.As<Item>(item);
    info.Label = _item.Name;

    string type = null;
    if (_item.DomNode.Type == Schema.plantItemType.Type)
        type = Resources.PlantImage;
    else if (_item.DomNode.Type == Schema.ornamentationItemType.Type)
        type = Resources.OrnamentationImage;

    info.ImageIndex = info.GetImageList().Images.IndexOfKey(type);

    info.Properties = new object[] { _item.Notes, _item.Rating };
}

```

Similarly to previous code changes, this code discriminates between the plant and ornament item types to set the information correctly: the image used for the item. The field `info.Properties` is set to an array of the attributes, other than "Name", associated with an item: "Notes" and "Rating". These changes result in the correct attribute information being displayed in the application's main list view for items.

The `GenericSelectionContext.cs` file is indeed generic and doesn't need to change at all other than the namespace already changed.

Document Modifications

The `eventSequence` and `itemList` types function very similarly in both `SimpleDomEditor` and `LandscapeGuide`. So much of the code from `SimpleDomEditor` carries over to `LandscapeGuide` with only type name changes.

The change from `EventSequenceDocument.cs` to `ItemListDocument.cs` is to change the file name and to simply make a few text changes from "EventSequence" to "ItemList". For example, "EventSequenceDocument" becomes "ItemListDocument".

Editor Changes

Again, most of these changes are simple text substitutions.

In going from `EventListEditor.cs` to `ItemListEditor.cs`, change the file name and make these text substitutions:

- "EventList" to "ItemList"
- "eventSequence" to "ItemList"
- "EventSequence" to "ItemList"
- "_event" to "_item"

`Editor.cs` changes involve the same text replacements. In addition, a few other changes are needed.

The namespace must change here:

```
m_scriptingService.ImportAllTypes("SimpleDomEditorSample");
```

becomes

```
m_scriptingService.ImportAllTypes("LandscapeGuide");
```

A few user interface changes are needed for the name of the list and for file extensions. Change

```
Localizer.Localize("Event Sequence"),  
new string[] { ".xml", ".esq" },
```

to

```
Localizer.Localize("Item list"),  
new string[] { ".xml", ".pil" },
```

Build Application

Building the application at this point may reveal some other changes you need to make, such as changing type names that were missed and adding resources, which is described in the next section.

Topics in this section

Links on this page to other topics

[Authoring Tools Framework](#)

Add New Code and Parts to Your Application

Add the application's unique features, using the appropriate ATF classes and components, starting with the most important features. This can entail creating components of your own, although it doesn't for `LandscapeGuide`.

Because it is simple and similar to `SimpleDOMEdition`, the only remaining things to add to `LandscapeGuide` are resources for the plant and ornamentation icons.

Topics

- Add Resources
- Add Data Persistence

Add Resources

Using a drawing tool of your choice, create plant and ornamentation icons the same size as the resource icons in `SimpleDOMEdition`. Add them to the project. When you add the icon files to the project, set their properties:

- Build Action: Embedded Resource
- Copy to Output Directory: Do not copy

If you do not embed the resources, you get compilation errors or an unhandled exception of type `System.ComponentModel.CompositionException` in `Atf.Core.dll` when you try to run the application.

Finally, add your resources to the `Resources` class in `Resources.cs`:

```
/// <summary>
/// Plant image resource filename</summary>
[ImageResource("plant.png")]
public static readonly string PlantImage;

/// <summary>
/// Ornamentation image resource filename</summary>
[ImageResource("ornamentation.png")]
public static readonly string OrnamentationImage;
```

Add Data Persistence

You generally want to save user edited data some way, usually in a file. `SimpleDOMEdition` does this by using ATF DOM classes for data persistence. The XML schema data model specified in the type definition file also specifies the XML format of saved data. If you want to use the ATF DOM and XML to save user data, you don't need to change anything. If you want to save data in some other format, you have additional code to write. You can base what you do on the ATF DOM `DomXmlReader` and `DomXmlWriter` classes. For information on data persistence with the ATF DOM, see the ATF Programmer's Guide: Document Object Model (DOM).

Topics in this section

Links on this page to other topics
No links

Build, Run, and Debug Your Application

Now that all changes have been made and all features added, you can finish development. In practice, of course, development cycles between debugging and fixing and adding features.

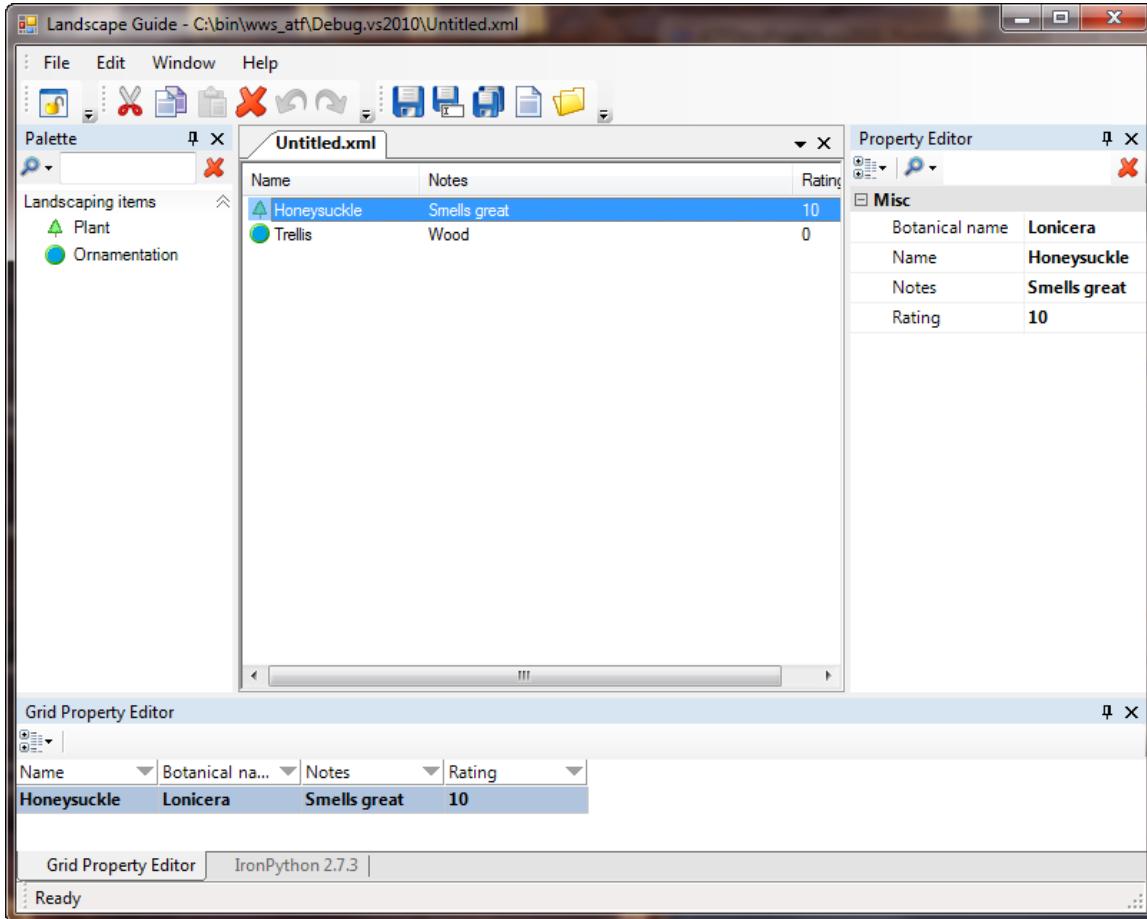
Build the Application

Try to build the application. Fix any compilation or other errors. You may need to add additional resources or fix type or attribute names, for instance, as previously noted.

Run and Debug the Application

Fix the cause for any exceptions you get. Exceptions may be caused by not including needed components or files not having the appropriate properties. You get exceptions for not embedding the schema file or icon files, as indicated previously.

Fix any deviations from the features. The properly running application looks like this:



LandscapeGuide operates in this way:

- Drag landscaping items from the Palette pane onto the list view list box.
- Select an item to display its attributes in the two property editor panes.
- Change item properties in either property editor.
- Save the file.
- When the application starts, the last saved file opens and displays its list of saved items.

Topics in this section

Links on this page to other topics

No links

MEF with ATF

ATF makes extensive use of Microsoft's Managed Extensibility Framework (MEF) to organize applications. This section shows how ATF uses MEF and how you can use it in your applications with existing ATF MEF components or your own MEF components.

- [What is MEF?](#) Brief description of MEF with links to more detailed information.
 - [How MEF is Used in ATF](#): Examine how ATF is used in sample applications to compose components and how components are decorated.
 - [Initializing Components](#): How component initialization is implemented in ATF.
 - [Using ATF Components](#): How to use ATF components, discovering what you need to provide in your application.
 - [Creating MEF Components](#): How to create components of your own, using the attributes you need.
 - [Important ATF Components](#): Description of some key ATF components in functional areas as well as some components in sample applications.
 - [Finding ATF Components](#): How to find components you need for your application.
-

What is MEF?

Microsoft's Managed Extensibility Framework (MEF) allows an application to easily add or remove components. A component is either an object or a C# type that is added to a special MEF container, so that the component can discover and use other components without having a hard-dependency between them.

Microsoft's MEF documentation calls these components extensions or parts; this document and the other ATF documents use the term component — or MEF component if required for clarity.

ATF contains many components to easily provide capabilities common to many applications, so components are an integral part of ATF. Many of these components can be used independently of MEF. Some key components are outlined in [Important ATF Components](#).

It is not in the scope of this reference to discuss MEF in depth, however, it does discuss [How MEF is Used in ATF](#).

You do need to know the basics of MEF to understand how ATF employs it. To learn about MEF, see the following:

- For a brief overview, see [What is MEF?](#) on CodePlex.
- For a short introduction showing how to use MEF, see [5 Minute Tutorial on Managed Extensibility Framework \(MEF\)](#).
- For more details on MEF usage, see [MEF Programming Guide](#) on CodePlex.
- For Microsoft's reference, see [Managed Extensibility Framework](#) on the MSDN Web site.
- For the MEF API Reference, see the [System.ComponentModel.Composition namespace](#) on MSDN.

Topics in this section

Links on this page to other topics

[How MEF is Used in ATF](#), [Important ATF Components](#)

How MEF is Used in ATF

The best way to see how ATF uses MEF is to look at the [ATF samples](#) and ATF components. This section examines code in detail to show what's required to use MEF.

Creating a Catalog

In MEF, you need to add components (parts) to a MEF catalog. There are several kinds of MEF catalogs, deriving from the base class `ComposablePartCatalog`.

Components are what MEF refers to as composable parts. These parts can be put together — composed — into a cohesive whole in the application, where the parts are instantiated and meet each others' import and export contracts, specified by their attributes. This process of putting parts together is called composition.

ATF mainly uses `TypeCatalog`, which creates a catalog from a collection of types. In ATF, you don't need to instantiate or directly access components to use them in your application. Instead, you simply list them in the `TypeCatalog` in the application's `Main()` function, followed by some standard initialization. Each of the parts listed in the catalog is instantiated by MEF, that is, an instance of each class is created during composition. In some cases, more than one instance may be created, depending on the [Part Creation Policy](#).

Topics

- [Creating a Catalog](#)
- [MEF Composition](#)
 - [Creating a CompositionContainer](#)
 - [Add Part for Main Form](#)
 - [Compose the Components \(Parts\)](#)
 - [Component Initialization](#)
 - [Disposal](#)
- [MEF Attributes](#)
 - [Export](#)
 - [Import](#)
 - [ImportingConstructor](#)
 - [Part Creation Policy](#)
 - [AllowRecomposition](#)
- [Other ComponentModel Classes](#)
 - [ExportProvider Class](#)
 - [IPartImportsSatisfiedNotification](#)

Here is a typical block of sample code setting up a MEF catalog from the `Main()` function in `Program.cs` for the [ATF Win Forms App Sample](#):

```

// Using MEF, declare the composable parts that will make up this application
TypeCatalog catalog = new TypeCatalog(
    typeof(SettingsService), // persistent settings and user preferences dialog
    typeof(CommandService), // handles commands in menus and toolbars
    typeof(ControlHostService), // docking control host
    typeof(ContextRegistry), // central context registry with change notification
    typeof(StandardFileExitCommand), // standard File exit menu command

    typeof(FileDialogService), // standard Windows file dialogs
    typeof(DocumentRegistry), // central document registry with change notification
    typeof(StandardFileCommands), // standard File menu commands for New, Open, Save,
SaveAs, Close
    typeof(AutoDocumentService), // opens documents from last session, or creates a
new document, on startup
    typeof(RecentDocumentCommands), // standard recent document commands in File menu
    typeof(MainWindowTitleService), // tracks document changes and updates main form
title
    typeof(AtfUsageLogger), // logs computer info to an ATF server
    typeof(WindowLayoutService), // service to allow multiple window layouts
    typeof(WindowLayoutServiceCommands), // command layer to allow easy switching between and
managing of window layouts

    // Client-specific plug-ins
    typeof(Editor), // editor class component that creates and saves
application documents
    typeof(SchemaLoader), // loads schema and extends types

    typeof(PythonService), // scripting service for automated tests
    typeof(ScriptConsole), // provides a dockable command console for entering
Python commands
    typeof(AtfScriptVariables), // exposes common ATF services as script variables
    typeof(AutomationService) // provides facilities to run an automated script
using the .NET remoting service
);

```

TypeCatalog's constructor lists all the types of the components the application uses, one parameter for each type.

Occasionally ATF uses other catalogs besides TypeCatalog. This code, from the StandardInteropParts class, creates a TypeCatalog and returns it as a ComposablePartCatalog type property:

```

/// Gets type catalog for all components</summary>
public static ComposablePartCatalog Catalog
{
    get
    {
        return new TypeCatalog(
            typeof(MainWindowAdapter),
            typeof(CommandServiceAdapter),
            typeof(ContextMenuService),
            typeof(DialogService),
            typeof(ControlHostServiceAdapter)
        );
    }
}

```

The [ATF Wpf App Sample](#) creates its own TypeCatalog and then uses it and the preceding ComposablePartCatalog from StandardInteropParts (plus a similar catalog from StandardViewModels) to create an AggregateCatalog:

```

protected override AggregateCatalog GetCatalog()
{
    var typeCatalog = new TypeCatalog(
        typeof(SettingsService), // persistent settings and user preferences
dialog
        typeof(CommandService), // handles commands in menus and toolbars
        typeof(ControlHostService), // docking control host
        ... //omitted types
        typeof(PythonService), // scripting service for automated tests
        typeof(ScriptConsole), // provides a dockable command console for
entering Python commands
        typeof(AtfScriptVariables), // exposes common ATF services as script
variables
        typeof(AutomationService) // provides facilities to run an automated script
using the .NET remoting service
    );

    return new AggregateCatalog(typeCatalog, StandardInteropParts.Catalog, StandardViewModels.Catalog
);
}

```

An `AggregateCatalog` is a catalog that combines elements of `ComposablePartCatalog` objects and also derives from `ComposablePartCatalog`.

The [ATF Wpf App Sample](#) uses the `AggregateCatalog` as a convenience to combine several catalogs. Later on, it uses this catalog similarly to the way other samples use `TypeCatalog`.

MEF Composition

After creating a catalog of components, samples use the catalog to initiate MEF's composition process.

Here's how [ATF Win Forms App Sample](#) does it:

```

// Create the MEF container for the composable parts
CompositionContainer container = new CompositionContainer(catalog);

// Create the main form, give it a toolStrip
ToolStripContainer toolStripContainer = new ToolStripContainer();
toolStripContainer.Dock = DockStyle.Fill;
MainForm mainForm = new MainForm(toolStripContainer);
mainForm.Text = Localizer.Localize("Sample Application");

// Create an MEF composable part from the main form, and add into container
CompositionBatch batch = new CompositionBatch();
AttributedModelServices.AddPart(batch, mainForm);
container.Compose(batch);

// Initialize components that require it. Initialization often can't be done in the constructor,
// or even after imports have been satisfied by MEF, since we allow circular dependencies
// between
// components, via the System.Lazy class. IInitializable allows components to defer some
operations
// until all MEF composition has been completed.
container.InitializeAll();

// Show the main form and start message handling. The main Form Load event provides a final
chance
// for components to perform initialization and configuration.
Application.Run(mainForm);

// Give components a chance to clean up.
container.Dispose();

```

This shows a fairly typical code sequence of how samples use the catalog and kick off the process of components discovering each other.

The following sections describe these MEF operations in detail. Some of the code here, such as creating the main form and toolbar, is not MEF-related.

Creating a CompositionContainer

A `CompositionContainer` is a container of components that manages composing the components (parts).

This line creates a `CompositionContainer` object for the catalog:

```
// Create the MEF container for the composable parts  
CompositionContainer container = new CompositionContainer(catalog);
```

Add Part for Main Form

A `CompositionBatch` is a set of parts (components) that can be added to a `CompositionContainer`. `MainForm` is also a component class. The next section of code adds the main form to the `CompositionBatch`:

```
// Create an MEF composable part from the main form, and add into container  
CompositionBatch batch = new CompositionBatch();  
AttributedModelServices.AddPart(batch, mainForm);
```

`AttributedModelServices` contains methods to assist with part composition. `AttributedModelServices.AddPart()` creates a composable part from `mainForm` and adds it to `batch`.

Compose the Components (Parts)

This is the main line of this whole sequence:

```
container.Compose(batch);
```

`Compose()` adds the parts in `batch` to the parts in the container and composes these parts, creating an object for each part and making sure they meet each other's import and export contracts.

Component Initialization

The comment here pretty well explains what's going on:

```
// Initialize components that require it. Initialization often can't be done in the constructor,  
// or even after imports have been satisfied by MEF, since we allow circular dependencies  
// between  
// components, via the System.Lazy class. IInitializable allows components to defer some  
// operations  
// until all MEF composition has been completed.  
container.InitializeAll();
```

Invoking `InitializeAll()` on the container causes the `IInitializable.Initialize()` method to be executed in every component. `InitializeAll()` is actually an extension method on `CompositionContainer` provided by ATF's `MefUtil` class. Every ATF component should implement `IInitializable`, that is, implement the `IInitializable.Initialize()` method. `InitializeAll()` performs other functions as well, described in [IInitializable](#).

Disposal

When the application terminates, this final statement releases all resources used by the `CompositionContainer`:

```
// Give components a chance to clean up.  
container.Dispose();
```

MEF Attributes

ATF uses a limited set of MEF attributes in its components. These are mainly used to specify exported and imported items. However, the Export attributes decorating ATF source are not just about exports and imports. For more information, see [Initializing Components](#).

Export

Export attributes nearly always apply to classes. That is, ATF mainly exports its component classes. Some samples export fields.

The `Export` attribute is used in the component to announce its presence to the world: its contract, which is essentially a string that importers use to find this component. It is nearly always of this form:

```
[Export(typeof(type))]
```

where `type` is the type of the component, usually its class name.

Multiple Export attributes are sometimes used, as in this example from `StandardFileExitCommand`:

```
[Export(typeof(StandardFileExitCommand))]
[Export(typeof(IInitializable))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class StandardFileExitCommand : ICommandClient, IInitializable, IPartImportsSatisfiedNotification
```

ATF often provides a variety of Export attributes on a class, one for each important interface that the class implements, and sometimes for whatever base classes there might be, too. The ATF designers try to anticipate how client code might use ATF components, so ATF provides more Exports than ATF's own code Imports. The attribute `[Export(typeof(IInitializable))]` has an additional purpose, described in [Initializing Components](#).

`[ExportMetadata()]` is not used.

Import

`[Import]` is often used, and it only rarely specifies a type, as in the `EditLabelCommand` component:

```
[Import(typeof(Sce.Atf.Applications.IContextRegistry))]
```

Nearly all imports apply to fields and constructors, and to a few properties. Imports are mainly used to connect components to each other. For more information, see [Using ATF Components](#).

Defaults are frequently allowed:

```
[Import(AllowDefault = true)]
```

This allows the importer to be set to its type's default value when a matching export could not be found. Defaults are only occasionally not permitted.

For an example of getting a component from an import, see [Getting a SettingsService Object](#).

ImportingConstructor

Note that during composition, the parameter-less constructor is used by default to instantiate the component. To make the composition engine use a different constructor, mark it with the `[ImportingConstructor]` attribute. ATF often uses this attribute.

Note also that all the constructor parameters for the constructor decorated with `[ImportingConstructor]` are in effect decorated as `[Import]`.

Part Creation Policy

This attribute is nearly always Shared:

```
[PartCreationPolicy(CreationPolicy.Shared)]
```

Shared means a single instance of the associated component is created by the `CompositionContainer` during composition and shared by all importers.

A few of these are used:

```
[PartCreationPolicy(CreationPolicy.Any)]
```

This means the most appropriate creation policy is used, and defaults to Shared.

AllowRecomposition

A few times `AllowRecomposition = true` is used, but never `AllowRecomposition = false`.

Other ComponentModel Classes

ComponentModel provides some other classes that ATF occasionally uses.

ExportProvider Class

The `System.ComponentModel.Composition.Hosting.ExportProvider` class has methods to get exported components, when it is not convenient to use an `[Import]` attribute for some reason.

For example, the [ATF Property Editing Sample](#) uses one of these methods to get references to several components in its `Program.cs` `Main()` function. These components are often used as parameters in an importing constructor, which doesn't apply for this sample.

The `ExportProvider.GetExportedValue<T>()` method gets the exported object with the contract name derived from the specified type parameter. In this example, `container` is an object in the `CompositionContainer` class, which derives from `ExportProvider`:

```
m_controlHostService = container.GetExportedValue<IControlHostService>();  
m_settingsService = container.GetExportedValue<ISettingsService>();  
m_contextRegistry = container.GetExportedValue<IContextRegistry>();
```

IPartImportsSatisfiedNotification

ATF implements this interface mainly in its classes that directly support WinForms and WPF, such as classes in the Application Shell Framework, as described in [ATF Frameworks](#).

Topics in this section

Links on this page to other topics

[ATF Code Samples](#), [ATF Frameworks](#), [ATF Property Editing Sample](#), [ATF Win Forms App Sample](#), [ATF Wpf App Sample](#), [Authoring Tools Framework](#), [Initializing Components](#), [SettingsService Component](#), [Using ATF Components](#)

Initializing Components

If a component requires initialization, it should implement and export the `IInitializable` interface. `IInitializable` and the `InitializeAll()` `CompositionContainer` extension method are used to ensure that all components requiring initialization are created and initialized.

Topics

- `IInitializable`
- `MefUtil`

`IInitializable`

The `IInitializable` interface contains the method `Initialize()`.

As previously mentioned in [Component Initialization](#), the `CompositionContainer` extension method `InitializeAll()` initializes components by calling their `IInitializable.Initialize()` method. This performs initialization on the component that can't be put in the component's constructor. Many ATF components require such initialization.

A component must have been created for it to be initialized. However, MEF creates components lazily, that is, on an as needed basis. Using `InitializeAll()` creates all the components so they can be initialized.

In ATF, any component that requires initialization implements `IInitializable` and also decorates its class with this export:

```
[Export(typeof(IInitializable))]
```

For a few components, this is the only export decorator. Obviously, this decorator's purpose is not to export the `IInitializable` interface, but is ATF's convention to mark components that need initialization. The next section explains the details of how initialization works.

`MefUtil`

The `MefUtil` class implements the `InitializeAll()` `CompositionContainer` extension method. Examining this implementation reveals how the `IInitializable` interface is used in ATF with MEF to initialize components that need it.

```

public static void InitializeAll(this CompositionContainer container)
{
    if (container == null)
        throw new ArgumentNullException("container");

    // ImportDefinition
    // - constraint: An expression that contains a Func<T, TResult> object that defines the
    conditions
    //      an Export must match to satisfy the ImportDefinition.
    // - contractName: The contract name.
    // - cardinality: One of the enumeration values that indicates the cardinality of the Export
    objects
    //      required by the ImportDefinition.
    // - ImportCardinality.ZeroOrMore: Zero or more Export objects are required by the
    ImportDefinition.
    // - isRecomposable: true to specify that the ImportDefinition can be satisfied multiple
    times
    //      throughout the lifetime of a ComposablePart object; otherwise, false.
    // - isPrerequisite: true to specify that the ImportDefinition must be satisfied before a
    //      ComposablePart can start producing exported objects; otherwise, false.
    var importDef = new ImportDefinition(constraint => true, string.Empty, ImportCardinality.ZeroOrMore
, true, true);

    try
    {
        // Create everything. This ensures that all objects are constructed before IInitializable
        // is used. Note that the order of construction is not deterministic. We have seen the
        same
        // code result in different orderings on different computers.
        foreach (Export export in container.GetExports(importDef))
        {
            // Triggers creation of object (otherwise lazy).
            // Also, IPartImportsSatisfiedNotification.OnImportsSatisfied() will be called here.
            object tmp = export.Value;
        }

        // Initialize components that require it. Initialization often can't be done in the
        constructor,
        // or even after imports have been satisfied by MEF, since we allow circular
        dependencies between
        // components, via the System.Lazy class. IInitializable allows components to defer some
        operations
        // until all MEF composition has been completed.
        foreach (IInitializable initializable in container.GetExportedValues<IInitializable>())
            initializable.Initialize();
    }
    catch (CompositionException ex)
    {
        foreach (var error in ex.Errors)
            Outputs.WriteLine(OutputMessageType.Error, "MEF CompositionException: {0}", error.Description
);

        throw;
    }
}
}

```

Analyzing this in detail, this line creates an `ImportDefinition` object:

```

var importDef = new ImportDefinition(constraint => true, string.Empty, ImportCardinality.ZeroOrMore
, true, true);

```

The first parameter returns true, so any Export satisfies this `ImportDefinition`, which is then used in the following loop:

```

foreach (Export export in container.GetExports(importDef))
{
    // Triggers creation of object (otherwise lazy).
    // Also, IPartImportsSatisfiedNotification.OnImportsSatisfied() will be called here.
    object tmp = export.Value;
}

```

`GetExports()` gets all exports that match the conditions specified in `importDef`, but this import imposed no conditions, so `GetExports()` obtains all exports in the container. That is, every component in the MEF container that has an `[Export]` attribute is obtained. The next line forces that component to be created. MEF creates objects lazily, so unless a component is imported, it might not be created otherwise.

Now that all components are created, the next lines perform any necessary initialization:

```

// Initialize components that require it. Initialization often can't be done in the constructor,
// or even after imports have been satisfied by MEF, since we allow circular dependencies
// between
// components, via the System.Lazy class. IInitializable allows components to defer some
// operations
// until all MEF composition has been completed.
foreach (IInitializable initializable in container.GetExportedValues<IInitializable>())
    initializable.Initialize();

```

`GetExportedValues<IInitializable>()` gets all the exported objects with the contract name derived from the specified type parameter, that is, all components that export `IInitializable`. It then calls `Initialize()` for each of these components.

In summary, if a component requires initialization, it should implement and export `IInitializable`. Calling `InitializeAll()` in the `Main()` function that sets up the MEF catalog ensures that each component is initialized.

Topics in this section

Links on this page to other topics

[How MEF is Used in ATF](#)

Using ATF Components

This section discusses what you need to do to use ATF components.

Add Components to Catalog

As shown in [Creating a Catalog](#), you add the components you want to a MEF catalog in your `Main()` function.

You also need to add `using` statements for the namespaces of whatever components you use, or fully qualify the component names in the catalog.

For guidance in choosing components, see [Important ATF Components](#) and [Finding ATF Components](#).

Topics

- [Add Components to Catalog](#)
- [Set Up MEF Infrastructure](#)
- [Satisfy Component Requirements](#)
 - [Discovering What's Required](#)
 - [Functionality Required From Other Components](#)
 - [Functionality Required in Application](#)

Set Up MEF Infrastructure

The section [MEF Composing](#) shows how most ATF samples provide the code that MEF requires. You can adapt this to your own application.

Note that if you are using any component that requires initialization that its constructor can't do, you must call the `InitializeAll()` extension method of `CompositionContainer`.

Satisfy Component Requirements

Most components are not self-contained and require something additional to work properly.

For some components, you need to do very little. You can simply add other components to your MEF catalog. For instance, the `StandardFileExitCommand` component adds the File > Exit menu item that exits the application. It requires the `CommandService` component, which handles commands in menus and toolbars. If you include `StandardFileExitCommand` but leave `CommandService` out of your catalog, the File > Exit menu item does not appear. In fact, your application would have no menus at all (unless you added them some other way). If a component requires other components in order to function properly, this is generally noted in the component's documentation.

Other components require the application to provide some functionality. For example, the `StandardFileCommands` component adds standard File menu commands for New, Open, Save, SaveAs, and Close. The application, in turn, must provide its own component containing code to handle these file commands: processing opened file data, displaying it, and so on. In other words, the application performs its own specialized functions for each menu command.

Discovering What's Required

To see what is needed by a component to function, look at its `Import` attributes. As mentioned, a component may require other components or require some functionality from the application itself.

If any ATF samples use a component of interest, any other components needed by that component are listed in the sample's MEF catalog.

Functionality Required From Other Components

Consider what `StandardFileExitCommand` needs by looking at its `Imports`. `StandardFileExitCommand.cs` contains three:

```
[ ImportingConstructor ]
public StandardFileExitCommand(ICommandService commandService)
...
[ Import(AllowDefault = true) ]
private IMainWindow m_mainWindow;

[ Import(AllowDefault = true) ]
private Form m_mainForm;
```

The latter two imports are common in components that work with a UI: a `System.Windows.Forms.Form` for a WinForm application, and a `Sce.Atf.Applications.IMainWindow`. `IMainWindow` abstracts the idea of a main window for the application and allows components to work with WinForms, WPF, and other UI toolkits. The `Main()` function in `Program.cs` creates a `MainForm` (derived from `Form`), which satisfies these two import requirements.

The parameter for the constructor decorated with `[ImportingConstructor]` indicates the component also imports `ICommandService`. In turn, the ATF component `CommandService` exports `ICommandService`:

```
[ Export(typeof(ICommandService))]
[ Export(typeof(IInitializable))]
[ Export(typeof(CommandService))]
[ PartCreationPolicy(CreationPolicy.Shared)]
public class CommandService : CommandServiceBase
```

Adding the `CommandService` component satisfies `StandardFileExitCommand`'s needs, so `CommandService` should be included in the MEF catalog whenever `StandardFileExitCommand` is.

Note that you can also satisfy this import by creating and adding your own component that exports `ICommandService`.

Functionality Required in Application

Now consider the `StandardFileCommands` component, which adds standard File menu commands. It stands to reason that this component would require support from the application, since these commands are clearly application-dependent. What exactly is required?

Here are `StandardFileCommands`'s imports:

```
[ ImportingConstructor ]
public StandardFileCommands(
    ICommandService commandService,
    IDocumentRegistry documentRegistry,
    IFileDialogService fileDialogService)
{
    ...
[ Import(AllowDefault = true) ]
private IStatusService m_statusService;

[ ImportMany ]
private Lazy<IDocumentClient>[] m_documentClients;
```

The constructor associated with the `[ImportingConstructor]` has three parameters. This import is very similar to the one for `StandardFileExitCommand`. It indicates components are required that export `ICommandService`, `IDocumentRegistry`, and `IFileDialogService`. These exports can be supplied by the ATF components `CommandService`, `DocumentRegistry`, and `FileDialogService`.

Similarly, the second import indicates that a component supplying `IStatusService` is needed, which is exported by the ATF component `StatusService`.

The last import indicates something the application itself must supply: something that exports `IDocumentClient`. This makes sense, because the `IDocumentClient` interface contains methods to support File commands, such as `CanOpen()` and `Open()` for the Open command. For example, the [ATF Simple DOM Editor Sample](#) uses `StandardFileCommands` and provides an `Editor` component that both exports and implements the `IDocumentClient` interface. Examining this sample shows that it also includes the ATF components listed above in its MEF catalog to meet the other import requirements.

There are two other things to note about the `IDocumentClient` import:

- An array is imported into and the attribute `[ImportMany]` is used. This allows an application to provide several implementations of `IDocumentClient` in case it handles multiple document types.
- It uses `Lazy<T>` for lazy initialization of components providing the `IDocumentClient` implementation. Such components might be large, and especially if there is more than one, all components might not be needed during an application's lifetime. Creating the

components as needed improves the application's performance and resource usage.

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [Finding ATF Components](#), [How MEF is Used in ATF](#), [Important ATF Components](#)

Creating MEF Components

Components are a fine way to add capabilities to your applications, taking advantage of the modularity that MEF components offer. In some cases, adding a component is the easiest or only way to interface with an existing ATF component. For instance, if your application uses the `ATF StandardFileCommands` component, you need to provide code that exports the `IDocumentClient` interface. The most straightforward way to do this is to follow the example of the ATF samples and create a component that implements and exports `IDocumentClient`.

This section describes what you need to provide in a component that works with ATF. You can also look at code for some of the many components in ATF and the ATF samples. For some key components in ATF, see [Important ATF Components](#).

Topics

- [Class Requirements](#)
- [Class Decorators](#)
- [Component Initialization](#)
- [Part Creation Policy](#)
- [Imports](#)
- [Deriving from Components](#)

Class Requirements

Your component is a C# class that is decorated appropriately for MEF.

For instance, here's the class declaration for `StandardFileCommands`:

```
public class StandardFileCommands : IDocumentService, ICommandClient, IInitializable
```

Your class should be `public`.

Class Decorators

Here are the decorators for the `StandardFileCommands` class:

```
[Export(typeof(IDocumentService))]
[Export(typeof(StandardFileCommands))]
[Export(typeof(IInitializable))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class StandardFileCommands : IDocumentService, ICommandClient, IInitializable
```

Note that there are several `[Export(typeof(xxx))]` attributes. ATF components often provide a variety of export attributes on a component, one for each important interface that the component implements, and sometimes for whatever base classes there might be, too. This is a good model to follow. Try to anticipate how client code might use the component, so provide more exports than your own code imports.

Exports fall in several categories:

- [Implemented Interfaces](#) (`[Export(typeof(IDocumentService))]`). You may want to export whatever important interfaces the component implements.
- [Base classes](#) (`[Export(typeof(StandardFileCommands))]`). This includes the class itself and perhaps others.
- [IInitializable](#) (`[Export(typeof(IInitializable))]`). Export this if the component needs initialization. For more details, see [Component Initialization](#)

Your class must have at least one export.

Component Initialization

Your component may need to finish its initialization after it has been instantiated or MEF has completed composition of the components. This is often the case when the component depends on other components.

If the component requires initialization that its constructor cannot perform, you should implement the `IInitializable` interface. You must

also export it, decorating the class:

```
[Export(typeof(IInitializable))]
```

Place the necessary initialization code in the implementation of the `IInitializable.Initialize()` method in the component.

Note that an application's MEF set up code must call the `InitializeAll()` extension method of `CompositionContainer` to actually perform this initialization.

For more information, see [Initializing Components](#).

Part Creation Policy

How many instances of the component need to be created? Generally, only one is needed, shared between the importers, so the creation policy should be `CreationPolicy.Shared`, which is what nearly all ATF components use. If a component is required for each importer, use `CreationPolicy.NonShared`. If the part can be shared or not, use `CreationPolicy.Any`; this is used when no policy is specified.

Imports

If your component depends on other components, you need to import items.

Sometimes you can import at least some of what you need in the constructor, as in this example from `StandardFileCommands`:

```
[ImportingConstructor]
public StandardFileCommands(
    ICommandService commandService,
    IDocumentRegistry documentRegistry,
    IFileDialogService fileDialogService)
```

Recall that using `[ImportingConstructor]` imports each constructor parameter. Create the appropriate fields for the imported parameters' values.

You can also import directly into a field if it is not a constructor parameter.

Be sure to use `[ImportMany]` on some kind of collection variable if you can import several instances of something, such as an `IDocumentClient` interface, one for each document client the application supports.

Deriving from Components

You can create a component by deriving from an existing component and supplying a similar set of Export attributes.

For example, the [ATF Property Editing Sample](#) contains a `CustomGridPropertyEditor` component that derives from the `GridPropertyEditor` component:

```
[Export(typeof(GridPropertyEditor))]
[Export(typeof(IInitializable))]
[PartCreationPolicy(CreationPolicy.Any)]
public class CustomGridPropertyEditor : GridPropertyEditor
```

This new component is decorated with exactly the same attributes as `GridPropertyEditor`, as its declaration shows:

```
[Export(typeof(GridPropertyEditor))]
[Export(typeof(IInitializable))]
[PartCreationPolicy(CreationPolicy.Any)]
public class GridPropertyEditor : IDisposable, IControlHostClient, IInitializable
```

You can then customize the new component for your application. For instance, the [ATF Property Editing Sample](#) overrides the derived from component's methods and also overrides methods of classes that `GridPropertyEditor` uses.

For more details on this example, see [Customizing Property Editor Components](#).

Links on this page to other topics

[ATF Property Editing Sample](#), [Authoring Tools Framework](#), [Important ATF Components](#), [Initializing Components](#), [Property Editing](#)

[Programming Discussion](#)

Important ATF Components

This section describes ATF components that provide capabilities needed by many applications, such as common menus and menu items. Each of these components is used in at least one ATF sample, and some components, such as `ControlHostService`, are used in nearly all samples.

You can also create your own components that perform similar functions and export the same types as these components. You can write the component from scratch, or derive from an existing component, as described in [Deriving from Components](#).

Topics

- [ATF Components](#)
 - [Application Shell Framework](#)
 - [Document Framework](#)
 - [Context Framework](#)
 - [Property Editing Framework](#)
 - [Instancing Framework](#)
 - [Commands](#)
 - [Window Management](#)
 - [Output, Scripts, and Logging](#)
- [ATF Samples Components](#)

ATF Components

These components are all in the `Sce.Atf.Applications` namespace, unless otherwise noted.

Application Shell Framework

- `ControlHostService`: provide a dock in which to hold controls, such as menus and toolbars. Its interface is `IControlHostService`. The docking control allows moving windows around on the dock, and the window configuration can be saved. For details, see [ControlHostService Component](#).
- `CommandService`: display currently available application commands to the user in menus and toolbars and provide keyboard shortcuts. Its interface is `ICommandService`. For more information, see [CommandService Component](#).
- `StatusService`: add a status bar at the bottom of the main form. Its interface is `IStatusService`. For details, see [StatusService Component](#).
- `SettingsService`: manage user-editable settings (preferences) and the persistence of these application settings. Its interface is `ISettingsService`. For more information, see [SettingsService Component](#).

For more information on the application shell, see [ATF Application Basics and Services](#).

Document Framework

- `StandardFileCommands`: implement File menu commands that modify the document registry: New, Open, Save, SaveAs, Save All, and Close. New and Open commands are created for each component that exports `IDocumentClient` in the application.
- `AutoDocumentService`: open documents from the previous application session or create a new empty document when an application starts.
- `RecentDocumentCommands`: add recently opened documents to the application's File > Recent Documents submenu.
- `DocumentRegistry`: hold documents, provide notifications when a document is added or removed, track the most recently active document and the open document contexts, and filter by document type. It implements `IDocumentRegistry`.
- `MainWindowTitleService`: update the main dialog's title to reflect the current document and its state, i.e., modified or not.
- `FileDialogService`: provide standard file dialogs for an application, using the ATF `OpenFileDialog`, `SaveFileDialog`, and `ConfirmationDialog` classes. Required by `StandardFileCommands`.

For more information on the document framework, see [Documents in ATF](#).

Context Framework

- `ContextRegistry`: tracks the active contexts in the application. It implements `IContextRegistry`.

For details on working with contexts, see [ATF Contexts](#).

Property Editing Framework

- **PropertyEditor**: edit object values and attributes using the ATF `PropertyGrid`, which is a two-column editing control. For more information, see [PropertyEditor Component](#).
- **GridPropertyEditor**: edit object values and attributes using the ATF `GridControl`, an ATF spreadsheet-style editing control. For more information, see [Grid Property Editor Component](#).
- **PropertyEditingCommands**: provide the property editing commands `Reset All` and `Reset Current` in a context menu that can be used inside property editor controls like `PropertyGrid`. `PropertyEditingCommands` can be used with ATF property editors or with custom property editors.

For more information about property editing, see [Property Editing in ATF](#). For details on the ATF Property Editing Sample, see [Property Editing Programming Discussion](#).

Instancing Framework

- **StandardEditCommands**: provide the Edit menu commands Cut, Copy, Paste, and Delete, using the `IInstancingContext` and (if available) `ITransactionContext` interfaces to perform the operations. For more details, see [StandardEditCommands and Instancing](#).
- **StandardEditHistoryCommands**: implement Edit menu Undo and Redo commands.
- **StandardSelectionCommands**: add standard selection commands in Edit menu: Select All, Deselect All, and Invert Selection.

For information about instancing, see [Instancing In ATF](#).

Commands

- **StandardFileExitCommand**: add the File > Exit command that closes the application's main form.
- **StandardPrintCommands**: add standard printing commands to File menu: Print, PageSetup, and PrintPreview.
- **StandardViewCommands**: add the standard viewing commands on contexts implementing the `IViewingContext` interface.

For information on creating and working with commands, see [Commands in ATF](#).

Window Management

- **WindowLayoutService**: save application window layouts.
- **WindowLayoutServiceCommands**: commands for handling application window layouts in the Window > Layouts submenu: Save Layout As, Manage Layouts, and Default.
- **PaletteService**: manage the global palette of objects that can be dragged onto other controls.
- **TabbedControlSelector**: enable the user to switch control focus using the Ctrl+Tab keyboard command, similar to Visual Studio, Windows, or any tabbed Internet browser application.
- **DefaultTabCommands**: provide the default commands related to document tab Controls, such as those that appear in the context menu when right-clicking on a tab.

Output, Scripts, and Logging

- **OutputService**: display text output to the user in a `RichTextBox`. You should also include `Outputs` component when you use `OutputService`.
- **Outputs**: provide static methods for easy access to the output message facilities in the application. In `Sce.Atf` namespace.
- **ErrorDialogService**: display error messages to users in an error dialog.
- **ScriptingService**: abstract base class for components that can expose C# objects to a scripting language, such as Python.
- **AtfScriptVariables**: expose common ATF services as script variables, so that the given `ScriptingService` can easily use these ATF services.
- **ScriptConsole**: provide a dockable command console for entering Python commands and import many common .NET and ATF types into the Python namespace.
- **PythonService**: scripting service for automated tests.
- **AutomationService**: provide facilities to run an automated script using the .NET remoting service.
- **CrashLogger**: log unhandled exceptions to a remote server. If used with `UnhandledExceptionService` component, put CrashLogger before `UnhandledExceptionService` in the `TypeCatalog`. In `Sce.Atf` namespace.
- **UnhandledExceptionService**: catch all unhandled exceptions from the UI thread and present the user with the option of continuing the application so work can be saved.
- **HelpAboutCommand**: add the Help > About command, which displays a dialog box with a description of the application plus the ATF version number. Applications should derive from this component.

ATF Samples Components

Samples also implement components applications can use or adapt. Each component lists the samples in which it appears.

- **DomTypes**: Use the `DomNodeType`, `ChildInfo`, and `AttributeInfo` DOM metadata classes to describe an application's

Document Object Model (DOM). Normally these are defined by a schema file. This component is in the [ATF Simple DOM No XML Editor Sample](#).

- **GroupingCommands:** Define circuit-specific commands for group and ungroup. Grouping takes modules and the connections between them and turns them into a single element that is equivalent. [ATF Circuit Editor Sample](#).
- **LayeringCommands:** Component to add an Add Layer command to an application. [ATF Circuit Editor Sample](#).
- **MasteringCommands:** Component that defines circuit-specific commands for Master and Unmaster commands. Mastering creates a new type of circuit element from a subcircuit. Any changes to this master element affect all its instances. [ATF Circuit Editor Sample](#).
- **ModelViewer:** View a 3D model. [ATF Model Viewer Sample](#).
- **ModulePlugin:** Add module types to an editor. This class adds sample audio modules. [ATF Circuit Editor Sample](#).
- **PaletteClient:** Populate a palette with types, such as basic Finite State Machine (FSM) types. [ATF Diagram Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF Simple DOM No XML Editor Sample](#), and [ATF State Chart Editor Sample](#).
- **RenderView:** Render a 3D scene from the active document. [ATF Model Viewer Sample](#).
- **ResourceListEditor:** Register a slave ListView control to display and edit resources that belong to the most recently selected event. It handles drag and drop and right-click context menus for the ListView control. [ATF Simple DOM Editor Sample](#) and [ATF Simple DOM No XML Editor Sample](#).
- **SchemaLoader:** Load the user interface schema, register data extensions on the DOM types, and annotate the types with display information and property descriptors. [ATF Circuit Editor Sample](#), [ATF Diagram Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [ATF Win Forms App Sample](#), and [ATF Wpf App Sample](#).
- **SourceControlContext:** Implement a context for source control. [ATF Code Editor Sample](#).
- **TreeLister:** Display a tree view of user interface data. Use the context registry to track the active UI data as documents are opened and closed. [ATF DOM Tree Editor Sample](#).

Topics in this section

Links on this page to other topics

[ATF Application Basics and Services](#), [ATF Circuit Editor Sample](#), [ATF Code Editor Sample](#), [ATF Contexts](#), [ATF Diagram Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Model Viewer Sample](#), [ATF Property Editing Sample](#), [ATF Simple DOM Editor Sample](#), [ATF Simple DOM No XML Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [ATF Win Forms App Sample](#), [ATF Wpf App Sample](#), [Commands in ATF](#), [CommandService Component](#), [ControlHostService Component](#), [Creating MEF Components](#), [Documents in ATF](#), [Instancing In ATF](#), [Property Editing in ATF](#), [Property Editing Programming Discussion](#), [Property Editor Components](#), [SettingsService Component](#), [StandardEditCommands and Instancing](#), [StatusService Component](#)

Finding ATF Components

This section discusses how to find existing ATF components that fulfill your application's needs. If a component does not precisely fit, you can modify it.

You can start by looking at [Important ATF Components](#).

You can also look at [ATF Code Samples](#) for samples that have capabilities in common with your application to see what components they have implemented. For example, several samples implement a `PaletteClient` that populates a palette with types, a common application task. [ATF Samples Components](#) also lists some useful sample components.

Finding Components in ATF

Every ATF component exports at least one type, so you can find all ATF components by searching for "[Export(typeof(" in Visual Studio. Performing this search on the ATF source in the `Frameworks` folder yields over 300 items like this:

```
C:\SonyDev\SVN\wws_sdk_trunk\components\wws_atf\Framework\Atf.Atgi\AtgiResolver.cs(18): [  
Export(typeof(AtgiResolver))]  
C:\SonyDev\SVN\wws_sdk_trunk\components\wws_atf\Framework\Atf.Atgi\AtgiResolver.cs(19): [  
Export(typeof(IInitializable))]  
C:\SonyDev\SVN\wws_sdk_trunk\components\wws_atf\Framework\Atf.Atgi\AtgiResolver.cs(20): [  
Export(typeof(IResourceResolver))]  
C:\SonyDev\SVN\wws_sdk_trunk\components\wws_atf\Framework\Atf.Core\AtfUsageLogger.cs(26): [  
Export(typeof(IInitializable))]  
C:\SonyDev\SVN\wws_sdk_trunk\components\wws_atf\Framework\Atf.Core\ConsoleOutputWriter.cs(10): [  
[ Export(typeof(IOutputWriter))]
```

This list provides some information about the component: its file location and the type it exports. The file location gives you the component name and what general category it falls in. You can now search this list for terms related to the capability you are looking for. For instance, you could search for "file", "curve", "label", or "property".

Topics in this section

Links on this page to other topics

[ATF Code Samples](#), [Important ATF Components](#)

ATF Application Basics and Services

This section discusses the basic structure of an ATF-based application, for both WinForms and WPF. It also discusses the Application Shell Framework, which consists of the core application services and interfaces needed for applications with a GUI.

- **WinForms Application:** Show a `Main()` function pattern common in WinForms application ATF samples and what it accomplishes.
 - **WPF Application:** Discuss how WPF applications differ from WinForms ones in basic application structure.
 - **ControlHostService Component:** This component is responsible for exposing client Controls in the application's main form.
 - **CommandService Component:** Show the basics of supporting commands in ATF.
 - **SettingsService Component:** This component manages preferences and their persistence.
 - **StatusService Component:** Show how to use this component to display status messages.
 - **Other ATF Services:** Describe other services: document management, file dialogs, Help dialog, and resources.
-

WinForms Application

WinForms applications in ATF can have a `Main()` function with a standard, simple structure, especially for applications with a GUI. Most of the [ATF Code Samples](#) have a structure similar to that described in this section.

ATF applications with a `Form` typically use a `MainForm` class derived from `System.Windows.Forms`, which allows ATF to have more control over the `Form`.

Topics

- [Initial Setup](#)
- [Application Methods](#)
- [Localization Support](#)
- [Metadata-Driven Property Editing](#)
- [MEF Processing](#)
- [MainForm Creation and Running the Application](#)

Initial Setup

This initial section of code is from the [ATF FSM Editor Sample](#):

```
static void Main()
{
    // It's important to call these before starting the app; otherwise theming and bitmaps
    // may not render correctly.
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.DoEvents() // see
    http://www.codeproject.com/buglist/EnableVisualStylesBug.asp?df=100&forumid=25268&exp=0&select=984714

    // Set up localization support early on, so that user-readable strings will be localized
    // during the initialization phase below. Use XML files that are embedded resources.
    Thread.CurrentThread.CurrentCulture = System.Globalization.CultureInfo.CurrentCulture;
    Localizer.SetStringLocalizer(new EmbeddedResourceStringLocalizer());

    // Enable metadata driven property editing for the DOM
    DomNodeType.BaseOfAllTypes.AddAdapterCreator(new AdapterCreator<CustomTypeDescriptorNodeAdapter
>());
}
```

Application Methods

The `Application` class methods called here are used by many applications to provide general Windows capabilities.

The comment on `DoEvents()` notes that this call must occur early to avoid problems with Controls for certain Windows versions.

Localization Support

These methods are called early so that user-readable strings are localized correctly in soon to follow initialization code.

Metadata-Driven Property Editing

If the application uses the ATF DOM, the call to `AddAdapterCreator` ensures that you can edit the properties of data types defined in the DOM for the application data. This statement (used in several samples) adds a DOM adapter to allow all DOM nodes to adapt to an appropriate property descriptor, if any. The adapter is called on `BaseOfAllTypes`, so it applies to all types.

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 details, see "Defining DOM Extensions and Adapters" in ATF Programmer's Guide: Document Object Model (DOM). For information

on property editors and descriptors, see [Property Editing in ATF](#).

MEF Processing

This MEF handling is from the [ATF Target Manager Sample](#). It includes part of the application getting its main form, because this object is also used by MEF.

```
var catalog = new TypeCatalog(
    typeof(SettingsService),           // persistent settings and user preferences dialog
    typeof(FileDialogService),         // provides standard Windows file dialogs, to let the
user open and close files. Used by SkinService.
    typeof(SkinService),              // allows for customization of an application's
appearance by using inheritable properties that can be applied at run-time
    typeof(StatusService),            // status bar at bottom of main Form
    typeof(CommandService),           // handles commands in menus and toolbars
    typeof(ControlHostService),        // docking control host

    typeof(StandardFileExitCommand),   // standard File exit menu command
    typeof(HelpAboutCommand),          // Help -> About command
    typeof(AtfUsageLogger),            // logs computer info to an ATF server
    typeof(CrashLogger),              // logs unhandled exceptions to an ATF server

    typeof(ContextRegistry),           // component that tracks application contexts; needed for
context menu

    // add target info plugins and TargetEnumerationService
    typeof(Deci4pTargetProvider),      // provides information about development devices
available via Deci4p
    typeof(TcpIpTargetProvider),       // provides information about development devices
available via TCP/IP
    typeof(TargetCommands),            // commands to operate on currently selected targets.
    typeof(TargetEnumerationService)   // queries and enumerates target objects, consuming
target providers created by the application
);

// Set up the MEF container with these components
var container = new CompositionContainer(catalog);

// Configure the main Form
var batch = new CompositionBatch();
var mainForm = new MainForm(new ToolStripContainer());
mainForm.Text = @"ATF TargetManager Sample";

// Add the main Form instance to the container
AttributedModelServices.AddPart(batch, mainForm);
container.Compose(batch);

...
// Initialize components that require it. Initialization often can't be done in the constructor,
// or even after imports have been satisfied by MEF, since we allow circular dependencies
between
// components, via the System.Lazy class. IInitializable allows components to defer some
operations
// until all MEF composition has been completed.
container.InitializeAll();

// Show the main form and start message handling. The main Form Load event provides a final
chance
// for components to perform initialization and configuration.
Application.Run(mainForm);

// Give components a chance to clean up.
container.Dispose();
```

Briefly, this code block creates a `TypeCatalog` listing all the ATF components used. This catalog is used to create a `CompositionContainer`. The new `MainForm` is added to a new `CompositionBatch`, which is then used to compose the MEF components. `InitializeAll()` initializes the MEF components. When the application ends, `Dispose()` performs final MEF cleanup on the container. For a more detailed explanation of these operations, see [How MEF is Used in ATF](#).

MainForm Creation and Running the Application

The previous code sample showed creating the `MainForm` as well as MEF manipulation. Here is a similar segment from the [ATF Win Forms App Sample](#), focusing on the application rather than MEF:

```
// Create the main form, give it a toolStrip
ToolStripContainer toolStripContainer = new ToolStripContainer();
toolStripContainer.Dock = DockStyle.Fill;
MainForm mainForm = new MainForm(toolStripContainer);
mainForm.Text = Localizer.Localize("Sample Application");

// Create an MEF composable part from the main form, and add into container
CompositionBatch batch = new CompositionBatch();
AttributedModelServices.AddPart(batch, mainForm);
container.Compose(batch);

// Initialize components that require it. Initialization often can't be done in the constructor,
// or even after imports have been satisfied by MEF, since we allow circular dependencies
// between
// components, via the System.Lazy class. IInitializable allows components to defer some
// operations
// until all MEF composition has been completed.
container.InitializeAll();

// Show the main form and start message handling. The main Form Load event provides a final
// chance
// for components to perform initialization and configuration.
Application.Run(mainForm);
```

A `System.Windows.Forms.ToolStripContainer` is created and its docking style is set.

A new `MainForm` is constructed with the `ToolStripContainer` and the `MainForm` object's `Text` property is set.

`MainForm` is also a MEF component, so it can be added to the `CompositionBatch` by `AttributedModelServices.AddPart()`.

The final operation needed is to call `Application.Run()` with the `MainForm` object to run the application.

Topics in this section

Links on this page to other topics

[ATF Code Samples](#), [ATF FSM Editor Sample](#), [ATF Target Manager Sample](#), [ATF Win Forms App Sample](#), [How MEF is Used in ATF](#), [Property Editing in ATF](#)

WPF Application

WinForms and WPF applications use ATF in similar ways. This section discusses their differences in basic application structure. The other sections in [ATF Application Basics and Services](#) also mention similarities and differences between WinForms and WPF.

For an example of a WPF ATF application, see the [ATF Wpf App Sample](#). For a WinForms application that shares code with this WPF sample and helps clarify the similarities and differences between the two, see the [ATF Win Forms App Sample](#).

Topics

- [App.xaml](#)
- [AtfApp Class](#)
- [MEF Setup](#)
- [Application Exit](#)

App.xaml

`App.xaml` is the XAML file in which the WPF application is set up:

```
<atf:AtfApp x:Class="WpfApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:atf="http://www.sce.net/Atf.Gui.Wpf">
</atf:AtfApp>
```

This partially defines the class `WpfApp.App`, which is derived from the ATF class `AtfApp`. The rest of the `WpfApp.App` class definition is in the C# file `App.xaml.cs`, which sets up the MEF catalog, as discussed in [MEF Setup](#).

AtfApp Class

The `AtfApp` class is the base ATF WPF application class and derives from `System.Windows.Application`. You should derive WPF applications from this class to avoid rewriting common code. `AtfApp` implements `IComposer`, which is an interface for `ComposablePart` and `CompositionContainer` used for WPF applications with MEF.

`AtfApp`'s constructor performs some of the same set up functions as the WinForms `Main()` function discussed in [Initial Setup](#):

```
public AtfApp()
{
    // Setup thread name and culture
    Thread.CurrentThread.CurrentCulture = System.Globalization.CultureInfo.CurrentCulture;
    Thread.CurrentThread.Name = "Main";

    // Ensure the current culture passed into bindings is the OS culture.
    // By default, WPF uses en-US as the culture, regardless of the system settings.
    FrameworkElement.LanguageProperty.OverrideMetadata(
        typeof(FrameworkElement), new FrameworkPropertyMetadata(
            XmlLanguage.GetLanguage(System.Globalization.CultureInfo.CurrentCulture.IetfLanguageTag
        )));
}
```

Much of `AtfApp` is dedicated to composing components with MEF, which is discussed next in [MEF Setup](#).

MEF Setup

If a WPF application uses MEF, it needs to specify the components it uses. This section discusses a way to do this using `AtfApp`.

`AtfApp` has an `OnStartup()` method that is called when a WPF application starts up:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);

    if (Compose())
    {
        OnCompositionBeginning();

        foreach (var initializable in m_initializables)
            initializable.Value.Initialize();

        OnCompositionComplete();
    }

    m_mainWindow.ShowMainWindow();
}
else
{
    Shutdown();
}
```

OnStartup() calls the Compose() method to compose the MEF components. In turn, Compose() calls the abstract method GetCatalog(), which actually sets up the MEF catalog. OnStartup() also initializes all the MEF components using the ATF IIInitialize.Initialize() method.

The ATF application should provide a GetCatalog() implementation that sets up the MEF catalog. Here's how the ATF Wpf App Sample does it in App.xaml.cs, defined in the partial class App:

```

public partial class App : AtfApp
{
    protected override AggregateCatalog GetCatalog()
    {
        var typeCatalog = new TypeCatalog(
            typeof(SettingsService), // persistent settings and user preferences
            dialog
                typeof(CommandService), // handles commands in menus and toolbars
                typeof(ControlHostService), // docking control host
                typeof(AtfUsageLogger), // logs computer info to an ATF server
                typeof(CrashLogger), // logs unhandled exceptions to an ATF server
                typeof(UnhandledExceptionService), // catches unhandled exceptions, displays
            info, and gives user a chance to save
                typeof(ContextRegistry), // central context registry with change
            notification
                typeof(StandardFileExitCommand), // standard File exit menu command
                typeof(FileDialogService), // standard Windows file dialogs
                typeof(DocumentRegistry), // central document registry with change
            notification
                typeof(StandardFileCommands), // standard File menu commands for New, Open,
            Save, SaveAs, Close
                typeof(AutoDocumentService), // opens documents from last session, or
            creates a new document, on startup
                typeof(RecentDocumentCommands), // standard recent document commands in File
            menu
                typeof(MainWindowTitleService), // tracks document changes and updates main
            form title
                typeof(WindowLayoutService), // service to allow multiple window layouts
                typeof(WindowLayoutServiceCommands), // command layer to allow easy switching
            between and managing of window layouts
                typeof(SchemaLoader), // loads schema and extends types
                typeof(MainWindow), // main app window (analog to 'MainForm' in
            WinFormsApp)
                typeof(Editor), // Sample editor class that creates and saves
            application documents
                typeof(PythonService), // scripting service for automated tests
                typeof(ScriptConsole), // provides a dockable command console for
            entering Python commands
                typeof(AtfScriptVariables), // exposes common ATF services as script
            variables
                typeof(AutomationService) // provides facilities to run an automated
            script using the .NET remoting service
        );
    }

    return new AggregateCatalog(typeCatalog, StandardInteropParts.Catalog, StandardViewModels.Catalog);
}
}

```

The `GetCatalog()` method returns an `AggregateCatalog`, which contains the `TypeCatalog` with the components the sample application uses — as well as a couple of `ComposablePartCatalog` objects containing WPF versions of ATF components that are generally useful for WPF applications: `StandardInteropParts.Catalog` and `StandardViewModels.Catalog`. For more information on `StandardInteropParts`, see [Using WinForms Controls in WPF](#).

Application Exit

`AtfApp` also provides a `OnExit()` called when the WPF application exits. `OnExit()` disposes of MEF components' resources. You can override it to perform additional clean up. Here is the base method:

```

protected override void OnExit(ExitEventArgs e)
{
    base.OnExit(e);

    if (Container != null)
    {
        Container.Dispose();
    }
}

```

Topics in this section

Links on this page to other topics

[ATF Application Basics and Services](#), [ATF Win Forms App Sample](#), [ATF Wpf App Sample](#), [Authoring Tools Framework](#), [Using WinForms Controls in WPF, WinForms Application](#)

ControlHostService Component

The ControlHostService component is responsible for exposing client controls in the application's main form. The control host service allows the client to specify a preferred location for the main form, and calls the client back when the control gets or loses focus, or is closed by the user. This component also provides a dock in which to hold controls, such as menus and tool strips. The docking control allows moving windows around on the dock. The window configuration can be saved, using the components WindowLayoutService and WindowLayoutServiceCommands.

Topics

- [Uses and Interfaces](#)
- [Registering a Control](#)

Uses and Interfaces

ControlHostService implements the IControlHostService interface. IControlHostService offers methods similar to ICommandService. IControlHostService offers methods to register, unregister, and show controls, instead of commands.

ControlHostService also implements IControlRegistry, which tracks active controls, noting when controls are activated or deactivated. For more information, see [Using Controls in ATF](#).

The ControlHostService component, or one like it, is useful to any application that has a GUI and is used by nearly all the ATF samples. Because it is a component, applications simply need to add ControlHostService to the MEF TypeCatalog. Applications also need to add the CommandService and SettingsService components, because ControlHostService imports ICommandService and ISettingsService.

For WinForms, ControlHostService also imports IContextMenuCommandProvider, which is used for getting a list of commands to present on a context menu. IContextMenuCommandProvider is imported using an [ImportMany] attribute, and you don't have to provide such an import for ControlHostService to work. This import is provided in case there are any controls in the application that have context menus.

For more information on satisfying components' import needs, see [Using ATF Components](#).

There are versions of ControlHostService and IControlHostService for both WinForms and WPF.

For more information, see [Using Controls in ATF](#).

Registering a Control

Here is a simple example of creating and registering a control from the PaletteService component. The call to RegisterControl() uses the ControlInfo class, which describes the control in the UI. This class is analogous to the CommandInfo class for commands, and it provides information on the control, such as a text description ("Palette").

```
m_control = new UserControl();
m_control.Dock = DockStyle.Fill;
m_control.SuspendLayout();
m_control.Name = "Palette".Localize();
m_control.Text = "Palette".Localize();
m_control.Controls.Add(m_searchInput);
m_control.Controls.Add(TreeControl);
m_control.Layout += controls_Layout;
m_control.ResumeLayout();

m_controlHostService.RegisterControl(
    m_control,
    new ControlInfo(
        "Palette".Localize(),
        "Creates new instances".Localize(),
        StandardControlGroup.Left),
    this);
```

To learn more, see [Registering Controls](#).

Topics in this section

Links on this page to other topics

[Registering Controls](#), [Using ATF Components](#), [Using Controls in ATF](#)

CommandService Component

A command is an action that is invoked from a menu item or tool strip button. You can create menus, menu items, and tool strip buttons and associate them with commands. The command specifies its appearance and behavior in the UI: text, icon, keyboard shortcut, and actual command action.

CommandService creates standard menus, displays the currently available application commands to the user in menus and tool strips, displays context menus, and processes keyboard shortcuts.

This component, or one like it, is useful to any application that activates commands through menus and tool strips and is used by nearly all the ATF samples. Because it is a component, applications simply need to add CommandService to the MEF TypeCatalog. The ATF standard command components, such as StandardEditCommands or StandardFileCommands, require CommandService to work. In fact, if you left out CommandService, your application would have no menus at all, unless you added them some other way.

For WinForms, applications also need to use the StatusService and SettingsService components, because CommandService imports IStatusService and ISettingsService.

For more information on satisfying components' import needs, see [Using ATF Components](#).

CommandService implements the `ICommandService` interface. `ICommandService` offers methods similar to `IControlHostService`. `ICommandService` offers methods to register and unregister menus and commands, instead of controls. The related interface `ICommandClient` contains methods to determine whether a command can be performed, update the command's menu, and actually perform the command.

There are versions of CommandService and ICommandService for both WinForms and WPF.

For more information about CommandService and commands, see [Commands in ATF](#).

Topics in this section

Links on this page to other topics

[Commands in ATF](#), [Using ATF Components](#)

SettingsService Component

The `SettingsService` component manages user-editable settings (preferences) and the persistence of these application settings. You can specify which settings to persist, and they are saved in an XML file. `SettingsService` implements the `ISettingsService` interface.

This component, or one like it, is useful to any application with preferences and is used by nearly all the ATF samples. To use it, add `SettingsService` to the MEF `TypeCatalog` of the application. Applications also need to add the `CommandService` component, because `SettingsService` imports `ICommandService`. After `SettingsService` is added, you can specify settings you want saved.

`SettingsService` is required by other components in the Application Shell: `ControlHostService` and `CommandService`. `ControlHostService` uses it to save the state of the dock panel and whether the UI is locked. `CommandService` saves keyboard shortcuts and the command icon size.

`SettingsService` does not interact with the rest of an application's GUI, so it can be used with both WinForms and WPF. It does present a dialog allowing a user to edit preferences.

Topics

- [ISettingsService Interface](#)
- [Getting a SettingsService Object](#)
- [Specifying Saved Settings](#)
- [Displaying User Settings](#)

ISettingsService Interface

The `ISettingsService` interface allows registering the settings you want to preserve and present them to the user with these methods:

- `RegisterSettings()`: Register all the settings you want to preserve, including user-editable ones.
- `RegisterUserSettings()`: Register the settings you allow a user to change.
- `PresentUserSettings()`: Present the settings to a user for editing.

Each setting is described by a `PropertyDescriptor`, and the registration methods take a `PropertyDescriptor` array parameter.

Note the following:

- Registering a setting adds it to the saved settings; it does not replace the existing set.
- Calling `RegisterUserSettings()` only indicates that the setting is to be user-visible; you must call `RegisterSettings()` for this setting to persist it.

Getting a SettingsService Object

You need a `SettingsService` object on which to invoke the `ISettingsService` methods to register settings. You get this by using MEF to import the `SettingsService` component you added to the application.

For example, the [ATF Timeline Editor Sample](#) uses an `[ImportingConstructor]` attribute to import the `SettingsService` object in the constructor parameter `settingsService`, which is then saved in the field `m_settingsService`:

```
[ImportingConstructor]
public TimelineEditor(
    IControlHostService controlHostService,
    ICommandService commandService,
    IContextRegistry contextRegistry,
    IDocumentRegistry documentRegistry,
    IDocumentService documentService,
    IPaletteService paletteService,
    ISettingsService settingsService)
{
    ...
    m_settingsService = settingsService;
```

For more information on using MEF exporting and importing, see [MEF Attributes](#).

Specifying Saved Settings

This example from the [ATF Timeline Editor Sample](#) creates property descriptors for the settings to be saved, and then registers these settings.

```
var settings = new BoundPropertyDescriptor[] {
    new BoundPropertyDescriptor(typeof(D2dTimelineRenderer),
        () => D2dTimelineRenderer.GlobalHeaderWidth,
        "Header Width", "Appearance", "Width of Group/Track Header"),
    new BoundPropertyDescriptor(typeof(D2dTimelineRenderer),
        () => D2dTimelineRenderer.GlobalKeySize, "Key Size", "Appearance", "Size of Keys"),
    new BoundPropertyDescriptor(typeof(D2dTimelineRenderer),
        () => D2dTimelineRenderer.GlobalMajorTickSpacing, "Major Tick Spacing", "Appearance",
        "Pixels between major ticks"),
    new BoundPropertyDescriptor(typeof(D2dTimelineRenderer),
        () => D2dTimelineRenderer.GlobalPickTolerance, "Pick Tolerance", "Behavior", "Picking
        tolerance, in pixels"),
    new BoundPropertyDescriptor(typeof(D2dTimelineRenderer),
        () => D2dTimelineRenderer.GlobalTrackHeight, "Track Height", "Appearance", "Height of
        track, relative to units of time"),

    //manipulator settings
    new BoundPropertyDescriptor(typeof(D2dSnapManipulator), () => D2dSnapManipulator.SnapTolerance
    , "Snap Tolerance", "Behavior",
        "The maximum number of pixels that a selected object will be snapped"),
    new BoundPropertyDescriptor(typeof(D2dSnapManipulator), () => D2dSnapManipulator.Color, "Snap
    Indicator Color", "Appearance",
        "The color of the indicator to show that a snap will take place"),
    new BoundPropertyDescriptor(typeof(D2dScaleManipulator), () => D2dScaleManipulator.Color,
    "Scale Manipulator Color", "Appearance",
        "The color of the scale manipulator");
};

m_settingsService.RegisterUserSettings("Timeline Editor", settings);
m_settingsService.RegisterSettings(this, settings);
```

Note that the `BoundPropertyDescriptor` array is used as a parameter in both `RegisterUserSettings()` and `RegisterSettings()` calls. The `m_settingsService` field was set to a `SettingsService` object as shown previously in [Getting a SettingsService Object](#).

This shorter example from the `ControlHostService` component illustrates that you can call `RegisterSettings()` repeatedly to add settings:

```
m_settingsService.RegisterSettings(this,
    new BoundPropertyDescriptor(this, () => DockPanelState, "DockPanelState", null, null));
m_settingsService.RegisterSettings(this,
    new BoundPropertyDescriptor(this, () => UILocked, "UILocked", null, null));
```

Displaying User Settings

`SettingService` implements the `PresentUserSettings()` method to show the settings to a user for editing. In addition, it registers and implements a command to display the dialog. The command is activated by the `Edit > Preferences` menu item. Preferences are displayed in a tree list editor.

Topics in this section

Links on this page to other topics

[ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [How MEF is Used in ATF](#)

StatusService Component

StatusService provides a global status text panel on the far left side of the Application's StatusBar. It also adds a display to show an operation's progress.

StatusService implements the `IStatusService` interface. There are versions of `StatusService` and `IStatusService` for both WinForms and WPF. Both `IStatusService` interfaces offer similar capabilities, but with a different API. In addition, the WinForms version allows adding an extra text item and an image to the status panel.

This component, or one like it, is useful to any application that wants to show status messages and is used by most of the ATF samples. To use it, add `StatusService` to the MEF TypeCatalog of the application.

In WinForms, `StatusService` is required by `CommandService`.

Showing a Status Message

To show a message, you have to acquire a `StatusService` object using a MEF import. The section [Getting a SettingsService Object](#) describes getting a `SettingService` object, and the process is similar for getting a `StatusService` object. For more details on MEF importing, see [MEF Attributes](#).

After a `StatusService` object is obtained, simply invoke `ShowStatus()` with the message, as in this line:

```
if (m_statusService != null)
    m_statusService.ShowStatus(Localizer.Localize("Rename Event"));
```

Topics in this section

Links on this page to other topics

[How MEF is Used in ATF](#), [SettingsService Component](#)

Other ATF Services

Document Services

ATF offers a Document Framework that supports using documents for application data.

Interfaces

These interfaces handle document services:

- `IDocument`: provide read-only and dirty properties.
- `IDocumentService`: provide document commands, such as Open, Close, Save, and Save As. These commands use the `IDocumentClient` implementation for a particular client to handle documents. Notifications allow tracking what the user is doing with the document.
- `IDocumentRegistry`: interface for the document registry, which holds documents, provides notifications when a document is added or removed, tracks the most recently active document and the open document contexts, and filters by document type.
- `IDocumentClient`: perform the actual open, close, and display methods for a client. Applications should implement a document client implementing `IDocumentClient` for every document type they handle.

Topics

- Document Services
- Interfaces
- Components
- File Dialog Service
- Help
- Resources

Components

The following components provide menus, a document registry, and other services:

- `StandardFileCommands`: implement File menu commands that modify the document registry: New, Open, Save, SaveAs, Save All, and Close.
- `AutoDocumentService`: open documents from the previous application session or create an empty document when an application starts.
- `RecentDocumentCommands`: add recently opened documents to the application's File > Recent Documents submenu.
- `DocumentRegistry`: track open documents in an application. It can retrieve the active document or most recently active document of a given type.
- `MainWindowTitleService`: update the main dialog's title to reflect the current document and its state, that is, modified or not.

For more information, see [Documents in ATF](#).

File Dialog Service

The `FileDialogService` component implements `IFileDialogService`, which contains methods to manage standard file dialogs. `FileDialogService` has file dialogs — for single and multiple files — and a save dialog. The dialog methods support standard file filtering and return the file path in a reference parameter.

If you use the `StandardFileCommands` component, you don't need to use `FileDialogService` directly. `StandardFileCommands` calls the `IFileDialogService` methods for these dialogs. However, you must include `FileDialogService` or a component that implements `IFileDialogService` in the MEF catalog, because `StandardFileCommands` imports `IFileDialogService`.

Help

Applications create their own component deriving from the ATF `HelpAboutCommand` component to display a Help dialog with a rich text message. Override the `ShowHelpAbout()` to customize it for the application information. Also modify the RTF file that `ShowHelpAbout()` uses, typically named `About.rtf`.

See any of the [ATF Code Samples](#) that use `HelpAboutCommand` for an example, such as [ATF Simple DOM Editor Sample](#).

Resources

The `Resources` class contains a variety of image resources for standard images. You can use these images for command icons, cursors, and so on. You only need to reference the image resource to have it automatically embedded in your application's resources by the `ResourceUtil` class.

Topics in this section

Links on this page to other topics

[ATF Code Samples](#), [ATF Simple DOM Editor Sample](#), [Documents in ATF](#)

Controls in ATF

You can use both .NET controls and custom ATF controls in your applications. The `ControlHostService` component, or one like it, manages controls for an ATF application. Similarly to commands, you register controls to use them.

- [Using Controls in ATF](#): Overview of using controls in ATF.
 - [ControlInfo and ControlDef Classes](#): Description of the `ControlInfo` (WinForms) and `ControlDef` (WPF) classes that describe a control's appearance and location in the UI.
 - [ATF Control Groups](#): About control groups, which provide an initial position for a top-level control.
 - [Registering Controls](#): How to register controls with the `ControlHostService` component.
 - [Creating Control Clients](#): Creating a control client that specifies the control's behavior, if any, when the control gains or loses focus, or is closed.
 - [Using WinForms Controls in WPF](#): How to use WinForms-based component controls and dialogs in a WPF based application.
 - [Adaptable Controls](#): Discussion of the `AdaptableControl` class, which provides a control with adapters.
 - [ATF Custom Controls](#): Overview of the custom controls ATF provides.
 - [ATF Custom Dialogs](#): Overview of the standard dialogs in ATF.
-

Using Controls in ATF

Controls

You can use any .NET control or derived .NET control in an ATF application. ATF also offers a large set of additional controls to supplement the standard .NET controls. These include utility controls for editing numbers and file paths, tree and list controls, and palettes. ATF also offers special purpose controls that are useful for game development applications, such as timeline and curve controls. For more information on these controls, see [ATF Custom Controls](#).

Topics

- [Controls](#)
- [ControlHostService](#)
- [Control Groups](#)
- [Registering Controls](#)
- [Control Behavior](#)
- [WinForms and WPF](#)

ControlHostService

The `ControlHostService` component is responsible for handling client controls in the application's main form. `ControlHostService` automatically manages docking, pinning, and show/hide behavior for controls registered with this service. `ControlHostService` implements the `IControlHostService` interface. For more information, see [ControlHostService Component](#).

`ControlHostService` also implements `IControlRegistry`, which tracks active controls, noting when controls are activated or deactivated. This interface provides:

- `ActiveControl` property.
- `Controls` property listing open controls `ControlInfo` objects with control information.
- Events before and after an active control change.
- Events for a control being added or removed.

Control Groups

Top-level controls are controls that are immediate children of your application's main form and are docked to that main form. Top-level controls make up the main palettes, editors, panels, and document windows of your application.

Control groups determine the appearance and initial docking location for top-level controls in an application's main form. Controls may be on the top, bottom, left, right, or center of the main form, or can float, free of the dock. Top-level controls in any position other than center permanent can be undocked, hidden, and then shown and docked again by the user.

Multiple controls in the same position are overlapped and displayed with tabs. Center controls always have tabs, even if there is only one. Tabs are shown with name labels, and you can also specify an icon for the tab when you register the control.

All the control group identifiers are listed in the `StandardControlGroup` enumeration. For details, see [ATF Control Groups](#).

Registering Controls

All controls must be registered with `ControlHostService`. Only top-level controls need to be directly registered with `ControlHostService`, although all controls must either be themselves registered or have a registered control as a parent.

`ControlHostService` implements `IControlHostService`, which contains methods to register, unregister, and show controls.

You can register a control with a `ControlInfo` object for WinForms or a `ControlDef` object for WPF. These classes provide information about a control: its name for a control title, description for a tooltip, group, initial location in the dock, and an optional image appearing on a control tab.

For more information about the control information classes, see [ControlInfo and ControlDef Classes](#).

To learn more about control registration, see [Registering Controls](#).

Control Behavior

Define registered control behavior with the methods in the `IControlHostClient` interface. Control behavior includes what actions should occur, if any, when the control gains or loses focus (activated or deactivated), or is closed. The class on which you implement this interface is called the control client. You can define the client to be the same class as the one in which you register the control, or some other class. You specify the client when you register the control.

For details, see [Creating Control Clients](#).

WinForms and WPF

The `IControlHostService` interface differs slightly for WinForms and WPF, and their `ControlHostService` components also differ. To register a control, use a `ControlInfo` object in WinForms and a `ControlDef` object in WPF. Though these components and interfaces differ somewhat, both offer comparable capability.

The `IControlHostClient` interfaces are also very similar for both. For details, see [Creating Control Clients](#).

Topics in this section

Links on this page to other topics

[ATF Control Groups](#), [ATF Custom Controls](#), [ControlHostService Component](#), [ControlInfo and ControlDef Classes](#), [Creating Control Clients](#), [Registering Controls](#)

ControlInfo and ControlDef Classes

The `ControlInfo` and `ControlDef` classes describe a control's appearance and location in the UI. WinForms uses `ControlInfo`; WPF uses `ControlDef`. Both have similar information. Information in these classes is needed to register a control.

Topics

- [ControlInfo Class](#)
- [ControlDef Class](#)

ControlInfo Class

`ControlInfo` offers a variety of constructors with varying information. This one has all the options:

```
public ControlInfo(string name, string description, StandardControlGroup group, Image image)
{
    m_name = name;
    m_description = description;
    m_group = group;
    m_image = image;
}
```

These parameters are:

- `name`: Control name, which may be displayed as the title of a hosting control or form.
- `description`: Control description, which may be displayed as a tooltip.
- `group`: `StandardControlGroup` for initial control hosting group. For details, see [ATF Control Groups](#).
- `image`: Optional control image, which can be displayed on a hosting control tab or form.

ControlDef Class

`ControlDef` does not provide a constructor with parameters, but has the following gettable and settable properties:

- `Name`: Control's name, which may be displayed as the title of a hosting control tab or form.
- `Description`: Description, which may be displayed as a tooltip.
- `Group`: `StandardControlGroup` indicating where controls are initially docked. For details, see [ATF Control Groups](#).
- `ImageSourceKey`: Optional control image, which may be displayed on a hosting control or form.
- `Id`: Unique ID for control.

This information is the same as in `ControlInfo` except for the additional `ID`.

This line, from the [ATF Model Viewer Sample](#), shows instantiating a `ControlInfo` for a 3D rendering control:

```
ControlInfo cinfo = new ControlInfo("3D View", "3d viewer", StandardControlGroup.CenterPermanent
);
```

This example, from the `IControlHostService` interface for WPF, shows instantiating a `ControlDef`:

```
var def = new ControlDef() { Name = name, Description = description, Group = group, Id = id,
    ImageSourceKey = imageSourceKey };
```

Topics in this section

Links on this page to other topics

[ATF Control Groups](#), [ATF Model Viewer Sample](#)

ATF Control Groups

Control Groups

The control group is the initial position for a top-level control within the application's main form or dock. All the control group identifiers are listed in the `StandardControlGroup` enumeration.

Topics

- [Control Groups](#)
- [Tab Icons on Centered Controls](#)

You can specify a control's group when:

- You create a `ControlInfo` or `ControlDef` object, as described in [ControlInfo and ControlDef Classes](#).
- You register a control. For more information, see [Registering Controls](#).

Controls that are registered with a control group on the perimeter of the application window (`Left`, `Right`, `Top`, `Bottom`, `Floating`) are displayed as docked (pinned) windows that can be unpinned, floated, or hidden. Multiple controls in the same positions are overlapped and tabbed. Palettes, item listers, and property editing controls are most often registered as perimeter controls.

Centered controls (`Center` and `CenterPermanent`) are displayed in the middle of the application as overlapping tabbed windows. Even single center controls are displayed with a tab.

All controls except for those registered as `CenterPermanent` can be undocked, unpinned, or hidden by the user at run time. Controls registered as `CenterPermanent` are anchored in place and cannot be moved, hidden, or closed.

This example creates a `ControlInfo` object for a control in the left position (`StandardControlGroup.Left`):

```
ControlInfo controlInfo = new ControlInfo("Items", "Property Editing Sample List", StandardControlGroup.Left);
```

Controls are given tabs if they are centered (`Center` or `CenterPermanent`) or if there are multiple controls docked in the same perimeter position. By default, the tab is given a text label (the `Name` property for the control, if specified, or "(Untitled)").

Tab Icons on Centered Controls

You can also define an image icon for a control, which is displayed on the control's tab. The icon appears on the tab, in addition to the name.

Use the optional `image` argument in `ControlInfo`, `ControlDef`, or `RegisterControl()` to add an icon to a tab. The icon is a `System.Drawing.Image` object.

You can use your own image or get a standard image with one of the `ResourceUtil.GetImage()` methods.

Topics in this section

Links on this page to other topics

[ControlInfo and ControlDef Classes](#), [Registering Controls](#)

Registering Controls

The `ControlHostService` component registers controls, which makes them available in the user interface. You only need to register top-level controls, although all the controls in your application must either be registered or have a registered control as a parent.

In addition to information about the control, you specify a control client that is notified when the control is activated, deactivated, or requested to close. For details, see [Creating Control Clients](#).

Topics

- [IControlHostService](#)
- [Registering Controls](#)

IControlHostService

`ControlHostService` implements `IControlHostService`. There are different versions of this component and interface for WinForms and WPF, but both offer similar capabilities.

`IControlHostService` contains these methods:

- `RegisterControl()`: Register the control, using the information in either a `ControlInfo` for WinForms or `ControlDef` for WPF, and also specify the control client. Several `RegisterControl()` versions are offered as extension methods with parameters to get control information directly, rather than through a `ControlInfo` or `ControlDef` object.
- `UnregisterControl()` (WinForms) or `UnregisterContent()` (WPF): Unregister the control and its contents, that is, any child controls or other content.
- `Show()`: Make the control visible.

Registering Controls

You need to provide the following (or equivalent information) to register a control:

- `Control`: Control to be registered.
- `ControlInfo` or `ControlDef`: Describe the control's appearance in the UI, as described in [ControlInfo and ControlDef Classes](#). This information may also be specified in parameters in `RegisterControl()`.
- `IControlHostClient`: Client to be notified for activation, deactivation, and close events. To learn more, see [Creating Control Clients](#).

`IControlHostService` provides several `RegisterControl()` methods, including convenience methods with parameters containing the information `ControlInfo` or `ControlDef` provides, so you don't need to explicitly create a `ControlInfo` or `ControlDef` object.

The order in which `RegisterControl()` is called in your application determines the order in which controls and groups of controls appear in the dock.???

The simplest registration method for WinForms is:

```
void RegisterControl(Control control, ControlInfo controlInfo, IControlHostClient client);
```

and for WPF:

```
IControlInfo RegisterControl(ControlDef definition, object control, IControlHostClient client);
```

This line from the [ATF Simple DOM Editor Sample](#) registers a `ListView` control:

```
m_controlHostService.RegisterControl(context.ListView, controlInfo, this);
```

The field `m_controlHostService` is an `IControlHostService` instance, which is imported using MEF. The argument `controlInfo` is a `ControlInfo` object containing the control's UI description. The `Editor` class containing this code not only registers this control, but provides an implementation of the control client, because it implements `IControlHostClient`. In the line above, `this` provides the required `IControlHostClient` parameter.

This longer call in the `PaletteService` component specifies the control information parameters and control client for a palette control:

```
m_controlInfo = m_controlHostService.RegisterControl(  
    new PaletteContent(this),  
    "Palette",  
    "Creates new instances",  
    Sce.Atf.Applications.StandardControlGroup.Left,  
    s_paletteControl.ToString(), this);
```

This example in the WPF `OutputService` component specifies the control information parameters, including an ID, and the control client for an output control:

```
ControlHostService.RegisterControl(  
    m_view,  
    "Output".Localize(),  
    "View errors, warnings, and informative messages".Localize(),  
    Sce.Atf.Applications.StandardControlGroup.Bottom,  
    kId.ToString(),  
    this);
```

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample, ControlInfo and ControlDef Classes, Creating Control Clients](#)

Creating Control Clients

The control client specifies the control's behavior, if any, when the control gains or loses focus, or is closed. You can implement this interface in the same class in which you call `RegisterControl()` or in a separate class.

`ControlHostService` implements `IControlRegistry`, which tracks active controls, noting when controls are activated or deactivated. This interface provides:

- `ActiveControl` property.
- `Controls` property, listing open controls' `ControlInfo` objects with control information.
- Events before and after an active control change.
- Events for a control being added or removed.

Topics

- [IControlHostClient interface](#)
 - [Activate\(\) Method](#)
 - [Deactivate\(\) Method](#)
 - [Close\(\) Method](#)

IControlHostClient interface

A control client implements the `IControlHostClient` interface, which consists of these methods for WinForms:

```
void Activate(Control control);
void Deactivate(Control control);
bool Close(Control control);
```

These are the methods for WPF, which are almost the same, except that the parameter is `object` rather than `Control`, and `Close()` has an extra parameter:

```
void Activate(object control);
void Deactivate(object control);
bool Close(object control, bool mainWindowClosing);
```

Activate() Method

The `Activate()` method notifies the client that its control is activated, that is, gains focus. Activation occurs when the control gets focus, or the control's parent "host" control gets focus.

Use this method to update other objects or perform operations that should occur when the control gains input focus. A common use of this method is to notify command clients that the context has changed by the control being activated.

Use the `ICommandService.SetActiveClient()` method to switch the command client when the control gains focus, as in this example from the `OutputService` component:

```
public void Activate(Control control)
{
    if (m_commandService != null)
        m_commandService.SetActiveClient(this);
}
```

For information about command clients, see [Creating Command Clients](#).

This example from the [ATF Simple DOM Editor Sample](#) shows a document control setting the active document and context when it is activated:

```

void IControlHostClient.Activate(Control control)
{
    EventSequenceDocument document = control.Tag as EventSequenceDocument;
    if (document != null)
    {
        m_documentRegistry.ActiveDocument = document;

        EventSequenceContext context = document.As<EventSequenceContext>();
        m_contextRegistry.ActiveContext = context;
    }
}

```

For a discussion of contexts, see the topics in [ATF Contexts](#).

Deactivate() Method

`Deactivate()`'s purpose is to notify the client that its control is deactivated, that is, loses focus. Deactivation occurs when another control gets focus or the control's "host" control loses focus.

In this example from the `OutputService` component, the active command client is set to `null` when the output control is deactivated:

```

public void Deactivate(Control control)
{
    if (m_commandService != null)
        m_commandService.SetActiveClient(null);
}

```

Close() Method

This method requests permission to close the client's control. It returns true if the control can close, or false to not close. For WPF, the `mainWindowClosing` parameter is true if the application's main window is closing.

If true is returned, `ControlHostService` calls `UnregisterControl()`. The application has to call `RegisterControl()` again if it wants to re-register this control.

This method is not called when the user toggles control visibility by using Windows menu commands.

This sample, from the editor common code of [ATF Win Forms App Sample](#) and [ATF Wpf App Sample](#), allows the control to close if its document can be closed:

```

bool IControlHostClient.Close(Control control)
{
    bool closed = true;

    WinGuiCommonDataDocument document = control.Tag as WinGuiCommonDataDocument;
    if (document != null)
    {
        closed = m_documentService.Close(document);
        if (closed)
            m_contextRegistry.RemoveContext(document);
    }

    return closed;
}

```

NOTE: If a particular control is closed when an application terminates, its control client's `Close()` method should usually call `IDocumentService.Close()`, as the previous example does. This gives the opportunity to check whether the document is dirty and offer the user the option of saving it; otherwise changes are lost. For instance, the often used `StandardFileCommands` component implements `IDocumentService` and, if the document's `IDocument.Dirty` property is true, its `IDocumentService.Close()` method asks a user whether or not the document's changes should be saved or discarded.

Links on this page to other topics

[ATF Contexts](#), [ATF Simple DOM Editor Sample](#), [ATF Win Forms App Sample](#), [ATF Wpf App Sample](#), [Creating Command Clients](#)

Using WinForms Controls in WPF

You can use WinForms-based component controls and dialogs in a WPF based application. This is made possible by the `ControlHostServiceAdapter` component, which implements the WinForms `IControlHostService` so that it works in a WPF environment. In other words, it adapts `Sce.Atf.Wpf.Applications.IControlHostService` to `Sce.Atf.Applications.IControlHostService`.

The `ControlHostServiceAdapter` component exports `Sce.Atf.Applications.IControlHostService`. If this component is included in an application, the WinForms-based components that need `IControlHostService` import it from `ControlHostServiceAdapter`, rather than the WinForms-based `ControlHostService` component. This allows the components to function with the WPF application. In this case, the application would not include the WinForms-based `ControlHostService` component — or any other component implementing the WinForms `IControlHostService` — in its MEF catalog.

`ControlHostServiceAdapter` also creates an object of the `ControlHostClientAdapter` class, which adapts to WPF the `IControlHostClient` interface for WinForms-based control clients. This allows these control clients implementing `Sce.Atf.Applications.IControlHostClient` to operate properly in WPF.

The `StandardInteropParts` class includes the `ControlHostServiceAdapter` component in a MEF TypeCatalog:

```
public class StandardInteropParts
{
    /// <summary>
    /// Gets type catalog for all components</summary>
    public static ComposablePartCatalog Catalog
    {
        get
        {
            return new TypeCatalog(
                typeof(MainWindowAdapter),
                typeof(CommandServiceAdapter),
                typeof(ContextMenuService),
                typeof(DialogService),
                typeof(ControlHostServiceAdapter)
            );
        }
    }
}
```

This catalog can be included in an `AggregateCatalog`, as in this line from the ATF Wpf App Sample:

```
return new AggregateCatalog(typeCatalog, StandardInteropParts.Catalog, StandardViewModels.Catalog
);
```

After this, the WPF application can use the WinForms-based component controls — with no additional support from the application.

Topics in this section

Links on this page to other topics

[ATF Wpf App Sample](#)

Adaptable Controls

The `AdaptableControl` class provides an adaptable control, which is a control decorated with adapters. An adaptable control can be converted into any of its adapters by using the `IAdaptable.As` method. Such a control is very versatile.

The `AdaptableControl` class provides the `Adapt()` method to adapt the control to all the specified adapters:

```
public void Adapt(params IControlAdapter[] adapters);
```

Each adapter provided in the array is an object of the `ControlAdapter` class, implementing `IControlAdapter`. This interface consists of methods to bind and unbind the adapter to or from its underlying control. The `Bind()` method for each adapter is called in the order that the adapters were listed in the `Adapt()` method. By convention, the bottom-most layer should appear first in the `AdaptableControl`'s `Adapt()` method, and so the `Paint` event should be subscribed to in `Bind()`.

For more information on adaptation, see [Adaptation in ATF](#).

This example from the `Editor` class in the [ATF FSM Editor Sample](#) illustrates setting up an `AdaptableControl`. The steps in this process are:

1. Create the `AdaptableControl`. This example creates an object of the class `D2dAdaptableControl`, which derives from `AdaptableControl`.
2. Invoke `SuspendLayout()` on the new `AdaptableControl` to suspend layout operations while the control is being configured.
3. Set any `AdaptableControl` properties desired, such as `BackColor`.
4. Create the `ControlAdapter` objects. All of these objects, such as `ViewingAdapter`, eventually derive from `ControlAdapter`. These adapters all perform different tasks, as noted in the comments. For instance, `ViewingAdapter` implements `IViewingContext` for framing or ensuring that items are visible. Configure these adapters as needed. For instance, `HoverAdapter` needs to subscribe to some events.
5. Call `AdaptableControl.Adapt()` with an array of the `ControlAdapter` objects. You can use as many `ControlAdapter` objects as you want.
6. Invoke `ResumeLayout()` on the new `AdaptableControl` to resume layout operations.
7. Perform any other initialization needed, such as setting up contexts, documents, initializing DOM adapters, and so on.
8. Register the `AdaptableControl` by invoking `RegisterControl()` on the `ControlHostService` instance.

```
// set up the AdaptableControl for editing FSMS
var control = new D2dAdaptableControl();
control.SuspendLayout();

control.BackColor = SystemColors.ControlLight;
control.AllowDrop = true;

var transformAdapter = new TransformAdapter(); // required by several of the other adapters
transformAdapter.UniformScale = true;
transformAdapter.MinScale = new PointF(0.25f, 0.25f);
transformAdapter.MaxScale = new PointF(4, 4);

var viewingAdapter = new ViewingAdapter(transformAdapter); // implements IVIEWINGCONTEXT for
framing or ensuring that items are visible

var canvasAdapter = new CanvasAdapter(); // implements a bounded canvas to limit scrolling

var autoTranslateAdapter = // implements auto translate when the user drags out of control's
client area
    new AutoTranslateAdapter(transformAdapter);
var mouseTransformManipulator = // implements mouse drag translate and scale
    new MouseTransformManipulator(transformAdapter);
var mouseWheelManipulator = // implements mouse wheel scale
    new MouseWheelManipulator(transformAdapter);
var scrollbarAdapter = // adds scroll bars to control, driven by canvas and transform
    new ScrollbarAdapter(transformAdapter, canvasAdapter);

var hoverAdapter = new HoverAdapter(); // add hover events over pickable items
hoverAdapter.HoverStarted += control_HoverStarted;
hoverAdapter.HoverStopped += control_HoverStopped;

var annotationAdaptor = new D2dAnnotationAdapter(m_theme); // display annotations under diagram
```

```

var fsmAdapter = // adapt control to allow binding to graph data
    new D2dGraphAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, transformAdapter);

var fsmStateEditAdapter = // adapt control to allow state editing
    new D2dGraphNodeEditAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, fsmAdapter, transformAdapter
);

var fsmTransitionEditAdapter = // adapt control to allow transition
    new D2dGraphEdgeEditAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, fsmAdapter, transformAdapter
);

var mouseLayoutManipulator = new MouseLayoutManipulator(transformAdapter);

// apply adapters to control; ordering is from back to front, that is, the first adapter
// will be conceptually underneath all the others. Mouse and keyboard events are fed to
// the adapters in the reverse order, so it all makes sense to the user.
control.Adapt(
    hoverAdapter,
    scrollbarAdapter,
    autoTranslateAdapter,
    new RectangleDragSelector(),
    transformAdapter,
    viewingAdapter,
    canvasAdapter,
    mouseTransformManipulator,
    mouseWheelManipulator,
    new KeyboardGraphNavigator<State, Transition, NumberedRoute>(),
    //new GridAdapter(),
    annotationAdaptor,
    fsmAdapter,
    fsmStateEditAdapter,
    fsmTransitionEditAdapter,
    new LabelEditAdapter(),
    new SelectionAdapter(),
    mouseLayoutManipulator,
    new DragDropAdapter(m_statusService),
    new ContextMenuAdapter(m_commandService, m_contextMenuCommandProviders)
);

control.ResumeLayout();

// associate the control with the viewing context; other adapters use this
// adapter for viewing, layout and calculating bounds.
ViewingContext viewingContext = node.Cast<ViewingContext>();
viewingContext.Control = control;

// set document URI
document = node.As<Document>();
ControlInfo controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center);

//Set IsDocument to true to prevent exception in command service if two files with the
// same name, but in different directories, are opened.
controlInfo.IsDocument = true;

document.ControlInfo = controlInfo;
document.Uri = uri;

// now that the data is complete, initialize the rest of the extensions to the Dom data;
// this is needed for adapters such as validators, which may not be referenced anywhere
// but still need to be initialized.
node.InitializeExtensions();

// set control's context to main editing context
EditingContext editingContext = node.Cast<EditingContext>();
control.Context = editingContext;

```

```
// show the FSM control  
m_controlHostService.RegisterControl(control, controlInfo, this);
```

Topics in this section

Links on this page to other topics

[Adaptation in ATF, ATF FSM Editor Sample](#)

ATF Custom Controls

ATF offers a variety of controls to augment standard .NET controls. You can use WinForms controls in WPF applications; for more information, see [Using WinForms Controls in WPF](#). WPF also offers some controls of its own, described in [WPF Controls](#).

Topics

- [WinForms Controls](#)
- [WPF Controls](#)

WinForms Controls

Category	Control Class	Description
Adaptable	AdaptableControl	Control with adapters (decorators). The adaptable control can be converted into any of its adapters using the <code>IAdaptable.As()</code> method. It is used in several samples, such as the ATF Circuit Editor Sample . For more information, see Adaptable Controls .
	D2dAdaptableControl	Control with adapters (decorators), derived from <code>AdaptableControl</code> . Specialized for Direct2D graphic drawing. Used in several samples, such as the ATF Circuit Editor Sample .
Containers	CollectionEditingControl	Collection editing control using a <code>ListBox</code> .
	QuadPanelControl	Control holding 4 panels with a 2-way splitter.
Canvases	Canvas2d	Canvas for displaying graphics.
	CanvasControl	Control for viewing and editing a 2D bounded canvas.
	CanvasControl3D	3D OpenGL canvas control.
	Cartesian2dCanvas	Cartesian 2D canvas.
	CurveCanvas	Canvas for drawing and picking curves, derived from <code>Cartesian2dCanvas</code> .
	DesignControl	Extends <code>CanvasControl3D</code> to provide Scene graph rendering and picking. Used in the ATF Model Viewer Sample .
OpenGL	Panel3D	Control for using OpenGL to do painting.
	TranslatorControl	Reusable 3D control for doing translations.
Property Editors	GridControl	Wrapper control for the spreadsheet-style <code>GridView</code> control, combining it with a toolbar. Use this as a replacement for the <code>.NET System.Windows.Forms.DataGridView</code> control. Used in the <code>GridPropertyEditor</code> component. For more information, see GridPropertyEditor Component .
	GridView	Spreadsheet-like control for displaying properties of many objects simultaneously. Only properties that are in common with all selected objects are displayed. Derived from the <code>PropertyView</code> abstract class.
	PropertyEditingControl	Universal property editing control that can be embedded in complex property editing controls. It uses <code>System.ComponentModel.TypeConverter</code> and <code>System.Drawing.Design.UITypeEditor</code> classes to provide a GUI for every kind of .NET property. It gets used by default if no custom value editing control is provided.
	PropertyGrid	Wrapper control for a two-column <code>PropertyGridView</code> control combined with a toolbar. Use this as a replacement for the <code>.NET System.Windows.Forms.PropertyGrid</code> control. Used in the <code>PropertyEditor</code> component. For more information, see PropertyEditor Component .

	PropertyGridView	Control for displaying properties in a two column grid, with property names on the left and property values on the right, derived from the <code>PropertyView</code> abstract class. Properties with associated <code>IPropertyEditor</code> instances can embed controls into the right column, while all other properties are edited in a standard .NET way with a <code>PropertyEditingControl</code> .
	PropertyView	Abstract base class for complex property editing controls, providing formats, fonts, data binding, persistent settings, and category/property information.
Special Purpose	CurveEditingControl	Container control for <code>CurveCanvas</code> control. Used by <code>CurveEditor</code> component and the ATF DOM Tree Editor Sample .
	Direct2DControl	Control for Direct2D. Consider using <code>D2dAdaptableControl</code> instead.
	D2dTimelineControl	Control to display a timeline. In a multiple document application, there is one of these per tab or document. Used in the ATF Timeline Editor Sample .
	InteropControl	Support interoperability for events between Windows and ATF.
	OutputService	Component that displays text output to the user in a <code>RichTextBox</code> . Used in several samples, such as ATF State Chart Editor Sample .
	PerformanceMonitorControl	Control that displays the rendering performance of a control.
	SyntaxEditorControl	Encapsulates the Actipro <code>SyntaxEditor</code> control, implementing <code>ISyntaxEditorControl</code> . This class is internal to abide by <code>SyntaxEditor</code> licensing terms. Used in the ATF Code Editor Sample .
	ThumbnailControl	Thumbnail viewer control.
Trees	FilteredTreeControlEditor	Component derived from the <code>TreeControlEditor</code> class that adds filtering support for tree editors.
	TreeControl	Tree control to display hierarchical data. Handles user input, including editing. Decides the placement of the images, check box, and label of each item. The <code>TreeItemRenderer</code> class decides how to draw the items. The <code>TreeControlAdapter</code> class adapts this tree control to the <code>ITreeView</code> interface. Used in ATF File Explorer Sample .
	TreeControlEditor	Base class for tree editors. It is a component and is not abstract so it can be used as a generic tree editor. It is built from a <code>TreeControl</code> . Used in ATF DOM Tree Editor Sample .
	TreeListView	Provide a tree <code>ListView</code> . Used in ATF Tree List Editor Sample .
Trees, Specialized	LayerLister	Component derived from <code>TreeControlEditor</code> that presents an <code>ILayeringContext</code> , which controls item visibility and provides a tree view of layers. Used in ATF Circuit Editor Sample and ATF Diagram Editor Sample .
	PaletteService	Component derived from <code>TreeControlEditor</code> that manages a global palette of objects that can be dragged onto other controls. Used in numerous samples, such as ATF FSM Editor Sample .
	ProjectLister	Component derived from <code>FilteredTreeControlEditor</code> that provides a hierarchical tree control, listing the contents of a loaded document.
	PrototypeLister	Component derived from <code>TreeControlEditor</code> that presents an <code>IPrototypingContext</code> , which which can present a tree view of its contents and create <code>System.Windows.Forms.IDataObject</code> objects from them. Used in several samples to contain prototypes that can be reused, such as ATF Circuit Editor Sample and ATF Diagram Editor Sample .
	ResourceLister	Component using a <code>TreeControl</code> to browse and organize resource folders and resources, such as models, images, and so on.
	TemplateLister	Component derived from <code>TreeControlEditor</code> that presents an <code>ITemplatingContext</code> , which can present a tree view of its contents and create <code>System.Windows.Forms.IDataObject</code> objects from them. Used in ATF Circuit Editor Sample to hold a template that can be reused.
Value Editing	ArrayEditingControl	Control for editing arrays of arbitrary values.

	BoolInputControl	Control for editing a Boolean value.
	FloatInputControl	Control for editing a bounded float value.
	IntInputControl	Control for editing a bounded int value.
	NumericMatrixControl	Control for editing a numeric matrix.
	NumericTupleControl	Control for editing a numeric tuple (vector) of arbitrary dimensions.

WPF Controls

Control Class	Description
FormattingTextBox	TextBox with a bindable StringFormat property.
OutputView	Interaction logic for OutputView.xaml for dialogs to display text output to user.
PropertyGrid	Grid of PropertyNode objects, which encapsulate an object or collection of objects and a property common to them all.
PropertyGridToolBar	Toolbar for a PropertyGrid control.
SliderBox	Combine a slider with a formatted TextBox. Also provide the option of acting in deferred mode, where the slider value is not propagated to the binding source while it is being dragged. Used with property editors.

Topics in this section

Links on this page to other topics

[Adaptable Controls](#), [ATF Circuit Editor Sample](#), [ATF Code Editor Sample](#), [ATF Diagram Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF File Explorer Sample](#), [ATF FSM Editor Sample](#), [ATF Model Viewer Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [ATF Tree List Editor Sample](#), [Authoring Tools Framework](#), [Property Editor Components](#), [Using WinForms Controls in WPF](#)

ATF Custom Dialogs

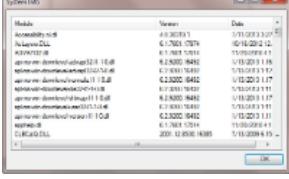
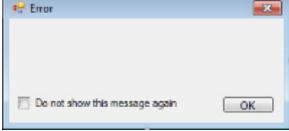
ATF offers a variety of standard dialogs for WinForms applications, described in the following table. You can use WinForms dialogs in WPF applications; for more information, see [Using WinForms Controls in WPF](#). WPF has dialogs of its own, described in [WPF Dialogs](#).

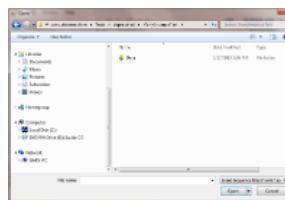
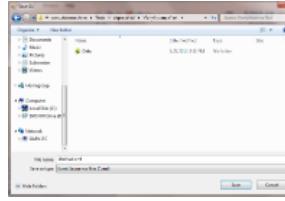
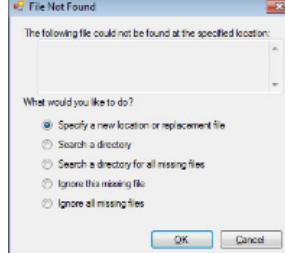
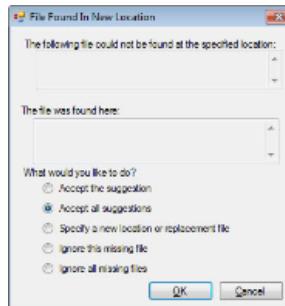
For details on using dialogs, see [Using Dialogs](#) below.

Topics

- [WinForms Dialogs](#)
- [WPF Dialogs](#)
- [Using Dialogs](#)
- [Component Dialogs](#)
- [Directly Using Dialogs](#)

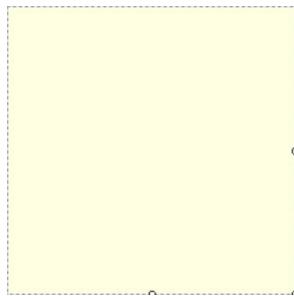
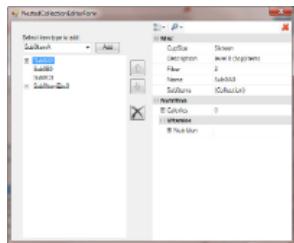
WinForms Dialogs

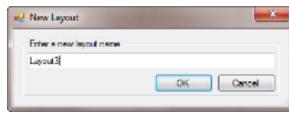
Category	Dialog Class	Appearance	Description
About	AboutDialog		A "Help About..." dialog, with Sony Logo and assembly list. Used in numerous samples, such as ATF Circuit Editor Sample .
	AboutSysInfoDialog		A system information dialog, with Sony Logo and assembly list.
Errors	ErrorDialog		Dialog to display error messages to a user. It is used by the <code>ErrorDialogService</code> component, which is imported by several samples, such as ATF FSM Editor Sample .
	UnhandledExceptionDialog		Unhandled exception dialog, which is used by the <code>UnhandledExceptionService</code> component employed in several samples, such as ATF Diagram Editor Sample .
File Handling	CustomFileDialog		Abstract base class for a custom file dialog, which <code>CustomSaveFileDialog</code> and <code>CustomOpenFileDialog</code> derive from. Uses <code>System.Windows.Forms.OpenFileDialog</code> .

	Custom OpenFileDialog		Custom open file dialog derived from <code>CustomFileDialog</code> . This class behaves the same as the <code>System.Windows.Forms.OpenFileDialog</code> class. Allows setting "Read Only" check box. Can indicate whether supports multiple file name selections. Used by <code>FileDialogService</code> component, which is used by numerous samples like ATF Code Editor Sample .
	Custom SaveFileDialog		Custom save file dialog derived from <code>CustomFileDialog</code> . This class behaves the same as the <code>System.Windows.Forms.SaveFileDialog</code> class. Allows prompting user when file does not exist and when file already exists. Used by the <code>FileDialogService</code> component, which is imported by numerous samples like ATF Code Editor Sample .
	FilteredFileDialogBase		Base class for filtered file dialogs, including <code>OpenFilteredFileDialog</code> . Its properties are equivalent to those with the same names in <code>System.Windows.Forms.FileDialog</code> . Unlike <code>System.Windows.Forms.FileDialog</code> or <code>Sce.Atf.CustomFileDialog</code> , this is a Windows Forms implementation that does not use Win32 methods. It has a <code>CustomFileFilter</code> callback, which can be used to exclude files in the dialog's <code>ListView</code> of files.
	FindFileDialog		Dialog to assist the user in finding a missing file.
	FindFileWithSuggestionDialog		Dialog for finding missing files that allows the user to suggest a path.
	FolderSelectDialog		Folder selection dialog that wraps <code>System.Windows.Forms.OpenFileDialog</code> to make it present a Vista-style dialog.

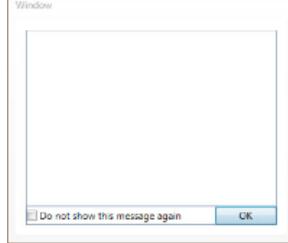
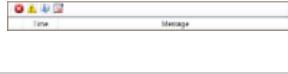
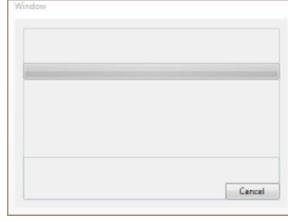
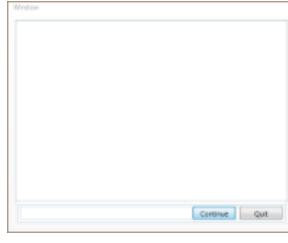
	OpenFilteredFileDialog		Open file dialog with a filter, derived from FilteredFileDialogBase. This class behaves the same as the System.Windows.Forms.OpenFileDialog class, but allows a custom file filter to be added to exclude files in the dialog's ListView of files. Use it for importing assets, for example.
Perforce	Connections		Form for connecting to a Perforce server. Used in the PerforceService component.
	LoginDialog		Login dialog to a Perforce server. Used in the PerforceService component.
	UsersList		Perforce server users list dialog.
	WorkspaceList		Perforce workspace list dialog.
Progress	ProgressDialog		Dialog to show progress of a task.
	ThreadSafeProgressDialog		Thread-safe dialog to show task progress. Uses System.Windows.Forms.ProgressBar.
Source Control	CheckInForm		Source control check-in form. Used in the SourceControlCommands component, which is imported in the ATF Code Editor Sample .
	ReconcileForm		Source control reconcile form. Used in the SourceControlCommands component, which is imported in the ATF Code Editor Sample .

Special Purpose	ColorPicker		Color picker adapted from the Adobe Color Picker Clone .
	ConfirmationDialog		Simple customizable "Yes/No/Cancel" dialog box. Used by the <code>FileDialogService</code> component, which is imported by numerous samples like ATF Code Editor Sample .
	CustomizeKeyboardDialog		Dialog to edit and assign keyboard shortcuts to registered commands. Used in <code>CommandService</code> component, that is imported by nearly all the samples, such as ATF Circuit Editor Sample .
	FeedbackForm		Form for submitting bugs to the SourceForge bug tracker. Note that each project has a unique identifier used to map to the SourceForge project (for example, "com.scea.screamtool"). This identifier can be specified with the <code>ProjectMappingAttribute</code> . For more information, see Bug Submittal . Used in the <code>UserFeedbackService</code> component, which is imported by the ATF File Explorer Sample .
	GridControlShowHidePropertiesDialog		Grid control with check boxes to show or hide properties.
	HoverBase		Base class for tool tips that appear when the mouse hovers over an item. Used in ATF Circuit Editor Sample .

	HoverLabel		Control to display a string when the mouse hovers over an item. Derives from <code>HoverBase</code> . Used in ATF Circuit Editor Sample , ATF FSM Editor Sample , and ATF State Chart Editor Sample .
	NestedCollectionEditorForm		Nested collections editor form. Used by the <code>NestedCollectionEditor</code> editor class, which is employed by the ATF Property Editing Sample .
	OscDialog		Dialog for Open Sound Control (OSC), which allows devices to get or set properties on C# objects. It is used by <code>OscCommands</code> , a component that provides menu commands for users of the <code>OscService</code> component.
	RenameCommandDialog		Rename command dialog used to perform the action provided by the <code>RenameCommand</code> component that defines a mass rename command.
	TabbedControlSelectorDialog		Dialog displayed by the <code>TabbedControlSelector</code> component, which enables a user to switch focus between controls accessible to a specified <code>IControlHostService</code> . <code>TabbedControlSelector</code> is used in several samples, such as ATF Circuit Editor Sample and ATF Code Editor Sample
Target Handling	TargetDialog		Dialog to add or edit a target device, such as a PS3 controller. Used in the <code>TargetService</code> component.
	TargetEditDialog		Form for editing target devices.
Window Layout	WindowLayoutManageDialog		Dialog to manage window layouts. Used by the <code>WindowLayoutServiceCommands</code> component, which is imported by numerous samples, such as the ATF Circuit Editor Sample .

	WindowLayoutNewDialog		<p>New window layout dialog. Used by the <code>WindowLayoutServiceCommands</code> component, which is imported by numerous samples, such as the ATF Circuit Editor Sample</p> <p>ATF Circuit Editor Sample.</p>
--	-----------------------	--	---

WPF Dialogs

Dialog Class	Appearance	Description
AboutDialog		About dialog.
CommonDialog	None	Base class for WPF dialogs.
ErrorDialog		Error dialog.
MessageBoxDialog		Message boxes. An alternative to the native Win32 message box if WPF styling is required.
OutputView		Dialog for text output to user. Used in the WPF <code>OutputService</code> component.
ProgressDialog		Dialog to show progress of a task.
UnhandledExceptionDialog		Unhandled exception dialog. Used in the WPF <code>UnhandledExceptionService</code> component.

Using Dialogs

You can use dialogs in two ways: indirectly through a component or directly by constructing the dialog class.

Component Dialogs

Some ATF components do all the dialog handling for you. For instance, the `FileDialogService` component creates and displays the `Custom OpenFileDialog`, `Custom SaveFileDialog`, and `ConfirmationDialog` dialogs, as directed by the `StandardFileCommands` component in response to a user selecting File menu commands. Similarly, `UnhandledExceptionDialog` is used by the `UnhandledExceptionService` component to inform the user when an unhandled exception occurs. Simply use these components in your application to employ the dialogs.

Directly Using Dialogs

You can also manage dialogs by creating dialog objects yourself with dialog constructors and their parameters. For example, `FindFileDialog()` takes a string for the path to the original (missing) file.

Dialogs also have properties appropriate for the dialog. For instance, `FindFileDialog` has an `Action` property of type `FindFileAction` indicating the action the user took, such as accepting the suggested file name found, based on which button the user pressed on the dialog.

Dialogs have a `ShowDialog()` method, which returns a `System.Windows.Forms.DialogResult` value, so you can determine the result of the user's interaction with the dialog.

This example is from the class `FindFileResolver`. It creates dialog objects using a file path and checks their return value and `Action` property, which gets the user's choice of action from the radio buttons after the OK button was pressed. It sets the `FindFileAction` variable accordingly.

```
FindFileAction userAction;  
...  
// Ask the user what we should do. There are two possible dialog boxes to use.  
if (suggestedUri == null)  
{  
    // There are a few options and slightly reorganized dialog box if there  
    // is no suggested replacement for the missing file.  
    FindFileDialog dialog = new FindFileDialog(uri.LocalPath);  
    if (dialog.ShowDialog() == DialogResult.Cancel)  
        userAction = FindFileAction.Ignore;  
    else  
        userAction = dialog.Action;  
}  
else  
{  
    // We have a suggested replacement already, so allow the user to accept the  
    // suggestion.  
    FindFileWithSuggestionDialog dialog =  
        new FindFileWithSuggestionDialog(uri.LocalPath, suggestedUri.LocalPath);  
    if (dialog.ShowDialog() == DialogResult.Cancel)  
        userAction = FindFileAction.Ignore;  
    else  
        userAction = dialog.Action;  
}
```

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF Code Editor Sample](#), [ATF Diagram Editor Sample](#), [ATF File Explorer Sample](#), [ATF FSM Editor Sample](#), [ATF Property Editing Sample](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [Using WinForms Controls in WPF](#)

Commands in ATF

You generally need to add commands to your application's menus and tool strips. ATF manages the command GUI and infrastructure for menus, menu items, and toolstrips. You provide command descriptions and the command action. ATF also provides components for creating and implementing many standard menus and menu items, requiring minimal extra coding.

- [Using Commands in ATF](#): Overview of how commands are handled in ATF.
 - [CommandInfo and CommandDef Classes](#): Description of classes that describes command UI info.
 - [ATF Command Groups](#): Description of command groups and how commands are added to them.
 - [Using Standard Command Components](#): Outline of the components that add common commands to applications.
 - [Registering Menus and Commands](#): How to create new menus and register commands.
 - [Creating Command Clients](#): Creating command clients that implement command actions.
 - [Using WinForms Commands in WPF](#): How to use WinForms-based command components in a WPF based application.
 - [Using Context Menus](#): How to use context menus in ATF.
-

Using Commands in ATF

Commands

A command is an action that is invoked from a menu item or tool strip button. You can create menu items and tool strip buttons and associate them with commands. The command specifies its appearance and behavior in the UI: text, icon, keyboard shortcut, and actual command action.

All standard commands are listed in the enum `StandardCommand`. You can also create new commands. For details, see [Registering Menus and Commands](#).

Topics

- [Commands](#)
- [Command Groups](#)
- [Standard and Custom Commands](#)
- [Registering Commands](#)
- [Command Behavior](#)
- [Command Contexts](#)

Command Groups

Commands are grouped by related functions, both in menus and tool strips. At the top level, menus contain related functions. The Edit menu contains editing commands, for instance. Menu items are also grouped together, separated from other groups by divider lines. For example, File > Save and File > SaveAs are closely related commands, grouped together. ATF provides a set of standard command groups, and you can add custom groups.

Tool strip buttons are grouped by menu; all the Edit commands are in one tool strip and all the File commands in another, for instance.

For details, see [ATF Command Groups](#).

Standard and Custom Commands

You can add both standard and custom commands to your application.

ATF provides components to add commands that are common to many applications. For instance, `StandardFileCommands` adds the standard File commands New, Open, Save, SaveAs, Save All, and Close. For details, see [Using Standard Command Components](#).

An application can also create custom menus, tool strips, and commands. For more information, see [Registering Menus and Commands](#).

Registering Commands

The `CommandService` component provides a service to handle commands in menus and tool strips. All commands must be registered with `CommandService`. `CommandService` implements `ICommandService`, which contains methods to register and unregister commands, display context menus, and process keyboard shortcuts.

Note that there is a separate `CommandService` component for WinForms and WPF. Each implements a different `ICommandService` as well. Though the components and interfaces differ somewhat, both offer comparable capability, tailored to the platform.

You can register a command with a `CommandInfo` (WinForms) or `CommandDef` (WPF) object. The `CommandInfo` and `CommandDef` classes provides information about a command: its visibility, location, and appearance. The `CommandInfo` class also creates `CommandInfo` objects for common commands, such as the standard File and Edit menu item commands.

Visibility determines whether a command appears in a menu or tool strip, both, or neither. Commands that are not visible can be added to menus or tool strips later on.

Location indicates what group a command is a member of, its order in the group, and where the group is in the menus and tool strips. In addition, menu items can be in submenus.

Appearance specifies what text, tool tips, and icons are associated with the command.

`CommandInfo` can also specify keyboard shortcuts. Also set up accelerator keys (as in Help > About) by using the "&" character before the

designating character in the menu name text. On the other hand, `CommandDef` can specify an `InputGesture`, which describes input device gestures.

For more information about `CommandInfo` and `CommandDef`, see [CommandInfo and CommandDef Classes](#).

Tool strips can also contain text entry and combo box controls for commands. These controls are registered through `CommandService`. How are these handled???

To learn how to register commands, see [Registering Menus and Commands](#).

Command Behavior

Define command behavior with the methods in the `ICommandClient` interface; WinForms and WPF use the same `ICommandClient`. In addition to performing the command, this interface provides methods to determine whether the command can be performed and to update the `CommandState`. Command state includes the menu item name and whether the command menu item has a check mark. For instance, the command state's name may switch back and forth between "Group" and "Ungroup", depending on what actions have occurred.

The command client is the client class in which you implement `ICommandClient` for the command. The command client can be the same class as the one in which you register the command, or some other class. You specify the client when you register the command, although you can switch between different clients at any time.

For details, see [Creating Command Clients](#).

Command Contexts

A command may behave differently depending on what context is active. A context can be a document, editor, palette, or other object in your application.

Your application can define different contexts, each with a different set of data, and each requiring different command behavior. For example, Select All in a text editor is a very different operation from Select All on a drawing canvas. You can switch the command client for a command to the appropriate one for the context. An application can also have different contexts of the same type, for example, multiple open documents.

The command context is used in building context menus. For more information, see [Using Context Menus](#).

The command context changes when a control gains or loses focus. Your application needs to keep track of the current command context, notify the command client when the context has changed, and implement context behavior in your client.

For more information about contexts, see [ATF Contexts](#).

Topics in this section

Links on this page to other topics

[ATF Command Groups](#), [ATF Contexts](#), [CommandInfo and CommandDef Classes](#), [Creating Command Clients](#), [Registering Menus and Commands](#), [Using Context Menus](#), [Using Standard Command Components](#)

CommandInfo and CommandDef Classes

The `CommandInfo` and `CommandDef` classes describe a command's interaction with the UI: its visibility, location, and appearance, as described in [Registering Commands](#). This information is needed to register a command.

Topics

- [CommandInfo Class](#)
- [CommandDef Class](#)

CommandInfo Class

`CommandInfo` offers a variety of constructors, and the following one offers the most options:

```
public CommandInfo(
    object commandTag,
    object menuTag,
    object groupTag,
    string menuText,
    string description,
    IEnumerable<Keys> shortcuts,
    string imageName,
    CommandVisibility visibility);
```

The tag parameters identify the command itself, as well as the menu and group the command belongs to. The `commandTag` parameter can be one of the `StandardCommand` enums. The `menuTag` tag is typically one of the `StandardMenu` enums. Similarly, `groupTag` is a `StandardCommandGroup` enum. For details on groups, see [ATF Command Groups](#).

String parameters give the menu item name and a more detailed tool tip description. In the menu item name:

- A forward slash indicates the command is in a submenu of the menu. For example, "Size/Make Heights Equal" creates a "Size" menu with a submenu item "Make Heights Equal".
- The "&" character in the name indicates that the next character is underlined and used as the menu item's accelerator key for this menu item (used with the Alt key), as in standard Windows menu conventions. For example, "&About" underlines the "A" in "About" and enables Alt+A to be used to choose the "About" menu item.

You can also specify a collection of keyboard shortcuts. Shortcuts are specified with the `Keys` enum, specifying both qualifier keys and the key, as in "Keys.Control | Keys.S".

You can provide an image for the command's tool strip and menu item, typically a member of the `Resources` class.

The `CommandVisibility` enum tells where command is visible, as on menus, tool strips, and so on.

This example, from the `CommandInfo` class, shows instantiating the `CommandInfo` for the `FileSave` command:

```
public static CommandInfo FileSave =
    new CommandInfo(
        StandardCommand.FileSave,
        StandardMenu.File,
        StandardCommandGroup.FileSave,
        "Save".Localize("Save the active file"),
        "Save the active file".Localize(),
        Keys.Control | Keys.S,
        Resources.SaveImage);
```

The constructor used here differs from the other constructor in that it provides a single `Keys` object rather than a collection of them for the command shortcut in the second-last parameter. It doesn't specify a `CommandVisibility` either, which defaults to the command being visible on both a menu and tool strip.

CommandDef Class

Similarly to `CommandInfo`, `CommandDef` has a variety of constructors, and this one has the most parameters:

```
public CommandDef(  
    object commandTag,  
    object menuTag,  
    object groupTag,  
    string text,  
    string[] menuPath,  
    string description,  
    object imageSourceKey,  
    InputGesture[] inputGestures,  
    CommandVisibility visibility)
```

The parameters are similar to those of `CommandInfo` constructors, providing information on the command, the menu and group the command belongs to, command image, and its visibility. It does not provide a keyboard shortcut, but instead provides a more general `InputGesture` array, for describing input device gestures to execute the command.

This example, from the WPF HelpCommands component, creates a Help command:

```
var commandItem = m_commandService.RegisterCommand(  
    new CommandDef(  
        new ContextMenuHelpTag() { Index = i },  
        null,  
        Groups.Help,  
        "Help".Localize(),  
        new string[] { "Help".Localize() },  
        "Help".Localize(),  
        null,  
        null,  
        Sce.Atf.Applications.CommandVisibility.None), this);
```

Topics in this section

Links on this page to other topics

[ATF Command Groups](#), [Using Commands in ATF](#)

ATF Command Groups

A command group is a set of logically related commands. For example, the Edit commands Cut, Copy, Paste, and Delete fall in a group. In a menu, groups are separated with divider lines; a tool strip contains a command group.

When you register a command, you specify its command group.

ATF defines a standard set of groups, and you can define your own groups. Each group also has a standard order for its commands.

Standard Command Groups

The `StandardCommandGroup` enumeration specifies ATF's standard command groups, used for both WinForms and WPF. ATF's standard command components, such as `StandardFileCommands`, use these groups, and you can use these groups for commands you define.

The group is named after the group's first command. For instance, the "FileSave" command group contains the FileSave, FileSaveAs, FileSaveAll, and FileClose standard commands.

You can add any command to a group, standard or custom. In general, the added command goes to the end of the group, with the exceptions noted in the following table. This table indicates where these command groups place new commands in their group.

Command Group	Notes
FileExit	New commands are added before Exit.
EditCut	New commands are added in between Paste and Delete.
EditPreferences	New commands are added at the top of the list.
ViewControls	Placeholder for top-level controls that can be hidden or shown. New commands are added at the top of the list.
FormatAlign	New commands are added at the top of the list.
WindowDocuments	List of open documents. New commands are added at the top of the list.
HelpAbout	New commands are added before this command.

This table does not show all command groups; consult the `StandardCommandGroup.cs` file for the current list.

The command groups include groups for miscellaneous commands in the group, such as the `FileOther` and `EditOther` groups.

Topics in this section

Links on this page to other topics

No links

Using Standard Command Components

ATF provides components that add commands commonly used in applications.

To use one of these components, do the following:

1. Add the component to the MEF type catalog, as described in [How MEF is Used in ATF](#).
2. Add the `CommandService` component to the MEF type catalog. `CommandService` is required for commands.
3. Add a command client to implement `ICommandClient` for each command the component provides.

These components handle all the command registration, so you only need to implement the command clients. For details, see [Creating Command Clients](#).

Each of these components is used in at least one sample.

The following table shows the ATF components that add standard commands and the commands they add. The "Added Command" column contains the `StandardCommand` enum for the command unless marked with an asterisk *, in which case it is an `enum` internal to the defining class.

Component	Component purpose	Menu	Added Menu Item	Added Command (<code>StandardCommand</code>)	Command Function
DefaultTabCommands	Provide default commands related to document tab controls	Context menu	Close	*CloseCurrentTab	Close the current Tab panel
			Close All But This	*CloseOtherTabs	Close all but the current Tab panel
			Copy Full Path	*CopyFullPath	Copy the file path for the tab's document
			Open Containing Folder	*OpenContainingFolder	Open the folder containing the tab's document in Windows Explorer
HelpAboutCommand	Display a dialog box with a description of the application	Help	About	HelpAbout	Display information about the application
PropertyEditingCommands	Provide property editing commands that can be used inside <code>PropertyGrid</code> -like controls, such as property editors	Context menu	Reset Current	*ResetCurrent	Reset the current property to its default value
			Reset All	*ResetAll	Reset all properties to their default values
			Copy Property	*Copy	Copy this property's value to the clipboard
			Paste Property	*Paste	Paste the clipboard into this property's value

			View In Text Editor	*ViewInTextEditor	Open the file in the associated text editor
RecentDocumentCommands	Provide menu commands to open and pin recent documents	File	Document name		Open the document named in menu item
		File	Pin active document name	*Pin	Pin the active document to the recently used list
RenameCommand	Define a mass rename command	Edit	Rename...	*Rename	Rename selected objects
SourceControlCommands	Implement source control commands	File > Source Control	Enable	*Enabled	Enable source control
			Open Connection...	*Connection	Open the source control connection
			Add	*Add	Add to source control
			Check In	*CheckIn	Check in to source control
			Check Out	*CheckOut	Check out from source control
			Get Latest Version	*Sync	Get latest version from source control
			Revert	*Revert	Revert add or check out from source control
			Refresh Status	Refresh	Refresh status in source control
			Reconcile Offline Work...	*Reconcile	Reconcile offline work
StandardEditCommands	Implement standard Edit menu commands. For more information, see StandardEditCommands and Instancing .	Edit	Cut	EditCut	Cut the selection and place it on clipboard
			Copy	EditCopy	Copy the selection and place it on clipboard
			Paste	EditPaste	Paste the contents of clipboard and make that the new selection

			Delete	EditDelete	Delete the selection
StandardEditHistoryCommands	Implement standard Edit Undo and Redo commands	Edit	Undo	EditUndo	Undo the last change
			Redo	EditRedo	Redo the last edit
StandardFileCommands	Add standard file commands	File	Save	FileSave	Save the active file
			Save As ...	FileSaveAs	Save the active file under a new name
			Save All	FileSaveAll	Save all open files
			Close	FileClose	Close the active file
StandardFileExitCommand	Add File/Exit command that closes application's main form	File	Exit	FileExit	Exit the application
StandardLayoutCommands	Provide standard layout commands	Format > Align	Lefts	FormatAlignLefts	Align left sides of selected items
		Format > Align	Tops	FormatAlignTops	Align tops of selected items
		Format > Align	Rights	FormatAlignRights	Align right sides of selected items
		Format > Align	Centers	FormatAlignCenters	Align centers of selected items
		Format > Align	Bottoms	FormatAlignBottoms	Align bottoms of selected items
		Format > Align	Middles	FormatAlignMiddles	Align middles of selected items
		Format > Size	Make Widths Equal	FormatMakeWidthEqual	Make selected items have the same width
		Format > Size	Make Heights Equal	FormatMakeHeightEqual	Make selected items have the same height
		Format > Size	Make Equal	FormatMakeSizeEqual	Make selected items have the same size

StandardLockCommands	Implement standard Lock and Unlock commands	Edit	Lock	EditLock	Lock the selection to disable editing
			Unlock	EditUnlock	Unlock the selection to enable editing
StandardPrintCommands	Implement standard printing commands	Print	Print...	Print	Print the active document
			Page Setup...	FilePageSetup	Set up page for printing
			Print Preview...	FilePrintPreview	Show a print preview of the active document
StandardSelectionCommands	Implement standard selection commands	Edit	Select All	EditSelectAll	Select all items
			Deselect All	EditDeselectAll	Deselect all items
			Invert Selection	EditInvertSelection	Select unselected items and deselect selected items
StandardShowCommands	Implement standard Show commands	View	Hide	ViewHide	Hide all selected objects
			Show	ViewShow	Show all selected objects
			Show Last Hidden	ViewShowLast	Show the last hidden object
			Show All	ViewShowAll	Show all hidden objects
			Isolate	ViewIsolate	Show only the selected objects and hide all others
StandardViewCommands	Implement standard viewing commands	View	Frame Selection	ViewFrameSelection	Frame all selected objects in the current view
			Frame All	ViewFrameAll	Frame all objects in the current view
TargetCommands	Commands to operate on currently selected targets	Context menu	Edit Vita Target in Neighborhood	VitaNeighborhood	Edit Vita Target in Neighborhood
			Add New TTT	Add New TTT	Create a new target of type TTT

			Remove TTT	Remove TTT	Remove selected target of type TTT
WindowLayoutServiceCommands	Provide menu options and GUIs for managing and using layouts	Window	Layouts > Save Layout As...	*SaveLayoutAs	Save layout in selected file
			Layouts > Manage Layouts...	*ManageLayouts	Display Manage Layouts dialog

Topics in this section

Links on this page to other topics

[Creating Command Clients](#), [How MEF is Used in ATF](#), [StandardEditCommands](#) and [Instancing](#)

Registering Menus and Commands

The `CommandService` component handles creating menus and registering commands, which makes them available for use in menus and tool strips. After registration, you create command clients for the commands you have just registered to specify what the commands actually do. For details, see [Creating Command Clients](#).

`CommandService` implements `ICommandService`, which provides methods to create menus and register commands. `ICommandService` is different for WinForms and WPF, though both offer similar capabilities.

If you use one of ATF's components for standard commands, as described in [Using Standard Command Components](#), these standard commands are already registered for you. You only need to register your custom commands.

Topics

- Registering Menus
 - WinForms Menus
 - WPF Menus
- Registering Commands
 - Registering Commands in WinForms
 - Registering Commands in WPF

Registering Menus

You don't need to create most menus, because `CommandService` creates a standard set of menus in the `StandardMenus` class for both WinForms and WPF:

- File
- Edit
- View
- Modify
- Format
- Window
- Help

New menus appear between the standard Format and Window menus in the order in which they are added. A new menu also defines a new command group. All commands you add to the new menu that have toolbar visibility are grouped in a single toolbar.

WinForms Menus

Menus are created with the `ICommandService.RegisterMenu()` method:

```
void RegisterMenu(MenuInfo info);
```

The constructor for `MenuInfo` simply requires a tag and text for the UI and description:

```
public MenuInfo(  
    object menuTag,  
    string menuText,  
    string description);
```

The `menuTag` parameter is one of the `StandardMenu` enumerations.

For example, the standard File menu is created in ATF by first creating a `MenuInfo` in the `MenuInfo` class:

```
public static MenuInfo File =  
    new MenuInfo(StandardMenu.File, "File".Localize(), "File Commands".Localize());
```

and then registering the menu using that `MenuInfo` object in `CommandService` (which derives from `CommandServiceBase` and that implements `ICommandService`, so the `RegisterMenu()` method is available):

```
RegisterMenu(StandardMenus.File);
```

You can create custom menus similarly, using the application's `CommandService` object.

WPF Menus

Menus are created with the `ICommandService.RegisterMenu()` method:

```
void RegisterMenu(MenuDef definition);
```

The constructor for `MenuDef`, similarly to `MenuItemInfo`, simply requires a tag and text for the UI and description:

```
public MenuDef(object menuTag, string text, string description);
```

The `menuTag` parameter is one of the `StandardMenu` enumerations.

For example, the standard File menu is created in ATF by first creating a `MenuDef` in the `StandardMenus` class:

```
public static MenuDef File =
    new MenuDef(StandardMenu.File, Localizer.Localize("_File"), Localizer.Localize("File
Commands"));
```

and then registering the menu using that `MenuDef` object in `CommandService`, which implements `ICommandService` so the `RegisterMenu()` method is available:

```
RegisterMenu(StandardMenus.File);
```

You can create custom menus similarly, using the application's `CommandService` object.

Registering Commands

You need to provide two objects (or equivalent information) to register a command:

- `CommandInfo` or `CommandDef`: Describe the command's interaction with the UI, as described in [CommandInfo and CommandDef Classes](#).
- `ICommandClient`: Tell what the command does. To learn more about this, see [Creating Command Clients](#).

`ICommandService` provides several `RegisterCommand()` methods, including convenience methods with parameters containing the information `CommandInfo` or `CommandDef` provides, so you don't need to explicitly create a `CommandInfo` or `CommandDef`.

The order in which `RegisterCommand()` is called in your application determines the order in which commands and groups of commands appear in menus and tool strips.

Registering Commands in WinForms

The simplest registration method is this:

```
void RegisterCommand(CommandInfo info, ICommandClient client);
```

This line from `StandardEditCommands` registers the `EditCut` command:

```
m_commandService.RegisterCommand(CommandInfo.EditCut, this);
```

`CommandInfo.EditCut` was already created in the `CommandInfo` class. `m_commandService` is an `ICommandService` instance.

`StandardEditCommands` not only registers this command, but provides an implementation, because it implements `ICommandClient`. In the line above, `this` specifies the required `ICommandClient` parameter.

This longer call in the `SourceControlCommands` component specifies all the command info parameters for the `Enabled` command:

```
m_sourceControlEnableCmd = m_commandService.RegisterCommand(
    Command.Enabled,
    StandardMenu.File,
    SourceControlCommandGroup.OnOff,
    "Source Control/Enable".Localize(),
    "Enable source control".Localize(),
    Keys.None,
    Resources.SourceControlEnableImage,
    CommandVisibility.Menu | CommandVisibility.Toolbar,
    this);
```

This registration specifies the command visibility explicitly. This component also provides the `ICommandClient` for the command.

Registering Commands in WPF

The simplest registration method is this:

```
ICommandItem RegisterCommand(CommandDef definition, ICommandClient client);
```

`ICommandItem` is an interface for command items, which provides properties to get the various pieces of information in a `CommandDef` object.

This call in the WPF `HelpCommands` component registers a Help command:

```
var commandItem = m_commandService.RegisterCommand(
    new CommandDef(
        new ContextMenuHelpTag() { Index = i },
        null,
        Groups.Help,
        "Help".Localize(),
        new string[] { "Help".Localize() },
        "Help".Localize(),
        null,
        null,
        Sce.Atf.Applications.CommandVisibility.None), this);
```

This registration specifies the command visibility explicitly. This component also provides the `ICommandClient` for the command.

Topics in this section

Links on this page to other topics

[CommandInfo and CommandDef Classes](#), [Creating Command Clients](#), [Using Standard Command Components](#)

Creating Command Clients

ICommandClient interface

A command client implements the `ICommandClient` interface, which consists of these methods:

```
bool CanDoCommand(object commandTag);
void DoCommand(object commandTag);
void UpdateCommand(object commandTag, CommandState commandState);
```

Both WinForms and WPF use the same `ICommandClient` interface.

Topics

- `ICommandClient` interface
- `CanDoCommand()` Method
- `UpdateCommand()` Method
 - `CommandState` Name
 - `CommandState` Check Mark
- `DoCommand()` Method

CanDoCommand() Method

The `CanDoCommand()` method simply indicates whether or not the command can be performed, usually considering the current context. It is called in response to application events. When the method returns false, the command is disabled, which is indicated in the UI by graying out menu items and tool strip controls. The method's `commandTag` argument is the same tag the command was registered with.

Here is an example from the [ATF Circuit Editor Sample MasteringCommands](#) class. The main thing to notice is that the current context is used to determine whether or not the command is possible by checking if at least one item is selected. And if there is no current context, the command can't be done either.

```
public bool CanDoCommand(object commandTag)
{
    bool enabled = false;
    var context = m_contextRegistry.GetActiveContext<CircuitEditingContext>();
    if (context != null)
    {
        ISelectionContext selectionContext = context.As<ISelectionContext>();
        if (CommandTag.CreateMaster.Equals(commandTag))
        {
            enabled = selectionContext.GetLastSelected<Element>() != null; // at least one module
selected
        }
        else if (CommandTag.ExpandMaster.Equals(commandTag))
        {
            enabled = selectionContext.GetLastSelected<SubCircuitInstance>() != null; // at least
one mastered instance selected
        }
    }

    return enabled;
}
```

UpdateCommand() Method

`UpdateCommand()`'s purpose is to update the command's `CommandState`, which contains two properties: the command name and menu item check mark. Application events trigger calling `UpdateCommand()`.

CommandState Name

This is the menu's name, initially specified in the command's `CommandInfo` when the command was registered. You can change this to reflect

the status of the command. For instance, the `RecentDocumentCommands` component updates this text with a document name:

```
public virtual void UpdateCommand(object commandTag, CommandState state)
{
    if (commandTag is RecentDocumentInfo)
    {
        RecentDocumentInfo info = (RecentDocumentInfo)commandTag;
        state.Text = info.Uri.LocalPath;
    }
}
```

CommandState Check Mark

The check mark indicates whether or not to display a check on the command's menu. For example, the ATF Timeline Editor Sample uses this feature to place a check on the Edit > Interval Splitting Mode menu item when this mode is active:

```
public void UpdateCommand(object commandTag, CommandState commandState)
{
    TimelineDocument document = m_contextRegistry.GetActiveContext<TimelineDocument>();
    if (document == null)
        return;

    if (commandTag is Command)
    {
        switch ((Command)commandTag)
        {
            case Command.ToggleSplitMode:
                commandState.Check = document.SplitManipulator != null ? document.SplitManipulator
.Active : false;
                break;
        }
    }
}
```

DoCommand() Method

This method performs the command when the command is triggered by the user selecting the command in the menu or tool strip. Its argument is the command tag with which the command was registered. It may simply call other methods, as in this example from the `StandardEditCommands` component. This method uses `commandTag` to determine which command is being done, and switches accordingly:

```
void ICommandClient.DoCommand(object commandTag)
{
    switch ((StandardCommand)commandTag)
    {
        case StandardCommand.EditCut:
            Cut();
            break;

        case StandardCommand.EditDelete:
            Delete();
            break;

        case StandardCommand.EditCopy:
            Copy();
            break;

        case StandardCommand.EditPaste:
            Paste();
            break;
    }
}
```

The `RecentDocumentCommands` component's `DoCommand()` opens the document associated with the menu item, removing the menu item when the document is invalid:

```
public virtual void DoCommand(object commandTag)
{
    if (commandTag is RecentDocumentInfo) // recently used file?
    {
        RecentDocumentInfo info = (RecentDocumentInfo)commandTag;
        IDocumentClient client;
        if (m_typeToClientMap.TryGetValue(info.Type, out client))
        {
            IDocument document = m_documentService.OpenExistingDocument(client, info.Uri);
            if (document == null)
                RemoveDocument(info);
        }
    }
}
```

Note that this example from the [ATF Circuit Editor Sample](#) `MasteringCommands` class uses the current context to perform the command:

```
public void DoCommand(object commandTag)
{
    var context = m_contextRegistry.GetActiveContext<CircuitEditingContext>();
    if (CommandTag.CreateMaster.Equals(commandTag))
    {
        SubCircuitInstance subCircuitInstance = null;

        var masterContext = context.DomNode.GetRoot().Cast<CircuitEditingContext>();
        ...
    }
}
```

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF Timeline Editor Sample](#)

Using WinForms Commands in WPF

You can use WinForms-based command components in a WPF based application. This is made possible by the `CommandServiceAdapter` component, which implements the WinForms `ICommandService` so it works in a WPF environment. In other words, it adapts `Sce.Atf.Wpf.Applications.ICommandService` to `Sce.Atf.Applications.ICommandService`.

The `CommandServiceAdapter` component exports `Sce.Atf.Applications.ICommandService`. If this component is included in an application, the WinForms-based components that need `ICommandService` import it from `CommandServiceAdapter`, rather than the WinForms-based `CommandService` component. This allows the components to function with the WPF application. The application would not include the WinForms-based `CommandService` component in its MEF catalog in this case.

`CommandServiceAdapter` also creates an object of the `CommandClientAdapter` class, which adapts the `ICommandClient` interface for WinForms-based command clients to WPF. This allows command clients implementing `Sce.Atf.Applications.ICommandClient` to operate properly in WPF.

The `StandardInteropParts` class includes the `CommandServiceAdapter` component in a MEF `TypeCatalog`:

```
public class StandardInteropParts
{
    /// <summary>
    /// Gets type catalog for all components</summary>
    public static ComposablePartCatalog Catalog
    {
        get
        {
            return new TypeCatalog(
                typeof(MainWindowAdapter),
                typeof(CommandServiceAdapter),
                typeof(ContextMenuService),
                typeof(DialogService),
                typeof(ControlHostServiceAdapter)
            );
        }
    }
}
```

This catalog can be included in an `AggregateCatalog`, as in this line from the [ATF Wpf App Sample](#):

```
return new AggregateCatalog(typeCatalog, StandardInteropParts.Catalog, StandardViewModels.Catalog
);
```

After this, the WPF application can use the WinForms-based commands component — with no additional support from the application.

Topics in this section

Links on this page to other topics

[ATF Wpf App Sample](#)

Using Context Menus

Context menus display a list of commands, typically in response to mouse clicks on a control.

For WinForms, the `ICommandService` interface provides this method to display a context menu for commands, given a collection of commands:

```
void RunContextMenu(IEnumerable<object> commandTags, Point screenPoint);
```

This section discusses how to get the command list and display it in a context menu.

Topics

- [Getting a Command List](#)
- [Implementing a Command List Provider](#)
- [Importing a Command List](#)
- [Displaying the Context Menu](#)

Getting a Command List

The `IContextMenuCommandProvider` interface contains the following method to collect commands appropriate for a context:

```
IEnumerable<object> GetCommands(object context, object target);
```

The parameter `target` is the object clicked on, and `context` is the context containing `target`.

Thus any class implementing `IContextMenuCommandProvider` can provide a collection of commands for a given context.

Implementing a Command List Provider

This example from `StandardEditCommands` provides a list of commands, if the provided context is a selection and instancing context and thus appropriate for editing:

```
IEnumerable<object> IContextMenuCommandProvider.GetCommands(object context, object clicked)
{
    ISelectionContext selectionContext = context.As<ISelectionContext>();
    IInstancingContext instancingContext = context.As<IInstancingContext>();
    if ((selectionContext != null) && (instancingContext != null))
    {
        return new object[]
        {
            StandardCommand.EditCut,
            StandardCommand.EditCopy,
            StandardCommand.EditPaste,
            StandardCommand.EditDelete,
        };
    }

    return Enumerable<object>.Instance;
}
```

Importing a Command List

The `ControlHostService` for WinForms has the following import:

```
[ImportMany]
private IEnumerable<Lazy<IContextMenuCommandProvider>> m_contextMenuCommandProviders;
```

This allows the component to get a collection of objects on which it can call `IContextMenuCommandProvider.GetCommands()` to get a list

of commands appropriate to the context. A variety of components export `IContextMenuCommandProvider`, including several that create standard commands, such as `StandardEditCommands` shown previously and `DefaultTabCommands`.

Displaying the Context Menu

`ControlHostService` for WinForms uses the collection `contextMenuCommandProviders` it imported, as shown previously, to implement this method that displays a context menu in response to a right-click event:

```
private void dockPaneStrip_MouseUp(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        DockPaneStripBase dockPaneStrip = sender as DockPaneStripBase;

        ControlInfo info = FindControlInfo(m_activeDockContent.Controls[0]);
        IEnumerable<object> commands =
            m_contextMenuCommandProviders.GetCommands(null, info);

        Point screenPoint = dockPaneStrip.PointToScreen(new Point(e.X, e.Y));

        m_commandService.RunContextMenu(commands, screenPoint);
    }
}
```

Topics in this section

Links on this page to other topics

No links

Adaptation in ATF

Adaptation allows an object of one type to behave like an object of another type.

- [What is Adaptation?](#): General discussion of what adaptation is and how it is used in ATF.
 - [General Adaptation Interfaces](#): Survey of interfaces for adaptation.
 - [General Adaptation Classes](#): Describe fundamental classes in adaptation.
 - [Control Adapters](#): Discussion of control adapters, which add abilities to controls without changing the control.
 - [Other Adaptation Classes](#): Survey of non-control adapter classes that perform various kinds of adaptation.
 - [Adapting to All Available Interfaces](#): A detailed example of how adaptation works and what you need to do to make it work showing how adapters can be obtained for any interface DOM adapters implement.
-

What is Adaptation?

Adaptation is the process of allowing an object of one type to behave like another type. In particular, this means allowing the object to use the API of another type: methods, properties, and events. You can think of adaptation as dynamic casting.

An adapter for an object `A` to type `T` is an object that has all the capability and data of `A` and also behaves like an object of type `T`. In this case, `A` would be the adaptee of the adapter. An object can be its own adapter.

Adapters implement the `IAdapter` interface. Adaptees implement the `IAdaptable` interface, which obtains an adapter to a given type.

Adaptation is a powerful tool used widely in ATF. Adaptation gives you the ability to extend the behavior of objects without using inheritance. To add new functionality to an object's class, you don't need to make it implement an interface or derive from another class: simply add an adapter with the new capabilities.

DOM adapters are one of the most important kinds of adapters. For more information, see the [ATF Programmer's Guide: Document Object Model \(DOM\)](#).

You can also use adaptation with controls. `AdaptableControl`, and its Direct2D version `D2dAdaptableControl`, are base controls that can take control adapters. Classes that derive from `ControlAdapter` add new functions to controls without changing the controls. ATF has a large set of `ControlAdapter` classes, and you can create your own by deriving from `ControlAdapter` or one of its ancestors.

The Adaptation Framework's interfaces and classes are in two namespaces:

- `Sce.Atf.Adadaptation`: contains general interfaces and classes.
 - `Sce.Atf.Controls.Adaptable`: has the interfaces and classes for control adapters.
-

Topics in this section

Links on this page to other pages

[ATF Documentation](#)

General Adaptation Interfaces

The namespace `Sce.Atf.Adaptation` contains several general purpose, simple interfaces for adaptation.

IAdaptable Interface

`IAdaptable` is implemented by every class that wants to be adaptable, that is, have adapters to extend its reach. Its method comes right to the point:

```
object GetAdapter(Type type);
```

Contents

- [IAdaptable Interface](#)
- [IDecoratable Interface](#)
- [IAdapter Interface](#)
- [IAdapterCreator Interface](#)

This method returns the adapter for the given type. Note that the object returned is of the type specified, so by definition, it provides the desired interface for its type. Only one adapter is returned. The `IDecoratable` interface can be used to get a collection of adapters. Also, the `Adapters.AsAll()` method returns all adapters for a type. For information on this class, see [Adapters Class](#).

IDecoratable Interface

This interface's job is to get all decorators for a type:

```
IEnumerable<object> GetDecorators(Type type);
```

In this case, decorator means adapter, so this method returns all adapters of the given type. `IDecoratable` complements `IAdaptable`, whose `GetAdapter()` returns a single adapter.

IAdapter Interface

`IAdapter` is implemented by adapters, which adapt an adaptee. `IAdapter`'s declaration indicates it implements two other interfaces:

```
public interface IAdapter : IAdaptable, IDecoratable
```

That this interface also implements `IAdaptable` underscores the fact that an adapter is also an adaptee. Its one property, `Adaptee`, gets and sets the adaptee for the adapter.

The `Adapter` class implements `IAdapter`.

IAdapterCreator Interface

`IAdapterCreator` works for adapter creators and is used to build adapter factories with its methods:

- `bool CanAdapt(object adaptee, Type type)`: can an adapter of the desired type be created for this potential adaptee?
- `object GetAdapter(object adaptee, Type type)`: get an adapter of the given type for the adaptee or `null` if none is available.

The `AdapterCreator` class implements `IAdapterCreator`. `AdapterCreator` can be used to make property editors show the attributes listed in property descriptor definitions. For more information, see [Metadata-Driven Property Editing](#).

The class `Sce.Atf.Dom.TypeAdapterCreator<T>` implements `IAdapterCreator` for `DomNodes`.

Topics in this section

Links on this page to other pages

[General Adaptation Classes, WinForms Application](#)

General Adaptation Classes

The namespace `Sce.Atf.Adaptation` offers several classes fundamental to adaptation.

Adapters Class

`Adapters` provides the key adaptation methods that actually provide adapters. `Adapters` provides these methods as extension methods. Note that these extension methods are on `object`, so they apply to any object. Simply adding `using Sce.Atf.Adaptation;` to a class file makes these extension methods available.

As() Methods

Examining the `As()` method in detail illuminates how adaptation works. Here's the entire method:

Contents

- Adapters Class
- As() Methods
- Cast() Methods
- Is() Methods
- IEnumerable Adaption Methods
- Using Adaptation Methods
- Adapter Class
- Classes On Collections
- AdapterCreator Class
- AdaptablePath Class

```
public static object As(this object reference, Type type)
{
    if (reference == null)
        return null;

    if (type == null)
        throw new ArgumentNullException("type");

    // is the adapted object compatible?
    if (type.IsAssignableFrom(reference.GetType()))
        return reference;

    // try to get an adapter
    var adaptable = reference as IAdaptable;
    if (adaptable != null)
    {
        object adapter = adaptable.GetAdapter(type);
        if (adapter != null)
            return adapter;
    }

    return null;
}
```

After the parameter checks, the method calls `type.IsAssignableFrom(reference.GetType())` to determine whether an object of the given `type` is assignable from an object of the type of `reference`. The comments on `public virtual bool IsAssignableFrom(Type c)` say it returns true if any of these are true:

- `c` and the current `Type` (of the object on which `IsAssignableFrom()` is invoked) represent the same type.
- The current `Type` is in the inheritance hierarchy of `c`.
- The current `Type` is an interface that `c` implements.
- `c` is a generic type parameter and the current `Type` represents one of the constraints of `c`.

The method returns `false` if none of these conditions are true or `c` is `null`. These conditions spell out the type compatibility for the assignment.

When `IsAssignableFrom()` is true, the object reference itself is returned. The object serves as its own adapter, because it can do whatever an object of type can do. In other words, it can be cast as type.

If casting is not possible, the method then tries to get an adapter for reference. It can only do this when reference implements `IAdaptable`, so `GetAdapter()` can be invoked on reference to get an adapter to return. Failing this, `As()` returns null to indicate no adapter is available.

The method `As<T>(this object reference)` works similarly:

```
public static T As<T>(this object reference)
    where T : class
{
    if (reference == null)
        return null;

    // try a normal cast
    var converted = reference as T;

    // if that fails, try to get an adapter
    if (converted == null)
    {
        var adaptable = reference as IAdaptable;
        if (adaptable != null)
            converted = adaptable.GetAdapter(typeof(T)) as T;
    }

    return converted;
}
```

After parameter checks, the method tries the cast `var converted = reference as T;`. If that fails, it attempts to get an adapter. Again, this requires that `reference` implement `IAdaptable`.

The method `As<T>(this IAdaptable adaptable)` is almost the same.

Note that the `As()` methods always return an object of the type requested, or a type that is assignable from that type.

Cast() Methods

The `Cast()` methods are almost the same as their `As()` counterparts, returning an adapter, and are implemented by calling `As()` methods. The only difference is that if no adapter is found, they raise an exception instead of returning null.

Like `As()` methods, `Cast()` methods always return an object of the type requested, or a type that is assignable from that type.

Is() Methods

`Is()` methods simply determine whether an adapter is available. They are implemented by calling the appropriate `As()` method.

IEnumerable Adaption Methods

Some methods in `Adapters` work on an `IEnumerable` of adapters in a variety of ways:

- `IEnumerable<object> AsAll(this object reference, Type type)`: Get all adapters that can convert a reference to the given type.
- `IEnumerable<T> AsAll<T>(this object reference)`: Get all adapters that can convert a reference to the given type `T`.
- `IEnumerable<T> AsAll<T>(this IDecoratable decoratable)`: Get an enumeration of all adapters that can convert a reference to the given type.
- `IEnumerable<object> AsIEnumerable(this IEnumerable enumerable, Type type)`: Get an adapter that converts an enumerable to an enumerable of another type.
- `IEnumerable<T> AsIEnumerable<T>(this IEnumerable enumerable)`: Return an enumeration for an adapter that converts an enumerable to an enumerable of another type `T`.
- `bool Any(this IEnumerable enumerable, Type type)`: Returns whether any of the items in the enumerable are adaptable to the given type.
- `bool Any<T>(this IEnumerable enumerable)`: Return whether any of the items in the enumerable are adaptable to the given type `T`.
- `bool All(this IEnumerable enumerable, Type type)`: Return whether all of the items in the enumerable are adaptable to the given type.
- `bool All<T>(this IEnumerable enumerable)`: Return whether all of the items in the enumerable are adaptable to the given type `T`.

Using Adaptation Methods

The adaptation methods in `Adapters` are widely used in ATF, especially for the ATF DOM. It is very common to use the `As<>T()` and `Cast<>T()` methods to cast a `DomNode` to another type for which it has a DOM adapter. The DOM adapter class implements the adaptation interfaces:

```
public abstract class DomNodeAdapter : IAdapter, IAdaptable, IDecoratable
```

The adaptation process in the DOM follows this sequence:

1. A type is defined, typically in a type definition file. The ATF DOM provides excellent support for type definitions in the XML Schema Definition (XSD) language.
2. The XML schema is loaded, if one is used.
3. A DOM adapter is defined for types whose data is represented by a `DomNode` in the application data.
4. The DOM adapter is initialized to its `DomNode` by calling any of the `Adapters` methods `As<>T()`, `Cast<>T()`, or `Is<>T()`. The returned value from `As<>T()` or `Cast<>T()` can then be used with the API of the DOM adapter and holds the application data of the `DomNode`, because it is still a `DomNode`.

For example, the [ATF Simple DOM No XML Editor Sample](#) uses this line to treat a document as a context:

```
EventSequenceContext context = Adapters.As<EventSequenceContext>(document);
```

`EventSequenceContext` is a DOM adapter derived from `EditingContext` that ultimately derives from `DomNodeAdapter`:

```
EventSequenceContext : EditingContext
```

`EventSequenceContext` is also defined as a DOM adapter for the root type of the data, "eventSequenceType", which is also the type of the document:

```
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceContext>());
```

For details on using adaptation in the DOM, see the [ATF Programmer's Guide: Document Object Model \(DOM\)](#).

Adapter Class

`Adapter` provides an example of an adapter and is used as an adapter a few places in ATF:

```
public class Adapter : IAdapter, IAdaptable, IDecoratable
```

In addition to `IAdapter`, `Adapter` implements `IAdaptable` and `IDecoratable`. (This is more of a reminder than a requirement, because `IAdapter` implements both of these anyway.)

The `Adaptee` property gets and sets the adaptee, that is, the object being adapted:

```
public object Adaptee
{
    get { return m_adaptee; }
    set
    {
        if (m_adaptee != value)
        {
            object oldAdaptee = m_adaptee;
            m_adaptee = value;
            OnAdapteeChanged(oldAdaptee);
        }
    }
}
```

The `Adapter` class gets adapters using the methods in `Adapters`. For details on `Adapters` methods, see [Adapters Class](#).

The casting methods (`As<>T()`, `Cast<>T()`, and `Is<>T()`) simply use the corresponding `Adapters` methods. For example, `As<T>()` calls

```
Adapters.As<T>():
```

```
public T As<T>()
    where T : class
{
    return Adapters.As<T>(this);
}
```

The `GetAdapter()` method is more involved than the casting methods:

```
public object GetAdapter(Type type)
{
    // see if this can adapt
    object adapter = Adapt(type);

    // if not, let the adaptee handle it
    if (adapter == null)
        adapter = m_adaptee.As(type);

    return adapter;
}
```

It first tries the `Adapt()` method to determine if the object itself can serve as an adapter:

```
protected virtual object Adapt(Type type)
{
    // default is to return this if compatible with requested type
    if (type.IsAssignableFrom(GetType()))
        return this;

    return null;
}
```

`Adapt()` calls `type.IsAssignableFrom(GetType())` to determine whether an object of the given `type` is assignable from an object of the type of the `Adapter` object. When `IsAssignableFrom()` is true, `Adapt()` returns `this`, the object itself. `GetAdapter()` returns `this` in turn.

If the object can't serve as its own adapter, `GetAdapter()` makes this assignment:

```
adapter = m_adaptee.As(type);
```

`m_adaptee.As(type)` invokes `Adapters.As()` on the adaptee (the object being adapted) because:

- `m_adaptee` holds the `Adaptee` property value.
- `Adapters.As()` is an extension method on `object`.
- `Adapter` is in the `Sce.Atf.Adaptation` namespace, same as `Adapters`.

Here's another example, returning an `IEnumerable` of adapters:

```
public IEnumerable<object> GetDecorators(Type type)
{
    // see if this can adapt
    object adapter = Adapt(type);
    if (adapter != null)
        yield return adapter;

    foreach (object obj in m_adaptee.AsAll(type))
        yield return obj;
}
```

Classes On Collections

Several `Sce.Atf.Adaptation` classes operate on collections:

- `AdaptableActiveCollection`: Collection representing active items that uses adaptation on items implementing `IAdaptable` to

convert them to other types. It's used by both the `ContextRegistry` and `DocumentRegistry` components.

- `AdaptableCollection`: Wrap an `ICollection` of one type to implement `ICollection` adapted to another type.
- `AdaptableList`: Wrap an `IList` of one type to implement `IList` adapted to another type.
- `AdaptableSelection`: Collection representing a selection. It uses adaptation to convert to other types. It is used in `Sce.Atf.Dom.EditingContext` from which several samples derive editing contexts, as well as in `Sce.Atf.Dom.SelectionContext`.

AdapterCreator Class

`AdapterCreator` implements `IAdapterCreator` and appears in all the samples that use the ATF DOM in a line like this:

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

`AdapterCreator` is used here to make property editors show the attributes listed in the property descriptor definitions. For more information, see [Metadata-Driven Property Editing](#).

AdaptablePath Class

`AdaptablePath` represents a path in a tree or graph, such as a `DomNode` tree representing application data. `AdaptablePath` can adapt the last element of the path to a requested type. For example, DOM property descriptors (like `AttributePropertyDescriptor`) try to adapt an `AdaptablePath` object to a `DomNode`, and this succeeds if the last element of the path can be adapted to a `DomNode`. `AdaptablePath` is used in various classes that use paths in trees and lists.

Topics in this section

Links on this page to other pages

[ATF Documentation](#), [ATF Simple DOM No XML Editor Sample](#), [Authoring Tools Framework](#), [WinForms Application](#)

Control Adapters

Control adapters give you an easy way to add abilities to certain controls without changing the control at all. ATF has a large number of control adapters performing a variety of functions that you can use on your controls.

These adapters and their interfaces are in the `Sce.Atf.Controls.Adaptable` namespace.

Using Control Adapters

To understand how control adapters work, consider how they are used.

Control adapters are used on adaptable controls, which derive from `AdaptableControl`. ATF provides two: `AdaptableControl` itself and `D2dAdaptableControl`, which is the one that is most used.

Consider this simple example from the `Editor` class's constructor in [ATF Circuit Editor Sample](#). It sets up a control that is displayed when the cursor hovers over a subcircuit in a circuit:

Contents

- Using Control Adapters
- ControlAdapter Class
- AdaptableControl Class
 - Adapt() Method
 - AdaptableControl Adapter Methods
 - AdaptableControl Context Methods
- Control Adapter Examples
- Control Adapter Interfaces
- Control Adapter Descriptions

```
m_d2dHoverControl = new D2dAdaptableControl();
m_d2dHoverControl.Dock = DockStyle.Fill;
var xformAdapter = new TransformAdapter();
xformAdapter.EnforceConstraints = false; //to allow the canvas to be panned to view negative
coordinates
m_d2dHoverControl.Adapt(xformAdapter,
    new D2dGraphAdapter<Module, Connection, ICircuitPin>(m_circuitRenderer, xformAdapter));
```

After the `D2dAdaptableControl` is constructed and configured, the control adapter `TransformAdapter` is created to provide transform information about the control. Next, `AdaptableControl.Adapt()` is called with a list of the adapters to add to the control. This adds the `TransformAdapter` plus a `D2dGraphAdapter` constructed inside the `Adapt()` call. This is all you need to do to get the benefits of the adapters for the control.

For further discussion, see [Adaptable Controls](#).

ControlAdapter Class

Control adapters all derive directly or indirectly from the abstract `ControlAdapter` class, which is never used as a control adapter itself. `ControlAdapter` implements `IControlAdapter`, which contains the following property and methods:

- `AdaptedControl`: Get the adapted control.
- `Bind(AdaptableControl control)`: Bind the adapter to the adaptable control. It is called in the order that the adapters were defined on the control.
- `BindReverse(AdaptableControl control)`: Bind the adapter to the adaptable control. It is called in the reverse order that the adapters were defined on the control.
- `Unbind(AdaptableControl control)`: Unbind the adapter from the adaptable control.

Control adapters override these methods.

Note that the binding related methods get the control they are binding to as a parameter, so it is available for the `AdaptedControl` property. Binding occurs when `AdaptableControl.Adapt()` is called. What binding accomplishes is discussed in the next section.

AdaptableControl Class

`AdaptableControl` augments `Control`, from which it derives. It has events before and after when adapters are adapted. It also overrides the base `Control` `OnMouseDown()` and `OnMouseMove()` methods. It overrides the `WndProc()` method so that it can call its overridden `OnPainting()` method.

Adapt() Method

The main thing the `Adapt()` method does is bind the control adapters to the control:

```
public void Adapt(params IControlAdapter[] adapters)
{
    OnAdapting(EventArgs.Empty);
    Adapting.Raise(this, EventArgs.Empty);

    // allow existing adapters to detach from context
    Context = null;

    foreach (IControlAdapter adapter in m_adapters)
    {
        adapter.Unbind(this);
        Dispose(adapter);
    }
    m_adapters.Clear();
    m_adapterCache.Clear();
    m_contextAdapterPreferred.Clear();

    m_adapters.AddRange(adapters);

    foreach (IControlAdapter adapter in m_adapters)
    {
        adapter.Bind(this);
    }

    for (int i = m_adapters.Count - 1; i >= 0; i--)
    {
        m_adapters[i].BindReverse(this);
    }

    OnAdapted(EventArgs.Empty);
    Adapted.Raise(this, EventArgs.Empty);

    Invalidate();
}
```

The first section of this method removes any previous binding, calling the `Unbind()` for each control adapter. It calls `Bind()` for each adapter in the order they were specified, and then calls `BindReverse()` for each adapter in the reverse order specified.

The desired order depends on whether the control adapter handles painting or mouse events. For painting events, the order is bottom up, because the top layer covers lower layers. For mouse events, the order is top up, because top layers handle events before layers below. Typically, a painting type adapter overrides only the `Bind()` method, while a mouse type adapter overrides only the `BindReverse()` method. It's just a matter of convention, however. The benefit of offering both `Bind()` and `BindReverse()` that are called in reverse orders is that you can put both kinds of control adapters in a single list and order them the way you want. For instance, you could put all the painting control adapters first in the list in the order that you want, followed by the mouse control adapters in their desired order.

AdaptableControl Adapter Methods

The `ControlAdapters` property gets a list of all the adapters in their specified order.

Adapters are also retrieved and handled by these methods:

- `T As<T>()`: Get an adapter of the specified type, or `null` if none is available.
- `T Cast<T>()`: Get an adapter of the specified type, throwing an exception if none is available.
- `bool Is<T>()`: Return whether the control has the specified adapter of type `T`.
- `IEnumerable<T> AsAll<T>()` where `T : class`: Get all adapters of the specified type, or `null` if none.

The `AdaptableControl` can be converted into any of its adapters using the `IAdaptable.As()` method.

These convenience methods are implemented using utility private methods, like `As()`, that looks through the adapter list to find what's desired:

```

private object As(Type type)
{
    // try cache
    object adapter;
    if (m_adapterCache.TryGetValue(type, out adapter))
        return adapter;

    // try adapters (from top to bottom)
    for (int i = m_adapters.Count - 1; i >= 0; i--)
    {
        adapter = m_adapters[i];
        if (type.IsAssignableFrom(adapter.GetType()))
        {
            m_adapterCache.Add(type, adapter);
            return adapter;
        }
    }

    // finally, try the adaptable control itself
    if (type.IsAssignableFrom(GetType()))
    {
        m_adapterCache.Add(type, this);
        return this;
    }

    return null;
}

```

The `m_adapterCache` `Dictionary<Type, object>` caches the adapters for quick access by type. If not yet cached, the method goes through the list and uses the `IsAssignableFrom()` method previously mentioned to find a suitable adapter. Failing this, it tries the control itself, and returns what it finds.

AdaptableControl Context Methods

Adaptable controls have a `Context` property so their users can specify a desired context for using the control. For instance, the `Editor` class in [ATF Circuit Editor Sample](#) sets a context:

```
control.Context = editingContext;
```

In addition, a preferred context for a type (adapter) can be specified with the `RegisterContextAdapter(Type type, object adapter)` method. Once that's done, an adapter for a specified context can be obtained with these methods:

- `T ContextAs<T>()`: Get an adapter to the context of the given type, or `null` if none is available.
- `T ContextCast<T>()`: Get an adapter to the context of the given type. If none is available, throw an `AdaptationException`.
- `bool ContextIs<T>()`: Return whether the context can be adapted.

Control Adapter Examples

Examine `MouseWheelManipulator`, a simple adapter that scales the adaptable control viewing area when the user turns the mouse wheel. The constructor for this adapter takes an `ITransformAdapter`:

```
public MouseWheelManipulator(ITransformAdapter transformAdapter)
```

`MouseWheelManipulator`'s binding methods are:

```

protected override void BindReverse(AdaptableControl control)
{
    control.MouseWheel += control_MouseWheel;
}

protected override void Unbind(AdaptableControl control)
{
    control.MouseWheel -= control_MouseWheel;
}

```

`BindReverse()` simply subscribes to the `MouseWheel` event; `Unbind()` unsubscribes. Note that `MouseWheelManipulator` does not override `IControlAdapter.Bind()`: control adapters typically override one but not both. This is a mouse event adapter, so it overrides `BindReverse()` rather than `Bind()`.

The event handler `control_MouseWheel` changes the view's scale by manipulating the `ITransformAdapter` implementer it was given in its constructor.

`TransformAdapter` implements `ITransformAdapter` and is a special case adapter. It does not override either `Bind()` or `BindReverse()`, but is instead used by other adapters, such as `MouseWheelManipulator`. Note that an adapter does not need to have an `ITransformAdapter` given in its constructor to access it. It can obtain it from the `AdaptableControl` with adapter methods, as `D2dAnnotationAdapter` does:

```
m_transformAdapter = control.As<ITransformAdapter>();
```

For details on these adapter methods like `As()`, see [AdaptableControl Adapter Methods](#).

For an example of a painting control adapter, consider `CanvasAdapter`, which adjusts the viewing window when its bounds change. It has a `Bind()` override, rather than `BindReverse`:

```
protected override void Bind(AdaptableControl control)
{
    m_windowBounds = control.ClientRectangle;

    m_transformAdapter = control.As<ITransformAdapter>();
    if (m_transformAdapter != null)
        m_transformAdapter.TransformChanged += transformAdapter_TransformChanged;

    control.ClientSizeChanged += control_ClientSizeChanged;
}
```

This method, too, obtains an `ITransformAdapter` using `AdaptableControl.As()`. As would be expected, it subscribes to the `ClientSizeChanged` event; its `Unbind` unsubscribes.

Control Adapter Interfaces

This table describes the interfaces various control adapters use.

Control adapter interface	Description	Used by control adapters
<code>IAutoTranslateAdapter</code>	For auto-translate adapters, which track the mouse when enabled and adjust any <code>ITransformAdapter</code> .	<code>AutoTranslateAdapter</code> , <code>D2dAnnotationAdapter</code> , <code>D2dGraphEdgeEditAdapter</code> , <code>D2dGraphNodeEditAdapter</code> , <code>GraphEdgeEditAdapter</code> , <code>MouseLayoutManipulator</code> , <code>RectangleDragSelector</code>
<code>ICanvasAdapter</code>	For adapters that define a rectangular canvas, with a rectangular window for viewing it.	<code>CoordinateAxisAdapter</code> , <code>ScrollbarAdapter</code>
<code>IDragSelector</code>	For drag-selecting adapters. Uses <code>DragSelectionEventArgs</code> for drag selection event data.	<code>RectangleDragSelector</code> , <code>SelectionAdapter</code>
<code>IIItemDragAdapter</code>	For control adapters that can drag items. This interface provides a mechanism for any control adapter that can drag items to initiate drags of items in other layers at the same time. In this way, all selected items may be dragged together even if they are managed by different adapter layers.	<code>D2dAnnotationAdapter</code> , <code>D2dGraphNodeEditAdapter</code> , <code>SelectionAdapter</code>
<code>ILabelEditAdapter</code>	For adapters that can do label editing operations.	<code>LabelEditAdapter</code>
<code>IPickingAdapter2</code>	For control adapters that perform picking. NOTE: Use this instead of the obsolete <code>IPickingAdapter</code> .	<code>ContextMenuAdapter</code> , <code>D2dAnnotationAdapter</code> , <code>D2dGraphAdapter</code> , <code>KeyboardGraphNavigator</code> , <code>KeyboardIOGraphNavigator</code> , <code>HoverAdapter</code> , <code>SelectionAdapter</code> , <code>ViewingAdapter</code>

ISelectionAdapter	For control adapters that perform selection.	LabelEditAdapter, SelectionAdapter
ITransformAdapter	For adapters that define a transform for the adapted control consisting of a scale, followed by a translation. The TransformAdapters class adds extension methods on ITransformAdapter to transform coordinates and so on.	AutoTranslateAdapter, CanvasAdapter, CoordinateAxisAdapter, D2dAnnotationAdapter, D2dGridAdapter, D2dGraphAdapter, D2dGraphEdgeEditAdapter, D2dGraphNodeEditAdapter, D2dSubgraphAdapter, KeyboardGraphNavigator, KeyboardIOLGraphNavigator, LabelEditAdapter, MouseLayoutManipulator, MouseTransformManipulator, MouseWheelManipulator, RectangleDragSelector, ScrollbarAdapter, TransformAdapter, ViewingAdapter

Control Adapter Descriptions

The table lists the control adapters, what they derive from (if not `ControlAdapter`) and implement, and a description.

Some of these adapters operation on graphs in an adaptable control. For more information on graphs, see [Graphs in ATF](#). Also see the discussions of samples that use graphs, such as [Circuit Editor Programming Discussion](#).

Control adapter	Derived from, Implements	Description
AutoTranslateAdapter	IAutoTranslateAdapter, IDisposable	Auto-translate adapter that tracks the mouse when enabled and adjusts an <code>ITransformAdapter</code> .
CanvasAdapter	ICanvasAdapter, ILayoutConstraint	Canvas adapter that defines the canvas bounds and visible window size.
ContextMenuAdapter		Adapter to run a context menu on a <code>MouseUp</code> event with the right button.
CoordinateAxisAdapter		Adapter that renders horizontal and vertical coordinate axes.
D2dAnnotationAdapter	DraggingControlAdapter IPickingAdapter2, IItemDragAdapter, IDisposable	Adapter that allows displaying and editing diagram annotations (comments).
D2dGraphAdapter	IPickingAdapter2, IDisposable	Adapter to reference and render a graph diagram. Also provides hit testing with the <code>Pick()</code> method, and viewing support with the <code>Frame()</code> and <code>EnsureVisible()</code> methods.
D2dGraphEdgeEditAdapter	DraggingControlAdapter	Adapter that adds graph edge dragging capabilities with a graph adapter.
D2dGraphNodeEditAdapter	DraggingControlAdapter IItemDragAdapter	Adapter that adds graph node dragging capabilities with a graph adapter. The Shift key can be pressed to constrain dragging to be parallel to either the x-axis or y-axis.
D2dGridAdapter	ILayoutConstraint	Adapter to draw a grid on a diagram and to perform layout constraints using that grid.
D2dSubgraphAdapter	D2dGraphAdapter	Control adapter to reference and render a subgraph diagram. Also provides hit testing with the <code>Pick()</code> method, and viewing support with the <code>Frame()</code> and <code>EnsureVisible()</code> methods.
DragDropAdapter		Adapter to add drag and drop support.
DraggingControlAdapter		Abstract base class for control adapters that drag. It tracks the mouse and handles the logic of determining when the mouse has moved enough to be considered a drag.
GroupPinEditor	DraggingControlAdapter IItemDragAdapter, IDisposable	Adapter for adding floating group pin location and label editing capabilities to a subgraph control.

HoverAdapter	IDisposable	Adapter that uses an IPickingAdapter and a timer to generate events when the user hovers over items. For a discussion of how to use this adapter, see Hover Adapter .
KeyboardGraphNavigator		Allows for navigating a graph using the arrow keys. It requires that the adaptable control's context be adaptable to ISelectionContext and IGraph . Optionally, this context should be adaptable to IViewingContext . It requires that the adaptable control can be adapted to IPickingAdapter2 . Consider using KeyboardIOGraphNavigator instead, if the graph has inputs and outputs on specific sides of the nodes.
KeyboardIOGraphNavigator		Allows for navigating an "input-output" graph using the arrow keys. This kind of graph has inputs on only one side of a node and outputs on the other side. It requires that the adaptable control's context be adaptable to ISelectionContext and IGraph . Optionally, this context should be adaptable to IViewingContext . It requires that the adaptable control can be adapted to IPickingAdapter2 .
LabelEditAdapter	ILabelEditAdapter, IDisposable	Adapter for adding in-place label editing.
MouseLayoutManipulator	DraggingControlAdapter	Adapter to add a layout adorner to complex states, such as marking states as selected.
MouseTransformManipulator		Adapter that converts mouse drags into translation and scaling of the adapted control using an ITransformAdapter .
MouseWheelManipulator		Adapter that converts mouse wheel rotation into scaling the adapted control using an ITransformAdapter .
RectangleDragSelector	DraggingControlAdapter IDragSelector	Adapter that allows a user to drag-select rectangular regions on the adapted control to modify the selection. It uses DragSelectionEventArgs for drag selection event data.
ScrollbarAdapter		Adapter that adds horizontal and vertical scrollbars to the adapted control. It requires an ITransformAdapter and ICanvasAdapter .
SelectionAdapter	ISelectionAdapter, IItemDragAdapter, ISelectionPathProvider	Adapter that adds mouse click and drag selection to the adapted control. The associated context must be convertible to ISelectionContext . It uses DragSelectionEventArgs for drag selection event data.
TransformAdapter	ITransformAdapter	Adapter that defines a transform for the adapted control consisting of a scale followed by a translation. This transform affects how the adapted control is viewed. The TransformAdapters class adds extension methods on ITransformAdapter to transform coordinates and so on.
ViewingAdapter	IViewingContext	Adapter that defines a viewing context on the adapted control. It requires an ITransformAdapter .

The following control adapters are considered obsolete and are replaced by Direct2D classes.

Control adapter	Derived from, Implements	Use
AnnotationAdapter	DraggingControlAdapter IPickingAdapter, IPrintingAdapter, IItemDragAdapter, IDisposable	Use D2dAnnotationAdapter instead.
GraphAdapter	IGraphAdapter, IPickingAdapter, IPrintingAdapter, IDisposable	Use D2dGraphAdapter instead.
GraphEdgeEditAdapter	DraggingControlAdapter	Use D2dGraphEdgeEditAdapter instead.
GraphNodeEditAdapter	DraggingControlAdapter IItemDragAdapter	Use D2dGraphNodeEditAdapter instead.

GridAdapter

ILayoutConstraint

Use D2dGridAdapter instead.

Topics in this section

Links on this page to other pages

[Adaptable Controls](#), [ATF Circuit Editor Sample](#), [Authoring Tools Framework](#), [Circuit Editor Programming Discussion](#), [Graphs in ATF](#), [State Chart Editor Programming Discussion](#)

Other Adaptation Classes

These classes provide a variety of adaptations. Some of these types of adaptation, as in `CommandServiceAdapter`, differ from the adaptation of one class to another previously discussed.

Adapter class	Derives from, Implements	Description
<code>CollectionAdapter<T, U></code>	<code>ICollection<U></code>	Wrap an <code>ICollection</code> of one type to implement an <code>ICollection</code> of another type. This adapter class can be used to simulate interface covariance, where an <code>ICollection</code> of <code>Type1</code> can be made to implement an <code>ICollection</code> of <code>Type2</code> , as long as <code>Type1</code> implements or can be adapted to <code>Type2</code> . To observe changes to the underlying collection, consider passing in an <code>ObservableCollection</code> of type <code>T</code> .
<code>CommandServiceAdapter</code>	<code>Sce.Atf.Applications.ICommandService</code>	Present commands in menu and toolbar controls. This class adapts <code>Sce.Atf.Wpf.Applications.ICommandService</code> to <code>Sce.Atf.Applications.ICommandService</code> . This allows legacy code to run in a WPF based application. For more information, see Using WinForms Commands in WPF .
<code>ControlHostServiceAdapter</code>	<code>Sce.Atf.Applications.IControlHostService</code>	Provide control host services. Class to adapt <code>Sce.Atf.Wpf.Applications.IControlHostService</code> to <code>Sce.Atf.Applications.IControlHostService</code> . This allows WinForms-based applications to run in a WPF based application. For more information, see Using WinForms Commands in WPF .
<code>CustomTypeDescriptorNodeAdapter</code>	<code>DomNodeAdapter</code> <code>ICustomTypeDescriptor</code>	Node adapter to get <code>PropertyDescriptor</code> s from a <code>DomNodeType</code> and other metadata. This is very useful for metadata-driven property editing in property editors. For more details, see AdapterCreator Class .
<code>DomNodeListAdapter<T></code>	<code>IList<T></code>	Wrapper that adapts a node child list to a list of <code>T</code> items. The adapted list item type <code>T</code> adapts a <code>DomNode</code> or is a <code>DomNode</code> and should implement <code>IAdaptable</code> . Examples include <code>DomNodeAdapter</code> , <code>DomNode</code> , and <code>IAdapter</code> . For more information on this class, see the ATF Programmer's Guide: Document Object Model (DOM) .
<code>ListAdapter<T, U></code>	<code>CollectionAdapter<T, U></code> <code>IList<U></code>	Wrap an <code>IList</code> of one type to implement an <code>IList</code> of another type. This adapter class can be used to simulate interface covariance, where an <code>IList</code> of <code>Type1</code> can be made to implement an <code>IList</code> of <code>Type2</code> , as long as <code>Type1</code> implements or can be adapted to <code>Type2</code> .
<code>ListViewAdapter</code>		Adapter that adapts a <code>System.Windows.Forms.ListView</code> control to an <code>IListView</code> . For more information, see ListViewAdapter Class in File Explorer Programming Discussion . For another example, see Property Editing Programming Discussion .
<code>MainFormAdapter</code>	<code>IMainWindow</code>	Class to adapt a <code>System.Windows.Forms.Form</code> to <code>IMainWindow</code> . This class can be used as a lightweight adapter for components that need to support both <code>System.Windows.Forms.Form</code> and <code>IMainWindow</code> for backwards compatibility.
<code>MainWindowAdapter</code>	<code>IMainWindow, IInitializable</code>	Component to manage lifetime and notifications for the main window. For an example of its use, see Using WinForms Controls in WPF .

RetrieveVirtualNodeAdapter	EventArgs	Wrapper around ListView's RetrieveVirtualItem to allow client code to supply adapted user data. This is used when the underlying TreeListView's style is set to VirtualList and the underlying ListView is requesting an item that it doesn't know about. It makes a data query of the TreeListView, which then queries the TreeListViewAdapter for the data.
StatusServiceAdapter	Sce.Atf.Applications.IStatusService	Component that manages a status display. Class to adapt Sce.Atf.Wpf.Applications.IStatusService to Sce.Atf.Applications.IStatusService. This allows WinForms-based applications to be run in a WPF based application.
TreeControlAdapter		<p>Class to adapt a TreeControl to a data context that implements ITreeView. For more information, see TreeControlAdapter Class in File Explorer Programming Discussion. The following optional interfaces may also be used by a context:</p> <ol style="list-style-type: none"> IItemView is used to determine the item's label, icon, small icon, whether it's checked, and so on. IObservableContext is used to keep the TreeControl nodes in sync with the data. IValidationContext is used to defer updates until the data becomes stable. This allows more efficient updates of the TreeControl and saves IObservableContext implementations the trouble of calculating indices for ItemInserted and ItemRemoved events. These may be set to -1 in this case, and the TreeControl can still update itself correctly.
TreeListViewAdapter		Wrap a TreeListView and allow a user to supply data to the TreeListView through the ITreeListView interface and the View property of the TreeListViewAdapter. The ATF Tree List Editor Sample uses TreeListViewAdapter.

Topics in this section

Links on this page to other pages

[ATF Documentation](#), [ATF Tree List Editor Sample](#), [File Explorer Programming Discussion](#), [General Adaptation Classes](#), [Property Editing Programming Discussion](#), [Using WinForms Commands in WPF](#), [Using WinForms Controls in WPF](#)

Adapting to All Available Interfaces

This section goes into detail on how adaptation works for a specific example. This case is for DOM adapters, which are widely used in ATF, and shows how an adapter can be obtained for any interface the DOM adapters implement. It also points out what you need to do to make adaptation work in this example, which represents a common case.

State Chart Editor Example

The DOM adapter class `LockingValidator` in the [ATF State Chart Editor Sample](#) contains this line in its `OnNodeSet()` method:

Contents

- [State Chart Editor Example](#)
- [Adapting to Interfaces](#)
 - [AddAdapterCreator\(\) Method](#)
 - [InitializeExtensions\(\) Method](#)
 - [Adaptation Actions](#)

```
m_lockingContext = this.Cast<ILockingContext>(); // required ILockingContext
```

This code is attempting to get an adapter for `LockingValidator` to the interface `ILockingContext`. `LockingValidator` does not implement `ILockingContext`. How this cast works is what this section is about.

Adapting to Interfaces

You can adapt a DOM adapter to all interfaces implemented by a set of DOM adapters by doing the following:

- Define a set of DOM adapters for some type.
- Call `AddAdapterCreator()` for that type.

When you do this, you can adapt a `DomNode` defined for a type to any interface that any of the DOM adapters defined for that type implement.

Note: This technique is most useful when the type is the type of the root of the `DomNode` tree.

AddAdapterCreator() Method

Most samples have the following line, typically in their `Program.cs` file:

```
// Enable metadata driven property editing for the DOM
DomNodeType.BaseOfAllTypes.AddAdapterCreator(new AdapterCreator<CustomTypeDescriptorNodeAdapter
>());
```

This call to `DomNodeType.AddAdapterCreator()` results in `AddAdapterCreator()` being called for every type in the application's data model. As a result, it creates a list of adapters for every type defined in the data model. In general, you should call `AddAdapterCreator()` for `DomNodeType.BaseOfAllTypes` like this example, rather than call it for individual types.

`AddAdapterCreator()` is very simple:

```
public void AddAdapterCreator(IAdapterCreator creator)
{
    if (m_extensions == null)
        FreezeExtensions();

    AddCreator(creator);
}
```

The method `DomNodeType.AddCreator()` adds the adapter to a list:

```

private void AddCreator(IAdapterCreator creator)
{
    if (m_adapterCreators == null)
        m_adapterCreators = new List<IAdapterCreator>();

    m_adapterCreators.Add(creator);
}

```

`IAdapterCreator` is an interface for adapter creators and is implemented by `ExtensionAdapterCreator`, which serves as a wrapper for adapter creators, or, in this case, for adapters.

When you call `DomNodeType.AddAdapterCreator()` as in the example shown, `AddCreator()` is called for every DOM adapter defined on the type, for every type. As a result, the field `m_adapterCreators` ends up containing all DOM adapters defined for that `DomNodeType`, for every `DomNodeType` in the application. In other words, these `m_adapterCreators` (one for each `DomNodeType`) contain all the DOM adapters in the entire application.

InitializeExtensions() Method

It is also useful for applications to call `DomNode.InitializeExtensions()` after defining DOM adapters (also known as extensions). It is especially useful to do this for the root `DomNode` of the tree. A good time to do this is after opening a file and creating a `DomNode` tree from this application data.

When you call `InitializeExtensions()`, the `OnNodeSet()` method is called for each DOM adapter defined on the type of `DomNode` that `InitializeExtensions()` was invoked on.

Call `InitializeExtensions()` after calling `DomNodeType.AddAdapterCreator()`.

Adaptation Actions

What exactly happens as a result of this call to `Adapters.Cast<>()`?

```
m_lockingContext = this.Cast<ILockingContext>(); // required ILockingContext
```

`Cast<T>()` calls `Adapters.As<T>()`, where `adaptable` is the instance of `LockingValidator`:

```

public static T Cast<T>(this IAdaptable adaptable)
    where T : class
{
    T converted = As<T>(adaptable);
    if (converted == null)
        throw new AdaptationException(typeof(T).Name + " adapter required");
    return converted;
}

```

So `Adapters.Cast<T>()` calls `Adapters.As<T>()`, where `adaptable` is again `LockingValidator`:

```

public static T As<T>(this IAdaptable adaptable)
    where T : class
{
    if (adaptable == null)
        return null;

    // try a normal cast
    var converted = adaptable as T;

    // if that fails, try to get an adapter
    if (converted == null)
        converted = adaptable.GetAdapter(typeof(T)) as T;

    return converted;
}

```

Here, the first attempt at adaptation (`var converted = adaptable as T`) results in `null`, because `LockingValidator` does not implement `ILockingContext` and can't be so simply adapted. Next, `DomNodeAdapter.GetAdapter()` is called, because `LockingValidator` (in the parameter `adaptable`) is a DOM adapter:

```
public virtual object GetAdapter(Type type)
{
    return m_domNode.GetAdapter(type);
}
```

The parameter `type` is the type of `ILockingContext`. The field `m_domNode` contains the `DomNode` that is adapted to the DOM adapter. In this case, it is the root DOM node for which the DOM adapter `LockingValidator` is defined. As a result, `DomNode.GetAdapter()` is called next:

```
public object GetAdapter(Type type)
{
    // try a normal cast
    if (type.IsAssignableFrom(typeof(DomNode)))
        return this;

    // try to get an adapter
    return DomNodeType.GetAdapter(this, type);
}
```

The call `type.IsAssignableFrom(typeof(DomNode))` returns `false`, because the type of `LockingValidator` is not assignable from the type of `ILockingContext` — `LockingValidator` does not implement `ILockingContext`. So `DomNodeType.GetAdapter()` is called:

```
internal static object GetAdapter(DomNode node, Type type)
{
    IEnumerable<IAdapterCreator> adapterCreators = GetAdapterCreators(node, type);
    foreach (IAdapterCreator creator in adapterCreators)
        return creator.GetAdapter(node, type);

    return null;
}
```

Diving deeper, `DomNodeType.GetAdapterCreators()` is called to try to get an adapter from the list of adapter creators for `type`:

```

private static IEnumerable<IAdapterCreator> GetAdapterCreators(DomNode node, Type type)
{
    DomNodeType nodeType = node.Type;
    if (nodeType.m_adapterCreatorCache == null)
        nodeType.m_adapterCreatorCache = new Dictionary<Type, IEnumerable<IAdapterCreator>>();

    IEnumerable<IAdapterCreator> adapterCreators;
    if (!nodeType.m_adapterCreatorCache.TryGetValue(type, out adapterCreators))
    {
        // build an array of adapter creators that can adapt the node
        List<IAdapterCreator> creators = new List<IAdapterCreator>();
        while (nodeType != null)
        {
            if (nodeType.m_adapterCreators != null)
            {
                foreach (IAdapterCreator creator in nodeType.m_adapterCreators)
                {
                    if (creator.CanAdapt(node, type))
                        creators.Add(creator);
                }
            }
        }

        nodeType = nodeType.BaseType;
    }

    // for empty arrays, use global instance
    adapterCreators = (creators.Count > 0) ? creators.ToArray() : Enumerable<IAdapterCreator>.Instance;

    // cache the result for subsequent searches
    node.Type.m_adapterCreatorCache.Add(type, adapterCreators);
}

return adapterCreators;
}

```

This method first sets `nodeType` to the `DomNodeType` of the `DomNode` for which the adapter is to be obtained, which is the type of the root `DomNode` in this case. If the method can't find an adapter for this type, it later resets the value to the type the original `DomNode` type is derived from, if any:

```
nodeType = nodeType.BaseType;
```

On the first pass through `GetAdapterCreators()`, the `Dictionary m_adapterCreatorCache` is `null`. As a result, when this line is executed, the test condition is `true`, because the dictionary is empty:

```
if (!nodeType.m_adapterCreatorCache.TryGetValue(type, out adapterCreators))
```

So the subsequent block of code is entered and the `while` loop executed, starting with this line:

```
if (nodeType.m_adapterCreators != null)
```

In `AddAdapterCreator()` Method, it was noted that calling that method resulted in `m_adapterCreators` being populated with all the DOM adapters that are defined for that `DomNodeType` in the application. Therefore, `GetAdapterCreators()` iterates through all adapters defined for the root `DomNodeType`:

```
foreach (IAdapterCreator creator in nodeType.m_adapterCreators)
```

Recall that `IAdapterCreator` is an interface for adapter creators and is implemented by `ExtensionAdapterCreator`, which serves as a wrapper for adapters.

In the loop, it makes this test:

```
if (creator.CanAdapt(node, type))
```

The `CanAdapt()` method is in the private class `ExtensionAdapterCreator` and determines whether the adapter can be adapted to `type`, which is the type of `ILockingContext`:

```
public bool CanAdapt(object adaptee, Type type)
{
    DomNode node = adaptee as DomNode;
    return
        node != null &&
        type != null &&
        type.IsAssignableFrom(m_extensionInfo.Type);
}
```

The field `m_adapterCreators` is actually a `List<IAdapterCreator>`, which is a list of `ExtensionAdapterCreator` instances in this case, because `ExtensionAdapterCreator` implements `IAdapterCreator`. In `ExtensionAdapterCreator`, the field `m_extensionInfo` holds the actual adapter.

`IsAssignableFrom()` tests whether the type of `ILockingContext` is assignable from the type of the adapter, `m_extensionInfo.Type`. This expression returns `false` until the adapter is `EditingContext`, which implements `ILockingContext`. This condition in `DomNodeType.GetAdapterCreators()` is finally satisfied, and the adapter `EditingContext` is added to the list:

```
if (creator.CanAdapt(node, type))
    creators.Add(creator);
```

`EditingContext` is the only adapter in State Chart Editor that meets this requirement, so when the loop ends, `EditingContext` is the only adapter in the list.

From this point, control goes back up the stack:

- `DomNodeType.GetAdapterCreators` returns an adapter list containing `EditingContext`.
- `DomNodeType.GetAdapter()` returns `EditingContext`.
- `DomNode.GetAdapter()` returns `EditingContext`.
- `DomNodeAdapter.GetAdapter()` returns `EditingContext`.
- `Adapters.As<T>` returns `EditingContext`.
- `Adapters.Cast<T>` returns `EditingContext`, which is where this all started:

```
m_lockingContext = this.Cast<ILockingContext>(); // required ILockingContext
```

The field `m_lockingContext` is set to the DOM adapter `EditingContext`, which implements `ILockingContext`, so it fills the bill.

Topics in this section

Links on this page to other pages

[ATF State Chart Editor Sample](#), [Authoring Tools Framework](#)

ATF Contexts

ATF provides services for application contexts with interfaces and classes.

- [What is a Context?](#): Define contexts and discuss how ATF provides services for them.
 - [Context Registry](#): The `ContextRegistry` component tracks all context objects in an application.
 - [Context Interfaces](#): ATF's context interfaces and their usage.
 - [Context Classes](#): Classes that implement services for a context.
 - [Context Related Classes](#): Classes that use context interfaces and classes for their operation.
 - [Implementing a Context Interface](#): Implementing context interfaces.
-

What is a Context?

Context Definition

A context is the environment or interrelated conditions in which something exists or occurs. A context can also be seen as a set of circumstances or certain situations in an application.

A context can have various aspects or “logical views” of an application and its data. For example, a context may have items that can be viewed, or it may have items that can be positioned and sized, or both. Some contexts have transactions that can be undone and redone, such as editing data.

An application can have a variety of contexts. These contexts may be active at different times, typically depending on which controls are active. The current active context is usually associated with the current active window in the user interface. For instance, a text editing application could have an active context in which text can be modified while a text window is active. This application would have an altogether different context active when a dialog is open to edit the application’s preferences.

Topics

- [Context Definition](#)
- [Context Interfaces and Classes](#)
- [Contexts and Data](#)
- [DOM Adapter Context Classes](#)

Context Interfaces and Classes

ATF has interfaces and classes to provide services for an application’s data and operations in various contexts. These interfaces can apply to different types of contexts or aspects of contexts. For example, the `IViewingContext` interface applies to contexts with items that can be viewed, whereas `ILayoutContext` works with contexts containing items that can be positioned and sized. `ITransactionContext` works with contexts in which transactions can occur.

The term context is often used to refer to an instance of a context interface or class.

Interfaces and classes that work with contexts have names that end with "Context", such as `IViewingContext` and `EditingContext`.

A context interface may also have a standard ATF implementation class. It may implement more than one context interface, as `TransactionContext` does:

```
public class TransactionContext : DomNodeAdapter, ITransactionContext, IValidationContext
```

More than one context can apply to a certain environment, so a class may implement several context interfaces. A few context interfaces even derive from each other. For instance:

```
public interface ILayeringContext : IVisibilityContext, ITreView, IItemView
```

Some of the context classes also have a chain of derivation. For instance, `EditingContext` has this derivation ancestry: `EditingContext`, `HistoryContext`, and `TransactionContext`.

Contexts and Data

Data is important in some contexts, for instance, where data objects are selectable. The `IEnumerableContext` interface is for a context that can provide an enumeration of all its items. It complements `ISelectionContext` for contexts where items can be selected.

A context interface can be very general, working with different kinds of data. For instance, `ILockingContext` is for contexts where items may be locked so they can’t be modified. Such items could be text or graphical elements — anything lockable. And `ISelectionContext` supports a context where items can be selected — any kind of items.

DOM Adapter Context Classes

Some context classes, such as `TransactionContext`, are also DOM adapters. Because many contexts involve application data, the ability to adapt the data to a context’s interface is very useful.

An application can define DOM adapters that implement context interfaces for its root `DomNode` type and other types. The `DomNodes` of those types can then use these context interfaces.

For example, the [ATF Simple DOM Editor Sample](#) defines several DOM adapters in its `SchemaLoader` class:

```
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceDocument>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceContext>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<MultipleHistoryContext>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequence>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<DomNodeQueryable>());

Schema.eventType.Type.Define(new ExtensionInfo<Event>());
Schema.eventType.Type.Define(new ExtensionInfo<EventContext>());
```

This application's data model defines two main data types: `eventType` for events, and `eventSequenceType`, which is a sequence of `eventType` objects.

The type `eventSequenceType` has several DOM adapters defined. `EventSequenceContext` derives from `EditingContext`, which implements `ISelectionContext` and several other context interfaces through its ancestor classes, such as `IHistoryContext` and `ITransactionContext`. `DomNodeQueryble` implements `IQueryableContext`, `IQueryableResultContext`, and `IQueryableReplaceContext`. This means that this application's document objects can use all of these context interfaces. Note that the `EventSequenceDocument` DOM adapter is also defined for this type.

The type `eventType` has the defined DOM adapter `EventContext`, which also implements `EditingContext`, so event objects can also use all `EditingContext`'s context interfaces.

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#)

Context Registry

The context registry tracks all context interface or class objects in an application, maintaining a collection of contexts. In particular, the registry tracks the active context, which is usually determined by the currently active control. The `IContextRegistry` interface provides the framework to do all of this.

IContextRegistry Interface

`IContextRegistry` contains the following items that help an application keep track of contexts, including the active one:

- `Contexts` property: Get an enumeration of all open contexts, that is, all objects that implement context interfaces and have been added to the context registry.
- `ActiveContext` property: Get or set the active context.
- `GetActiveContext<T>()`: Get the active context as the given type.
- `GetMostRecentContext<T>()`: Get the most recently active context of the given type, which may not be the same as `ActiveContext`.
- `RemoveContext()`: Remove the given context from the registry.
- `ActiveContextChanging` and `ActiveContextChanged`: Events before and after the active context changes.
- `ContextAdded` and `ContextRemoved`: Events to indicate a context has been added to or removed from the registry.

Setting the `ActiveContext` property also adds the context to the registry; there is no method to simply add a context to the registry.

Topics

- [IContextRegistry Interface](#)
- [ContextRegistry Component](#)
- [Adding Contexts](#)
- [Removing Contexts](#)
- [Subscribing to IContextRegistry Events](#)
- [Testing For Contexts](#)

ContextRegistry Component

ATF's `ContextRegistry` component implements the `IContextRegistry` interface. Most applications need to use this component or one like it, so they can determine what operations are meaningful in any given context. You can create your own `ContextRegistry` component by implementing the `IContextRegistry` interface.

`ContextRegistry` uses `AdaptableActiveCollection`, an extension of `ActiveCollection`, as a container for contexts. `ActiveCollection` handles the collection per se. `AdaptableActiveCollection` allows adapting contexts to other types, because a given context object may implement several context interfaces.

The `ContextRegistry` is often passed as one of the parameters in the `ImportingConstructor` of components used in an application. The constructor can then save this instance for later use. The `ContextRegistry` object can also be obtained other ways, as with the `System.ComponentModel.Composition.Hosting.ExportProvider` class, discussed in [How MEF is Used in ATF](#).

Adding Contexts

As previously mentioned, `IContextRegistry` doesn't have a method to add contexts to the registry; setting the active context adds that context to the registry.

For instance, the `Editor` class of the [ATF Simple DOM Editor Sample](#) creates and saves documents that contain sequences of events. `Editor` also serves as the control host client for the `ListView` control that displays document data. When the `ListView` is activated by the user clicking on it, the control's `Activate()` method is called:

```

void IControlHostClient.Activate(Control control)
{
    EventSequenceDocument document = control.Tag as EventSequenceDocument;
    if (document != null)
    {
        m_documentRegistry.ActiveDocument = document;

        EventSequenceContext context = document.As<EventSequenceContext>();
        m_contextRegistry.ActiveContext = context;
    }
}

```

This method adapts the `EventSequenceDocument` document to a context for the event sequence, `EventSequenceContext`, and then sets the `IContextRegistry.ActiveContext` property to this context to make it the active context. If this context is not already in the context registry, it is added.

A control becoming active is a common way to set the active context

Removing Contexts

When a context is no longer needed, it should be removed from the registry. For instance, a context associated with a document may be removed from the registry once the document is closed. The `Editor` class in the [ATF Simple DOM Editor Sample](#) does precisely that when the document successfully closes:

```

bool IControlHostClient.Close(Control control)
{
    bool closed = true;

    EventSequenceDocument document = control.Tag as EventSequenceDocument;
    if (document != null)
    {
        closed = m_documentService.Close(document);
        if (closed)
            m_contextRegistry.RemoveContext(document);
    }

    return closed;
}

```

Subscribing to IContextRegistry Events

An application often needs to track context changes, so it subscribes to `IContextRegistry` events. A component's constructor or `IInitializable.Initialize()` method could subscribe to events in `IContextRegistry`.

The `EventListEditor` component of the [ATF Simple DOM Editor Sample](#) tracks contexts and also handles drag and drop and right-click context menus for a `ListView` control. In this sample, the component subscribes to several `IContextRegistry` events in its constructor:

```

public EventListEditor(
    IControlHostService controlHostService,
    ICommandService commandService,
    IContextRegistry contextRegistry)
{
    m_controlHostService = controlHostService;
    m_commandService = commandService;
    m_contextRegistry = contextRegistry;

    m_contextRegistry.ActiveContextChanged += new EventHandler(contextRegistry_ActiveContextChanged);
}
m_contextRegistry.ContextAdded += new EventHandler<ItemInsertedEventArgs<object>>(contextRegistry_ContextAdded);
}
m_contextRegistry.ContextRemoved += new EventHandler<ItemRemovedEventArgs<object>>(contextRegistry_ContextRemoved);
}
}
...
private void contextRegistry_ContextAdded(object sender, ItemInsertedEventArgs<object> e)
{
    EventSequenceContext context = Adapters.As<EventSequenceContext>(e.Item);
    if (context != null)
    {
        context.ListView.DragOver += new DragEventHandler(listView_DragOver);
        context.ListView.DragDrop += new DragEventHandler(listView_DragDrop);
        context.ListView.MouseUp += new MouseEventHandler(listView_MouseUp);

        context.ListViewAdapter.LabelEdited +=
            new EventHandler<LabelEditedEventArgs<object>>(listViewAdapter_LabelEdited);
    }
}

```

In this case, the `ContextAdded` event handler, `contextRegistry_ContextAdded`, takes the opportunity to subscribe to various events on the `ListView` control associated with the context. The `ContextRemoved` handling method unsubscribes from these events.

For more information on context handling in this sample, see [Simple DOM Editor Programming Discussion](#).

In the following example, the `DomRecorder` component initially obtains the `ContextRegistry` instance using a MEF import:

```

[Import(AllowDefault = true)]
public IContextRegistry ContextRegistry
{
    get { return m_contextRegistry; }
    set
    {
        if (m_contextRegistry != null)
            m_contextRegistry.ActiveContextChanged -= m_contextRegistry_ActiveContextChanged;

        m_contextRegistry = value;

        if (m_contextRegistry != null)
            m_contextRegistry.ActiveContextChanged += m_contextRegistry_ActiveContextChanged;
    }
}
...
private void m_contextRegistry_ActiveContextChanged(object sender, EventArgs e)
{
    ValidationContext = m_contextRegistry.ActiveContext.As<IValidationContext>();
}

```

After the active context changes, the handler `m_contextRegistry_ActiveContextChanged` sets the `ValidationContext` property to the active context, adapted to an `IValidationContext`. Instead of using the `ActiveContext` property, this event handler could also have called `GetActiveContext<IValidationContext>()`.

Testing For Contexts

It is useful to determine whether the active context supports a given context interface, so the application can determine what operations are meaningful in the current context. For example, the `RenameCommand` component tests whether it can perform a rename command by checking the available context interfaces:

```

bool ICommandClient.CanDoCommand(object commandTag)
{
    // The dialog box is modal and not dockable, so only allow it to pop up if it can be used.
    bool canDo = false;
    if (Command.Rename.Equals(commandTag))
    {
        // Note that the ITransactionContext can be null, so we don't need to check it here.
        var selectionContext = m_contextRegistry.GetActiveContext<ISelectionContext>();
        var namingContext = m_contextRegistry.GetActiveContext<INamingContext>();

        if (selectionContext != null &&
            namingContext != null)
        {
            foreach (object item in selectionContext.Selection)
            {
                if (namingContext.CanSetName(item))
                {
                    canDo = true;
                    break;
                }
            }
        }
    }
    return canDo;
}

```

This method calls `GetActiveContext()` to adapt the active context to selection and naming context interfaces. It checks that both these contexts be non-null to be able to rename objects, which is a sensible requirement.

In this example from the `StandardEditCommands` component, `GetActiveContext()` returns an `IInstancingContext` interface object, which is then adapted to an `ITransactionContext`, so that context interface is required, too:

```

public void Paste()
{
    IInstancingContext instancingContext = m_contextRegistry.GetActiveContext<IInstancingContext>();
    if (instancingContext != null &&
        instancingContext.CanInsert(Clipboard))
    {
        ITransactionContext transactionContext = instancingContext.As<ITransactionContext>();
        transactionContext.DoTransaction(
            delegate
            {
                instancingContext.Insert(Clipboard);
            }, CommandInfo.EditPaste.MenuText);

        OnPasted(EventArgs.Empty);
    }
}

```

Note that `GetActiveContext()` returns the actual context object. It is first adapted to an `IInstancingContext` and then adapted to `ITransactionContext`. The object's underlying class must therefore implement `ITransactionContext` for this to work.

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [How MEF is Used in ATF](#), [Simple DOM Editor Programming Discussion](#)

Context Interfaces

This section describes each of ATF's context interfaces and their usage. Other sections discuss [Context Classes](#) and [Context Related Classes](#).

Topics

- [Context Interface List](#)

Context Interface List

The following table lists the context interfaces and describes the kind of context where the interface can be used.

Context Interface	Description, used where...	Methods, Properties, Events	Used in
IColoringContext	Items can be colored.	GetColor(), CanSetColor(), SetColor()	CircuitEditingContext class
IEnumerableContext	Context can provide an enumeration of all its items. Complements ISelectionContext.	Items	Numerous samples, including ATF FSM Editor Sample, ATF Simple DOM Editor Sample, ATF Timeline Editor Sample
IHierarchicalInsertionContext	Context can insert new objects (e.g., via drag and drop) under a specific parent object. Normally implemented along with IIInstancingContext.	CanInsert(), Insert()	ATF Tree List Editor Sample to allow specific tree editors like ProjectLister to tell the context which node a user has dragged an object to. This context is used by ApplicationUtil's CanInsert() and Insert() methods and is preferred over IIInstancingContext if both are implemented.
IHistoryContext	Context has history, such as commands with history. It enables implementing "Undo"/"Redo" commands, tracking what has changed since the document was last saved.	CanUndo(), CanRedo(), UndoDescription(), RedoDescription(), Undo(), Redo().Dirty, Event DirtyChanged	HistoryContext class
IIInstancingContext	Context can instance objects, which requires the ability to copy, insert, and delete items. Consider implementing IHierarchicalInsertionContext, too, if this context has data that is exposed to any kind of tree control, like ProjectLister. IIInstancingContext should be used to create new instances, as in "Group"/"Ungroup" commands. In a WinForms application, the object type is probably System.Windows.Forms.IDataObject, and in a WPF application, this object can be a System.Windows.IDataObject. For more details, see Instancing In ATF .	CanCopy(), Copy(), CanInsert(), Insert(), CanDelete(), Delete()	Several samples, including ATF DOM Tree Editor Sample, ATF FSM Editor Sample, ATF Simple DOM Editor Sample

ILabelEditingContext	Context supports editing item labels. WPF only.	<code>CanEditLabel()</code> , <code>EditLabel()</code> , <code>GetLabel()</code> , <code>SetLabel()</code> . Event <code>BeginLabelEdit</code>	EditLabelCommand class
ILayeringContext	Context with layering. Layering contexts control item visibility and provide a tree view of layers. Implements IVisibilityContext , ITreeView , and IItemView .	<code>SetActiveItem()</code>	LayerLister component
ILayoutContext	Items can be positioned and resized. Contains methods for getting and setting item bounding rectangles, information on which parts of the bounds are meaningful, and which parts of bounds can be set. Has both WinForms and WPF versions, which have exactly the same methods and extension methods, with slightly different implementations.	<code>GetBounds()</code> , <code>CanSetBounds()</code> , <code>SetBounds()</code>	StandardLayoutCommands component . ATF FSM Editor Sample , ATF State Chart Editor Sample
ILockingContext	Items can be locked, so they can't be modified or deleted. Designed for locking individual objects within a document.	<code>IsLocked()</code> , <code>CanSetLocked()</code> , <code>SetLocked()</code>	StandardLockCommands . For locking documents, see ISourceControlContext and SourceControlCommands .
INamingContext	Objects can be named.	<code>GetName()</code> , <code>CanSetName()</code> , <code>SetName()</code>	ATF DOM Tree Editor Sample , ATF FSM Editor Sample , ATF State Chart Editor Sample , ATF Timeline Editor Sample
IObservableContext	Context can provide update events for changes to its contents. Allow observers of list and tree data to track changes. If an application finds that the active context implements IObservableContext , it can implement an editable view that stays in sync with the data. If applications do not find IObservableContext , they can assume the view is read-only and still consume data.	Events <code>ItemInserted</code> , <code>ItemRemoved</code> , <code>ItemChanged</code> , <code>Reloaded</code>	Numerous samples, including ATF DOM Tree Editor Sample , ATF File Explorer Sample , ATF Simple DOM Editor Sample , ATF Simple DOM No XML Editor Sample
IPropertyEditingContext	Properties can be edited by controls. Allow property editing controls, such as PropertyGrid or GridControl , to get properties and items with properties, enabling property editing controls to display objects and their properties.	<code>Items</code> , <code>PropertyDescriptor</code> s	PropertyView class
IPrototypingContext	Prototyping context, which can present a tree view of its contents and create IDataObjects from them. Implements ITreeView and IItemView .	<code>SetActiveItem()</code> , <code>GetInstance()</code>	PrototypeLister component . ATF FSM Editor Sample , ATF State Chart Editor Sample
IQueryableContext	Classes in which objects may be searched.	<code>Query()</code>	DomNodeQueryable class
IQueryableReplaceContext	Classes in which containing objects may be replaced.	<code>Replace()</code>	DomNodeQueryable class
IQueryableResultContext	Accessing the results of a query and being notified when those results change.	Results. Event <code>ResultsChanged</code>	DomNodeQueryable class
ISearchableContext	Context provides search and replace capabilities through a client-defined UI.	<code>SearchUI</code> , <code>ReplaceUI</code> , <code>ResultsUI</code>	ATF Simple DOM Editor Sample

ISelectionContext	Items can be selected. This interface distills what it means to have a selection in a context. It provides filtering, so you can get the last selected object of a certain type. It has an efficient method for determining if a selection contains an object.	GetSelection<T>(), GetLastSelected<T>(), SelectionContains(). Selection, LastSelected, SelectionCount. Events SelectionChanging, SelectionChanged	SelectionContext class. Several samples, including ATF File Explorer Sample , ATF Property Editing Sample , ATF Simple DOM Editor Sample , ATF Tree List Editor Sample
ISourceControlContext	Source control context. Users of this interface can use the IResource to examine URLs and adapt the IResource to IDocument to track a document's dirty flag.	Resources	SourceControlCommands
ISubSelectionContext	Context supports selection and can optionally specify a "sub-selection" context, i.e., the selection context of sub-elements, related to the selection in this context. This interface serves to abstractly allow consumers of selection contexts to hook into events from sub-selections. Implements ISelectionContext .	SubSelectionContext	ATF Simple DOM Editor Sample
ITemplatingContext	Prototyping context, which can present a tree view of its contents and create IDataObjects from them. Implements ITreeView and IItemView .	SetActiveItem(), GetInstances(), CanReference(), CreateReference()	TemplatingContext class. TemplateLister component
ITransactionContext	Data is changed in transactions, which can be rolled back, simplifying editor design. Transactions can be cancelled any time before calling End() . Different modules can contribute to an operation without having to know about each other, and if an exception occurs in one module, the whole transaction can be rolled back.	Begin(), Cancel(), End(). InTransaction	TransactionContext class. Several samples, including ATF DOM Tree Editor Sample , ATF FSM Editor Sample , ATF Timeline Editor Sample
IValidationContext	Context provides events so listeners can perform validation. It's useful for marking when the user begins and ends logical commands. Validation events allow listeners to update themselves and check constraints more efficiently.	Events Beginning, Cancelled, Ending, Ended	ATF Tree List Editor Sample
IViewingContext	Items can be viewed. Its methods check whether objects are or can be made visible, or are framed or can be framed in the current view.	CanFrame(), Frame(), CanEnsureVisible(), EnsureVisible()	StandardViewCommands component. Several samples, including ATF FSM Editor Sample , ATF State Chart Editor Sample
IViewingContext	Items can be viewed. WPF version, derived from Sce.Atf.Applications.IViewingContext	Adaptable, ViewingAdapter, PickingAdapters, LayoutConstraints	
IVisibilityContext	Items can be shown and hidden.	IsVisible(), CanSetVisible(), SetVisible()	StandardShowCommands component

Topics in this section

Links on this page to other topics

[ATF DOM Tree Editor Sample](#), [ATF File Explorer Sample](#), [ATF FSM Editor Sample](#), [ATF Property Editing Sample](#), [ATF Simple DOM Editor Sample](#), [ATF Simple DOM No XML Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [ATF Tree List Editor Sample](#), [Context Classes](#), [Context Related Classes](#), [Instancing In ATF](#)

- It also provides filtering: you can get the last selected object of a certain type. It has events detecting before and after a selection changes. It has an efficient method for determining if a selection contains an object.<!-- /* Style Definitions */ table.MsoNormalTable

Unknown macro: {mso-style-name}

-->

Context Classes

Several classes implement services for a context. Most, but not all, implement one or more of the context interfaces described in [Context Interfaces](#). Another section discusses [Context Related Classes](#), which use context interfaces in their operation.

Note that many of these classes are DOM adapters. The samples often define these adapters for their data model's types.

Because `Sce.Atf.Dom.EditingContext` is widely used in the samples, it merits special mention.

Topics

- [Context Classes](#)
- [Sce.Atf.Dom.EditingContext Class](#)

Context Classes

This table describes classes that provide services for a context.

Context Class	Description	DOM Adapter?	Context Interfaces Implemented	Used in
BitmapContext	Rendering context for OpenGL for creating bitmaps.	No		<code>ThumbnailGenerator</code> class
CircuitEditingContext	Class that defines a circuit editing context. Each context represents a circuit, with a history, selection, and editing capabilities. There may be multiple contexts within a single circuit document, because each sub-circuit has its own editing context. The class has a default implementation of <code>IInstancingContext</code> if no other <code>IInstancingContext</code> adapters are associated with the <code>DomNode</code> . Derives from <code>EditingContext</code> .	Yes	<code>IEnumerableContext</code> , <code>INamingContext</code> , <code>IInstancingContext</code> , <code>IObservableContext</code> , <code>IColoringContext</code>	<code>CircuitControlRegistry</code> , <code>GroupingCommands</code> components
EditingContext	A general editing context, which is a history context with a selection, providing a basic, self-contained editing context. There may be multiple <code>EditingContexts</code> in a document. Derives from <code>HistoryContext</code> .	Yes	<code>ISelectionContext</code>	Various samples, including ATF DOM Tree Editor Sample , ATF Simple DOM Editor Sample , ATF State Chart Editor Sample , ATF Timeline Editor Sample . For a further discussion, see Sce.Atf.Dom.EditingContext Class .
GlobalHistoryContext	Adapter that implements a single global history on a DOM tree containing multiple local <code>HistoryContexts</code> . The adapter tracks all other <code>HistoryContexts</code> in the subtree rooted at a <code>DomNode</code> and passes their transactions to the global <code>HistoryContext</code> , which must be on the same <code>DomNode</code> as this adapter.	Yes		<code>EditingContext</code> class
HistoryContext	Adapter that adds history to a <code>TransactionContext</code> . If the adapter can adapt the <code>DomNode</code> to an <code>ISelectionContext</code> , the history includes selection changes. Derives from <code>TransactionContext</code> .	Yes	<code>IHistoryContext</code>	<code>EditingContext</code> , <code>GlobalHistoryContext</code> classes

LayeringContext	Context for LayerLister component (or other component implementing <code>ILayeringContext</code>). This context has its own independent selection, but uses the main context's <code>HistoryContext</code> for undo/redo operations. <code>IInstancingContext</code> and <code>IHierarchicalInsertionContext</code> implementations control drag/drop and copy/paste operations within the LayerLister (internal), pastes/drops to the LayerLister and drag/copies from the LayerLister (external). The <code>IObservableContext</code> implementation notifies the LayerLister's TreeControl when a change occurs that requires an update of one or more tree nodes. Derives from <code>SelectionContext</code> . For more information on instancing contexts, see IInstancingContext and IHierarchicalInsertionContext Interfaces .	Yes	<code>ILayeringContext</code> , <code>IInstancingContext</code> , <code>IHierarchicalInsertionContext</code> , <code>IObservableContext</code> , <code>INamingContext</code>	LayeringCommands component. ATF Circuit Editor Sample
MultipleHistoryContext	Adapter that observes a <code>DOMNode</code> subtree and synchronizes multiple history contexts by setting the <code>IHistoryContext.Dirty</code> property to the same value for every context. The root DOM node must be adaptable to <code>IDocument</code> .	Yes		Used as a DOM adapter in several samples, including ATF Circuit Editor Sample , ATF FSM Editor Sample , ATF State Chart Editor Sample
PropertyEditingContext	Context in which properties can be edited, holding the selected items whose properties are to be edited. The context provides a way to get descriptors for properties being edited. The context performs property edits as logical commands for the undo/redo system.	No	<code>IPropertyEditingContext</code>	<code>EmbeddedCollectionEditor</code> , <code>GridControl</code> , <code>PropertyGrid</code> classes
PropertyEditorControl-Context	Context for embedded property editing controls, which provides a safe, convenient interface for getting and setting values without worrying about the intricacies of property descriptors and transaction contexts. Is only constructed by <code>PropertyView</code> .	No	<code>ITypeDescriptorContext</code>	PropertyEditingControl class. Also used by numerous value editors, such as <code>ArrayEditingControl</code> , <code>BoundedFloatEditor</code> , <code>UniformArrayEditor</code> . ATF Property Editing Sample
PrototypingContext	Editing context for prototypes in the circuit document. This context is bound to the <code>PrototypeLister</code> component when a circuit document becomes the active context. This context implements instancing differently than the <code>CircuitContext</code> . It can insert modules and connections, converting them into prototypes. It can only copy and delete prototypes. Derives from <code>SelectionContext</code> .	Yes	<code>IPrototypingContext</code> , <code>IInstancingContext</code> , <code>IObservableContext</code> , <code>INamingContext</code>	Used as DOM adapter in ATF Circuit Editor Sample , ATF FSM Editor Sample , ATF State Chart Editor Sample
SelectionContext	Adapts a DOM node to an <code>ISelectionContext</code> , using an <code>AdaptableSelection</code> object.	Yes	<code>ISelectionContext</code>	<code>LayeringContext</code> , <code>PrototypingContext</code> , <code>TemplatingContext</code> classes

SelectionPropertyEditingContext	Class that supports property editing on any <code>ISelectionContext</code> . It implements <code>IObservableContext</code> by raising the <code>Reloaded</code> event when the selection changes. It only raises the <code>Reloaded</code> event (although it tries to do that efficiently using an <code>IValidationContext</code> interface) if the selection context can be converted to an <code>IValidationContext</code> interface.	No	<code>IPropertyEditingContext</code> , <code>IObservableContext</code>	<code>GridPropertyEditor</code> , <code>PropertyEditingCommands</code> , <code>PropertyEditor</code> components. Numerous samples, such as ATF DOM Tree Editor Sample , ATF FSM Editor Sample , ATF Timeline Editor Sample .
TemplatingContext	Editing context for templates library. This context (or one implementing <code>ITemplatingContext</code>) is bound to the <code>TemplateLister</code> component when a circuit document becomes the active context. This context has its own independent selection. Derives from <code>SelectionContext</code> .	Yes	<code>ITemplatingContext</code> , <code>IInstancingContext</code> , <code>IObservableContext</code> , <code>INamingContext</code>	<code>TemplatingCommands</code> component. ATF Circuit Editor Sample
TransactionContext	Adapts a DOM node to an <code>ITransactionContext</code> .	Yes	<code>ITransactionContext</code> , <code>IValidationContext</code>	<code>HistoryContext</code> class
TypeDescriptorContext	Lightweight implementation of <code>ITypeDescriptorContext</code> for convenient interaction with the <code>System.ComponentModel</code> framework.	No	<code>ITypeDescriptorContext</code>	<code>PropertyEditingControl</code> , <code>PropertyGridView</code> classes. ATF Property Editing Sample
ViewingContext	Provides viewing functions for a <code>Circuit</code> object. It holds a reference to the <code>AdaptableControl</code> for viewing the <code>Circuit</code> object. It updates the control's canvas bounds during validation. Derives from <code>Validator</code> .	Yes	<code>IViewingContext</code> , <code>ILayoutContext</code>	<code>CircuitControlRegistry</code> , <code>GroupingCommands</code> components. ATF Circuit Editor Sample

Sce.Atf.Dom.EditingContext Class

This `EditingContext` is a generic editing context for applications and is a history context with a selection, providing a basic self-contained editing context. There may be multiple `EditingContexts` in an application. As a result of its utility and generality, `EditingContext` is used in several samples.

`EditingContext` derives from `HistoryContext`, which derives from `TransactionContext`.

`TransactionContext` implements both `ITransactionContext` and `IValidationContext`. `ITransactionContext` brackets changes to data between `Begin()`/`End()` transaction calls, which allow transactions to be monitored and cancelled, if necessary. As a complement, `IValidationContext` provides `Beginning`, `Cancelled`, `Ending`, and `Ended` events to allow transactions to be validated. `TransactionContext` provides transactions that can be undone.

`HistoryContext` adds history to a `TransactionContext`, providing a history context for undo/redo operations. `HistoryContext` implements `IHistoryContext`, whose methods test if operations can be undone or redone, and also perform that operation. If the implementing class can adapt to an `ISelectionContext`, the history includes selection changes.

`EditingContext` also implements `ISelectionContext` that provides the events, properties, and methods to support selection of some kind of item in the context.

For more information on `IHistoryContext`, `ITransactionContext`, and `ISelectionContext`, see [Context Interfaces](#).

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Property Editing Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [Context Interfaces](#), [Context Related](#)

Context Related Classes

Some classes use context interfaces or classes for their operation, so are good examples of context usage. The classes discussed here fall in a few general categories. These classes are nearly all MEF components.

Topics

- Lister Classes
- Command Handlers
- Property Editing Classes
- Miscellaneous

Lister Classes

These classes list various items in a tree type control.

Class	Context Interfaces or Classes Used	Description
LayerLister	ILayeringContext	Editor that presents an ILayeringContext using a TreeControl.
ProjectLister	IHierarchicalInsertionContext	Editor providing a hierarchical tree control, listing the contents of a loaded document.
PrototypeLister	IPrototypingContext	Editor that presents an IPromotypingContext using a TreeControl.
TemplateLister	ITemplatingContext	Editor that presents an ITemplatingContext using a TreeControl.

Command Handlers

These classes provide commands for applications.

Class	Context Interfaces or Classes Used	Description
GroupingCommands	ViewingContext	Component that defines circuit-specific commands for Group and Ungroup commands. Grouping takes modules and the connections between them and turns them into a single element that is equivalent.
LayeringCommands	LayeringContext	Component to add an "Add Layer" command to an application.
SourceControlCommands	ISourceControlContext	Component that implements source control commands.
StandardLayoutCommands	ILayoutContext	Component that provides standard layout commands, like align left, or make widths equal. The component tries to work with the active context, and requires both the ISelectionContext and ILayoutContext interfaces to be implemented on the context. Further, it examines the selected items to see if the layout context can set their x, y, width, and height. Any commands that can be done on the selection are enabled in the GUI. All commands are also available programmatically through public methods.
StandardLockCommands	ILockingContext	Implements standard Lock and Unlock commands on contexts implementing the ILockContext interface.
StandardShowCommands	IVisibilityContext	Class that implements the standard Show commands: Show Selected, Hide Selected, Show Last Hidden, Show All, Isolate.
StandardViewCommands	IViewingContext	Class that implements the standard viewing commands on contexts implementing the IVViewingContext interface.
TemplatingCommands	TemplatingContext	Component to add an "Add Template Folder" command to an application.

Property Editing Classes

These classes provide property editing functionality for applications.

Class	Context Interfaces or Classes Used	Description
GridControl	PropertyEditingContext	Wrapper control for the spreadsheet-style <code>GridView</code> control, combining it with a toolbar. Use this as a replacement for the .NET <code>System.Windows.Forms.DataGridView</code> .
GridPropertyEditor	SelectionPropertyEditingContext	Component to edit DOM object values and attributes using <code>GridControl</code> .
PropertyEditingCommands	SelectionPropertyEditingContext	Component to provide property editing commands that can be used inside <code>PropertyGrid</code> -like controls. It defines context-menu commands to reset the current property and all properties.
PropertyEditingControl	PropertyEditorControlContext, TypeDescriptorContext	Universal property editing control that can be embedded in complex property editing controls. It uses <code>TypeConverters</code> and <code>UITypeEditors</code> to provide a GUI for every kind of .NET property.
PropertyEditor	SelectionPropertyEditingContext	Component to edit DOM object values and attributes using <code>PropertyGrid</code> .
PropertyGrid	PropertyEditingContext	Wrapper control for a two-column <code>PropertyGridView</code> combined with a toolbar. Use this as a replacement for the .NET <code>System.Windows.Forms.PropertyGrid</code> .
PropertyGridView	TypeDescriptorContext	Control for displaying properties in a two-column grid, with property names on the left and property values on the right.
PropertyView	IPropertyEditingContext	Base class for complex property editing controls, providing formats, fonts, data binding, persistent settings, and category/property information.
Various value editors	PropertyEditorControlContext	A value editor calls the <code>IPropertyEditor</code> interface's <code>GetEditingControl()</code> method, which takes a <code>PropertyEditorControlContext</code> as its parameter. <code>GetEditingControl()</code> uses the context to construct the associated control. For more information, see Value Editors .

Miscellaneous

A few other classes that use contexts are mentioned here.

Class	Context Interfaces or Classes Used	Description
CircuitControlRegistry	ViewingContext, CircuitEditingContext	Component to provide a convenient service to register/unregister circuit controls created for circuit groups, to synchronize UI updates for circuit controls due to group renaming, to insert/delete circuit elements, and to close documents/controls.
DomNodeQueryable	IQueryableContext, IQueryableReplaceContext, IQueryableResultContext	DOM adapter enabling <code>DomNodes</code> to be searched, to supply the search results, and to have those results replaced with other data.

Topics in this section

Links on this page to other topics

[Value Editors](#)

Implementing a Context Interface

You implement a context interface by creating a class that implements all the context (and other) interfaces you want to use. The objects created from this class can later be obtained from the `ContextRegistry`, which tracks the existing context objects. These objects can then be cast to objects implementing the various interfaces.

Various context interfaces have various points of view about the context. Hence there is a great variety in these implementations.

Topics

- [Implementing Needed Interfaces](#)
- [Related Context Interfaces](#)
- [Variations on Implementing a Context Interface](#)
- [Simple DOM Editor Examples](#)

Implementing Needed Interfaces

Make sure you implement all the appropriate context (and other) interfaces required for a class. For example, the `StandardEditCommands` component has a `Paste()` method that gets the active context from the `ContextRegistry`, adapted to an `IInstancingContext` interface. It then adapts it to an `ITransactionContext`. The underlying context class must also implement `ITransactionContext` for this to work.

```
public void Paste()
{
    IInstancingContext instancingContext = m_contextRegistry.GetActiveContext<IInstancingContext>();
    if (instancingContext != null && instancingContext.CanInsert(Clipboard))
    {
        ITransactionContext transactionContext = instancingContext.As<ITransactionContext>();
        transactionContext.DoTransaction(
            delegate
            {
                instancingContext.Insert(Clipboard);
            }, CommandInfo.EditPaste.MenuText);

        OnPastedEventArgs.Empty);
    }
}
```

To meet this requirement, a class could derive from `EditingContext` and also implement `IInstancingContext`. `EditingContext` derives from `HistoryContext`, which derives from `TransactionContext`, which implements `ITransactionContext`.

For example, the [ATF Simple DOM Editor Sample](#) uses the `StandardEditCommands` component. It implements several context handling classes, such as the `EventContext` class, which does what's needed:

```
public class EventContext : EditingContext,
    IListview,
    IItemView,
    IObservableContext,
    IInstancingContext,
    IEnumerableContext
```

Related Context Interfaces

Some context interfaces go together. For instance, if you implement a class derived from `SelectionContext`, you probably want to implement `IInstancingContext` as well. If you can select objects, you can typically edit them, too, and `IInstancingContext` provides editing methods. For more details, see [Instancing In ATF](#).

For example, the `LayeringContext` class derives from `SelectionContext` and implements `IInstancingContext` as well as other context interfaces:

```
public abstract class LayeringContext : SelectionContext,
    IInstancingContext,
    IHierarchicalInsertionContext,
    ILayeringContext,
    IObservableContext,
    INamingContext
```

Variations on Implementing a Context Interface

An application determines its contexts, and a context interface's implementation can be wildly different from application to application.

For example, here is a partial implementation of `IInstancingContext` for [ATF Simple DOM Editor Sample](#):

```
public bool CanCopy()
{
    return Selection.Count > 0;
}
...
public object Copy()
{
    IEnumerable<DomNode> resources = Selection.AsIEnumerable<DomNode>();
    List<object> copies = new List<object>(DomNode.Copy(resources));
    return new DataObject(copies.ToArray());
}
```

Here is the corresponding implementation in [ATF Timeline Editor Sample](#):

```
public bool CanCopy()
{
    return Selection.Count > 0;
}
...
public object Copy()
{
    object[] selection = Selection.GetSnapshot();

    // Cut + Paste needs to know the original tracks of the cut objects.
    m_copyObjToTrack = new Dictionary<ITimelineObject, ITrack>(selection.Length);
    foreach (ITimelineObject source in selection.AsIEnumerable<ITimelineObject>())
    {
        ITrack sourceTrack;
        IGroup sourceGroup;
        GetTrackAndGroup(source, out sourceTrack, out sourceGroup);
        if (sourceTrack != null)
            m_copyObjToTrack[source] = sourceTrack;
    }

    return new DataObject(selection);
}
```

These examples are taken from their samples' versions of an editing context class, which both derive from `EditingContext` and implement `IInstancingContext`.

The `CanCopy()` method is identical in both samples. Both use the `EditingContext.Selection` property and check that its `Count` property is positive. The `Copy()` methods, on the other hand, look totally different. `Copy()` depends on the nature of selectable objects — the data model — which is quite different in the two samples.

Simple DOM Editor Examples

The [ATF Simple DOM Editor Sample](#) implements several context interfaces. For programming details, see [Working With Contexts](#).

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [ATF Timeline Editor Sample](#), [Instancing In ATF](#), [Simple DOM Editor Programming Discussion](#)

Instancing In ATF

Instancing is working with object instances that can be copied, inserted, or deleted. The `IInstancingContext` provides editing methods for instances, which can be used in any kind of editing application.

- **What is Instancing?**: Generally describes instancing, which also requires instances to be selectable.
 - **IInstancingContext and IHierarchicalInsertionContext Interfaces**: Discuss the `IInstancingContext` interface and its editing methods.
 - **StandardEditCommands and Instancing**: Show how the `StandardEditCommands` component works with instancing.
 - **Drag and Drop and Instancing**: Explain how drag and drop is used with instancing.
 - **Implementing Instancing**: Point to examples of implementing instancing in the [ATF Simple DOM Editor Sample](#).
-

What is Instancing?

The Instancing Framework works with object instances that can be edited, that is, copied, inserted, or deleted. For example, it enables inserting objects, such as typing text onto a page, pasting from a clipboard, or dropping a circuit element onto a canvas. This framework allows making copies or clones of objects and deleting instances.

In ATF, such instancing is done through the `IInstancingContext` interface, which provides methods to determine if objects can be edited and to perform editing operations. Most ATF editing applications should implement `IInstancingContext`.

Because selected objects are the ones edited with `IInstancingContext`, an editing application needs to have a selection mechanism. It can do this by creating a context class that derives from `EditingContext` or `SelectionContext`, or implements `ISelectionContext`. For more general information on contexts, see [ATF Contexts](#).

Topics in this section

Links on this page to other topics

[ATF Contexts](#)

IInstancingContext and IHierarchicalInsertionContext Interfaces

Instancing is implemented in two interfaces: primarily in `IInstancingContext` and also with `IHierarchicalInsertionContext`.

Topics

- [IInstancingContext Interface](#)
- [IHierarchicalInsertionContext Interface](#)

IInstancingContext Interface

The `IInstancingContext` interface provides all the editing commands customarily available under application Edit menus. It is primarily used for contexts without any kind of hierarchy to the objects. It contains these methods:

- `CanCopy()`: Return whether the context can copy the selection.
- `Copy()`: Copy the selection, returning a data object representing the copied items.
- `CanInsert()`: Return whether the context can insert the data object.
- `Insert()`: Insert the data object into the context.
- `CanDelete()`: Return whether the context can delete the selection.
- `Delete()`: Delete the selection.

Note that these methods all operate on the current selection, so the current context must allow selection. As previously mentioned in [What is Instancing?](#), you need to implement some mechanism for selection, as well as instancing.

These methods are very general, so they apply to editing objects of any type. `IInstancingContext` could apply to graphic objects on a canvas as for [ATF FSM Editor Sample](#) and [ATF State Chart Editor Sample](#). It can also apply to objects dragged from a palette, as in the [ATF Simple DOM Editor Sample](#). It can apply to composite objects, such as prototypes, as in the [ATF FSM Editor Sample](#).

This interface is used in various editing mechanisms, too. Instancing supports both editing with menu items from either the Edit menu or context menus, as well as drag and drop editing.

IHierarchicalInsertionContext Interface

The `IHierarchicalInsertionContext` interface is used for contexts that can insert new objects under a specific parent object, by drag and drop for example. Its methods are a subset of `IInstancingContext`:

- `CanInsert()`: Can the child object be inserted under the given parent.
- `Insert()`: Insert the child object under the given parent.

`IHierarchicalInsertionContext` works with objects of any type that can be selected, just as for `IInstancingContext`.

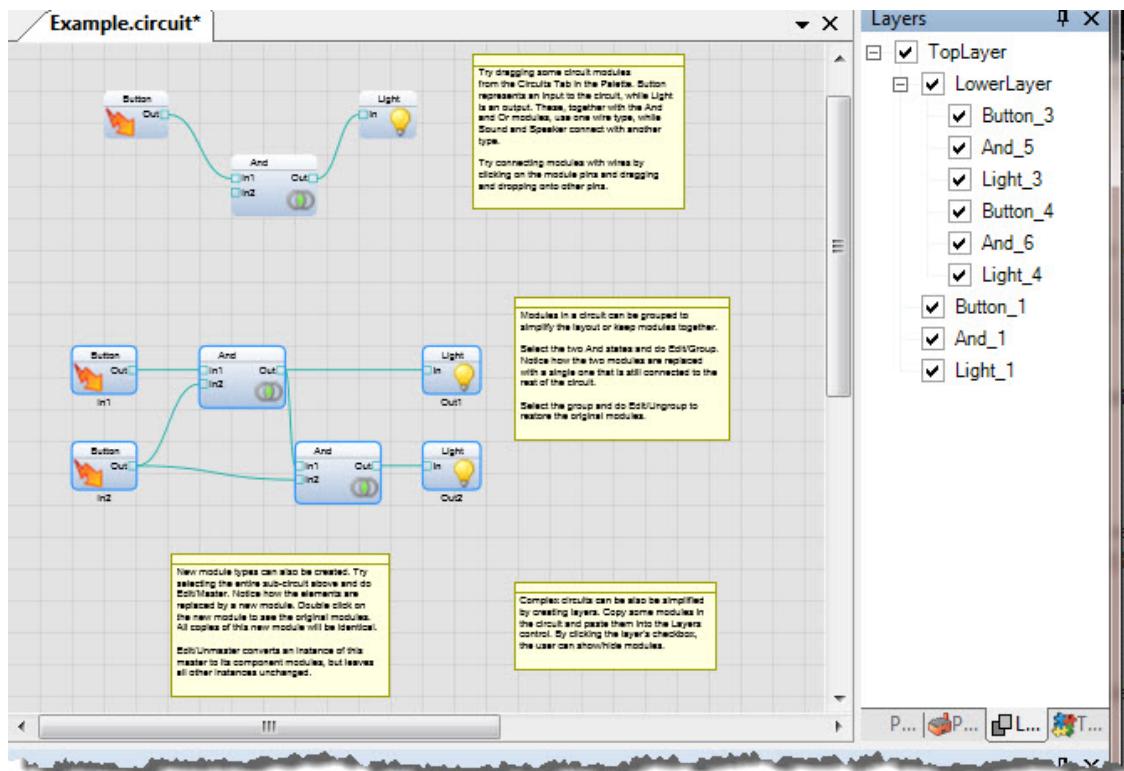
If implementing instancing in a context with hierarchical items, it can be more appropriate to use the insertion methods of `IHierarchicalInsertionContext` rather than `IInstancingContext`.

For instance, the `ApplicationUtil` class provides utilities for inserting objects in contexts. This class is used by several hierarchical editor components:

- `ListViewEditor`: Base class for list editors.
- `TreeControlEditor`: Base class for tree editors.
- `TreeListViewEditor`: Tree editor with right click context menu editing. It is used in the [ATF Tree List Editor Sample](#).

These components use `ApplicationUtil`'s `CanInsert()` and `Insert()` methods, which have parameters for context, child, and parent objects. If the given context implements `IHierarchicalInsertionContext`, the `IHierarchicalInsertionContext` interface's `CanInsert()` and `Insert()` methods are used to insert the child under the parent. If only `IInstancingContext` is implemented in the context, then its methods are used.

In addition, the `LayeringContext` class implements `IHierarchicalInsertionContext` for contexts with a hierarchy of layer objects. The [ATF Circuit Editor Sample](#) uses this context for its Layers window, which allows you to create layers of objects. You can control a layer and its individual object's visibility with check boxes in this window. You can also drag layers under other layers, so there is a hierarchy of layers, as shown in this illustration:



The visibility of lower layers is controlled by its parent layers' visibility.

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Tree List Editor Sample](#), [What is Instancing?](#)

StandardEditCommands and Instancing

The Instancing Framework's component is `StandardEditCommands` (in the `Sce.Atf.Applications` namespace), which provides editing commands. `StandardEditCommands` implements the standard Edit menu's Cut, Copy, Paste, and Delete commands.

`StandardEditCommands` also provides edit commands on a context menu. For details on how it does this, see [Using Context Menus](#).

To use `StandardEditCommands` in an application, simply include it in the MEF TypeCatalog. Create a class that implements `IInstancingContext`. When this class's context is active, `StandardEditCommands` automatically calls the `IInstancingContext` editing methods in response to menu item selections.

Topics

- Executing Commands
- `ICommandClient.CanDoCommand()` Implementation
- `ICommandClient.DoCommand()` Implementation
 - Copy Operation
 - Delete Operation

Executing Commands

`StandardEditCommands` is a command client and uses the `ICommandClient` interface to execute commands in the `IInstancingContext` interface. For information on `ICommandClient`, see [Commands in ATF](#) and [Creating Command Clients](#) in particular.

`ICommandClient.CanDoCommand()` Implementation

Observing the `ICommandClient` implementation illustrates the process. First, the "Cut" case from `CanDoCommand()`:

```
bool ICommandClient.CanDoCommand(object commandTag)
{
    bool canDo = false;

    IInstancingContext instancingContext = m_contextRegistry.GetActiveContext<IInstancingContext>();
    switch ((StandardCommand)commandTag)
    {
        case StandardCommand.EditCut:
            canDo =
                instancingContext != null &&
                instancingContext.CanCopy() &&
                instancingContext.CanDelete();
            break;
        ...
    }
}
```

`CanDoCommand()` uses the `ContextRegistry` to get the current context to determine whether that context supports editing or not. The current context must implement `IInstancingContext` for edit commands to be executed. For the "Cut" command case shown, when `instancingContext` is non-null, `instancingContext` is used to invoke its `IInstancingContext.CanCopy()` and `IInstancingContext.CanDelete()` methods. Both these methods must return true to be able to perform a cut operation. The other editing commands in `ICommandClient.CanDoCommand()` are similar to "Cut".

`ICommandClient.DoCommand()` Implementation

Next, look at the "Cut" case in `ICommandClient.DoCommand()`:

```
void ICommandClient.DoCommand(object commandTag)
{
    switch ((StandardCommand)commandTag)
    {
        case StandardCommand.EditCut:
            Cut();
            break;
        ...
    }
}
```

The StandardEditCommands.Cut() method does a copy and delete:

```
public void Cut()
{
    Copy();
    Delete("Cut".Localize("Cut the selection"));
}
```

Next, examine these copy and delete operations.

Copy Operation

StandardEditCommands.Copy() calls IInstancingContext.CanCopy() and IInstancingContext.Copy():

```
public void Copy()
{
    IInstancingContext instancingContext = m_contextRegistry.GetActiveContext<IInstancingContext>();
    if (instancingContext != null && instancingContext.CanCopy())
    {
        object rawObject = instancingContext.Copy();
        IDataObject dataObject = rawObject as IDataObject ?? new DataObject(rawObject);

        OnCopyingEventArgs.Empty);

        Clipboard = dataObject;

        OnCopiedEventArgs.Empty);
    }
}
```

StandardEditCommands.Copy() gets the IInstancingContext instance as before, and after verifying it's non-null, invokes IInstancingContext.CanCopy() and IInstancingContext.Copy(). It casts the returned data as a System.Windows.Forms.IDataObject object, which allows copying the data to the Windows clipboard. StandardEditCommands.Copy() also raises the StandardEditCommands.Copying and StandardEditCommands.Copied events at the appropriate times.

Delete Operation

The StandardEditCommands.Delete() method is similar, calling IInstancingContext.CanDelete() and IInstancingContext.Delete(). However, it has one key difference: it does the deletion in a transaction using ITransactionContext:

```
private void Delete(string commandName)
{
    OnDeletingEventArgs.Empty);

    IInstancingContext instancingContext = m_contextRegistry.GetActiveContext<IInstancingContext>();
    if (instancingContext != null && instancingContext.CanDelete())
    {
        ITransactionContext transactionContext = instancingContext.As<ITransactionContext>();
        transactionContext.DoTransaction(
            delegate
            {
                instancingContext.Delete();

                ISelectionContext selectionContext = instancingContext.As<ISelectionContext>();
                if (selectionContext != null)
                    selectionContext.Clear();
            }, commandName);
    }

    OnDeletedEventArgs.Empty);
}
```

In addition to adapting the current context to `IInstancingContext`, it also adapts it to `ITransactionContext`. This means that the context has to implement `ITransactionContext` or the operation fails. The `IInstancingContext.Delete()` call occurs inside the transaction.

The current context is also cast as a `ISelectionContext`, so it must implement `ISelectionContext` as well. The `ISelectionContext.Clear()` call also occurs inside the transaction. `StandardEditCommands.Delete()` also raises the `StandardEditCommands.Deleting` and `StandardEditCommands.Deleted` events outside the transaction, before and after the deletion.

The other editing cases in `ICommandClient.DoCommand()` are similar to the "Cut" case in their use of `IInstancingContext` methods. `StandardEditCommands.Paste()` also employs a transaction.

Topics in this section

Links on this page to other topics

[Commands in ATF](#), [Creating Command Clients](#), [Using Context Menus](#)

Drag and Drop and Instancing

You also use instancing with drag and drop editing, as well as with editing using menu items. Drag events are in the context of a control, so you define drag event handlers for the control.

Handling Dragging Events

Create handlers for the `DragOver` and `DragDrop` events on the control in which items can be dropped. `DragOver` occurs when the cursor is dragged over the control. `DragDrop` occurs when the mouse is released over the control after dragging.

Topics

- Handling Dragging Events
- DragOver Event
- DragDrop Event

This handling is well illustrated in the [ATF Simple DOM Editor Sample](#), from which the subsequent code examples are taken. This editor creates lists of sequences. Sequences can have resources added to them by dragging a resource icon from a palette onto the Resources window's `ListView` control. The `ResourceListEditor` component creates this `ListView` control. This component also defines handlers on the `ListView` for the dragging events in its `IInitializable.Initialize()` method. For other programming details of instance handling in that sample, see [IInstancingContext Interface](#).

DragOver Event

The drag over event indicates whether an insertion is possible:

```
private void resourcesListView_DragOver(object sender, DragEventArgs e)
{
    if (m_event != null)
    {
        EventContext context = m_event.Cast<EventContext>();
        e.Effect = DragDropEffects.None;
        if (context.CanInsert(e.Data))
        {
            e.Effect = DragDropEffects.Move;
            ((Control)sender).Focus(); // Focus the list view; this will cause its context to
become active
        }
    }
}
```

The event argument is an instance of the `EventContext` class, which implements `IInstancingContext`. If `IInstancingContext.CanInsert()` indicates that insertion is possible, the cursor changes to show where the resource can be dropped. The `ListView` control is also activated.

DragDrop Event

The drag over event performs the resource insertion:

```
private void resourcesListView_DragDrop(object sender, DragEventArgs e)
{
    if (m_event != null)
    {
        IInstancingContext context = m_event.Cast<IInstancingContext>();
        if (context.CanInsert(e.Data))
        {
            ITransactionContext transactionContext = context as ITransactionContext;
            TransactionContexts.DoTransaction(transactionContext,
                delegate
                {
                    context.Insert(e.Data);
                },
                Localizer.Localize("Drag and Drop"));

            if (m_statusService != null)
                m_statusService.ShowStatus(Localizer.Localize("Drag and Drop"));
        }
    }
}
```

Similarly to the drag over event, this handler first checks that the insertion is possible. It adapts the event parameter to both an `IInstancingContext` and an `ITransactionContext`. `EventArgs` derives from `EditingContext`, which ultimately implements `ITransactionContext` in one of its derived classes, `TransactionContext`. Similarly to the paste operation from a menu item, the insertion is done by calling `IInstancingContext.Insert()` inside a transaction using an `ITransactionContext` object.

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [Simple DOM Editor Programming Discussion](#)

Implementing Instancing

Implement the `IInstancingContext` interface in a class that provides services for a context in the application. This usually means implementing other interfaces as well. As previously noted, the `ISelectionContext` interface must be implemented, so items can be selected in that context.

For example, the [ATF Simple DOM Editor Sample](#) implements a context for its Resources window's `ListView` with its `EventContext` class. This class implements `IInstancingContext` and is derived from `EditingContext`, which implements `ISelectionContext`:

```
public class EventContext : EditingContext,
    IListview,
    IItemView,
    IObservableContext,
    IInstancingContext,
    IEnumerableContext
```

Topics

- [Implementing IInstancingContext](#)
- [Implementing Drag and Drop](#)

Implementing IInstancingContext

The [ATF Simple DOM Editor Sample](#) implements `IInstancingContext` in two context classes: `EventContext` as noted previously, and `EventSequenceContext` for the `ListView` of its main window. To see how all the `IInstancingContext` methods are implemented in `EventContext`, see [IInstancingContext Interface](#). This also shows how the `StandardEditCommands` component interacts with these methods.

Implementing Drag and Drop

You can also use `IInstancingContext` with drag and drop editing. To learn how the [ATF Simple DOM Editor Sample](#) uses drag and drop for its editing with `IInstancingContext`, see the previous topic, [Drag and Drop and Instancing](#).

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [Drag and Drop and Instancing](#), [Simple DOM Editor Programming Discussion](#)

Documents in ATF

Applications store their data in documents. ATF provides a Document Framework with interfaces and components to facilitate working with application documents.

The section covers these topics:

- [What is a Document in ATF?](#): Introduces the `IDocument` and `IDocumentClient` interfaces that an individual document uses.
 - [Document Registry and Services](#): Discusses the `DocumentRegistry` to track documents and other document services offered in `IDocumentService`.
 - [Document Handling Components](#): Details the MEF components providing services document handlers need.
 - [Implementing a Document and Its Client](#): Shows how to implement both `IDocument` and `IDocumentClient` interfaces for a document type.
-

What is a Document in ATF?

A document is a file containing application data. Nearly all applications process data, so most have documents that can be saved and opened for later modification.

IDocument Interface

In ATF, a document object implements the `IDocument` interface. `IDocument` derives from `IResource`, an interface for resources that have a type and unique resource identifier (URI).

`IDocument` is very simple, consisting of:

- `IsReadOnly` property: indicate whether the document is read-only.
- `Dirty` property: is the version of the document in memory different than the version in its file?
- `DirtyChanged` event: `Dirty` changed.
- `Type` property (`IResource`): get the text string that identifies this document type to the end-user.
- `Uri` property (`IResource`): get or set the document's URI as an absolute path.
- `UriChanged` event (`IResource`): `Uri` changed.

Topics

- [IDocument Interface](#)
- [IDocumentClient Interface](#)
- [DocumentClientInfo Class](#)

IDocumentClient Interface

Each type of document has one or more `IDocumentClient` interfaces associated with it. A document client does the work of handling a document and implements `IDocumentClient` for operations on the document, mainly file-related:

- `Info` property: get `DocumentClientInfo` with information about the document client, such as file type and its file extensions, and so on. For details, see [DocumentClientInfo Class](#) below.
- `CanOpen()`: can the document be opened?
- `Open()`: open the document from a file.
- `Show()`: display the document.
- `Save()`: save the document to a file.
- `Close()`: close the file.

An application typically provides one `IDocumentClient` interface for each type of document it handles. Different applications can handle the same document in different ways, so they could have different `IDocumentClient` implementations for the same document. Applications processing text files may do so in very different ways, for instance. In addition, a single application could provide different views of application data and have several implementations of `IDocumentClient` for the same document type. For example, an application could open an XML file containing a Schema as plain text or as a tree of nodes, as Visual Studio does.

DocumentClientInfo Class

`DocumentClientInfo` holds information about a document for a particular document client. This information can be used to determine if a file can be opened, for example.

`DocumentClientInfo` has a set of constructors, who between them, provide the following information:

- Name describing the type of document, unique for each document type in the application.
- File extensions (without the initial '!').
- Name of a "New document" icon used in the UI, such as on a menu item or button.
- Name of an "Open document" icon used in the UI, such as on a menu item or button.
- Whether the client can open multiple documents simultaneously.

Links on this page to other topics

Authoring Tools Framework

Document Registry and Services

ATF provides services for documents and also has a registry to keep track of documents.

IDocumentRegistry Interface and DocumentRegistry Component

The document registry's purpose is to keep track of all the documents an application has open. A document in this context is any object implementing [IDocument](#).

Topics

- [IDocumentRegistry Interface and DocumentRegistry Component](#)
- [IDocumentService Interface](#)

The [IDocumentRegistry](#) provides the following properties:

- `ActiveDocument`: get or set the active [IDocument](#) object.
- `Documents`: get an enumeration of all active [IDocument](#) objects.

A document is typically some kind of object, so the following methods are useful:

- `GetActiveDocument<T>()`: get the active document as a type `T`, if possible.
- `GetMostRecentDocument<T>()`: get the most recently active document of type `T`, which may not be the same as the currently active document.
- `GetDocument()`: get the document specified by a given URI.

Finally, these events track active document changes:

- `ActiveDocumentChanging`: active document is about to change.
- `ActiveDocumentChanged`: active document has changed.
- `DocumentAdded`: a document was added to the document registry's collection.
- `DocumentRemoved`: a document was removed from the document registry's collection.

The [DocumentRegistry](#) component implements [IDocumentRegistry](#) to track open documents. All you need to do to use this component is to add it to your application's MEF catalog. Nearly all the ATF samples use [DocumentRegistry](#).

IDocumentService Interface

The [IDocumentService](#) interface provides a way to handle the UI for document handling: opening, saving, and closing documents. Its methods, which do exactly what they say, include:

- `OpenNewDocument()`
- `OpenExistingDocument()`
- `Save()`
- `SaveAs()`
- `SaveAll()`
- `Close()`
- `CloseAll()`

[IDocumentService](#) also contains the accompanying events for documents being saved, closed, and opened.

The [IDocumentClient](#) interface also provides `Save()` and `Close()` methods. What's the difference between these and the [IDocumentService](#) methods of the same name?

A document service provider implementing [IDocumentService](#), such as [StandardFileCommands](#), can deal with or provide the user interface, such as a menu item to close a document, and directly interacts with a user. When asked to close a document, the [IDocumentService](#) figures out which document is involved and calls the `Close()` method in the appropriate document client implementing [IDocumentClient](#). The document client then does everything needed to actually close the document.

The [StandardFileCommands](#) component implements the [IDocumentService](#) interface. For details, see [DocumentRegistry Component](#).

Links on this page to other topics

[Document Handling Components](#)

Document Handling Components

Several ATF components provide services document handlers can use.

Many of the sample applications use all of these components. Some of these components depend on each other, as noted in the discussion below on component requirements.

DocumentRegistry Component

As previously discussed, the `DocumentRegistry` component tracks open documents (objects implementing `IDocument`) for an application and implements `IDocumentRegistry`.

Topics

- [DocumentRegistry Component](#)
- [StandardFileCommands Component](#)
- [AutoDocumentService Component](#)
- [MainWindowTitleService Component](#)

To use this component, simply add it to the application's MEF catalog. It doesn't import anything, so no other components are required to use it.

However, other components import and require `IDocumentRegistry`: `StandardFileCommands`, `AutoDocumentService`, and `MainWindowTitleService` — the other components discussed here. Therefore, you must provide `DocumentRegistry` or some other component implementing `IDocumentRegistry` to use these other components. Other components besides these also import `IDocumentRegistry`, such as `CommandLineArgsService`, which parses and validates command line arguments. These components all need and take advantage of the document tracking services `DocumentRegistry` offers.

For additional information on this component, see [IDocumentRegistry Interface](#) and [DocumentRegistry Component](#).

StandardFileCommands Component

`StandardFileCommands` implements File menu commands that modify the document registry: New, Open, Save, SaveAs, Save All, and Close. New and Open commands are created for each `IDocumentClient` component in the application.

In other words, including this component in your application automatically adds all the usual menu items you need to handle documents.

`StandardFileCommands` is the only ATF component that implements `IDocumentService` and can be used to satisfy such an import. For instance, the `AutoDocumentService` component imports `IDocumentService`.

`StandardFileCommands` requires (except as noted) the following:

- `DocumentRegistry` component or some other component implementing `IDocumentRegistry`.
- `CommandService` component or component implementing `ICommandService`.
- `FileDialogService` component or component implementing `IFileDialogService`.
- `StatusService` component or component implementing `IStatusService` (requested but not required).
- `IDocumentClient` implementer for each document type.

These requirements are easily met by including the indicated components in the application's MEF catalog — except for the last one: document clients implementing `IDocumentClient`. The application must implement `IDocumentClient` for every document type it supports. For details on doing this, see [Implementing a Document and Its Client](#).

Note that the application can implement as many `IDocumentClient` implementers as it wishes, because of the form of the import:

```
[ ImportMany ]
private Lazy<IDocumentClient>[] m_documentClients;
```

AutoDocumentService Component

When an application starts up, `AutoDocumentService` automatically opens the application's last open document or a new, empty document.

AutoDocumentService requires the following (except as noted):

- DocumentRegistry component or some other component implementing `IDocumentRegistry`.
- SettingsService component or component implementing `ISettingsService` (requested but not required).
- CommandLineArgsService component (requested but not required).
- IDocumentService implementer, such as `StandardFileCommands`, to open an existing document.
- IMainWindow implementer, which is met by creating a `MainForm` component.
- Form implementer, which is met by creating a `MainForm` component.
- IDocumentClient implementer for each document type to open a document.

Most of these requirements are easily met by including the indicated components. `MainForm` derives from `Form` and implements `IMainWindow`, so it satisfies two of the imports, as noted.

The last requirement is the most demanding: the application must implement `IDocumentClient` for every document type it supports. For details on doing this, see [Implementing a Document and Its Client](#). Just as for `StandardFileCommands`, the application can have as many `IDocumentClient` implementers as it wants.

MainWindowTitleService Component

`MainWindowTitleService` updates the application main form's title to reflect the current document and indicate whether it is dirty or not.

`MainWindowTitleService` requires the following:

- DocumentRegistry component or some other component implementing `IDocumentRegistry`.
- IMainWindow implementer, which is met by creating a `MainForm` component.

Topics in this section

Links on this page to other topics

[Document Registry and Services](#), [Implementing a Document and Its Client](#)

Implementing a Document and Its Client

This section reviews how a document and its client are implemented, as demonstrated in the [ATF Simple DOM Editor Sample](#).

Implementing IDocument

Simple DOM Editor implements `IDocument` in its `EventSequenceDocument` class by deriving from `DomDocument`, which implements `IDocument`.

Here are the properties:

```
public virtual bool IsReadOnly
{
    get { return false; }
}
...
public virtual bool Dirty
{
    get
    {
        return m_dirty;
    }
    set
    {
        if (value != m_dirty)
        {
            m_dirty = value;

            OnDirtyChanged(EventArgs.Empty);
        }
    }
}
```

And here is the event handler that raises the event:

```
protected virtual void OnDirtyChanged(EventArgs e)
{
    DirtyChanged.Raise(this, e);
}
```

Topics

- [Implementing IDocument](#)
- [Implementing a Document Client](#)
 - [DocumentClientInfo Property](#)
 - [CanOpen\(\) Method](#)
 - [Open\(\) Method](#)
 - [Show\(\) Method](#)
 - [Save\(\) Method](#)
 - [Close Method](#)

Implementing a Document Client

The document client handles the details of how file contents are retrieved and turned into a document, and vice versa. The items in this interface, discussed in [IDocumentClient Interface](#) are implemented in the `Editor` class of the Simple DOM Editor sample.

DocumentClientInfo Property

First, the implementation of the `DocumentClientInfo` property is straightforward:

```

public DocumentClientInfo Info
{
    get { return DocumentClientInfo; }
}

...
public static DocumentClientInfo DocumentClientInfo = new DocumentClientInfo(
    Localizer.Localize("Event Sequence"),
    new string[] { ".xml", ".esq" },
    Sce.Atf.Resources.DocumentImage,
    Sce.Atf.Resources.FolderImage,
    true);

```

This indicates the application can open files with extensions of ".xml" and ".esq", and multiple documents can be open at once. Icons are also provided for the UI.

CanOpen() Method

The `CanOpen()` method uses `DocumentClientInfo.IsCompatibleUri()` to determine if the file's extension is one of the ones set up in the `DocumentClientInfo`:

```

public bool CanOpen(Uri uri)
{
    return DocumentClientInfo.IsCompatibleUri(uri);
}

```

Open() Method

Simple DOM Editor uses the ATF DOM for its application data. The ATF DOM supports saving its data to XML using the `DomXmlWriter` class and reading it with `DomXmlReader`, and Simple DOM Editor takes full advantage of this.

The `Open()` method uses `DomXmlReader` if the file exists, and otherwise creates a new document. The `DomNode`, `node`, is the root node of the `DomNode` tree.

Through adaptation, a `DomNode` can be dynamically cast to many different kinds of objects. In this method, the root `DomNode` is cast to both an `EventSequenceDocument` and an `EventSequenceContext`. This can be done, because both `EventSequenceDocument` and `EventSequenceContext` are DOM node adapters: both ultimately derive from the `DomNodeAdapter` class. For more information about adaptation, see [Adaptation in ATF](#) and the ATF Programmer's Guide: Document Object Model (DOM).

The second part of this method creates an `EventSequenceDocument`, `document`, which derives from `DomDocument`. The method casts the root `DomNode` `node` as an `EventSequenceDocument` object. It goes on to set various properties for the `EventSequenceDocument` object.

The document is displayed in a `ListView` control, which is part of the `EventSequenceContext` that is also cast from the root `DomNode`. The method sets up a `ControlInfo` for this control and also registers it.

This shows that the object implementing the `IDocument` interface can be adapted to various classes, as needed.

```

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.eventSequenceType.Type, Schema.eventSequenceRootElement);
    }

    EventSequenceDocument document = null;
    if (node != null)
    {
        // Initialize Dom extensions now that the data is complete
        node.InitializeExtensions();

        EventSequenceContext context = node.As<EventSequenceContext>();

        ControlInfo controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center
    );
        context.ControlInfo = controlInfo;

        // set document URI
        document = node.As<EventSequenceDocument>();
        document.Uri = uri;

        // set document GUIs for search and replace
        document.SearchUI = new DomNodeSearchToolStrip();
        document.ReplaceUI = new DomNodeReplaceToolStrip();
        document.ResultsUI = new DomNodeSearchResultsListView(m_contextRegistry);

        context.ListView.Tag = document;

        // show the ListView control
        m_controlHostService.RegisterControl(context.ListView, controlInfo, this);
    }

    return document;
}

```

Show() Method

Show() uses an EventSequenceContext object, which it also casts from the EventSequenceDocument, document. The method then uses the ControlHostService object to show the document, which is displayed in a ListView in the context:

```

public void Show(IDocument document)
{
    EventSequenceContext context = Adapters.As<EventSequenceContext>(document);
    m_controlHostService.Show(context.ListView);
}

```

Save() Method

To complete the symmetry of the Open() method, Save() uses the DomXmlWriter class to write the document to an XML file. The DomXmlWriter.Write() method needs a DomNode, so the document is adapted to one:

```

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);
        EventSequenceDocument eventSequenceDocument = (EventSequenceDocument)document;
        writer.Write(eventSequenceDocument.DomNode, stream, uri);
    }
}

```

Close Method

To close this DocumentClientInfo interface implementation discussion, Close() again adapts the document to an EventSequenceContext. It gets the ListView control from the context and unregisters it using the ControlHostService object. It performs various context clean up. Finally, it uses the DocumentRegistry object to invoke the Remove() method to remove the document from the document registry.

```

public void Close(IDocument document)
{
    EventSequenceContext context = Adapters.As<EventSequenceContext>(document);
    m_controlHostService.UnregisterControl(context.ListView);
    context.ControlInfo = null;

    // close all active EditingContexts in the document
    foreach (DomNode node in context.DomNode.Subtree)
        foreach (EditingContext editingContext in node.AsAll<EditingContext>())
            m_contextRegistry.RemoveContext(editingContext);

    // close the document
    m_documentRegistry.Remove(document);
}

```

Topics in this section

Links on this page to other topics

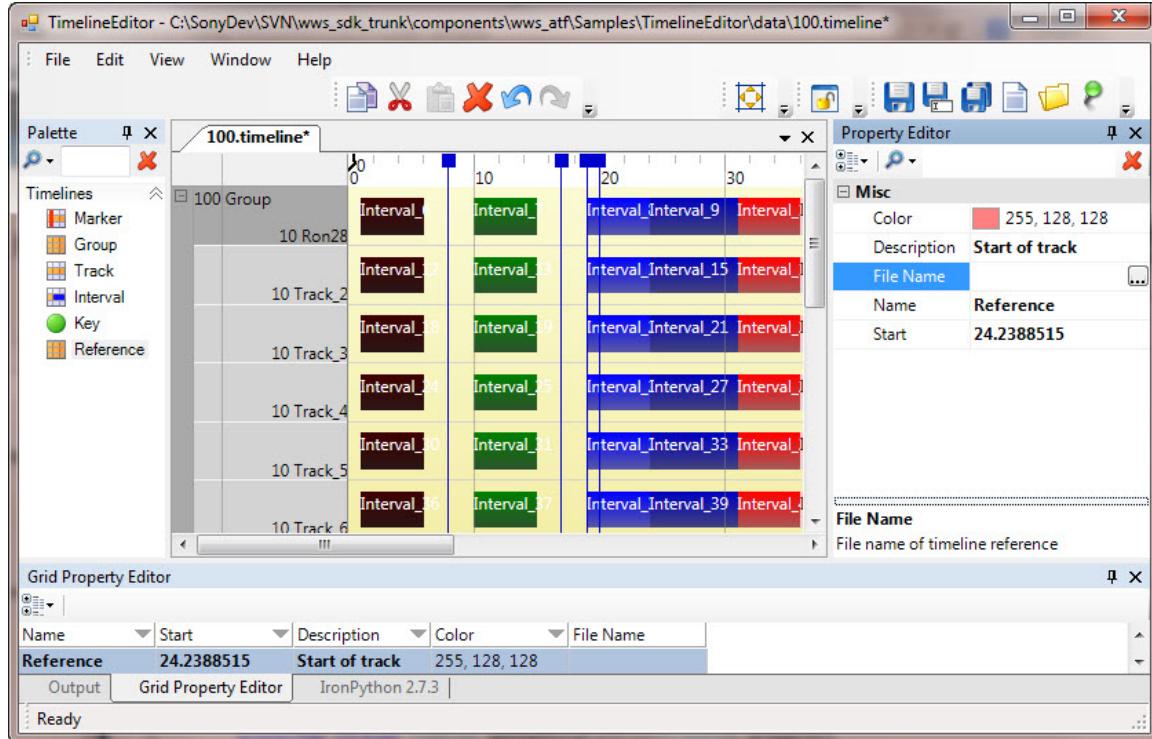
[Adaptation in ATF](#), [ATF Simple DOM Editor Sample](#), [What is a Document in ATF?](#)

Property Editing in ATF

Editing data objects' properties is usually one of the fundamental purposes of an application, so ATF provides comprehensive facilities for property editing.

Note that property in the context of this topic refers to an attribute of application data — not to be confused with a C# property.

The [ATF Timeline Editor Sample](#) in the figure features two different property editors in the "Property Editor" and "Grid Property Editor" tabs:



This section covers the following topics:

- [Using Properties in ATF](#): Overview of the ATF property editing process.
- [Selection Property Editing Context](#): How the `SelectionPropertyEditingContext` class is used in editing properties.
- [Property Descriptors](#): Discusses various kinds of `PropertyDescriptor` classes and how they are created from constructors or XML Schema annotations.
- [Value Editors](#): Types of value editors and how they are implemented.
- [Value Editors and Value Editing Controls](#): Survey of the various kinds of ATF value editors and their associated value controls.
- [Value Converters](#): How value converters convert values between value editors and value controls.
- [Property Editor Components](#): Discussion of widely used ATF property editor components.
- [Implementing a Property Editor](#): How to implement a property editor.

Using Properties in ATF

Property Editing Overview

Applications typically create and edit data, so editing data objects' properties is one of the fundamental purposes of an application. For instance, one of the main functions of a vector graphics application is to change graphical object's properties, such as line width, thickness, and color.

Applications usually provide property editor controls for the application's data objects. These editors may go by various names, such as object inspectors, but their function is the same. Typically the property editor displays a collection of properties belonging to the selected object or properties common to all selected objects. For each property, the editor must provide an appropriate value editor control for that property's data type. A color editor control is very different from a numerical editor control, for instance.

Topics

- [Property Editing Overview](#)
- [Selection Property Editing Context](#)
- [Property Descriptors](#)
- [Value Editors](#)
- [Value Editing Controls](#)
- [Value Converters](#)
- [Property Editor Components](#)

Because the property editor shows properties for a selection, it is dependent on the current context. Property editors can only be used when there is a selection and the active context is one in which properties can be edited. ATF uses the context to determine which property editors are appropriate. For more information, see [Selection Property Editing Context](#) and [ATF Contexts](#).

If an application uses a data model for its application data based on the ATF Document Object Model (DOM), ATF provides a great deal of support for property editors of the attributes of the application's data types. The DOM defines the data types used in the application. Property editing support is even better when the DOM uses the XML Schema Definition (XSD) to specify the data types. A data type can have attributes, which are equivalent to properties. For more details on the ATF DOM, see the ATF Programmer's Guide: Document Object Model (DOM), which you can download at [ATF Documentation](#).

ATF provides several kinds of classes to handle property editing: a selection property editing context, property descriptors, value editors, value editing controls, value converters, and property editors. Although these classes integrate well with the ATF DOM, they can be useful for property editing even if the application does not use the ATF DOM.

Selection Property Editing Context

The default context for property editors is `SelectionPropertyEditingContext`, which is constructed when property editors are created. The `SelectionPropertyEditingContext` class supports property editing on any `ISelectionContext`, which provides properties and methods to get the selected items, last selected item, and so forth. This interface also provides events for selection changes. For details, see [Selection Property Editing Context](#).

Property Descriptors

A property descriptor is a metadata class describing a property/attribute of a data type. It contains information about the property, such as its name, description, and an appropriate editor for the property, given its data type. Property descriptors provide the information needed to actually edit properties.

Property descriptors can be described in a variety of ways: from constructors and from XML Schemas in the ATF DOM.

For more information, see [Property Descriptors](#).

Value Editors

Every data object property has a data type, which may be defined in the application's data model. ATF provides a large set of value editors for the various kinds of data types, listed in [Value Editors](#) and [Value Editing Controls](#). For example, the `BoundedIntEditor` edits bounded integers. Value editors use a value editing control for the user to interact with in modifying the property value.

Value Editing Controls

Every value editor has a corresponding value editing control that users employ to set the desired property value. A value editing control for a Boolean value could simply be a CheckBox control, for instance. A bounded integer could use a spinner control. A color editor can use a color picker control. For details, see [Value Editors and Value Editing Controls](#).

Value Converters

The value converter converts the property value between its internal data representation in a value editor and a form that appears in a value editing control. For instance, a value converter for an integer value could go back and forth between an internal `int` value and a string representation of the integer that can be displayed in a value editing control. The conversion typically goes in both directions. For more information, see [Value Converters](#).

Property Editor Components

Property editor controls contain the value editing control components. There are two main classes: `GridView` and `PropertyGridView`, which both derive from `PropertyView`. `PropertyGridView` is used for two column style editors. `GridView` shows a spreadsheet view, listing properties of several items.

You don't need to use `GridView` and `PropertyGridView` directly. A `GridView` is usually provided by the `GridPropertyEditor` component, `PropertyGridView` by the `PropertyEditor` component. These components make these property editor controls very easy to use.

These property editors are designed to show the common properties of selected objects, so you must set up the selection context for the items whose properties you want to edit. These editors ultimately require a list of property descriptors for the properties to be edited.

For details, see [Property Editor Components](#).

Topics in this section

Links on this page to other topics

[ATF Contexts](#), [ATF Documentation](#), [Property Descriptors](#), [Property Editor Components](#), [Selection Property Editing Context](#), [Value Converters](#), [Value Editors and Value Editing Controls](#)

Selection Property Editing Context

Property editors show properties for selected items, so property editing is context dependent. The key interface for contexts is `ISelectionContext`, which provides properties and methods to get the selected items and last selected item, change the selection, and so forth. `ISelectionContext` also provides events for selection changes: `SelectionChanging` and `SelectionChanged`. Raising these events allows selections to be tracked and the properties of selected items obtained for property editors.

The `SelectionPropertyEditingContext` class supports property editing on any `ISelectionContext`. `SelectionPropertyEditingContext` implements `IPropertyEditingContext`, which is for contexts in which properties can be edited by controls, such as `PropertyGrid` and `GridControl`, on which the components `PropertyEditor` and `GridPropertyEditor` are built.

`IPropertyEditingContext` defines two properties:

- `Items`: Get an enumeration (`IEnumerable<object>`) of the selected items that have editable properties.
- `PropertyDescriptor`s: Get an enumeration (`IEnumerable<PropertyDescriptor>`) of the property descriptors for selected items. These are the property descriptors common to all items in the selection.

These properties are obtained every time the selection changes, that is, when the events in the `ISelectionContext` interface are raised.

Thus the context can provide a collection of property descriptors whose properties all apply to the current selection. Each property descriptor, in turn, provides all the information needed to edit its property.

Topics in this section

Links on this page to other topics

No links

Property Descriptors

PropertyDescriptor Class

The property descriptors ATF uses all derive from the `System.ComponentModel.PropertyDescriptor` class, a public abstract class. From this base class, ATF derives `Sce.Atf.Controls.PropertyEditing.PropertyDescriptor`. This ATF class is also abstract, so ATF in turn derives classes from it. In this documentation, `PropertyDescriptor` refers to ATF's `Sce.Atf.Controls.PropertyEditing.PropertyDescriptor` unless `System.ComponentModel.PropertyDescriptor` is specified.

`PropertyDescriptor` has a variety of constructors to specify the `PropertyDescriptor`'s associated information. Here is a list of all the various parameters; the last three are not required:

Topics

- [PropertyDescriptor Class](#)
- [AttributePropertyDescriptor Class](#)
- [ChildPropertyDescriptor Class](#)
- [ParseXml Method](#)
- [Specifying Property Descriptors](#)
 - [Constructing Property Descriptors](#)
 - [Using the DOM](#)
 - [Not Using the DOM](#)
 - [Creating Property Descriptors with XML Schema Annotations](#)

- `Name (string)`: Property name.
- `Type (System.Type)`: Type of the property.
- `Category (string)`: Category of the property.
- `Description (string)`: Description of the property.
- `ReadOnly (bool)`: Whether or not the property is read-only.
- `Editor (object)`: The value editor used to edit the property. For details on value editors, see [Value Editors](#).
- `Type converter (System.ComponentModel.TypeConverter)`: The value type converter used for this property. For more information, see [Value Converters](#).
- `Attributes (System.Attribute[])`: An array of `System.Attribute` custom attributes. These may be provided, but are not used by ATF. Instead, attributes are specified in the `AttributePropertyDescriptor` class, as described in [AttributePropertyDescriptor Class](#).

Note that the "Editor" and "Type converter" objects, when provided, indicate precisely which value editor and value type converter to use for the property.

All of the above parameters for the constructor are available as C# properties of `PropertyDescriptor`, except for the value editor. The `PropertyDescriptor.GetEditor(Type editorBaseType)` method gets the value editor for the given type, returning the value editor provided in the constructor, if there was one and it is compatible with the given `Type`. If no editor was provided, a default value editor and value type converter are supplied, based on the specified `Type`.

ATF derives two classes from `PropertyDescriptor`: `AttributePropertyDescriptor` and `ChildPropertyDescriptor`. These classes are used with the ATF DOM. Applications may also derive their own property descriptors, as in the [ATF Property Editing Sample](#) and the [ATF Tree List Editor Sample](#).

AttributePropertyDescriptor Class

In addition to the parameters for `PropertyDescriptor`, the `AttributePropertyDescriptor` constructor has an `AttributeInfo` parameter, which is required. The `AttributeInfo` class contains information used by the ATF DOM. A DOM's data type definitions describe attributes, that is properties, of data types. `AttributeInfo` has metadata that includes the underlying attribute type (`AttributeType`) and any restrictions the attribute was defined with in the DOM, such as bounds limitations for integer type attributes.

`AttributePropertyDescriptor` is the main type of property descriptor used in ATF. The samples show numerous examples of using it to specify property descriptors for application data types.

ChildPropertyDescriptor Class

In the DOM, data elements are stored in a tree of DOM nodes. A data type can specify that it has children of a certain type —

`ChildPropertyDescriptor` describes properties of types of child nodes. Child information includes the child's data type and whether the child is a list of nodes. `ChildInfo` also contains any restriction rules, such as limits on the number of children.

ParseXml Method

When an application uses the XML Schema Definition (XSD) language to specify the DOM data types, the data type definitions may include annotations that add information about the data type. In particular, the annotations can specify property descriptors.

`PropertyDescriptor.ParseXml()` parses the XML Schema for these types of annotations and creates property descriptors. For more details on these annotations, see [Creating Property Descriptors with XML Schema Annotations](#).

Specifying Property Descriptors

You can specify property descriptors two ways: constructing property descriptors directly or, with the ATF DOM, using XML Schema annotations. For full details on how to do this, see the ATF Programmer's Guide: Document Object Model (DOM).

Constructing Property Descriptors

You can create a property descriptor object directly with its constructor.

Using the DOM

The example here uses `AttributePropertyDescriptor` objects with the ATF DOM. When an application using the DOM starts up, it reads the data type definitions from a type definition file using a class called a schema type loader. The schema type loader creates metadata class objects for the type information, such as `AttributeInfo` objects. The loader can add information about the data types. In particular, it can define a property descriptor for an attribute of a data type.

Here's an example from the [ATF Simple DOM Editor Sample](#) that defines property descriptors for an "eventType" data type's attributes. It creates three `AttributePropertyDescriptor` objects, placing them in a `PropertyDescriptorCollection` object:

```
Schema.eventType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                Schema.eventType.nameAttribute,
                null,
                Localizer.Localize("Event name"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Time"),
                Schema.eventType.timeAttribute,
                null,
                Localizer.Localize("Event starting time"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Duration"),
                Schema.eventType.durationAttribute,
                null,
                Localizer.Localize("Event duration"),
                false),
        }));
});
```

These constructors for the `AttributePropertyDescriptor` objects use this form of the constructor:

```
public AttributePropertyDescriptor(
    string name,
    AttributeInfo attribute,
    string category,
    string description,
    bool isReadOnly);
```

These `AttributePropertyDescriptor` constructor examples specify the property descriptor's name, `AttributeInfo`, `description`, and `read only` properties; they leave `category` unspecified as `null`. The `AttributeInfo` object, as in `Schema.eventType.nameAttribute`, is a type metadata object, created by the schema type loader.

`AttributePropertyDescriptor` also has a constructor with an editor parameter:

```

public AttributePropertyDescriptor(
    string name,
    AttributeInfo attribute,
    string category,
    string description,
    bool isReadOnly,
    object editor);

```

This constructor form is used in this example from the schema type loader in the [ATF Win Forms App Sample](#) to specify an editor object `BoolEditor()` for a Boolean editor:

```

new AttributePropertyDescriptor(
    Localizer.Localize("Compressed"),
    Schema.animationResourceType.compressedAttribute,
    null,
    Localizer.Localize("Whether or not animation is compressed"),
    false,
    new BoolEditor())

```

Finally, the `AttributePropertyDescriptor` constructor can also specify the value type converter:

```

public AttributePropertyDescriptor(
    string name,
    AttributeInfo attribute,
    string category,
    string description,
    bool isReadOnly,
    object editor,
    TypeConverter typeConverter);

```

This constructor from the [ATF Simple DOM Editor Sample](#) creates a value type converter for an enumerated type:

```

string[] primitiveKinds = new string[]
{
    "Lines",
    "Line_Strips",
    "Polygons",
    "Polylist",
    "Triangles",
    "Triangle_Strips",
    "Bezier_Curves",
    "Bezier_Surfaces",
    "Subdivision_Surfaces"
};

...
new AttributePropertyDescriptor(
    Localizer.Localize("Primitive Kind"),
    Schema.geometryResourceType.primitiveTypeAttribute,
    null,
    Localizer.Localize("Kind of primitives in geometry"),
    false,
    new EnumUITypeEditor(primitiveKinds),
    new EnumTypeConverter(primitiveKinds))

```

The variable `primitiveKinds` contains a `string` array that the constructors for the value type editor and converter, `EnumUITypeEditor` and `EnumTypeConverter`, both use as the parameter.

Not Using the DOM

The [ATF Property Editing Sample](#) defines its own data types without a DOM in an `ItemBase` class, which defines a method to get a collection of property descriptors that is used to populate property editors. The class `FieldPropertyDescriptor` derives from ATF's `PropertyDescriptor` class. This method constructs `FieldPropertyDescriptor` objects and adds them to a `PropertyDescriptorCollection` collection:

```

public override PropertyDescriptorCollection GetProperties()
{
    PropertyDescriptorCollection props = new PropertyDescriptorCollection(null);
    foreach (FieldInfo field in GetType().GetFields())
    {
        FieldPropertyDescriptor fieldDesc = new FieldPropertyDescriptor(field, GetType());
        bool toAdd = true;
        foreach (Attribute attr in fieldDesc.Attributes)
        {
            if (attr.GetType() == typeof(ChildPropertyAttribute))
            {
                toAdd = false;
                break;
            }
        }

        if (toAdd)
            props.Add(fieldDesc);
    }
    return props;
}

```

For more information about the programming in this sample, see [Property Editing Programming Discussion](#).

Creating Property Descriptors with XML Schema Annotations

If you use the XML Schema Definition (XSD) language to define your data types in the DOM, you can specify property descriptors in annotations. The annotation's attributes provide the same information as property descriptor constructor parameters.

The `<scea.dom.editors.attribute>` tag is used for an annotation that describes a property descriptor.

The [ATF Timeline Editor Sample](#)'s data definition includes a "timelineRef" type, which contains an annotation defining property descriptors:

```

<xs:complexType name="timelineRefType">
    <xs:annotation>
        <xs:appinfo>
            <scea.dom.editors menuText="Reference" description="Timeline Reference"
                image="TimelineEditorSample.Resources.group.png" category="Timelines" />
            <scea.dom.editors.attribute name="name" displayName="Name" description="Name" />
            <scea.dom.editors.attribute name="start" displayName="Start" description="Start Time" />
            <scea.dom.editors.attribute name="description" displayName="Description" description="Event
                description" />
            <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" />
            <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"/>
        </xs:appinfo>
    </xs:annotation>
    <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 `<xs:attribute>` tags at the end specify the actual attributes of the "timelineRef" type, which includes their raw data type. For instance, the "name" attribute is a string, and the "start" attribute is a floating point number.

The `<scea.dom.editors.attribute>` tags define the property descriptors for each attribute/property and do not specify its data type; that is already specified in the `<xs:attribute>` tags.

The first three `<scea.dom.editors.attribute>` tags only specify three properties:

- "name": the attribute's name.
- "displayName": the property name, displayed in the value editor.
- "description": an optional full description of the attribute.

The last two property descriptor definitions include two more properties, followed by optional parameters:

- "editor": an optional fully qualified name (by namespace) of a `UITypeEditor` control for editing, followed by its assembly name and optional parameters.
- "converter": an optional fully qualified name (by namespace) of a `TypeConverter` value converter class used for this property, followed by optional parameters.

For instance, the value editor for the "ref" property is given by

```
editor="Sce.Atf.Controls.PropertyEditing.FileUriEditor,Atf.Gui.WinForms:Timeline files (*.timeline)|*.timeline"
```

This specifies the full path of the value editor class, a URI editor, followed by its assembly. After the colon is an optional parameter: a file filter string.

The value type converter for the "color" property is given by

```
converter="Sce.Atf.Controls.PropertyEditing.IntColorConverter"
```

Topics in this section

Links on this page to other topics

[ATF Property Editing Sample](#), [ATF Simple DOM Editor Sample](#), [ATF Timeline Editor Sample](#), [ATF Tree List Editor Sample](#), [ATF Win Forms App Sample](#), [Authoring Tools Framework](#), [Property Editing Programming Discussion](#), [Value Converters](#), [Value Editors](#)

Value Editors

Every property descriptor has an explicit or implicit value editor to allow a user to edit the property value described by the descriptor. The `PropertyDescriptor.GetEditor()` method returns the editor for a given type. If the property descriptor class derived from `PropertyDescriptor` does not implement `GetEditor()`, the base `PropertyDescriptor.GetEditor()` method is called:

```
public override object GetEditor(Type editorBaseType)
{
    if (m_editor != null &&
        editorBaseType.IsAssignableFrom(m_editor.GetType()))
    {
        return m_editor;
    }

    return base.GetEditor(editorBaseType);
}
```

Topics

- [UITypeEditor Class](#)
- [Implementing a UITypeEditor](#)
 - [ColorEditor Value Editor Class](#)
 - [BoundedIntEditor Value Editor Class](#)
 - [BoundedFloatEditor Value Editor Class](#)
 - [FolderBrowserDialogUITypeEditor Value Editor Class](#)
- [IPropertyEditor Interface](#)
- [Non-UITypeEditor Value Editors](#)
 - [Implementing a Non-UITypeEditor](#)
 - [BoolEditor Value Editor Class](#)
- [IAnotatedParams Interface](#)

If a value editor for the type `editorBaseType` is not found, the base method

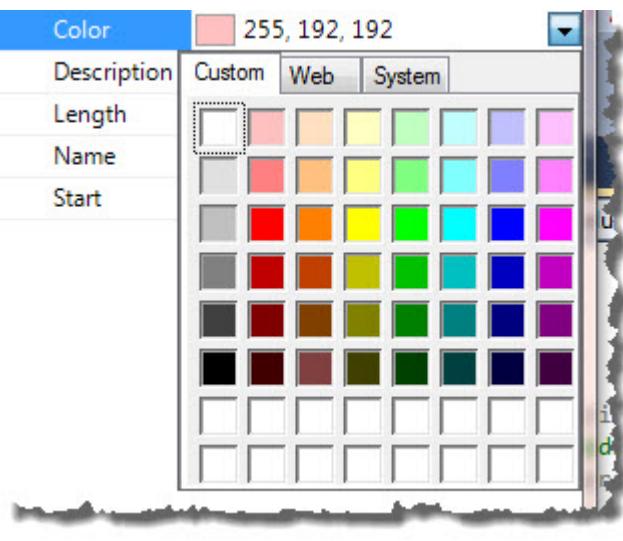
`System.ComponentModel.PropertyDescriptor.GetEditor()` is called, which returns a default editor for that type, assuming one can be found.

Note that the value editor is not a control itself. Instead, each value editor has a value editing control associated with it, described in [Value Editors and Value Editing Controls](#). This value editing control class is not necessarily in the value editor class's file.

There are two types of value editors: those that derive from `UITypeEditor` and those that don't. They can both implement the `IPropertyEditor` interface, which obtains a value editing control object.

UITypeEditor Class

`System.Drawing.Design.UITypeEditor` is a base class for a value editor with a value editing control that either drops down, such as a color swatch, or displays a modal dialog, such as a file selection dialog. It may also display no interactive control at all, although this option is not used in ATF. The property editor for this property typically shows either a tab ▾ or an ellipsis button ... to display either the drop down control or dialog. The illustration below, for instance, shows the `ColorEditor` drop down value editing control used in the [ATF Timeline Editor Sample](#).



Implementing a UITypeEditor

UITypeEditor includes the following methods:

- `GetEditStyle()`: Get the editor style used by `EditValue()`, a `System.Drawing.Design.UITypeEditorEditStyle` indicating the displayed control's style: Modal, DropDown, or None.
- `EditValue()`: Edit the value of the specified object using an appropriate value editing control. This displays the a value editing control or modal dialog and returns the value after user selection.
- `PaintValue()`: Paint a representation of the value of the specified object.

Most deriving classes should implement the first two methods.

The following sections demonstrate several UITypeEditor implementations.

ColorEditor Value Editor Class

The ATF Timeline Editor Sample's "timelineRef" type has a color attribute with a color value editor defined in an XML Schema annotation:

```
<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 value editing control for ColorEditor is seen in the earlier illustration.

The `Sce.Atf.Controls.PropertyEditing.ColorEditor.EditValue()` implementation is:

```
public override object EditValue(
    ITypeDescriptorContext context,
    IServiceProvider provider,
    object value)
{
    TypeConverter converter = context.PropertyDescriptor.Converter;
    if (value == null)
        value = Color.Transparent;
    Color oldColor = (Color)converter.ConvertTo(value, typeof(Color)); // convert from underlying
    type to Color
    Color newColor = (Color)base.EditValue(context, provider, oldColor);

    return converter.ConvertFrom(newColor);
}
```

Note that this method gets the property descriptor (`context.PropertyDescriptor`) from the context and then gets a value converter from the descriptor's `Converter`. It uses the converter to convert the old value to the `Color` Type with the `ConvertTo()` method.

ATF's `ColorEditor` derives from `System.Drawing.Design.ColorEditor`, so the following line invokes the base class's `EditValue()` method, which displays its own color picker:

```
Color newColor = (Color)base.EditValue(context, provider, oldColor);
```

This returned value is then converted to an appropriate form with the `ConvertFrom()` method, and that converted value is returned by `EditValue()`.

Here is `Sce.Atf.Controls.PropertyEditing.ColorEditor.PaintValue()`:

```
public override void PaintValue(PaintValueEventArgs e)
{
    if (e.Value is Color)
    {
        base.PaintValue(e);
    }
    else
    {
        TypeConverter converter = e.Context.PropertyDescriptor.Converter;
        object value = converter.ConvertTo(e.Value, typeof(Color)); // convert from underlying
        type to Color
        if (value is Color)
        {
            Color color = (Color)value;
            base.PaintValue(new PaintValueEventArgs(e.Context, color, e.Graphics, e.Bounds));
        }
    }
}
```

This method uses the value converter to convert the new color to a `Color` object, and then paints the new value using the base class's `PaintValue()` method. For information on converters, see [Value Converters](#).

BoundedIntEditor Value Editor Class

A `PaintValue()` method is not needed if the control handles the painting. For instance, the `BoundedIntEditor` value editor class implements just this `EditValue()` method:

```
public override object EditValue(ITypeDescriptorContext context, IServiceProvider provider,
object value)
{
    IWindowsFormsEditorService editorService =
        provider.GetService(typeof(IWindowsFormsEditorService)) as IWindowsFormsEditorService;
    if (editorService != null)
    {
        if (!(value is int))
            value = m_min;

        IntInputControl intInputControl = new IntInputControl((int)value, m_min, m_max);
        editorService.DropDownControl(intInputControl);
        // be careful to return the same object if the value didn't change
        if (!intInputControl.Value.Equals(value))
            value = intInputControl.Value;
    }

    return value;
}
```

This `EditValue()` method creates an `IntInputControl` control (a separate class) that handles all control painting. Similarly, `EnumUITypeEditor` provides a `ListBox` control to list enumerated values and does not implement `PaintValue`.

BoundedFloatEditor Value Editor Class

`BoundedFloatEditor` creates a dropdown value editing control with a slider and textbox to enter a floating point number.

It implements `GetEditStyle()` to specify how the control appears, dropdown in this case:

```

public override UITypeEditorDisplayStyle GetEditStyle(ITypeDescriptorContext context)
{
    return UITypeEditorDisplayStyle.DropDown;
}

```

Its `EditValue()` method gets a `IWindowsFormsEditorService` object to display a control in a drop-down area and uses it to display a `FloatInputControl` value editing control it creates and drops down with `IWindowsFormsEditorService.DropDownControl()`:

```

public override object EditValue(ITypeDescriptorContext context, IServiceProvider provider,
object value)
{
    IWindowsFormsEditorService editorService =
        provider.GetService(typeof(IWindowsFormsEditorService)) as IWindowsFormsEditorService;
    if (editorService != null)
    {
        if (!(value is float))
            value = m_min;

        FloatInputControl floatInputControl = new FloatInputControl((float)value, m_min, m_max);
        editorService.DropDownControl(floatInputControl);
        // be careful to return the same object if the value didn't change
        if (!floatInputControl.Value.Equals(value))
            value = floatInputControl.Value;
    }

    return value;
}

```

FolderBrowserDialogUITypeEditor Value Editor Class

This value editor displays a folder selection dialog, and its constructor creates this `FolderBrowserDialog` for later use:

```

public FolderBrowserDialogUITypeEditor(string description)
{
    // create instance of editing control,
    // reuse this instance for subsequent calls
    m_dialog = new FolderBrowserDialog();
    m_dialog.Description = description;
}

```

Its `GetEditStyle()` method indicates a modal dialog is displayed:

```

public override UITypeEditorDisplayStyle GetEditStyle(ITypeDescriptorContext context)
{
    return UITypeEditorDisplayStyle.Modal; // editor type is modal dialog.
}

```

The `EditValue()` method displays the `FolderBrowserDialog`, gets the user's selected path from it, and returns this value:

```

public override object EditValue(ITypeDescriptorContext context, IServiceProvider sp, object
value)
{
    IWindowsFormsEditorService editorService =
        (IWindowsFormsEditorService)sp.GetService(typeof(IWindowsFormsEditorService));

    if (editorService != null)
    {
        if (value != null)
            m_dialog.SelectedPath = value.ToString();
        m_dialog.ShowDialog();
        value = m_dialog.SelectedPath;
    }

    return value;
}

```

IPropertyEditor Interface

`IPropertyEditor` contains this method to get the value editing control associated with the value editor:

```
Control GetEditingControl(PropertyEditorControlContext context);
```

For example, the `BoundedIntEditor` value editor class implements `IPropertyEditor`:

```
public virtual Control GetEditingControl(PropertyEditorControlContext context)
{
    return new BoundedIntControl(context, m_min, m_max);
}
```

As noted previously, `UITypeEditor` classes, of which `BoundedIntEditor` is one, can also implement `IPropertyEditor` but are not required to do so, because the `UITypeEditor.EditValue()` method instantiates the value editing control. If a `UITypeEditor` does implement `IPropertyEditor`, it does not need to return the same control that it uses in `UITypeEditor.EditValue()`, and these controls are indeed different for `BoundedIntEditor`. As seen previously, `BoundedIntEditor.EditValue()` uses `IntInputControl` and `BoundedIntEditor.GetEditingControl()` returns a `BoundedIntControl`, a protected class inside `BoundedIntEditor`.

Property editors, containing value editors, ultimately derive from the `PropertyView` class. When a selection occurs, `PropertyView` calls `PropertyDescriptor.GetEditor()` to get the value editor (for a property of an item in the selection) and determines if it implements `IPropertyEditor`. If it does, `IPropertyEditor.GetEditingControl()` is called to get the value editing control, which is used to edit the value. Otherwise, `UITypeEditor.EditValue()` gets called to edit the value. Thus the `IPropertyEditor` value editing control has priority and would be used — in ATF. However, a `UITypeEditor` can be used with a .NET Framework two-column style property editor, because `UITypeEditor` is a .NET Framework class. In this case, the control specified in the `UITypeEditor.EditValue()` method would be used.

Non-UITypeEditor Value Editors

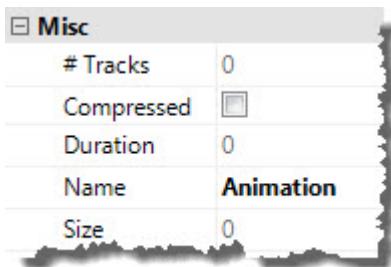
If a value editor does not derive from `UITypeEditor`, it must implement `IPropertyEditor` to specify the value editing control.

Implementing a Non-UITypeEditor

Such editors need to implement `IPropertyEditor.GetEditingControl()`, which is demonstrated here with `BoolEditor`.

BoolEditor Value Editor Class

`BoolEditor` is used in several samples, including the [ATF Simple DOM Editor Sample](#). This figure shows the property editor for an "Animation" object:



The "Compressed" Boolean property is shown with the check box associated with the `BoolEditor`.

Here is `IPropertyEditor.GetEditingControl()` for `BoolEditor`:

```
public Control GetEditingControl(PropertyEditorControlContext context)
{
    m_boolControl = new BoolControl(context);
    return m_boolControl;
}
```

`BoolEditor` also contains the implementation of the private `BoolControl` class, although it could be in a separate file.

`BoolControl`'s constructor creates a `CheckBox` control object, which is shown in the previous illustration of the value editor's control:

```

public BoolControl(PropertyEditorControlContext context)
{
    m_context = context;

    m_checkBox = new CheckBox();
    m_checkBox.Size = m_checkBox.PreferredSize;
    m_checkBox.CheckAlign = ContentAlignment.MiddleLeft;
    m_checkBox.CheckedChanged += checkBox_CheckedChanged;

    Controls.Add(m_checkBox);
    Height = m_checkBox.Height + m_topAndLeftMargin;

    RefreshValue();
}

```

The other methods in `BoolControl` do things like refresh the control, and check for changed state or size.

IAnnotatedParams Interface

If a value editor can have parameters in an XML Schema annotation, the value editor must implement `IAnnotatedParams`. This interface gets the parameters from the schema annotation and initializes the value editor for those parameters.

For example, the [ATF Timeline Editor Sample](#) defines a value editor for a file name that takes a file filter parameter in its schema definition for the "timelineRef" type:

```

<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"/>

```

The editor `FileUriEditor` implements `IAnnotatedParams`:`Initialize()` to obtain the filter string and place it in the `m_filter` field, which is used later for a file dialog:

```

public void Initialize(string[] parameters)
{
    if (parameters.Length >= 1)
        m_filter = parameters[0];
}

```

Value converters also use the `IAnnotatedParams` interface, as described in [IAnnotatedParams Interface](#).

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [ATF Timeline Editor Sample](#), [Value Converters](#), [Value Converters](#), [Value Editors and Value Editing Controls](#)

Value Editors and Value Editing Controls

A value editor is typically associated with a value editing control that provides the user interface to display and edit a property's value. The type of control is appropriate for the data type of the property.

A value editing control is embedded in a complex property editing control, such as the `GridControl` or `PropertyGrid` controls that the `GridPropertyEditor` and `PropertyEditor` components are built on. These property editor controls ultimately rely on the `PropertyView` class, which is the abstract base class for complex property editing controls, providing formats, fonts, data binding, persistent settings, and category/property information. For more information on property editor containers, see [Property Editor Components](#).

Contents

- [Value Editors and Controls](#)
- [Single Value Editors](#)
- [Multiple Value Editors](#)
- [Special Purpose](#)
- [WPF](#)
- [Other Value Editing Controls](#)

Some value editing controls contain a button ( or ) to display a dialog or drop down to get information. For example, clicking the button in the value editing control for `FolderUriEditor` displays a folder selection dialog. Similarly, the `ColorEditor`'s value editing control drops down color swatches on tabs.

The value editing control is specified by the value editor — not the property descriptor. As noted in [Value Editors](#), a value editor can specify the value editing control in two ways:

- Implement the `IPropertyEditor` interface, whose `GetEditingControl()` method returns a value editing control object.
- Derive from `System.Drawing.Design.UITypeEditor` and define an `EditValue()` method that creates a value editing control object, which gets the new value from the user and returns it.

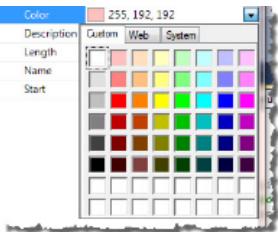
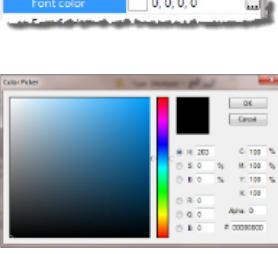
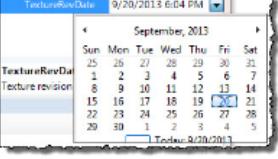
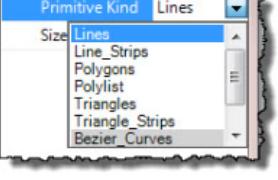
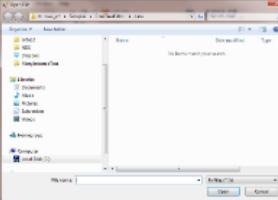
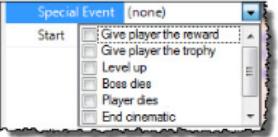
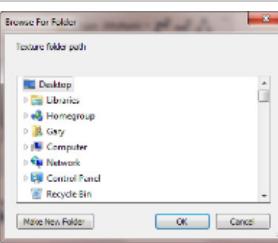
Some value editors, such as `BoundedIntEditor`, do both of these. The value editing controls obtained in these two ways for a single value editor may be different, as they are for `BoundedIntEditor`.

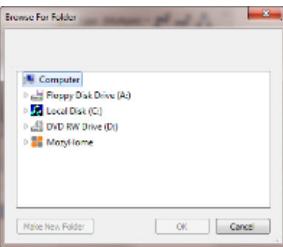
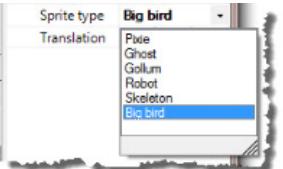
Value Editors and Controls

The tables show the value editors and their associated controls in various categories. The "Type" column indicates whether the editor implements `IPropertyEditor`, derives from `UITypeEditor`, or both. Some editors listed have no controls associated with them.

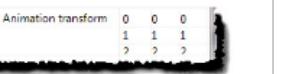
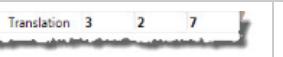
Single Value Editors

Value Editor	Value Editing Control	Appearance	Type	Description
BoolEditor	BoolControl (private class)		IPropertyEditor	Edit a Boolean value using a CheckBox. Used in several samples, such as ATF FSM Editor Sample .
BoundedFloat-Editor	BoundedFloat-Control		IPropertyEditor UITypeEditor	Edit a bounded float value. It displays a slider and TextBox in the GUI. Used in ATF DOM Tree Editor Sample .
BoundedInt-Editor	BoundedInt-Control		IPropertyEditor UITypeEditor	Edit a bounded int value. It displays a slider and TextBox in the GUI. Used in ATF DOM Tree Editor Sample .

ColorEditor	Uses control in System. Drawing. Design. ColorEditor. EditValue()		UITypeEditor	Derives from System.Drawing.Design.ColorEditor Edit color values. Clicking the button (▾) drops down color swatches on tabs to select another color. Used in ATF Timeline Editor Sample .
ColorPicker- Editor	ColorPicker		UITypeEditor	Edit colors, derived from System.Drawing.Design.ColorEditor . ColorPicker is similar to ColorEditor's value editor, displaying a color button and color components including alpha, shown in the top figure. Clicking the button (...) in the control displays an Adobe color picker clone dialog, shown in the bottom figure. In addition, the <code>Sce.Atf.Controls.PropertyEditing.</code> ColorPickerEditor derived from this value editor allows using color stored as an ARGB int. Used in ATF DOM Tree Editor Sample .
DateTime- Editor	Uses control in System. ComponentModel. Design. DateTimeEditor. EditValue()		UITypeEditor	Edit date and time values. Used in ATF DOM Tree Editor Sample .
EnumUIType- Editor	ListBox		UITypeEditor	Handle dynamic enumerations, where display names are associated with integer values. Used in several samples, such as ATF Simple DOM Editor Sample .
FileUri- Editor	OpenFileDialog		UITypeEditor	Use file open dialog box to allow the user to select a path as a string or URI. Used by ATF DOM Tree Editor Sample and in <code>PropertyEditingCommands</code> component.
FlagsUIType- Editor	CheckedListBox		UITypeEditor	Handle dynamic flag enumerations, selecting as many flags as desired in a list box. Used in ATF Timeline Editor Sample .
FolderBrowser - Dialog- UITypeEditor	FolderBrowser- Dialog		UITypeEditor	Select folder in folder browser dialog. Used in ATF DOM Tree Editor Sample .

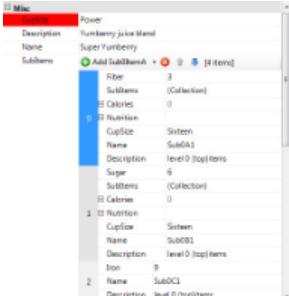
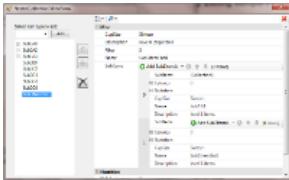
FolderUri-Editor	FolderBrowser-Dialog		UITypeEditor	Select folder path (stored as string or URI). Used in ATF DOM Tree Editor Sample .
LongEnum-Editor	LongEnum-Control-Wrapper (private class)		IPropertyEditor	Edit long enum-like attributes and values with an autocomplete ComboBox. The enum list can be updated programmatically. Used in ATF DOM Tree Editor Sample .
NumericEditor	NumericTextBox (private class)		IPropertyEditor	Text box for entering numeric values. Used in ATF DOM Tree Editor Sample .

Multiple Value Editors

Value Editor	Value Editing Control	Appearance	Type	Description
ArrayEditor	ArrayEditing-Control		IPropertyEditor	Edit the size, order, and values of an array whose elements are numeric. Used in ATF DOM Tree Editor Sample .
Collection-Editor	Collection-Editing-Control		IPropertyEditor	Editor that provides collection editing controls. Entering text selects the item with a matching first character. Used in ATF DOM Tree Editor Sample .
NumericMatrix-Editor	NumericMatrix-Control		IPropertyEditor UITypeEditor	Edit numeric matrices. Supports all numeric types except Decimal. Can be used with the standard .NET PropertyGrid, DataGridView, and DataGridView, as well as ATF's GridControl and PropertyGrid controls that the GridPropertyEditor and PropertyEditor components are built on. Used in ATF DOM Tree Editor Sample .
NumericTuple-Editor	NumericTuple-Control		IPropertyEditor UITypeEditor	Edit a numeric tuple (vector) of arbitrary dimensions. All numeric types except Decimal are supported. Can be used with the standard .NET PropertyGrid, DataGridView, and DataGridView, as well as ATF's GridControl and PropertyGrid controls that the GridPropertyEditor and PropertyEditor components are built on. Used in ATF DOM Tree Editor Sample .

UniformArray-Editor	UniformArray-TextBox		IPropertyEditor	Edit an array (of arbitrary length) of identical numeric values of the same type; the control displays only the single value. Used in ATF DOM Tree Editor Sample .
---------------------	----------------------	--	-----------------	--

Special Purpose

Value Editor	Value Editing Control	Appearance	Type	Description
Embedded-Collection-Editor	Collection - Control		IPropertyEditor	Embedded property editor for adding, removing, and editing items in a collection. Supports multiple item inserter functions, allowing you to insert child items of various types (typically derived from a common base class). Used in ATF Property Editing Sample . For a discussion and illustration of this editor in the sample, see Custom Property Editor Display .
Nested-Collection-Editor	Nested-Collection - EditorForm		UITypeEditor	Nested collections editor form. Used in ATF Property Editing Sample . For a discussion and illustration of this editor in the sample, see Custom Property Editor Display .

WPF

Value Editor	Value Editing Control	Appearance	Type	Description
MultiLine-TextValue-Editor	None			Class for editing a value with a multi-line TextBox and a SliderValueEditor.
SliderValue-Editor	None			Class for editing a value with a slider control.
ValueEditor	None			Base class for editing a value with a control. MultiLineTextValueEditor and SliderValueEditor derive from it.

Other Value Editing Controls

These controls are not used directly with value editors.

Value Editing Control	Description
PropertyEditingControl	Universal property editing control that can be embedded in complex property editing controls. It uses TypeConverter and UITypeEditor objects to provide a GUI for every kind of .NET property. It gets used by default if no custom value editing control is provided.
BoolInputControl	Edit a Boolean value.

Topics in this section

Links on this page to other topics

Value Converters

A value converter converts a property value between its internal data representation in a value editor and a form that appears in a value editing control.

For instance, a value converter for an integer value in a value editor would convert back and forth between an `int` value and a string representation of the integer that can be displayed in a value editing control. A color converter could convert color stored as an ARGB `int` value to or from color or string types.

Similarly to a value editor, a value converter is specified in the property descriptor with the `Converter` property. A given value editor usually has an associated value converter. The value converter must also work for the value editing control the value editor uses.

Topics

- [TypeConverter Class](#)
- [Defining Converters for Property Descriptors](#)
 - [Property Descriptor Constructors](#)
 - [XML Schema Annotations](#)
 - [IAnnotatedParams Interface](#)
- [ATF Value Converters](#)
- [Writing a Value Converter](#)

TypeConverter Class

The `System.ComponentModel.TypeConverter` class provides a unified way of converting types of values to other types. ATF value converters derive from `TypeConverter`. `TypeConverter` contains a large variety of methods, but it is not necessary to implement all of them in a value converter. The most useful methods indicate whether a conversion is possible to or from a given type in a given context — and perform such a conversion.

The `System.ComponentModel.TypeDescriptor` class provides information about the characteristics for a component, such as its attributes, properties, and events. This class provides the `GetConverter()` class method to get a `TypeConverter` for a given object. This `TypeConverter` may be useful for some converters, as shown for `IntColorConverter` in one of the examples in [Writing a Value Converter](#) below.

Defining Converters for Property Descriptors

Converters can be defined in a property descriptor by constructors or XML Schema annotations.

Property Descriptor Constructors

Property descriptors can be created with constructors, as described in [Constructing Property Descriptors](#).

For example, the [ATF DOM Tree Editor Sample](#) defines a property descriptor with the value editor `NumericTupleEditor` and the value converter `FloatArrayConverter` in `SchemaLoader.cs`:

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

XML Schema Annotations

Property descriptors can also be created from XML Schema annotations, as described in [Creating Property Descriptors with XML Schema Annotations](#).

For instance, the [ATF Timeline Editor Sample](#) specifies the color converter `IntColorConverter` for the color editor `ColorEditor` in an annotation defining a property descriptor in `timeline.xsd`:

```
<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" />
```

IAnnotatedParams Interface

Value converters, like value editors, can have parameters specified in annotations in an XML Schema.

If a value converter can have parameters in an XML Schema annotation, the value editor must implement `IAnnotatedParams`. This interface gets the parameters from the schema annotation and initializes the value converter for those parameters.

For example, `BoundedIntConverter`, which does conversion for the `BoundedIntEditor` value editor, can have parameters specifying the minimum and maximum values the converter handles. `BoundedIntConverter` implements `IAnnotatedParams:Initialize()` to obtain these limits and set its `m_min` and `m_max` fields:

```
public void Initialize( string[] parameters )
{
    if (parameters.Length < 2)
        throw new ArgumentException("Can't parse bounds");

    try
    {
        if (parameters[0].Length > 0)
            m_min = Int32.Parse(parameters[0]);
        if (parameters[1].Length > 0)
            m_max = Int32.Parse(parameters[1]);
    }
    catch
    {
        throw new ArgumentException("Can't parse bounds");
    }

    if (m_min.HasValue && m_max.HasValue && m_min.Value >= m_max.Value)
        throw new ArgumentException("Max must be > min");
}
```

For a description of how value editors use the `IAnnotatedParams` interface, see [IAnnotatedParams Interface](#).

ATF Value Converters

This table describes value converters associated with value editors.

Value Editor Class	Value Converter Class	Implements IAnnotatedParams	Description
<code>BoundedFloatEditor</code>	<code>BoundedFloatConverter</code>	Yes	Convert <code>float</code> values to a specified range. First argument, if not empty, is the minimum; second, if not empty, is the maximum.
<code>BoundedIntEditor</code>	<code>BoundedIntConverter</code>	Yes	Convert parsed <code>int</code> values to a specified range. First argument, if not empty, is the minimum; second, if not empty, is the maximum.
<code>ColorEditor</code>	<code>IntColorConverter</code>	No	Convert color stored as an ARGB <code>int</code> to or from <code>Color</code> or <code>string</code> types. Used in ATF Timeline Editor Sample .

EnumUITypeEditor	EnumTypeConverter ExclusiveEnumTypeConverter	Yes	Value converters for use with enum editors. Convert integers to strings. In ATF property editors. The user can enter any string, even if it doesn't match the list of names. Used in several samples, such as ATF Simple DOM Editor Sample .
FlagsUITypeEditor	FlagsTypeConverter	Yes	Convert int flags to or from a string.
Various	FloatArrayConverter	No	Convert float[] to or from a string of the form "c1,c2,...,cN". Used in ATF DOM Tree Editor Sample .
Various	ReadOnlyConverter	No	Converter for values that are not directly editable by the user. Used in ATF Timeline Editor Sample .
Various	UniformFloatArrayConverter	No	Convert a uniform float[] to or from a string of the form "c1,c2,...,cN". This is useful for uniform scale vectors, for example.

Writing a Value Converter

Consider what parameters the converter needs, based on the value editor it serves. You can provide these parameters in an XML Schema and implement [IAnnotatedParams](#). Or you can provide these parameters in the converter's constructor. You can also do both. For instance, [EnumTypeConverter](#) provides several constructors with and without parameters, in addition to implementing [IAnnotatedParams](#):

```
public EnumTypeConverter();
public EnumTypeConverter(string[] names);
public EnumTypeConverter(string[] names, int[] values);
```

The first constructor is used when the enumeration data is provided in the XML Schema. Enumeration data can also be passed to the converter in the constructors' parameters.

If the converter implements [IAnnotatedParams](#), implement [IAnnotatedParams.Initialize\(\)](#) to get the data from the XML Schema and convert it to the right data type, as shown in [IAnnotatedParams Interface](#).

Implement at least one conversion method in [System.ComponentModel.TypeConverter](#). For example, [BoundedFloatConverter](#) implements only this method (in addition to [IAnnotatedParams.Initialize\(\)](#)):

```
public override object ConvertFrom(ITypeDescriptorContext context, CultureInfo culture, object value)
{
    float floatValue = (float) base.ConvertFrom(context, culture, value);
    if (m_min.HasValue)
        floatValue = Math.Max(floatValue, m_min.Value);
    if (m_max.HasValue)
        floatValue = Math.Min(floatValue, m_max.Value);
    return floatValue;
}
```

The method converts the value and then checks it against the bounds. This converter is one way: from a string representing a floating point number to its float value.

By contrast, [IntColorConverter](#) implements these methods: [CanConvertFrom\(\)](#), [CanConvertTo\(\)](#), [ConvertFrom\(\)](#), and [ConvertTo\(\)](#). Here is [CanConvertTo\(\)](#):

```
public override bool CanConvertTo(ITypeDescriptorContext context, Type destType)
{
    return (destType == typeof(string) ||
            destType == typeof(Color));
}
```

This particular method is independent of context. Here is the corresponding [ConvertTo\(\)](#) method:

```
public override object ConvertTo(ITypeDescriptorContext context,
    CultureInfo culture,
    object value,
    Type destType)
{
    if (value == null)
        return null;

    if (value is Color)
        value = ((Color)value).ToArgb();

    if (value is int && destType == typeof(string))
    {
        // ARGB int -> string
        Color color = Color.FromArgb((int) value);
        TypeConverter colorConverter = TypeDescriptor.GetConverter(typeof(Color));
        return colorConverter.ConvertTo(context, culture, color, destType);
    }
    else if (value is int && destType == typeof(Color))
    {
        // ARGB int -> Color
        return Color.FromArgb((int) value);
    }

    return base.ConvertTo(context, culture, value, destType);
}
```

Note that this method calls `TypeConverter.GetConverter()` to get a converter from an ARGB int to a string, which uses the context. It also calls the base method `TypeConverter.ConvertTo()` to complete the conversion in some cases.

Topics in this section

Links on this page to other topics

[ATF DOM Tree Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [Property Descriptors](#), [Value Editors](#)

Property Editor Components

Several editors are MEF components. They are built on top of other controls to provide their capabilities.

Property Editor Containers

Several components encapsulate value editing controls. Many of the samples, such as [ATF Timeline Editor Sample](#), use both of these components, described below, to show different views of properties.

GridPropertyEditor Component

The `GridPropertyEditor` component displays items with properties in a spreadsheet type control, allowing you to see the properties of all selected items. On the other hand, you can't see categories or child properties with this control, because there is only one line per property.

Topics

- [Property Editor Containers](#)
- [GridPropertyEditor Component](#)
- [PropertyEditor Component](#)
- [PropertyView Class](#)
- [CurveEditor Component](#)
- [ListViewEditor Component](#)

Like the [ListViewEditor Component](#), `GridPropertyEditor` provides a `Configure()` method to set up the component and can be overridden to configure the control differently, such as with a different kind of grid control:

```
protected virtual void Configure(out GridControl gridControl, out ControlInfo controlInfo)
{
    gridControl = new GridControl();
    controlInfo = new ControlInfo(
        "Grid Property Editor".Localize(),
        "Edits selected object properties".Localize(),
        StandardControlGroup.Bottom);
}
```

`GridPropertyEditor` is built on top of the `GridControl` class, which is a wrapper for the spreadsheet-style `GridView` control class, combining it with a toolbar. You can use `GridControl` as a replacement for the .NET `System.Windows.Forms.DataGridView` control.

`GridView` is a spreadsheet-like control for displaying properties of many objects simultaneously. Only properties that are in common with all selected objects are displayed.

`GridView` is the real work horse of the `GridPropertyEditor` component, providing most of its functionality. `GridView` derives from [PropertyView](#), which also provides capabilities.

PropertyEditor Component

The `PropertyEditor` component displays items with properties in a two-column control, so you can see the properties of only one selected item. On the other hand, you can see categories and child properties with this control, because it is a multi-line control.

Like [ListViewEditor](#) and `GridPropertyEditor`, `PropertyEditor` provides a `Configure()` method to set up the component and can be overridden to configure the control differently, such as with a different kind of property grid control:

```

protected virtual void Configure(out PropertyGrid propertyGrid, out ControlInfo controlInfo)
{
    propertyGrid = new PropertyGrid();
    controlInfo = new ControlInfo(
        "Property Editor".Localize(),
        "Edits selected object properties".Localize(),
        StandardControlGroup.Right);
}

```

PropertyEditor is built on top of PropertyGrid, which is a wrapper for the two-column PropertyGridView control class, combining it with a toolbar. You can use PropertyGrid as a replacement for the .NET System.Windows.Forms.PropertyGrid control.

PropertyGridView is a control for displaying properties in a two column grid, with property names on the left and property values on the right. Properties with associated IPropertyEditor instances can embed controls into the right column, while all other properties are edited in a standard .NET way with a PropertyEditingControl.

PropertyEditingControl is a universal property editing control that can be embedded in complex property editing controls. It uses TypeConverter and UITypeEditor objects to provide a GUI for every kind of .NET property. It is used by default if no custom value editing control is provided.

PropertyGridView does most of the work of the PropertyEditor component. PropertyGridView derives from PropertyView, which also provides capabilities.

PropertyView Class

The GridView and PropertyGridView classes are the underlying controls that provide most of the functionality of the GridPropertyEditor and PropertyEditor components. Both derive from the abstract class PropertyView, which also provides fundamental capabilities.

PropertyView is a class for embedding complex property editing controls. It provides formats, fonts, data binding, property sorting, persistent settings of property editor layouts, and category/property information.

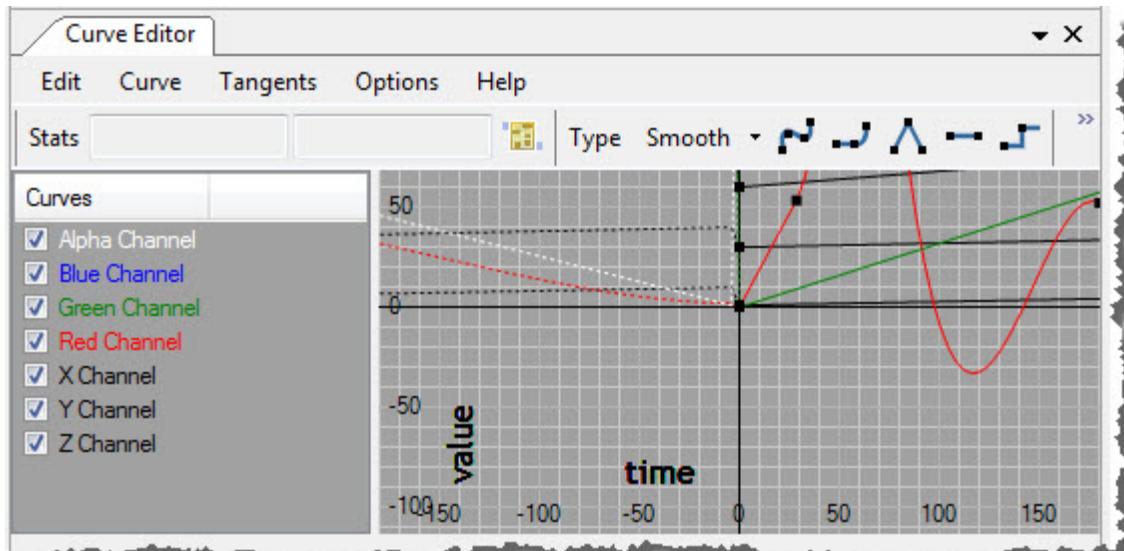
PropertyView helps handle the selection of items with properties in the application, tracking the active context. In response to selection events, PropertyView builds property descriptor lists of the common properties of selected items. It gets the value editor and value editing control for a property descriptor in the list of descriptors.

Property changes occur in a ITransactionContext in PropertyView, so that property changes can be undone and redone.

CurveEditor Component

You can use this component to edit curves. It integrates CurveEditingControl with ControlHostService and adds event handling. You can easily use some other curve editing control with CurveEditor.

For an example of using it, see the ATF DOM Tree Editor Sample, whose curve editing window is shown in this illustration:



ListViewEditor Component

ListViewEditor serves as a base class for list editors. It is not abstract, so it can be used as a generic list editor.

It creates a System.Windows.Forms.ListView control as its list view. It also creates a ListViewAdapter to adapt the ListView to an IListView interface, which enumerates objects that can be used in each row of the ListView.

Like the GridPropertyEditor and PropertyEditor components, its Configure() method sets up the component and can be overridden to configure the list differently, such as with a different kind of list control or adapter:

```
protected virtual void Configure(
    out ListView listView,
    out ListViewAdapter listViewAdapter)
{
    listView = new ListView();
    listView.SmallImageList = ResourceUtil.GetImageList16();
    listView.LargeImageList = ResourceUtil.GetImageList32();

    listViewAdapter = new ListViewAdapter(listView);
}
```

ListViewEditor offers a variety of other services to the control, such as setting up event handling and persistence for the list layout.

Topics in this section

Links on this page to other topics

[ATF DOM Tree Editor Sample](#), [ATF Timeline Editor Sample](#), [Authoring Tools Framework](#)

Implementing a Property Editor

You can implement a property editor in your application by following these steps:

1. Define the data whose properties you want to examine.
2. Set up a context for property editing.
3. Create property descriptors for the data's properties/attributes.
4. Use existing or create new value editors, value converters, and value editing controls.
5. Use existing or create new property editing components to display properties of selected items.
6. Add property editing components to the application's MEF type catalog and initialize them.

Define Application Data and Properties

You can describe data in your application many ways. If application data is defined in an ATF DOM, you can create property editors in a fairly straightforward fashion.

Topics

- [Define Application Data and Properties](#)
- [Set up Property Editing Context](#)
- [Create Property Descriptors Describing Data Properties](#)
- [Using Value Editors, Value Converters, and Value Editing Controls](#)
- [Using Property Editor Components](#)
- [Add Property Editing Components to MEF Catalog and Initialize](#)

You can also define your own non-DOM data and examine it with property editors, which is what the [ATF Property Editing Sample](#) does. For a detailed description of how this sample works, see [Property Editing Programming Discussion](#).

The rest of this section discusses how to implement property editors for application data in an ATF DOM, using the steps above.

When using an ATF DOM, you create a type definition file describing all the data types and their attributes/properties. You can do this in a variety of ways, but ATF provides the best support for property editors (and other things) when you define the data using the XML Schema Definition (XSD) language.

For details on using the ATF DOM for your data model, see the [ATF Programmer's Guide: Document Object Model \(DOM\)](#), which you can download at [ATF Documentation](#).

Set up Property Editing Context

Your application provides ways to edit its data's properties, and must provide a context in which to do this. There are several key interfaces the context should implement, including [ISelectionContext](#), which provides properties and methods to get the selected items and last selected item, change the selection, and so forth. It also provides events for selection changes. You may also want to implement [ITransactionContext](#), so that property editing changes can be undone and redone as part of transactions.

Several samples provide [EditingContext](#) classes, and some of these derive from or directly use `Sce.Atf.Dom.EditingContext`, which implements both [HistoryContext](#) and [ISelectionContext](#). [HistoryContext](#) derives from [TransactionContext](#), which implements [ITransactionContext](#).

Handling the selection changing events in [ISelectionContext](#) hooks into the event handling system that ultimately generates a list of property descriptors for the selected items' properties. Property editors use this list to display the common properties of selected items. If you provide the context, this processing happens for you.

For more information on using a context for property editing, see [Selection Property Editing Context](#). For general information on contexts, see [ATF Contexts](#).

Create Property Descriptors Describing Data Properties

Property descriptors can be created in two ways:

- With constructors, as described in [Constructing Property Descriptors](#).
- From XML Schema annotations, as shown in [Creating Property Descriptors with XML Schema Annotations](#).

The samples demonstrate both approaches. For instance, the [ATF FSM Editor Sample](#) shows constructing property descriptors in its schema

loader. The [ATF Timeline Editor Sample](#) shows specifying property descriptors in XML Schema annotations.

Using Value Editors, Value Converters, and Value Editing Controls

Property descriptors define which value editors and value converters are used for each data type. Each value editor specifies which value editing control users interact with to change the property's value.

If you are using existing ATF classes for the value editors, value converters, and value editing controls, you don't need to do anything else. For a list of existing value editors, see [Value Editors and Value Editing Controls](#). For a list of existing ATF value converters, see [ATF Value Converters](#). For existing value editing controls, see [Value Editors and Value Editing Controls](#).

If your data does not fit any of these existing value processing classes, you need to define your own:

- For information on creating value editors, see [Value Editors](#).
- For information on creating value converters, see [Value Converters](#).
- For information on creating controls, see [Controls in ATF](#).

Using Property Editor Components

The property editor components serve as containers for property editing controls:

- `GridPropertyEditor` displays items with properties in a spreadsheet type control, allowing you to see the properties of all selected items.
- `PropertyEditor` displays items with properties in a two-column control, so you can see the properties of only one selected item, with its categories or child properties.

These components automatically create the property editing controls, embed them, and display the values of currently selected items.

You can use these components as is, or customize them to suit your application. For an example of customizing them in the [ATF Property Editing Sample](#), see [Customizing Property Editor Components](#).

For general information on these components, see [Property Editor Containers](#).

Add Property Editing Components to MEF Catalog and Initialize

Your application needs to incorporate the Managed Extensibility Framework (MEF) to use the property editor components. For information on MEF, see [MEF with ATF](#).

You must add the property editor components to your MEF catalog, as in this code from the `Main()` function of [ATF Timeline Editor Sample](#):

```
TypeCatalog catalog = new TypeCatalog()
...
typeof(PropertyEditor),                      // property grid for editing selected objects
typeof(GridPropertyEditor),                  // grid control for editing selected objects
typeof(PropertyEditingCommands),             // commands for PropertyEditor and GridPropertyEditor
...
```

This sample also adds the `PropertyEditingCommands` component to provide property editing commands that can be used inside `PropertyEditor` and `GridPropertyEditor` from context menus. For more information on `PropertyEditingCommands`, see its entry in the table in [Using Standard Command Components](#).

After the components are added, call `MefUtil.InitializeAll()` to initialize them.

Before you set up the MEF catalog, you should also make this call:

```
DomNodeType.BaseOfTypeAdapterCreator<CustomTypeDescriptorNodeAdapter>();
```

Doing this guarantees that the properties you set up in the property descriptors are the ones that appear in the property editors. If this statement were removed, the properties visible in a property editor for an item would be properties of the selected object and classes it derives from — not the properties you provide in your property descriptor.

For more information, see [Metadata-Driven Property Editing](#).

Links on this page to other topics

[ATF Contexts](#), [ATF Documentation](#), [ATF FSM Editor Sample](#), [ATF Property Editing Sample](#), [ATF Timeline Editor Sample](#), [Controls in ATF](#), [MEF with ATF](#), [Property Descriptors](#), [Property Editing Programming Discussion](#), [Property Editor Components](#), [Selection Property Editing Context](#), [Using Standard Command Components](#), [Value Converters](#), [Value Editors](#), [Value Editors and Value Editing Controls](#), [WinForms Application](#)

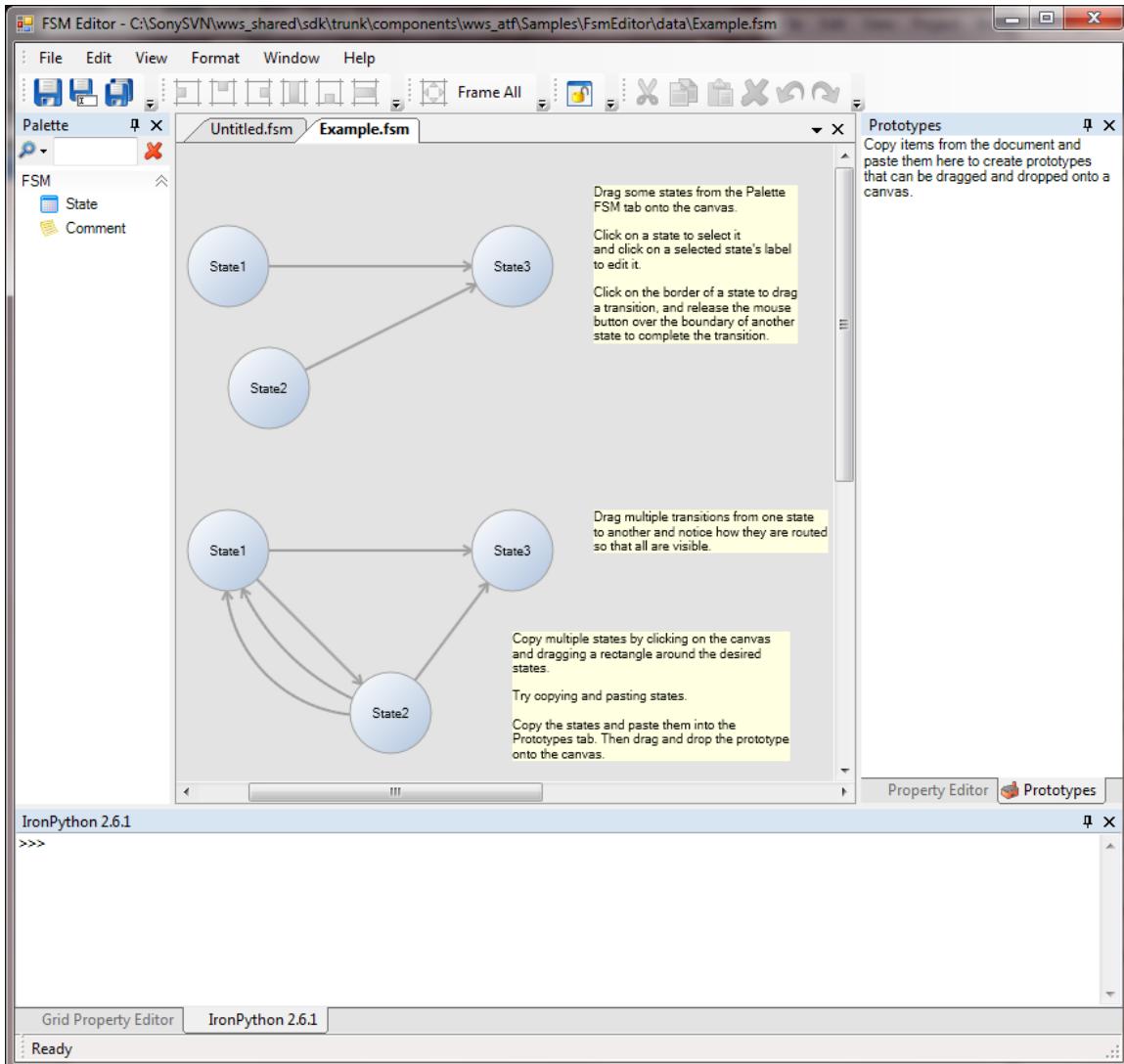
Graphs in ATF

A graph is a set of nodes with connecting edges. Graph editing applications are demonstrated in the [ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), and [ATF State Chart Editor Sample](#).

- [What is a Graph in ATF?](#): General description of graphs.
 - [Graph Data Model](#): Data model for graphs using the ATF DOM.
 - [Types of Graphs](#): Types of graphs supported in ATF.
 - [ATF Graph Interfaces](#): The main interface `IGraph` and other interfaces used in graphs.
 - [General Graph Support](#): Description of general graph support.
 - [Circuit Graph Support](#): How to use circuit graphs, which ATF provides the most support for.
 - [Statechart Graph Support](#): Support for statechart type graphs.
-

What is a Graph in ATF?

A graph is a collection of nodes with connecting edges. For example, this figure shows the [ATF FSM Editor Sample](#) application window, which contains a simple state machine. In this graph, states are the nodes and transitions are the connecting edges:



ATF supports a variety of graphs and demonstrates them in samples: basic graphs in [ATF FSM Editor Sample](#), statecharts in [ATF State Chart Editor Sample](#), and circuits in [ATF Circuit Editor Sample](#). Graph support resides in the `Sce.ATf.Controls.Adaptable.Graphs` namespace, split across two assemblies:

- `Atf.Gui`: Graph element classes and interfaces that are UI-platform-agnostic (such as Windows Forms or WPF).
- `Atf.Gui.WinForms`: Classes for working with graph elements, including adapters, renderers, documents, and validators, that are dependent on the Windows Forms UI platform. (There are no WPF counterparts yet).

Each of these assemblies has a folder for classes dedicated to circuit graphs, which have especially rich support in ATF:

- `wws_atf\Framework\Atf.Gui\Controls\Adaptable\Graphs\Circuit`
- `wws_atf\Framework\Atf.Gui.WinForms\Controls\Adaptable\Graphs\Circuit`

You can also see these folders in Visual Studio's Solution Explorer pane under these assemblies.

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF State Chart Editor Sample](#)

Graph Data Model

ATF's graph interfaces are not dependent on the ATF DOM. However, virtually all concrete types in the graph namespace use the ATF DOM for their data model and undo/redo support, so practically speaking, you need to use the ATF DOM to fully utilize ATF's default DOM-based graph implementation. Nevertheless, all interfaces and generic types (such as renderers) in the ATF graph are not DOM-aware, so you can probably go a long way just by using these generic types and interfaces, if it is too much work to support the required graph interfaces in your custom graph data model. This may be the case if your main goal is to visualize your data model using ATF's graph rendering engine, without a need for complicated and interactive graph editing. However, in almost all cases, it is far easier and productive to adopt the ATF DOM directly.

If you use an XML Schema to define the data model, you can use the DomGen utility to generate a `Schema` class containing metadata classes for the types.

The next step is to define your own DOM adapters to bind DOM data to the graph model at runtime. Typically this is an easy step, because you can derive from ATF's abstract DOM adapter classes and override the adapter's properties and methods as needed. In particular, override the properties that provide application-specific type metadata, typically using the metadata classes in the `Schema` class. For details on how the [ATF Circuit Editor Sample](#) derives from existing DOM adapters, see [DOM Adapters](#). There is an extensive set of DOM adapters for circuit graphs; for a list, see [Circuit DOM Adapters](#).

For every complex type defined in the schema, there is a corresponding `DomNodeType` that is created by the schema type loader class, which is usually named `SchemaLoader` and derives from `XmlSchemaTypeLoader`. You need to define your DOM adapters as extensions to each DOM type after the schema is loaded, but before attempting to load any application document. DOM adapters are usually defined on the data types in `SchemaLoader`.

For example, for a circuit graph, you could create DOM adapters for schema types of pin, element (an item with pins such as an OR gate), and wire. You can also define adapters for composite types, such as circuit and group.

After setting up these visual elements that form the building blocks of a graph, the next step is to attach higher level DOM adapters that coordinate the editing, interaction, rendering, and validating of these elements. For example, some adapters provide an editing context for editing circuit items. For a description, see [Circuit Context Classes](#). For an example of implementing contexts in [ATF Circuit Editor Sample](#), see [Context Classes](#). Other adapters can validate the graph as changes are made. For an example of circuit validation, see [CircuitValidator Class](#).

In the ATF DOM, each graph element is represented by a `DomNode`, and the graph is represented as a tree of `DomNodes`. The tree can be stored as a part of the application's `DomDocument`. The tree can be saved as XML using the `DomXmlWriter` class and read back with `DomXmlReader` for easy data serialization.

For more information on the ATF DOM, see the ATF Programmer's Guide: Document Object Model (DOM), downloadable from [ATF Documentation](#).

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF Documentation](#), [Circuit Editor Programming Discussion](#), [Circuit Graph Support](#)

Types of Graphs

ATF supports general purpose and specialized graphs, such as circuits and statecharts. This section briefly describes support for the different kinds of graphs. ATF graph support is described in further detail in other sections:

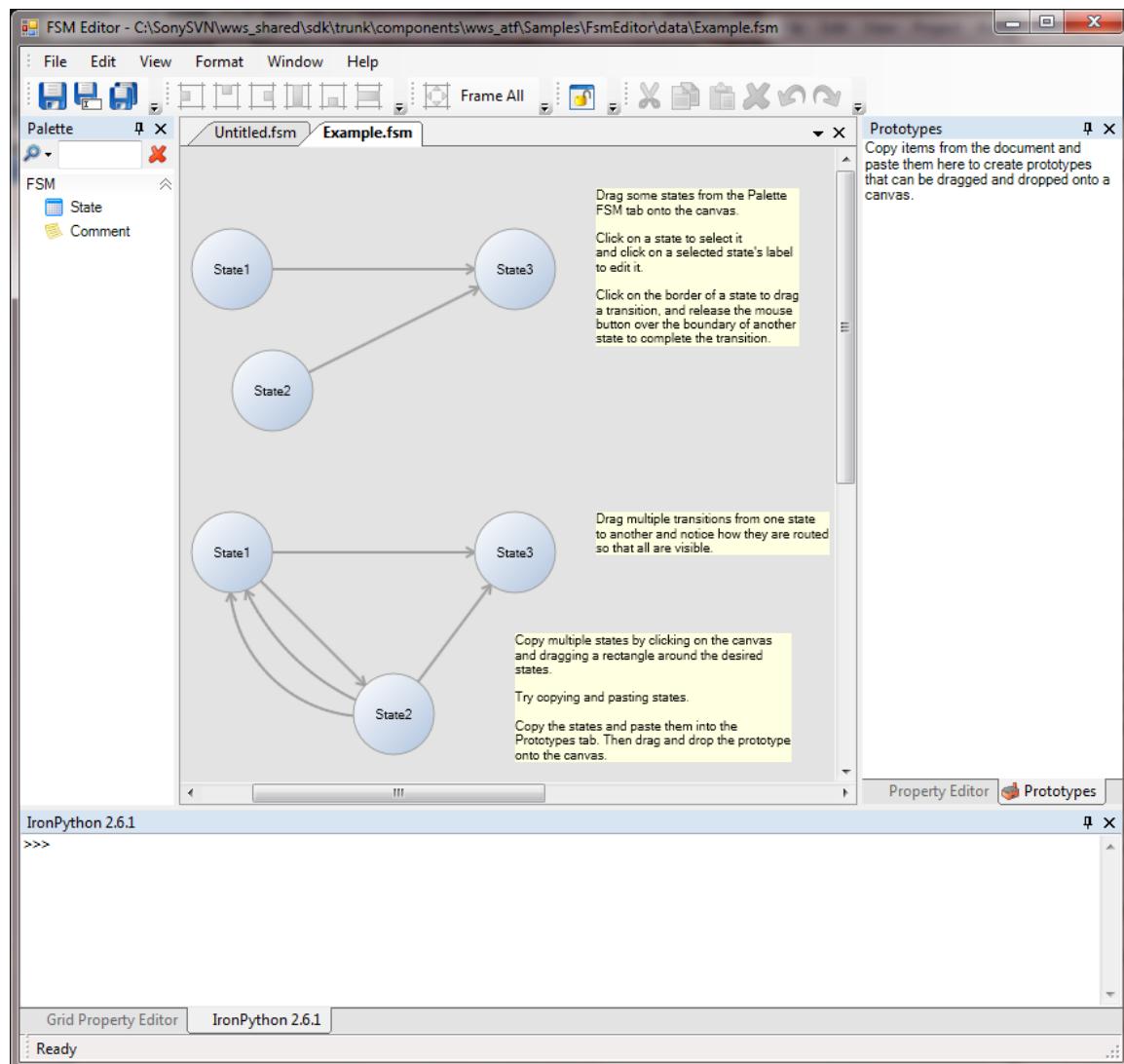
- General Graph Support
- Circuit Graph Support
- Statechart Graph Support

Topics

- General Purpose Graphs
- Circuit Graphs
- Statechart Graphs

General Purpose Graphs

The simplest graph has nodes and connecting edges, as in the [ATF FSM Editor Sample](#), representing a simple state machine. In this graph, states are the nodes and transitions are the connecting edges:



In ATF, the general interface for a graph is

```
IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>
```

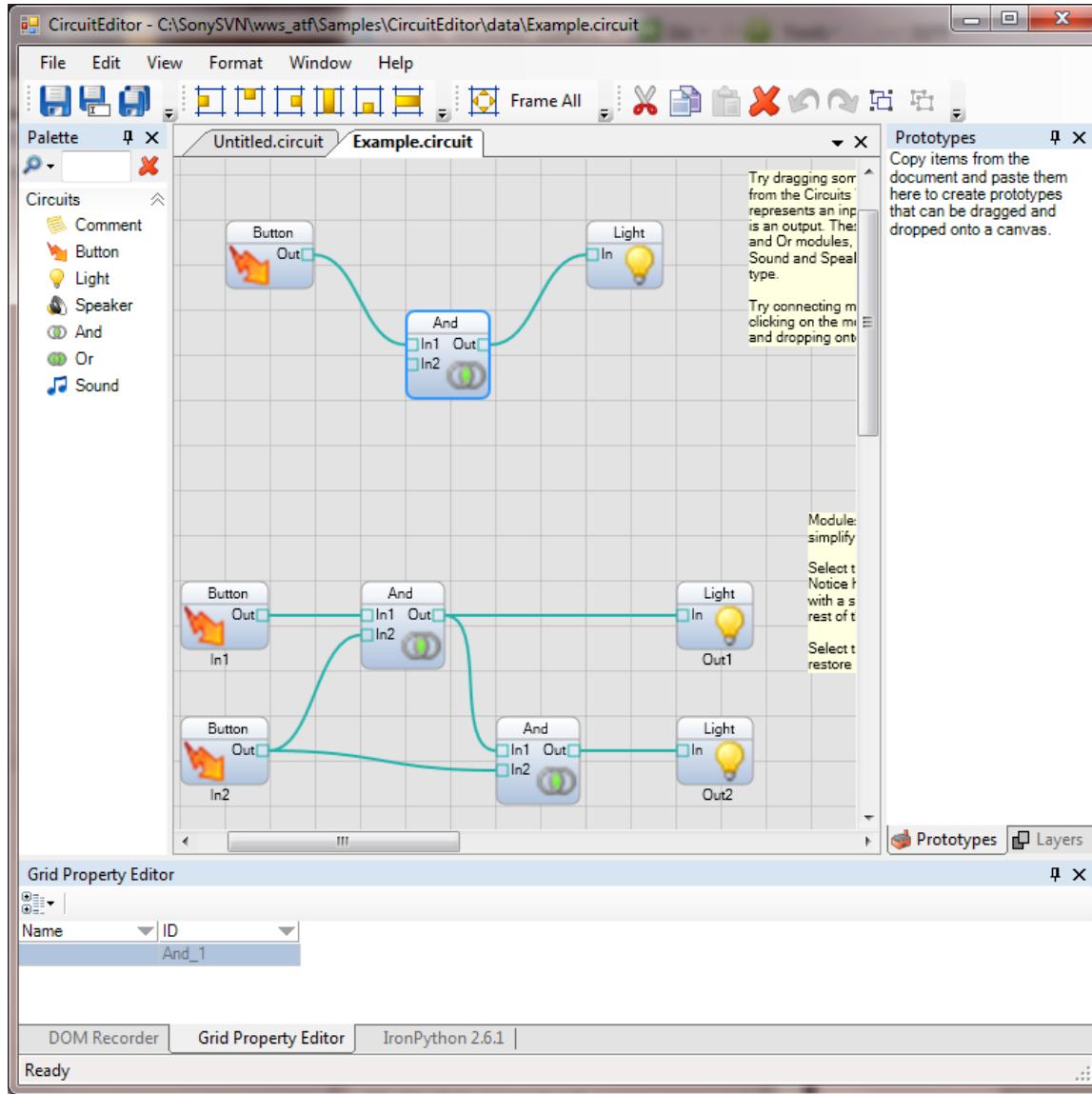
These parameters are:

- `IGraphNode`: Interface for a node in a graph.
- `IGraphEdge<IGraphNode, IEdgeRoute>`: Interface for edges in a graph.
- `IEdgeRoute`: Interface for edge routes, which act as sources and destinations for graph edges.

For more information on the general purpose `IGraph` interface, see [IGraph and Related Interfaces](#).

Circuit Graphs

ATF provides extensive support for circuit graphs. In a circuit, nodes are circuit elements, such as OR gates, that define input and output "pins", and edges are connecting wires connecting input pins to output pins. The [ATF Circuit Editor Sample](#) exercises the ATF circuit graph classes, and is illustrated in this figure:



The `Circuit` class specializes the `IGraph` interface:

```
public abstract class Circuit : DomNodeAdapter, IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram, ICircuitContainer
```

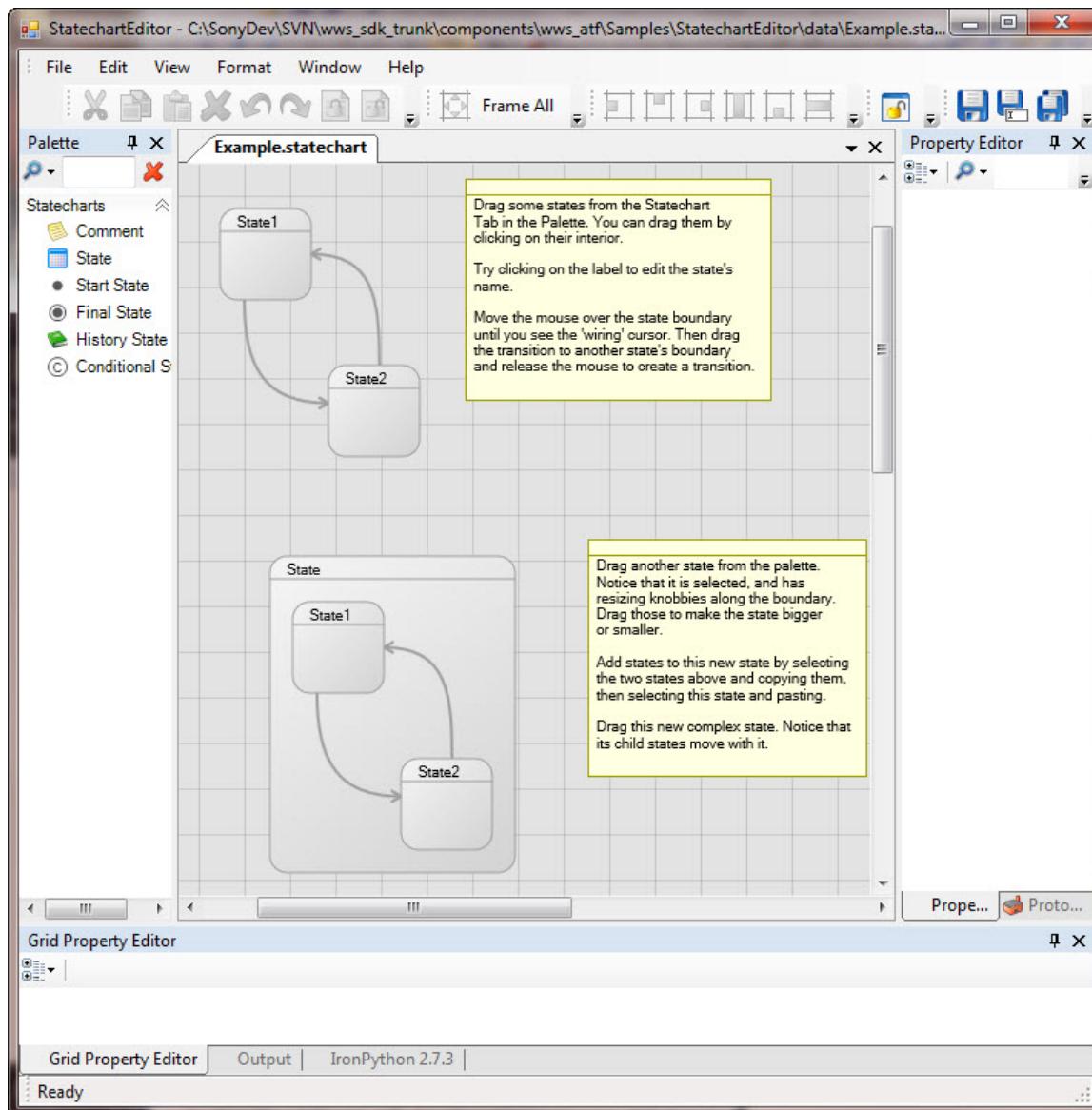
The parameters in `IGraph<Element, Wire, ICircuitPin>` are:

- `Element`: Adapts `DomNode` to an element, a circuit node with pins.
- `Wire`: Adapts `DomNode` to a connection in a circuit.
- `ICircuitPin`: Interface for pins, which are the sources and destinations (contact points) for wires between circuit elements.

Each of the interface parameters here derive from the parameters in the general `IGraph` interface. This is permitted, because the type parameters in `IGraph` are covariant.

Statechart Graphs

Statecharts, also known as state transition diagrams, show states and transitions, and allow embedding a child machine inside a state. The ATF State Chart Editor Sample employs statecharts, as in this figure:



Statecharts in ATF specialize the `IGraph` interface to this:

```
IGraph<IState, IGraphEdge<IState, BoundaryRoute>, BoundaryRoute>
```

The statechart unique items are:

`IState`: Interface for states in state-transition diagrams.

`BoundaryRoute`: Transition between states.

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF State Chart Editor Sample](#), [Circuit Graph Support](#), [General Graph Support](#), [Statechart Graph Support](#)

ATF Graph Interfaces

Graph support in ATF includes a variety of interfaces for general graphs, circuits, and statecharts. These interfaces reside in the `Sce.Atf.Controls.Adaptable.Graphs` namespace and `Atf.Gui` assembly.

IGraph and Related Interfaces

In ATF, the interface for a graph is `IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>`:

```
public interface IGraph<out TNode, out TEdge, out TEdgeRoute>
    where TNode : class, IGraphNode
    where TEdge : class, IGraphEdge<TNode, TEdgeRoute>
    where TEdgeRoute : class, IEdgeRoute
```

Topics

- [IGraph and Related Interfaces](#)
- [IGraphNode Interface](#)
- [IGraphEdge Interface](#)
- [IEdgeRoute Interface](#)
- [Other Graph Interfaces](#)

Graph objects, such as circuits, implement `IGraph`.

`IGraph` references several other interfaces:

- `IGraphNode`: Interface for a node in a graph; nodes are connected by edges.
- `IGraphEdge<IGraphNode, IEdgeRoute>`: Interface for routed edges in a graph. Routed edges connect nodes and have a defined source and destination route from and to the nodes.
- `IEdgeRoute`: Interface for edge routes, which contain information on source and destination nodes for graph edges.

Note that all the type parameters in `IGraph` are covariant, so that more derived types can be used than the ones specified. (As of C# 4.0, generic interfaces support covariance for type parameters marked with the `out` modifier. This modifier ensures that the marked type parameter is only used in an output position, for example, the return type of a method.) Other types of graphs use `IGraph` interfaces with parameters that are derived from the parameters specified in `IGraph`'s declaration, as shown in [Types of Graphs](#). For instance, the `Sce.Atf.Controls.Adaptable.Graphs.Circuit` class for circuit graphs implements `IGraph` this way:

```
public abstract class Circuit : DomNodeAdapter, IGraph<Element, Wire, ICircuitPin>, ...
```

In this circuit interface, `Element` is an element like an AND gate, `Wire` is a wire connecting elements, and `ICircuitPin` is a pin on an element. These classes and interface derive from or implement the interfaces in the general `IGraph` interface:

- `Element` implements `ICircuitElement` which implements `IGraphNode`.
- `Wire` implements `IGraphEdge<Element, ICircuitPin>`.
- `ICircuitPin` implements `IEdgeRoute`.

The `IGraph` interface contains the following properties:

- `Nodes`: Get the nodes in the graph as `IEnumerable<IGraphNode>` — or something derived from that, because the parameter is covariant.
- `Edges`: Get the edges in the graph as `IEnumerable<IGraphEdge<IGraphNode, IEdgeRoute>>` — or something derived from that.

`IGraph` also offers these extension methods:

- `GetEdges()`: Get the edges that connect to the given node.
- `GetNodes()`: Get the nodes that connect to the given node.
- `GetInputEdges()`: Get the "incoming" edges that connect to the given node. These are the edges whose `ToNode` matches the given node. For a description of the `ToNode` property, see [IGraphEdge Interface](#).
- `GetOutputEdges()`: Get the "outgoing" edges that connect to the given node. These are the edges whose `FromNode` matches the given node. For a description of the `FromNode` property, see [IGraphEdge Interface](#).
- `GetInputNodes()`: Get the "incoming" nodes that connect to the given node by an incoming edge. An incoming edge is an edge

- whose `ToNode` matches the given node.
- `GetOutputNodes()`: Get the "outgoing" nodes that connect to the given node by an outgoing edge. An outgoing edge is an edge whose `FromNode` matches the given node.

IGraphNode Interface

`IGraphNode` is the interface for a node in a graph. Nodes take a variety of forms, depending on the type of graph. In a circuit, for instance, a node is an element that contains pins that are connected to other pins on elements. `IGraphNode` contains these properties:

- `Name`: Get the node name.
- `Bounds`: Get the bounding rectangle for the node in world space (or local space if a hierarchy is involved, as with sub-circuits).

IGraphEdge Interface

`IGraphEdge` is the interface for edges in a graph and has two forms. The first form is `IGraphEdge<IGraphNode>`:

```
public interface IGraphEdge<out TNode>
    where TNode : class, IGraphNode
```

This interface has the properties:

- `FromNode`: Get the edge's source node as an `IGraphNode` — or something derived from `IGraphNode`, because the parameter is covariant.
- `ToNode`: Get the edge's destination node as an `IGraphNode` — or something derived from that, because the parameter is covariant.

The second form is `IGraphEdge<out IGraphNode, out IEdgeRoute>`:

```
public interface IGraphEdge<out TNode, out TEdgeRoute> : IGraphEdge<TNode>
    where TNode : class, IGraphNode
    where TEdgeRoute : class, IEdgeRoute
```

This interface includes the properties:

- `FromRoute`: Get the `IEdgeRoute` for the source node.
- `ToRoute`: Get the `IEdgeRoute` for the destination node.

Note that this form of the interface derives from the first one, so it includes both the source and destination nodes as well as the nodes' `IEdgeRoute` properties.

IEdgeRoute Interface

`IEdgeRoute` indicates whether nodes allow multiple input or output edges with these Boolean properties:

- `AllowFanIn`: Get whether this node can accept multiple edges as inputs.
- `AllowFanOut`: Get whether this node can accept multiple edges as outputs.

Other Graph Interfaces

This table lists the other graph interfaces and their derivation. It includes Interfaces for general, circuit, and statechart graphs.

Interface	Derives from	Description
<code>ICircuitContainer</code>		Interface for container of circuit items. The container should support element addition, removal, and insertion. For details, see ICircuitContainer Interface .
<code>ICircuitElement</code>	<code>IGraphNode</code>	Interface for circuit elements, which have a <code>Type</code> property defining their pins and appearance.

<code>ICircuitElementType</code>		Interface for circuit element types, containing properties to define the appearance, inputs, and outputs of an element. For instance, the <code>Image</code> property gets an element image.
<code>ICircuitGroupPin<out TElement></code> where <code>TElement : class, ICircuitElement</code>	<code>ICircuitPin</code>	Interface for group pins, which are virtual pins that expose real pins of elements inside the group to other circuit elements outside the group. Contains properties to get pin information: connections, bounds, and <code>CircuitGroupPinInfo</code> to specify behavior or appearance of <code>ICircuitGroupPin</code> .
<code>ICircuitGroupType<TElement, TWire, TPin></code> where <code>TElement : class, IGraphNode</code> <code>where TWire : class, IGraphEdge<TElement, TPin></code> <code>where TPin : class, IEdgeRoute</code>	<code>IHierarchicalGraphNode<TElement, TWire, TPin>, ICircuitElementType</code>	Interface for hierarchical circuit group element types. For more information on this interface, see Group Class .
<code>ICircuitPin</code>	<code>IEdgeRoute</code>	Interface for pins, which are the sources and destinations for wires between circuit elements. Its properties get the name, type, and pin index.
<code>IComplexState<TNode, TEdge></code> where <code>TNode : class, IState</code> where <code>TEdge : class, IGraphEdge<TNode, BoundaryRoute></code>	<code>IHierarchicalGraphNode<TNode, TEdge, BoundaryRoute></code>	Interface for states in statechart diagrams that are non-pseudo-states. For more information, see Statechart Interfaces .
<code>IEdgeStyleProvider</code>		Provide information about an edge's shape and appearance in an <code>EdgeStyle</code> enumeration or <code>EdgeStyleData</code> . For more information, see EdgeStyleData Class .
<code>IEditableGraph<in TNode, TEdge, in TEdgeRoute></code> where <code>TNode : class, IGraphNode</code> where <code>TEdge : class, IGraphEdge<TNode, TEdgeRoute></code> where <code>TEdgeRoute : class, IEdgeRoute</code>		Interface for a graph that can be edited by a control. Contains methods to change node connections.
<code>IEditableGraphContainer<in TNode, TEdge, in TEdgeRoute></code> where <code>TNode : class, IGraphNode</code> where <code>TEdge : class, IGraphEdge<TNode, TEdgeRoute></code> where <code>TEdgeRoute : class, IEdgeRoute</code>	<code>IEditableGraph<TNode, TEdge, TEdgeRoute></code>	Interface for container of graph objects. Its methods allow moving items in and out of the container and resizing it. For information on its use, see CircuitEditingContext Class .
<code>IGraphAdapter<TNode, TEdge, TEdgeRoute></code> where <code>TNode : class, IGraphNode</code> where <code>TEdge : class, IGraphEdge<TNode, TEdgeRoute></code> where <code>TEdgeRoute : class, IEdgeRoute</code>		Interface for graph adapters, which manage rendering and picking for a graph in the adapted control. This is currently unused.

<code>IGraphNodeOptionsProvider</code>		Interface for <code>IGraphNode</code> that can provide additional options for controlling the display and other aspects of a graph node.
<code>IHierarchicalCircuitElementType<TElement, TWire, TPin></code> <code> where TElement : class, ICircuitElement</code> <code> where TWire : class, IGraphEdge<TElement, TPin></code> <code> where TPin : class, ICircuitPin</code>	<code>IHierarchicalGraphNode<TElement, TWire, TPin>, ICircuitElementType</code>	Interface for a hierarchical circuit element type. Currently unused.
<code>IHierarchicalGraphNode<out TNode, TEdge, TEdgeRoute></code> <code> where TNode : class, IGraphNode</code> <code> where TEdge : class, IGraphEdge<TNode, TEdgeRoute></code> <code> where TEdgeRoute : class, IEdgeRoute</code>	<code>IGraphNode</code>	Interface for hierarchical nodes in a graph, which contain sub-nodes. For more information on this interface, see Group Class .
<code>IState</code>	<code>IGraphNode</code>	Interface for states in state-transition diagrams, such as statecharts. For more information, see Statechart Interfaces .

Topics in this section

Links on this page to other topics

[Authoring Tools Framework](#), [Circuit Graph Support](#), [General Graph Support](#), [Statechart Graph Support](#), [Types of Graphs](#)

General Graph Support

This section covers classes that offer general graph support. It's split in two parts by these assemblies:

- `Atf.Gui`: Graph element classes and interfaces that are UI-platform-agnostic (such as Windows Forms or WPF).
- `Atf.Gui.WinForms`: Classes for working with graph elements, including adapters, renderers, documents, and validators that are dependent on Windows Forms.

Most of ATF's graph support classes are for specific graph types. For more information, see [Circuit Graph Support](#) and [Statechart Graph Support](#).

Atf.Gui Assembly Graph Support

The `Atf.Gui` assembly contains all the graph interfaces. For details on these, see [ATF Graph Interfaces](#).

Topics

- [Atf.Gui Assembly Graph Support](#)
- [D2dGraphRenderer Class](#)
- [GraphHitRecord Class](#)
- [EdgeStyleData Class](#)
- [Atf.Gui.WinForms Assembly Graph Support](#)
- [Adapter and Utility Classes](#)
- [Direct2D Classes](#)
- [Obsolete Classes](#)

It also provides DOM-based default interface implementations of the circuit model (`Element`, `Pin`, `Wire`, `Group`, and `Circuit`). For information on circuits, see [Circuit Graph Support](#).

The rest of this section discusses non-interface items in the assembly.

D2dGraphRenderer Class

`D2dGraphRenderer` is an abstract base class for graph renderers, which render and hit-test a graph. It derives from another abstract class, `DiagramRenderer` in the `Sce.Atf.Controls.Adaptable` namespace, which is the base class for diagram renderers that render and hit-test diagrams.

Several specific type of graph renderers derive from this class to render circuits and statecharts: `D2dCircuitRenderer` and `D2dStatechartRenderer`. Several control adapters require a `D2dGraphRenderer` in their constructors: `D2dGraphAdapter` and `D2dGraphEdgeEditAdapter`. For information on these adapters, see [Direct2D Adapters](#).

`D2dGraphRenderer` contains several abstract `Draw()` methods to draw nodes and edges, which are overridden in the renderers `D2dCircuitRenderer` and `D2dStatechartRenderer`. `Draw()` is called in the adapter `D2dGraphAdapter`'s `OnRender()` method for specific renderers.

Similarly, `D2dGraphRenderer` offers `Pick()` methods to find a node or edge hit by the given point. These are overridden in `D2dCircuitRenderer`, `D2dStatechartRenderer`, `D2dDigraphRenderer`, and `D2dSubCircuitRenderer`. `Pick()` is called in the adapter `D2dGraphAdapter` for specific renderers.

The `Print()` method uses a `Draw()` method to draw a graph in a printer-friendly way.

GraphHitRecord Class

`GraphHitRecord` is a generic class that specifies the node, edge, and in/out edge routes from a graph picking operation, and is returned by some `D2dGraphRenderer`.`Pick()` methods. Additionally, an `IGraphNode` or `IGraphEdge` may be specified as the item that was hit in the various `GraphHitRecord` constructor forms. `GraphHitRecord` has methods to retrieve the hit `IGraphNode` or `IGraphEdge`. `GraphHitRecord` also includes the hit path for renderers that support hierarchical picking, such as the circuit renderer, which needs to transverse up or down the hit path along the expanded group hierarchy.

EdgeStyleData Class

`EdgeStyleData` contains information on edge styles:

- `EdgeShape`: enumeration for the shape of an edge. It includes alternatives, such as a line, a Bezier curve or spline, or not drawn at all.
- `ShapeType`: Property to get or set the edge's `EdgeShape`.
- `Thickness`: Property to get or set line thickness.
- `EdgeData`: Property to get or set the edge data that represents the edge shape.

Atf.Gui.WinForms Assembly Graph Support

The `Atf.Gui.WinForms` assembly contains control adapters, utilities, and Direct2D handling classes that, one way or another, reference types in the Windows Forms API. For example, all control adapters operate on an adapted control that derives from the `System.Windows.Forms.Control` class.

Adapter and Utility Classes

Control Adapter Classes

Control adapters add capabilities to adaptable controls. For a discussion of how adapters are used on a `D2dAdaptableControl` to handle circuits in the [ATF Circuit Editor Sample](#), see [Circuit Document Display](#).

KeyboardGraphNavigator Class

`KeyboardGraphNavigator` is a `ControlAdapter` derived class that adapts a control for navigating a graph using the arrow keys. It subscribes the control to key press events. When a key is pressed, it finds the nearest node to the starting node in the desired direction using its `FindNearestElement()` method, and selects that node. It uses the standard Windows convention of using the Control key to toggle the selection of the given item, the Shift key to add the item to the selection, and otherwise to set the selection to the item.

If the graph has inputs and outputs on specific sides of the nodes, as for a circuit, consider using the `KeyboardIOGraphNavigator` class instead.

It is used by the [ATF FSM Editor Sample](#) and the [ATF State Chart Editor Sample](#).

KeyboardIOGraphNavigator Class

`KeyboardIOGraphNavigator` is a `ControlAdapter` that operates similarly to `KeyboardGraphNavigator`, except that it navigates an "input-output" graph that has inputs and outputs on specific sides of the nodes, as in a circuit graph. It changes selection using the arrow keys, with the Shift key adding to the selection.

It is used by the [ATF Circuit Editor Sample](#).

Utility Classes

GraphViewCommands Class

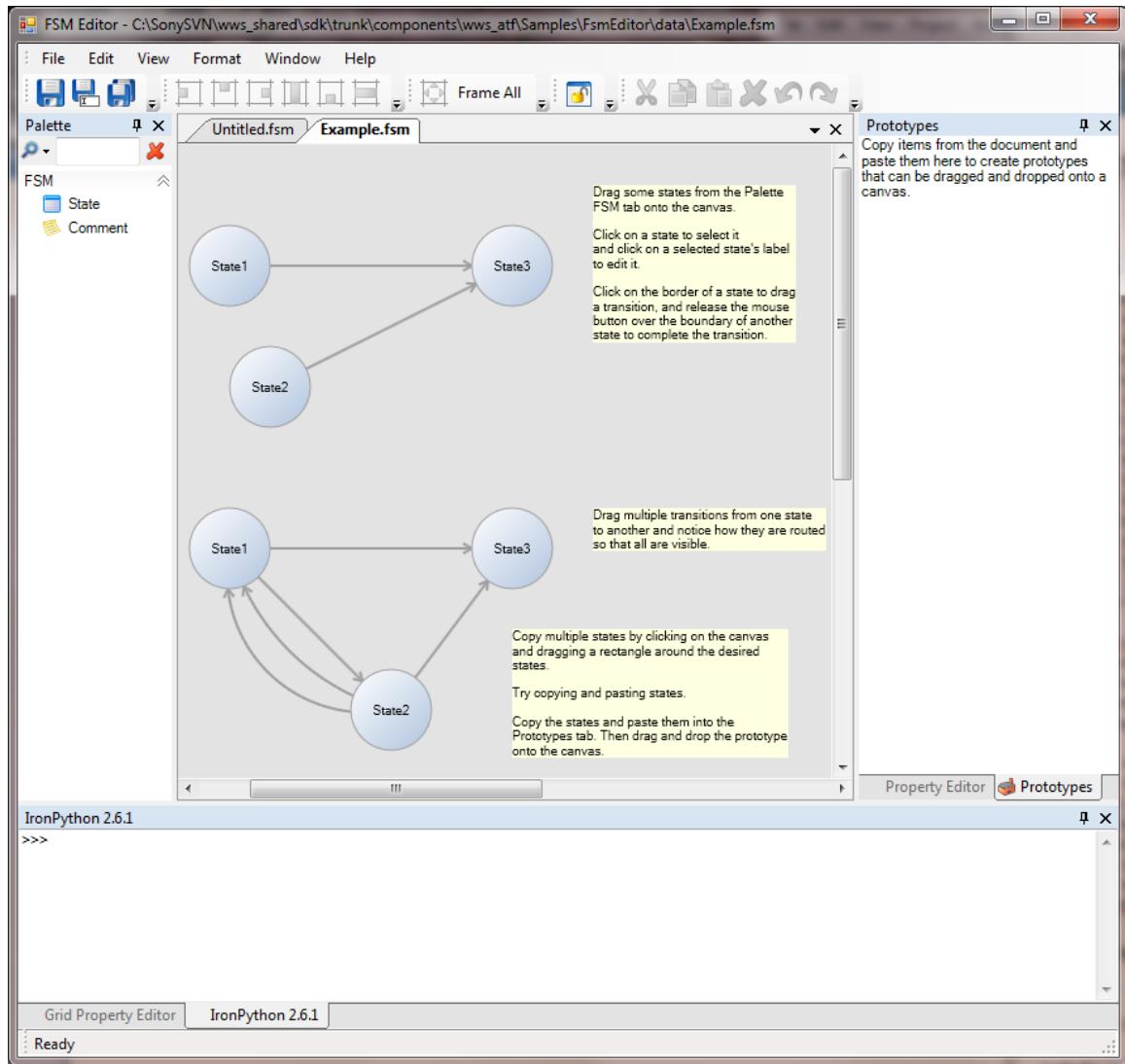
The `GraphViewCommands` component is the command client for zoom commands, implementing `ICommandClient`. It creates the View > Zoom menu items and their corresponding commands:

- Zoom In
- Zoom Out
- Zoom Reset

`GraphViewCommands` is used by the [ATF Circuit Editor Sample](#).

NumberedRoute Class

`NumberedRoute` is used to route directed graph edges, so that multiple edges connecting the same nodes do not overlap in rendering. An index indicates the degree of curving: route 0 is a straight line between nodes, route 1 is a slight arc, route 2 is a more curved arc, and so on. This illustration shows how multiple edges are displayed in the [ATF FSM Editor Sample](#) between "State1" and "State2" in the group of nodes at the bottom:



Direct2D Classes

These classes are control adapters and renderers for Direct2D controls, such as `D2dAdaptableControl`. For more information about Direct2D, see the MSDN article [About Direct2D](#).

Direct2D Adapters

Several control adapters work with Direct2D.

D2dGraphAdapter Class

`D2dGraphAdapter` is the key adapter here, because it adapts a control to displaying a graph. It also provides hit testing with its `Pick()` method. `D2dGraphAdapter` does not render a graph itself, but uses the `D2dGraphRenderer` passed in its constructor:

```
public D2dGraphAdapter(D2dGraphRenderer<TNode, TEdge, TEdgeRoute> renderer,
    ITransformAdapter transformAdapter)
```

`D2dGraphAdapter`'s `OnRender()` method is called to actually render the graph. `OnRender()` calls the renderer's `Draw()` methods to draw the nodes and edges of the graphs. `D2dGraphAdapter` may be used as is, or overridden. For example, the [ATF State Chart Editor Sample](#) derives its own `StatechartGraphAdapter` from `D2dGraphAdapter` and also overrides its `OnRender()` method.

The adapter `D2dSubgraphAdapter` derives from `D2dGraphAdapter` and has the same purpose, except that it works with a subgraph.

`D2dGraphAdapter` is required by the other adapters `D2dGraphEdgeEditAdapter` and `D2dGraphNodeEditAdapter` in their constructors. For example:

```

public D2dGraphEdgeEditAdapter(
    D2dGraphRenderer<TNode, TEdge, TEdgeRoute> renderer,
    D2dGraphAdapter<TNode, TEdge, TEdgeRoute> graphAdapter,
    ITransformAdapter transformAdapter)

```

Both the `D2dGraphEdgeEditAdapter` and `D2dGraphNodeEditAdapter` constructors also require a `D2dGraphRenderer`.

D2dGraphEdgeEditAdapter Class

`D2dGraphEdgeEditAdapter` adds graph edge dragging capabilities to an adapted control. It derives from `DraggingControlAdapter`, which is the base class for control adapters that handle mouse dragging, and overrides its methods to handle mouse manipulation.

Edges can be moved, disconnected, and reconnected to other nodes. `D2dGraphEdgeEditAdapter` has methods like `CanConnectTo()`, `CanConnectFrom()`, and `MakeConnection()` to check if connections can be made and make them.

The [ATF Circuit Editor Sample](#), the [ATF FSM Editor Sample](#), and the [ATF State Chart Editor Sample](#) all use `D2dGraphEdgeEditAdapter`, varying the constructor type parameters according to their graph model. For example, Circuit Editor does this to fit its variant of `IGraph`:

```

var circuitConnectionEditAdapter =
    new D2dGraphEdgeEditAdapter<Module, Connection, ICircuitPin>(m_circuitRenderer, circuitAdapter
, transformAdapter);

```

Because it uses states as nodes and transitions as edges, State Chart Editor does this:

```

var statechartTransitionEditAdapter =
    new D2dGraphEdgeEditAdapter<StateBase, Transition, BoundaryRoute>(m_statechartRenderer, statechartAdapter
, transformAdapter);

```

D2dGraphNodeEditAdapter Class

`D2dGraphNodeEditAdapter` complements `D2dGraphEdgeEditAdapter` by adding graph node dragging capabilities to an adapted control. `D2dGraphNodeEditAdapter` also derives from `DraggingControlAdapter`, overriding its methods. Dragging nodes does not change edge connections, but it does move the edges along with the nodes. This class does not handle dragging nodes out of groups of nodes; that is handled by the `IEditableGraphContainer` interface. For more information on using `IEditableGraphContainer`, see [CircuitEditingContext Class](#). `D2dGraphNodeEditAdapter` is used as an adapter for node dragging in the [ATF Circuit Editor Sample](#), the [ATF FSM Editor Sample](#), and the [ATF State Chart Editor Sample](#). In constructing `D2dGraphNodeEditAdapter`, these samples vary the type parameters, depending on the graph model, just as for `D2dGraphEdgeEditAdapter`.

Direct2D Renderers

ATF provides renderers for different types of graphs. The most general one is the abstract class `D2dGraphRenderer`, already discussed in [D2dGraphRenderer Class](#). The other graph rendering classes derive from `D2dGraphRenderer`.

Renderers are used with control adapters. For instance, `D2dGraphAdapter` and `D2dGraphEdgeEditAdapter` require a `D2dGraphRenderer` in their constructors.

Renderers use a `D2dGraphics` object to do the actual drawing. `D2dGraphics` represents an object that can receive drawing commands and uses Direct2D for drawing.

`D2dDigraphRenderer` is a fairly general graph rendering class derived from `D2dGraphRenderer`. It is a standard directed graph renderer that renders nodes as disks and edges as lines or arcs. Edge routes have integer indices, indicating which line or arc to draw for the edge, and this allows multiple edges between a pair of nodes to be distinguished.

`D2dDigraphRenderer` has `Draw()` methods for nodes and edges that use a `D2dGraphics` object to do the actual drawing. It also has `Pick()` methods to find the node and/or edge hit by a given point, returning a `GraphHitRecord` object. `D2dDigraphRenderer` is the renderer used in the [ATF FSM Editor Sample](#).

The other renderers are specific to the type of graph:

- Use `D2dCircuitRenderer` and `D2dSubCircuitRenderer` to render circuits and subcircuits, see [Circuit Rendering Classes](#).
- Use `D2dStatechartRenderer` to render statecharts, see [D2dStatechartRenderer Class](#).

Obsolete Classes

Do not use the following obsolete classes, which have been superseded by their Direct2D equivalents with a "D2d" prefix:

Obsolete class	Direct2D class to use	Comment
CircuitRenderer	D2dCircuitRenderer	
DigraphRenderer	D2dDigraphRenderer	
GraphAdapter	D2dGraphAdapter	
GraphEdgeEditAdapter	D2dGraphEdgeEditAdapter	
GraphNodeEditAdapter	D2dGraphNodeEditAdapter	
GraphRenderer	D2dGraphRenderer	GraphRenderer is referenced only in the obsolete classes. D2dGraphRenderer is in the Atf.Gui assembly.
StatechartRenderer	D2dStatechartRenderer	

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [Circuit Editor Programming Discussion](#), [Circuit Graph Support](#), [Statechart Graph Support](#)

Circuit Graph Support

ATF provides the most support for the circuit type of graph. The [ATF Circuit Editor Sample](#) shows how to use the ATF circuit graph facilities, which do much of the work of handling circuit graphs for you. For a discussion of this sample's programming, see [Circuit Editor Programming Discussion](#), which also refers back to this topic for classes and interfaces that are not part of the sample.

A circuit specializes a graph. The general graph interface

```
IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>
```

becomes this for a circuit:

```
IGraph<Element, Wire, ICircuitPin>
```

These specialized circuit items are:

- **Element**: Circuit element with pins.
- **Wire**: Connection between pins of **Elements**.
- **ICircuitPin**: Interface for pins, which are the sources and destinations for wires between circuit elements. It derives from **IEdgeRoute**; for further information, see [ATF Graph Interfaces](#).

The circuit graph classes are all in the `Sce.Atf.Controls.Adaptable.Graphs` namespace. The classes are split between two assemblies, just as for general graph support. Both assemblies have a "Circuit" folder containing circuit graph classes, but there are also some circuit classes outside that folder.

Contents

- [Atf.Gui Assembly Graph Support](#)
 - [Circuit DOM Adapters](#)
 - [Key DOM Adapter Classes](#)
 - [ICircuitContainer Interface](#)
 - [Options and Settings Classes](#)
 - [Utility and Information Classes](#)
- [Atf.Gui.WinForms Assembly Graph Support](#)
 - [CircuitDocument Class](#)
 - [Circuit Context Classes](#)
 - [Circuit Command Classes](#)
 - [Circuit Rendering Classes](#)
 - [Miscellaneous Support](#)

Almost all the circuit core classes are implemented as abstract classes, although their functions are fully implemented in the class. Clients can override a class and fill in the application-specific metadata for DOM attributes and child node types, typically using the classes in a `Schema` class created by DomGen. The next step is to register these derived class as DOM node extensions, typically in the schema loader class `SchemaLoader`.

`Element`, `Wire`, `Circuit`, and `Group` are the building block classes for circuit infrastructure in the circuit graph framework. They are all DOM adapters to bridge application code and data to the ATF circuit model.

Atf.Gui Assembly Graph Support

This assembly contains DOM adapter classes that represent circuit items, such as the important classes `Circuit` and `Group`. For more information on these classes, see [Circuit Class](#) and [Group Class](#). It also contains the `ICircuitContainer` interface and various support classes. For details on this interface, see [ICircuitContainer Interface](#).

Circuit DOM Adapters

A set of DOM adapter classes represents circuit items, such as circuits, groups, and pins. This means that each of these circuit items is represented by a `DomNode` of that item's type in the tree containing application data. The files for these classes are in the "Circuit" folder of the `Atf.Gui` assembly.

For a description of how the [ATF Circuit Editor Sample](#) overrides and uses these DOM adapters, see [DOM Adapters in Circuit Editor Programming Discussion](#).

DOM Adapter Summary

This table lists each DOM adapter, its derived class (if not from `DomNodeAdapter`), any interfaces it implements, and a description.

DOM adapter	Derives from (if not <code>DomNodeAdapter</code>)	Implements	Description
Annotation		IAnnotation	Text item that can be placed on a circuit to provide information.
Circuit		IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram, ICircuitContainer	Collection of Elements, Wires, Annotations, and LayerFolders representing a circuit. For more information, see Circuit Class .
Element		ICircuitElement, IVisible	The base circuit element containing Pins that connect to other Pins. For more details, see Element Class .
ElementRef			Reference to an Element, which is used within layer folders to represent circuit elements that belong to that layer.
Group	Element	ICircuitGroupType<Element, Wire, ICircuitPin>, IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram, ICircuitContainer	Collection of Elements, Wires, and Annotations. For more information, see Group Class .
GroupPin	Pin	ICircuitGroupPin<Element>	A pin on a group module, with extra information needed to associate the pin on the group with the internal module where it was connected before grouping.
LayerFolder			Folder containing layers. Layers contain references to Elements. Layers and their contents' visibility can be toggled.
Pin		ICircuitPin	Pin on an Element, which can be connected to another Pin with a Wire.
Prototype			Circuit prototype, which contains Elements and Wires that can be copied into a circuit. For a discussion of handling prototypes in the ATF Circuit Editor Sample , see Prototype Handling .
PrototypeFolder			Folder of Prototypes. For a discussion of handling prototypes in the ATF Circuit Editor Sample , see Prototype Handling .
SubCircuit	Circuit	ICircuitElementType	Subcircuit of Elements, Wires, and Annotations created when mastering. Subcircuits are used in mastering.
SubCircuitInstance	Element		Instance of a mastered SubCircuit. Note that this instance derives from Element — not Circuit as SubCircuit does. A SubCircuitInstance appears as a single element in a circuit, whereas a SubCircuit may appear as one or more elements in a circuit.
Wire		IGraphEdge<Element, ICircuitPin>	Connection between pins of Elements. For more details, see Wire Class .

General Characteristics

These DOM adapters have commonalities in how they handle attributes, and some of them also have their own `OnNodeSet()` method. These things may affect how you override these classes.

Attribute Properties

These DOM adapters generally have properties to access information about the underlying type that comes from the type definitions in the data model. If the application has a `Schema` class with type metadata classes, the properties would return type attributes. The [ATF Circuit Editor Sample](#) shows that kind of usage.

For example, the `Annotation` class has these two related properties:

```
protected abstract AttributeInfo TextAttribute { get; }
...
public string Text
{
    get { return (string)DomNode.GetAttribute(TextAttribute); }
    set { DomNode.SetAttribute(TextAttribute, value); }
}
```

The `TextAttribute` property must be overridden in the derived class, and this overridden property provides the `AttributeInfo`. The `Text` property can then use the `TextAttribute` property to get and set the value of the `DomNode` attribute. For instance, here is how the [ATF Circuit Editor Sample](#) overrides `TextAttribute`:

```
protected override AttributeInfo TextAttribute
{
    get { return Schema.annotationType.textAttribute; }
}
```

OnNodeSet() Method

Some of the DOM adapters implement their own `OnNodeSet()` method, which is called when a DOM adapter reference is obtained for a given `DomNode`. Here is `OnNodeSet()` for `Circuit`:

```
protected override void OnNodeSet()
{
    // cache these list wrapper objects
    m_elements = new DomNodeListAdapter<Element>(DomNode, ElementChildInfo);
    m_wires = new DomNodeListAdapter<Wire>(DomNode, WireChildInfo);
    if (AnnotationChildInfo != null)
        m_annotations = new DomNodeListAdapter<Annotation>(DomNode, AnnotationChildInfo);

    foreach (var connection in Wires)
        connection.SetPinTarget();

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

    base.OnNodeSet();
}
```

This method caches list wrappers of the `DomNode`'s circuit's elements, wires, and annotations. It also subscribes this `DomNode` to several events.

Note: Caching the lists with `DomNodeListAdapter` may give the impression that each item in the list (element, wire, or annotation) is adapted and cached. However, `DomNodeListAdapter` is a wrapper that adapts the entire underlying list to some type, and it does not cache or adapt each item in the list individually. Adding items simply adds items to the child list with no adaptation needed.

Classes that override the DOM adapter's `OnNodeSet()` method should generally call the DOM adapter's base `OnNodeSet()` method, too.

Key DOM Adapter Classes

This section covers the most important circuit DOM adapters.

Circuit Class

`Circuit` is the fundamental class of the DOM adapters, because it represents a circuit graph by implementing `IGraph<Element, Wire, ICircuitPin>`. `IGraph` has `Nodes` and `Edges` properties that get enumerations of the circuit's elements and wires: the building blocks of a circuit.

`Circuit` also implements `ICircuitContainer`, because it is a container for circuits, and this implementation comprises the bulk of the `Circuit` class. For details on what this interface does, see [ICircuitContainer Interface](#).

Element Class

A circuit consists of `Elements` connected by `Wires`. `Element` adapts a `DomNode` to a circuit element, which has `Pins` that connect to `Pins` on other `Elements`. Examples of elements include OR gates, buttons, and lights, as in the [ATF Circuit Editor Sample](#) in which these items can be dragged from a palette onto a circuit canvas.

`Element` maintains local name and bounds for faster circuit rendering during editing operations, such as dragging elements and wires.

`Element` contains properties describing the element, such as its position and name, which must be overridden. Other properties get the element type and level.

`Element` also contains methods to find an element and pin, given a `PinTarget` containing the pin's element and pin index. These `MatchPinTarget()` and `FullyMatchPinTarget()` are similar to methods in `ICircuitContainer`; for details, see [ICircuitContainer Interface](#).

Wire Class

`Wire` adapts to a connection in the circuit. `Wire` contains properties describing the connection, such as input and output element and pin, which must be overridden. It also has properties to get input and output `PinTargets`, encapsulating the pins' elements and pin indexes.

Group Class

`Group` is the most complicated of the circuit DOM adapters. It derives from `Element` because it is treated as a circuit element. Like `Circuit`, it implements `IGraph<Element, Wire, ICircuitPin>`, because it is a circuit in its own right.

In addition, it implements `ICircuitGroupType<Element, Wire, ICircuitPin>`. This interface defines properties to describe the group:

- `Expanded`: Get or set whether the subgraph is expanded.
- `SubEdges`: Get the group's internal edges.
- `AutoSize`: Get or set whether the hierarchical container is automatically resized to display its entire contents.
- `Info`: Get the `CircuitGroupInfo` object that controls various options on this circuit group.

`ICircuitGroupType` itself implements `IHierarchicalGraphNode` with its `SubNodes` property to get the sequence of nodes that are children of this group.

Much of this class is devoted to handling group pins and updating them as the group is edited and changes are validated. Such editing includes creating groups as well as moving objects in and out of groups.

ICircuitContainer Interface

`ICircuitContainer` represents a container for circuit items, and is implemented by the `Circuit` and `Group` classes.

`ICircuitContainer` contains properties to get or set lists of the constituent items and get circuit characteristics:

- `Annotations`: Get the modifiable list of annotations that are owned by this `ICircuitContainer` as `IList<Annotation>`.
- `Elements`: Get the modifiable list of elements and group elements that are inside this `ICircuitContainer` as `IList<Element>`.
- `Wires`: Get the modifiable list of wires that are completely contained within this container as `IList<Wire>`. Each `Wire` must connect to two circuit elements that are inside this container.
- `Dirty`: Get or set whether the contents of the container have changed.
- `Expanded`: Get or set whether the graph is expanded.

`ICircuitContainer` has these methods that are implemented by several of the circuit item DOM adapters:

- `FullyMatchPinTarget()`: Find the element and pin that match the pin target, including the template instance node.
- `MatchPinTarget()`: Find the element and pin that match the pin target for this circuit container.
- `Update()`: Synchronize internal data and contents that differ due to editing operations.

`ICircuitContainer` is mainly introduced so that in various types of circuit editing, the code does not need to distinguish whether you are editing a circuit (a top level graph container) or a group (a graph container at an arbitrary level), because both `Circuit` and `Group` implement

the editing interface `ICircuitContainer` to add and remove items.

Options and Settings Classes

The following simple classes provide various options and settings for circuit rendering:

- `CircuitDefaultStyle`: Properties for default settings for circuit rendering.
- `CircuitGroupInfo`: Properties for options specifying the behavior or appearance of an `ICircuitGroup`.
- `CircuitGroupPinInfo`: Properties for options specifying the behavior or appearance of a `ICircuitGroupPin`.

Utility and Information Classes

The remaining items in the assembly include utility and informational classes:

- `CircuitUtil`: Various circuit utility methods, such as `GetSubGraph()` that returns a list of all modules, internal connections between them, and connections to external modules; and `GetGroupPath()` that gets the graph path of a specified group.
- `ElementType`: Type of circuit element. This is a simple implementation of `ICircuitElementType`, an interface for circuit element types, which defines the appearance, inputs, and outputs of an `Element` object. It includes its own `Pin` subclass, an implementation of `ICircuitPin`, for element input and output pins.
- `PinTarget`: Encapsulate the circuit element and pin index for a given `Pin` object.

Atf.Gui.WinForms Assembly Graph Support

This assembly provides document, context, command, and rendering classes specific to circuit graphs.

CircuitDocument Class

`CircuitDocument` represents a circuit document. It derives from `DomDocument`, which implements `IDocument`, the interface for documents.

You can define `CircuitDocument` as a DOM adapter for a circuit document type, as the [ATF Circuit Editor Sample](#) does for its `CircuitDocument` derived class.

Circuit Context Classes

The main context provided is `CircuitEditingContext`. For more general information about contexts, see [ATF Contexts](#).

CircuitEditingContext Class

The `CircuitEditingContext` abstract class defines a circuit editing context. Each context represents a circuit or subcircuit, with a history, selection, and editing capabilities. There may be multiple contexts within a single circuit document, because each subcircuit has its own editing context.

You want to derive from `CircuitEditingContext` for any application that edits circuit graphs. For example, the [ATF Circuit Editor Sample](#) derives its own `CircuitEditingContext` from this class, as discussed in [CircuitEditingContext Class in Circuit Editor Programming Discussion](#).

`CircuitEditingContext` is a DOM adapter. You should define it for any type that contains circuit elements that can be displayed in their own window, such as circuits and groups. For an example in [ATF Circuit Editor Sample](#), see [CircuitEditingContext Class](#).

`CircuitEditingContext` derives from `Sce.Atf.Applications.EditingContext`, which is used in several samples. For more information about `EditingContext`, see [Context Classes](#) and other topics in [ATF Contexts](#).

`CircuitEditingContext` implements several interfaces that are needed for editing circuits:

```
public abstract class CircuitEditingContext : EditingContext,
    IEnumerableContext,
    INamingContext,
    IInstancingContext,
    IObservableContext,
    IColoringContext,
    IEditableGraphContainer<Element, Wire, ICircuitPin>
```

The only circuit-specific interface here is `IEditableGraphContainer<Element, Wire, ICircuitPin>`, which derives from `IEditableGraph<Module, Connection, ICircuitPin>`, and so both are implemented. `IEditableGraph` has methods, such as `CanConnect()`, `Connect()`, `CanDisconnect()`, and `Disconnect()` to make and break circuit connections.

`IEditableGraphContainer`'s methods, such as `CanMove()` and `Move()`, allow you to move circuit elements in and out of containers like groups and are fully implemented in `CircuitEditingContext`. These two interface implementations in `CircuitEditingContext` perform much of the work of editing a circuit for your application.

The other interfaces implemented by `CircuitEditingContext` also play their role:

- `IEnumerableContext`: Enumerate items in a circuit.
- `ISelectionContext`: Allow selecting items in a circuit. This is implemented by `EditingContext`. See the note below on implementing `ISelectionContext`.
- `IInstancingContext`: Handle copy and paste of selected circuit items.
- `INamingContext`: Name circuit items, such as elements.
- `IColoringContext`: Set and change the background color of annotations.
- `IObservableContext`: Events for circuit items being added or removed. This is required so the circuit display can be refreshed after it changes.

Some of these interfaces depend on each other. You need `ISelectionContext` to enable selection before you can use `IInstancingContext` to edit selections. `IEnumerableContext` provides the ability to enumerate circuit items, which is fundamental.

Note: `CircuitEditingContext` derives from `Sce.Atf.Dom.EditingContext`, which provides an `ISelectionContext` implementation in its base class `HistoryContext`. If an application already has its own editing context implementation, care must be taken to share the `ISelectionContext` between `CircuitEditingContext` and the custom editing context if it does not derive from `CircuitEditingContext`.

Suppose an application has its own custom editing context for application-specific logic in graph editing, say `GraphEditingContext`, that is not derived from `CircuitEditingContext`. In this case, it is recommended that both `CircuitEditingContext` and `GraphEditingContext` be defined as DOM adapters on the same types. For instance, custom circuit and group types would define both `CircuitEditingContext` and `GraphEditingContext` as DOM adapters. This is useful, because ATF calls `IContextRegistry.GetActiveContext<CircuitEditingContext>()` to obtain the current active circuit editing context to handle editing commands. If the current active context is `GraphEditingContext`, this call can obtain the `CircuitEditingContext` too, because both editing contexts are adapters for the same `DomNode` type; otherwise the call returns `null`.

LayeringContext Class

`LayeringContext` implements `ILayeringContext`, so that you can put circuit items in layers and then control visibility of layers and their members. The `LayerLister` component can present layered items in a tree control, using an `ILayeringContext` implementer. The [ATF Circuit Editor Sample](#) makes use of both this context and this component; for more details, see [Layer Handling](#).

`ILayeringContext` implements several interfaces:

```
public interface ILayeringContext : IVisibilityContext, ITreeView, IItemView
```

These interfaces do the following:

- `IVisibilityContext`: Control visibility of layers and items in layers.
- `ITreeView`: Define a view on hierarchical data so it can be displayed in a tree. For more information, see [ITreeView Interface](#).
- `IItemView`: Provide display information about an item in a `ItemInfo` object. `ItemInfo` holds information on the appearance and behavior of an item in a list or tree control. For more information, see [IItemView Interface](#).

`LayeringContext` derives from `SelectionContext`, which implements `ISelectionContext` to enable selection in the layering window:

```
public abstract class LayeringContext : SelectionContext,
    IInstancingContext,
    IHierarchicalInsertionContext,
    ILayeringContext, // : IVisibilityContext, ITreeView, IItemView
    IObservableContext,
    INamingContext
```

These interfaces perform similar functions as in `CircuitEditingContext`:

- `IInstancingContext` allows pasting items to be layered into the Layers window, which requires instancing. For more information about instancing, see [Instancing In ATF](#); in particular, see [IInstancingContext](#) and [IHierarchicalInsertionContext Interfaces](#).
- `IHierarchicalInsertionContext` is also implemented, so layer hierarchies can be established and displayed in the Layers window tree.
- `INamingContext`: Name layers.
- `IObservableContext`: Events to trigger refreshing the Layers window display.

Note how the `LayerFolder` DOM adapter fits into this layering scheme. There is a layer folder for each layer and a `DomNode` for each layer

folder. The `LayerFolder` DOM adapter can be defined for the layer folder type in the application's data model. `ElementRef` is the DOM adapter for an element reference that is added to a layer to place that element in the layer. `LayerFolder` `DomNodes` thus have `ElementRef` child nodes.

PrototypingContext Class

Prototypes allow you to copy circuit elements and paste them into a prototype window as a named prototype. You can then drag the prototype onto a circuit to copy all its circuit items into the circuit. `PrototypingContext` does with prototypes what `LayeringContext` does with layers, so the contexts implement similar interfaces and do similar things.

`PrototypingContext` implements `IPrototypingContext` to present a tree view of prototypes. The `PrototypeLister` component can present prototypes in a tree control, using an `IPrototypingContext` implementer. Again, the [ATF Circuit Editor Sample](#) incorporates both of these; for more information, see [Prototype Handling](#).

Like `LayeringContext`, `PrototypingContext` derives from `SelectionContext`, which implements `ISelectionContext` to enable selection in the prototype window. `PrototypingContext`'s other interfaces play similar roles as in `LayeringContext` to display a tree of information:

```
public abstract class PrototypingContext : SelectionContext,
    IInstancingContext,
    IPrototypingContext,
    IObservableContext,
    INamingContext
```

`IPrototypingContext` implements almost the same interfaces as `ILayeringContext`:

```
public interface IPrototypingContext : ITreeView, IItemView
```

These interfaces operate similarly as in `ILayeringContext`:

- `ITreeView`: Define a view on hierarchical data so it can be displayed in a tree. For more information, see [ITreeView Interface](#).
- `IItemView`: Provide display information about an item in a `ItemInfo` object. For more information, see [IItemView Interface](#).

The other interfaces `PrototypingContext` implements have a similar function here as they do in `LayeringContext`:

- `IInstancingContext`: Allow copying and pasting into the Prototypes window.
- `IObservableContext`: Update Prototypes window display.
- `INamingContext`: Name prototypes.

`PrototypingContext` does not implement `IHierarchicalInsertionContext`, so you can't create a hierarchy of prototypes.

In the realm of the DOM, `PrototypeFolder` type `DomNodes` have `Prototype` type children. `Prototype` `DomNodes` have `Element` and `Wire` children.

ViewingContext Class

`ViewingContext` is a DOM adapter that provides viewing functions for a control displaying a graph:

```
public class ViewingContext: Validator, IViewingContext, ILayoutContext
```

The class and interfaces do the following:

- `Validator`: Derives from this class to implement its `OnEnded()` method, which updates the control canvas's bounds large enough so that a user has room to work in.
- `IViewingContext`: Check whether objects are or can be made visible, or are framed or can be framed in the current view.
- `ILayoutContext`: Get and set item bounding rectangles, information on which parts of the bounds are meaningful, and which parts of bounds can be set.

The `ViewingContext` DOM adapter is useful for circuit and group types because objects of these types display circuits, and this is how the [ATF Circuit Editor Sample](#) uses it. [ATF FSM Editor Sample](#) and [ATF State Chart Editor Sample](#) also use `ViewingContext`.

Circuit Command Classes

Two components add commands to applications for graphs.

GroupingCommands Component

Grouping takes modules and the connections between them and turns them into a single equivalent element. `GroupingCommands` adds commands for grouping circuit elements that appear in a context menu for a group:

- Group: Group the selection into a single item.
- Ungroup: Ungroup any selected groups.
- Hide Unconnected Pins: Indicate whether to hide or show the unconnected pins in a group.
- Show Expanded Group Pins: Indicate whether to draw the orange box on the left and right borders of an expanded group that represents the group pin.
- Reset Group Pin Names: Set all the group pin names to their default values.

`GroupingCommands` is a command client implementing `ICommandClient`. It also implements `IContextMenuCommandProvider` to provide the commands for a context menu.

Simply add the `GroupingCommands` component to the application's MEF `TypeCatalog` to use it.

LayeringCommands Component

Layering adds modules to a layer. The layer and its individual items visibility can be toggled. Layers can be organized in a tree in a window. `LayeringCommands` adds a command related to the layering window in a context menu for the layering window:

- Add Layer: Create a new layer folder in the layer window.

`LayeringCommands` is a command client implementing `ICommandClient`. It also implements `IContextMenuCommandProvider` to provide the commands for a context menu.

To use the `LayeringCommands` component, add it to the application's MEF `TypeCatalog`.

Circuit Rendering Classes

There are several circuit renderers. Note that renderers are not used directly, but through control adapters.

D2dCircuitRenderer Class

The `D2dCircuitRenderer` class renders circuit graphs. It draws graph nodes as circuit elements, and edges as wires. Elements have zero or more output pins, where wires originate, and zero or more input pins, where wires end. Input pins are on the left side of elements, output pins on the right.

`D2dCircuitRenderer` derives from `D2dGraphRenderer`, which is discussed in [D2dGraphRenderer Class](#). `D2dCircuitRenderer` overrides `D2dGraphRenderer`'s `Draw()` methods using Direct2D (`D2dGraphics` class) to draw the circuit graph, including groups. It also overrides the `Pick()` methods. For more information about Direct2D, see [About Direct2D](#).

`D2dCircuitRenderer` requires a `D2dDiagramTheme`, a diagram rendering theme class for Direct2D rendering. For more information on this class, see [Document Client](#).

`D2dGraphAdapter` requires a `D2dCircuitRenderer` object to render the circuit. For more information on `D2dGraphAdapter`, see [General Graph Support](#) and [Direct2D Adapters](#) in particular. `D2dGraphEdgeEditAdapter` also requires a renderer, which can be a `D2dCircuitRenderer`. The [ATF Circuit Editor Sample](#) shows using a `D2dCircuitRenderer` with `D2dGraphAdapter` and `D2dGraphEdgeEditAdapter`. For specifics of how this is done, see [Circuit Document Display and Control Adapters](#).

D2dSubCircuitRenderer Class

`D2dSubCircuitRenderer` is a subgraph renderer that draws subnodes as circuit elements, and subedges as wires. It also draws virtual representations of group pins for editing. It is used for rendering groups.

`D2dSubCircuitRenderer` derives from `D2dCircuitRenderer`. It uses `D2dCircuitRenderer`'s `Draw()` methods directly. It also draws the visibility icon () that toggles the group pin's visibility and draws floating group pins. It provides a `Pick()` method to pick the visibility icon and floating group pins.

`D2dSubgraphAdapter` requires a `D2dSubCircuitRenderer` object to render the circuit. For more information on `D2dSubgraphAdapter`, see [Direct2D Adapters](#). `D2dGraphEdgeEditAdapter` also requires a renderer, which can be a `D2dSubCircuitRenderer`. The [ATF Circuit Editor Sample](#) shows using a `D2dSubCircuitRenderer` with both `D2dSubgraphAdapter` and `D2dGraphEdgeEditAdapter`. For a discussion of doing this, see [Circuit Document Display and Control Adapters](#).

Miscellaneous Support

These classes provide various kinds of circuit support, such as a circuit registry and validator.

CircuitControlRegistry Component

CircuitControlRegistry provides a convenient service to register and unregister circuit controls created for circuits, synchronizes UI updates for circuit controls due to group renaming, circuit element insertion and deletion, and closing of documents and controls. CircuitControlRegistry is analogous to ControlHostService, described in [ControlHostService Component](#), except that CircuitControlRegistry is used for controls that are used to render circuits, such as AdaptableControl and D2dAdaptableControl.

The RegisterControl() method registers controls with a DomNode:

```
public virtual void RegisterControl(DomNode circuitNode, Control control, ControlInfo controlInfo  
, IControlHostClient client)
```

The DomNode parameter is the root of the tree representing the circuit or subcircuit to be displayed in the control. The Control parameter is the control the circuit is rendered in and is usually a D2dAdaptableControl.

The [ATF Circuit Editor Sample](#) uses CircuitControlRegistry.RegisterControl() to register a D2dAdaptableControl to display a full circuit, as well as a group.

CircuitValidator Class

The CircuitValidator class is a DOM adapter deriving from Validator. It tracks changes to edges and updates their routing during validation. It updates edges on the Ending event to be part of the transactions themselves, and then validates all subgraphs in the current document on the Ended event.

CircuitValidator should be defined as a DOM adapter for the root type of the data model. This allows all DomNodes in the circuit tree to be validated. The DOM adapter ReferenceValidator must also be defined on this type.

Validator derives from Observer, a DOM adapter that tracks DOM nodes as they enter and exit the subtree rooted at the DOM node the adapter is bound to. CircuitValidator overrides Observer methods, such as OnChildInserted(), that are called when the DomNode tree is modified. This gives CircuitValidator the opportunity to update the circuit so it remains valid:

```
protected override void OnChildInserted(object sender, ChildEventArgs e)  
{  
    AddSubtree(e.Child);  
    if (!m_undoingOrRedoing && e.Child.Is<Wire>())  
        UpdateGroupPinConnectivity(e.Child.Cast<Wire>());  
}
```

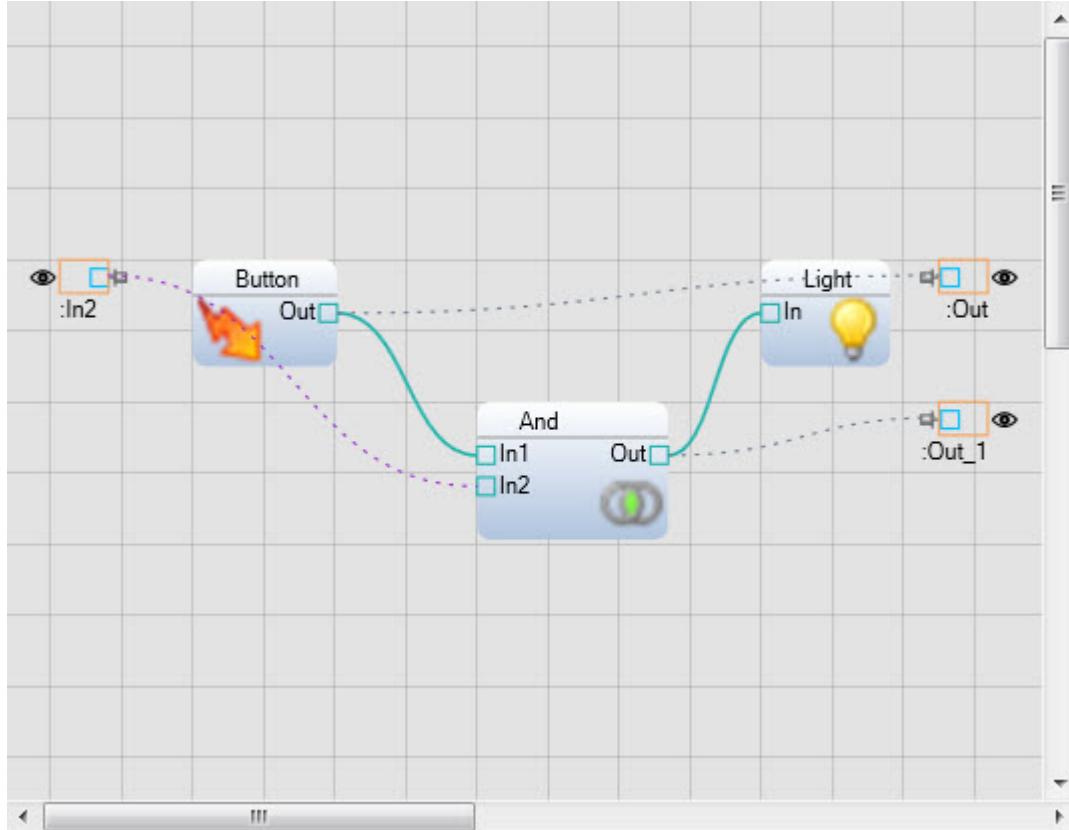
UpdateGroupPinConnectivity() updates the pin external connectivity of the connecting group.

CircuitValidator tracks changes made to the graph hierarchy during any transaction, doing its duties in three stages:

1. Editing tracking stage: During editing, it observes DomNode insertions, deletions, or attribute changes by marking the associated circuit or group node dirty.
2. Fixing stage: At the end of each editing operation (or more precisely, at the end of each transaction call to Validator.OnEnding()), it walks through all the graph nodes marked dirty and tries to fix up misaligned data. For example, when you delete a sub-node in a group, all its exposed group pins should also be deleted during this fix up stage. Another example is when you move a sub-node out of a group node, all edges previously connected to the sub-node internally need to be realigned if they become external edges in the parent container.
3. Validation stage: After each editing operation (that is, after each transaction call to Validator.OnEnded()), it walks through all the graph containers in the current document (circuits and groups), and verifies that each graph container is in a valid state. This stage does not change data, and currently is only activated in debug builds because of performance considerations.

GroupPinEditor Class

GroupPinEditor is a control adapter for adding floating group pin location and label editing capabilities to a subgraph control, for instance, a control containing a group. Here is a window showing a group in the [ATF Circuit Editor Sample](#):



GroupPinEditor allows a user to drag any of the group pins, boxed in the orange rectangles. GroupPinEditor also gives the ability to edit group pin labels by:

- selecting the group pin node in the group editor window and then pressing F2, or
- moving the mouse over the group pin label until it automatically enters label editing mode.

GroupPinEditor uses an `ITransformAdapter`, which is provided in its constructor. Here's an example from [ATF Circuit Editor Sample](#) where it is creating adapters for a `D2dSubgraphAdapter` to display a group:

```
var groupPinEditor = new GroupPinEditor(transformAdapter);
groupPinEditor.GetPinOffset = m_subGraphRenderer.GetPinOffset;
```

GroupPinEditor needs to have the `GetPinOffset()` callback to compute a group pin's y offset. In this example, the Circuit Editor uses the `D2dCircuitRenderer.GetPinOffset()` method.

After creating the `GroupPinEditor`, call `AdaptableControl.Adapt()` to use it, listing it with the other control adapters.

WireStyleProvider Class

`WireStyleProvider` is a DOM adapter that implements `IEdgeStyleProvider` to provide information about a wire's shape and appearance. You should define this on the type for connections in your data model. For instance, [ATF Circuit Editor Sample](#) defines it as an adapter for its "connectionType" in its schema loader:

```
Schema.connectionType.Type.Define(new ExtensionInfo<WireStyleProvider>Module, Connection, ICircuitPin
>>());
```

For more information on `IEdgeStyleProvider`, see [ATF Graph Interfaces](#).

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF Contexts](#), [ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF State Chart Editor Sample](#), [Authoring Tools](#)

Framework, Circuit Editor Programming Discussion, Context Classes, ControlHostService Component, File Explorer Programming Discussion, FSM Editor Programming Discussion, General Graph Support, IInstancingContext and IHierarchicalInsertionContext Interfaces, Instancing In ATF

Statechart Graph Support

Statecharts, or state transition diagrams, show states and transitions between them in a state machine. A child state machine can be embedded inside a state. Statecharts are illustrated in the [ATF State Chart Editor Sample](#).

A statechart, like a circuit, specializes a graph. The general graph interface

```
IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>
```

becomes this for a statechart:

```
IGraph<IState, IGraphEdge<IState, BoundaryRoute>, BoundaryRoute>
```

The distinguishing items for a statechart are:

- [IState](#): Interface for states in state-transition diagrams. For details, see [IState Interface](#).
- [BoundaryRoute](#): Edge route for statecharts. For more information, see [BoundaryRoute Class](#).

Topics

- [Atf.Gui Assembly Graph Support](#)
 - [Statechart Interfaces](#)
 - [BoundaryRoute Class](#)
 - [Statechart Enumerations](#)
 - [Atf.Gui.Winforms Assembly Graph Support](#)
 - [D2dStatechartRenderer Class](#)

Atf.Gui Assembly Graph Support

Most statechart classes and all the interfaces are in this assembly.

Statechart Interfaces

Statecharts can have regular states and pseudo-states. Pseudo-states provide information about actual states.

IComplexState Interface

[IState](#) is the interface for states in statechart diagrams that are non-pseudo-states. Its property:

- [Text](#): Get the state's interior text.

[IState](#) implements [IHierarchicalGraphNode](#), the interface for hierarchical nodes in a graph, which contain sub-nodes.

IState Interface

[IState](#) is the general interface for states in state-transition diagrams, and covers both states and pseudo-states. Its properties are:

- [Type](#): Get the state type, a [StateType](#) enumeration. For further details, see [StateType Enumeration](#).
- [Indicators](#): Get the visual indicators on the state, a [StateIndicators](#) enumeration. For more information, see [StateIndicators Enumeration](#).

In addition, [IState](#) implements [IGraphNode](#), the interface for a node in a graph. For more details, see [IGraphNode Interface](#).

BoundaryRoute Class

[BoundaryRoute](#) is an [IEdgeRoute](#) implementer for transitions between states. It is distinguished by its [Position](#) float property, which indicates the route position on the perimeter of a state. Its range is [0..4[, and it starts and ends at the top-left, going in a clockwise direction. The range from 0 to 1 is on the right side, from 1 to 2 on the bottom, and so on.

Statechart Enumerations

Two enumerations provide state information.

StateIndicators Enumeration

StateIndicators values enumerate visual indicators for state transition diagrams:

- Breakpoint: Display a breakpoint indicator on a state.
- Active: Display an "active" indicator on a state.

StateType Enumeration

StateType identifies the kind of state or pseudostate, which marks or points to states.

- Normal: A normal state.
- Start: The start pseudostate, which points to the first state that is active when the state machine starts running.
- Final: The final pseudostate, which is the last state that a state machine can be in.
- ShallowHistory: The shallow history pseudostate.
- DeepHistory: The deep history pseudostate.
- Conditional: A conditional pseudostate, which indicates a state with conditions to reduce the number of transitions.

History determines which states become active in a destination state's child state machines when the transition is taken to this destination. For a discussion of how history works in the StateMachine application built from ATF, see [State Machine History](#).

Atf.Gui.WinForms Assembly Graph Support

This assembly contains the rendering class for statecharts.

D2dStatechartRenderer Class

D2dStatechartRenderer is the class to handle rendering and hit testing on statechart graphs. It is analogous to the D2dCircuitRenderer that specializes in rendering circuit graphs. D2dStatechartRenderer draws graph nodes as states, and edges as transitions.

D2dStatechartRenderer derives from D2dGraphRenderer, which is described in [D2dGraphRenderer Class](#). D2dStatechartRenderer overrides D2dGraphRenderer's Draw() methods using Direct2D (D2dGraphics class) to draw the statechart. It also overrides the Pick() methods.

D2dStatechartRenderer requires a D2dDiagramTheme in its constructor to specify the rendering theme. For a description of its use in the [ATF FSM Editor Sample](#), see [Document Client in FSM Editor Programming Discussion](#).

Like the other rendering classes, D2dStatechartRenderer works with an adapter. For example, the [ATF State Chart Editor Sample](#) defines its own private class StatechartGraphAdapter derived from D2dGraphAdapter that takes a D2dStatechartRenderer in its constructor. Both D2dGraphNodeEditAdapter and D2dGraphEdgeEditAdapter also require a renderer, which can be a D2dStatechartRenderer. The [ATF State Chart Editor Sample](#) shows using a D2dStatechartRenderer with its StatechartGraphAdapter adapter for drawing statecharts, as well as with the adapters D2dGraphNodeEditAdapter and D2dGraphEdgeEditAdapter.

Topics in this section

Links on this page to other topics

[ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [FSM Editor Programming Discussion](#), [General Graph Support](#)

Timelines in ATF

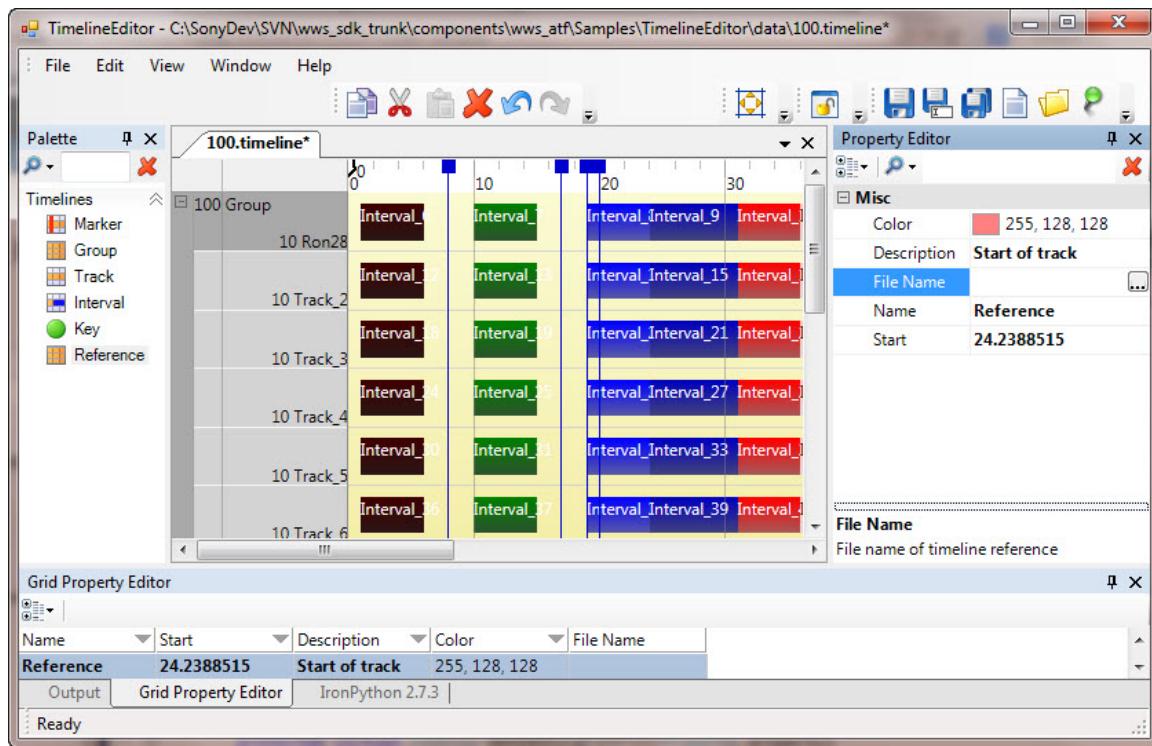
A timeline is a graphical representation of a time-sequence. Timelines contain groups that contain tracks that contain events: intervals, keys (zero-length intervals), and markers (zero-length events that are on all tracks in a timeline).

The [ATF Timeline Editor Sample](#) uses the ATF timeline facilities. For information on how the Timeline Editor uses these interfaces and classes, see [Timeline Editor Programming Discussion](#).

- [What is a Timeline in ATF?](#): General description of timelines, describing their objects, the timeline sample, and namespaces and assemblies in which support resides.
 - [Timeline Interfaces](#): Describes general and timeline object specific interfaces.
 - [WinForms Timeline Renderers, Controls, and Manipulators](#): Discusses the key objects in the `Sce.Atf.Controls.Timelines.Direct2D` namespace for timeline support of renderers, a canvas control, and manipulators for timelines.
 - [General Timeline Classes](#): Classes in the namespace `Sce.Atf.Controls.Timelines` that provide information needed for rendering and displaying timelines.
-

What is a Timeline in ATF?

A timeline is a graphical representation of a time sequence. It contains events, which start at some time and have an arbitrary duration. In the figure, the [ATF Timeline Editor Sample](#) shows a timeline in an opened file. The x-axis represents time.



A timeline consists of a hierarchy of objects:

- Timeline: Time sequence containing all the other timeline objects, such as groups. The figure shows a timeline with one group.
 - Marker: An event of zero-length time duration on all tracks in a timeline. The figure shows several markers, vertical lines through all tracks in the timeline.
 - Timeline Reference: Reference to a timeline (in a file) that can be displayed in another timeline.
 - Group: Container for zero or more tracks. The figure shows that the group has several tracks — labeled horizontal bars extending along the entire x-axis.
 - Track: Contains zero or more events.
 - Interval: An event of zero or greater time duration on a single track. Intervals on a single track do not overlap. The tracks in the illustration have several events — horizontal, labeled, colored bands along the track.
 - Key: An event of zero-length time duration on a single track.

There are several types of events:

- Interval
- Key
- Marker

Each of these objects has an associated interface. For a description of what these do, see [Timeline Interfaces](#).

Timeline support resides in two namespaces in two assemblies:

- `Sce.Atf.Controls.Timelines`: Timeline classes and interfaces that are UI-platform-agnostic (such as Windows Forms or WPF). In the `Atf.Gui` assembly.
- `Sce.Atf.Controls.Timelines.Direct2D`: Classes for working with timeline elements, including a renderer, control, and manipulators, that are dependent on the Windows Forms UI platform and use Direct2D to render timeline objects. (There are no WPF counterparts). In the `Atf.Gui.WinForms` assembly.

Topics in this section

Links on this page to other pages

[ATF Timeline Editor Sample](#), [Timeline Interfaces](#)

Timeline Interfaces

There are timeline interfaces for timelines themselves and for various kinds of objects in a timeline. These interfaces are UI-platform-agnostic (such as Windows Forms or WPF) and reside in the `Sce.Atf.Controls.Timelines` namespace and `Atf.Gui` assembly.

Timeline General Interfaces

This group of interfaces apply to timelines generally, including the timeline itself.

Contents

- [Timeline General Interfaces](#)
- [Timeline Object Interfaces](#)

Interface	Derives from/Implements	Properties, Methods	Description
<code>ITimeline</code>		Groups, Markers, <code>CreateGroup()</code> , <code>CreateMarker()</code>	For timelines, which contain groups, markers, and timeline references.
<code>IHierarchicalTimeline</code>	<code>ITimeline</code>	References	For <code>ITimeline</code> objects that can contain sub-timelines.
<code>IHierarchicalTimelineList</code>		References	For <code>ITimeline</code> objects that can contain sub-timelines, and support inserting, removing, and counting those sub-timelines via an <code>IList</code> interface.
<code>ITimelineDocument</code>	<code>IDocument</code> , <code>IObservableContext</code> , <code>IAdaptable</code>	Timeline	For editable timeline documents.

Timeline Object Interfaces

These interfaces are for objects in timelines. All of these objects are based on `ITimelineObject`, directly or indirectly.

Interface	Derives from/Implements	Properties, Methods	Description
<code>ITimelineObject</code>		None	For any object in a timeline.
<code>ITimelineReference</code>	<code>IEvent</code>	Target, Parent, Options	For objects that hold a reference to a timeline document.
<code>ITimelineObjectCreator</code>	<code>ITimelineObject</code>	<code>Create()</code>	For timeline objects that can create new instances of the same kind of object that they are. Before using the <code>Create()</code> method on an object, test that it implements <code>ITimelineObjectCreator</code> . <code>Create()</code> does not allow client code to determine the correct object to create in the case where there are multiple implementers of these interfaces: <code>IGroup</code> , <code>IMarker</code> , <code>ITrack</code> , <code>IInterval</code> , and <code>IKey</code> .
<code>IGroup</code>	<code>ITimelineObject</code>	Name, Expanded, Timeline, Tracks, <code>CreateTrack()</code>	For groups, which contain zero or more tracks, and can be expanded or collapsed in a timeline viewing control.
<code>ITrack</code>	<code>ITimelineObject</code>	Name, Group, Intervals, Keys, <code>CreateInterval()</code> , <code>CreateKey()</code>	For tracks, which contain zero or more events.
<code>IEvent</code>	<code>ITimelineObject</code>	Start, Length, Color, Name	For Events. This is the base interface for <code>IInterval</code> , <code>IKey</code> and <code>IMarker</code> .

IInterval	IEvent	Track	For intervals, which are zero or greater length events on a track.
IKey	IEvent	Track	For keys, which are zero length events on a track.
IMarker	IEvent	Timeline	For markers, which are zero length events on all tracks in a timeline.

Topics in this section

Links on this page to other pages

No links

WinForms Timeline Renderers, Controls, and Manipulators

These classes are responsible for drawing timelines in a control and providing extra capabilities. These classes use Direct2D to draw and work with timelines. These are for Windows Forms, and reside in the `Sce.Atf.Controls.Timelines.Direct2D` namespace and `Atf.Gui.WinForms` assembly.

There are both Direct2D and GDI versions of all the classes. You should use the much faster Direct2D versions, because the GDI versions are obsolete.

Timeline Renderers

The timeline renderers draw timeline objects on a canvas provided by a timeline control.

Contents

- [Timeline Renderers](#)
 - [D2dTimelineRenderer Class](#)
 - [D2dDefaultTimelineRenderer Class](#)
- [Timeline Controls](#)
- [Timeline Manipulators](#)
- [Putting It All Together: Using the Timeline Classes](#)
- [Obsolete Classes](#)

D2dTimelineRenderer Class

The abstract base class `D2dTimelineRenderer` renders the timeline and objects in it using Direct2D. It works with a timeline control, such as `D2dTimelineControl`.

`D2dTimelineRenderer` provides most of the timeline rendering capabilities:

- `Init()` method to initialize the renderer.
- Properties for various graphics objects used in rendering.
- `Dispose()` method for disposing of graphics objects.
- Numerous properties to configure the rendering, such as `TrackIndent` for track indentation and `HeaderWidth` for group and track header width.
- `Draw()` and `Print()` methods to draw the timeline to the display.
- Various private methods to draw timeline parts, such as sub-timelines, markers, groups, ghosts for moving and resizing objects, and so on.
- `Pick()` methods for getting a list of timeline objects hit in a rectangle.
- `GetBounds()` methods to find the bounds of timeline objects, such as intervals, keys, and markers.
- Abstract methods to draw particular objects, such as groups, tracks, and keys.
- Context class with useful information for laying out and drawing timeline elements, such as the `D2dTimelineRenderer` and `D2dGraphics` objects. This context is used as a parameter in several methods, such as the `Draw()` methods.

D2dDefaultTimelineRenderer Class

Because `D2dTimelineRenderer` is abstract, you must derive a class from it to produce an actual renderer, which is `D2dDefaultTimelineRenderer`'s role. A timeline application can use `D2dDefaultTimelineRenderer` or a class like it.

`D2dDefaultTimelineRenderer` has an `Init()` method to initialize the graphic objects it uses to draw timeline objects.

`D2dDefaultTimelineRenderer` also has properties to customize rendering, such as `TrackHeight` for the track height, relative to a unit of time.

`D2dDefaultTimelineRenderer` overrides the abstract `Draw()` methods in `D2dTimelineRenderer` to draw the following objects:

- Group
- Track
- Interval
- Key
- Marker

Timeline Controls

Timeline controls display the rendered timeline. There are two versions: `D2dTimelineControl` for Direct2D and `TimelineControl` for GDI. As already noted, you should use `D2dTimelineControl`.

`D2dTimelineControl` derives from `CanvasControl`, a control for viewing and editing a 2D bounded canvas. The constructor with the most arguments is:

```
public D2dTimelineControl(
    ITimelineDocument timelineDocument,
    D2dTimelineRenderer timelineRenderer,
    TimelineConstraints timelineConstraints,
    bool createDefaultManipulators)
```

All constructors require the `ITimelineDocument` argument. If `D2dTimelineRenderer` or `TimelineConstraints` are not supplied, default objects are constructed for them. The parameter `createDefaultManipulators` indicates whether default manipulators should be constructed for the control. For details about these manipulators, see [Timeline Manipulators](#).

`D2dTimelineControl` provides the Direct2D graphics device used for drawing in its `D2dGraphics` property. This value is used whenever drawing occurs.

`D2dTimelineControl` has a set of `Create()` methods to create these objects using an `ITimelineObjectCreator`:

- Group
- Track
- Interval
- Key
- Marker

It also has enumerator properties to get all instances of objects: `AllEvents`, `AllMarkers`, `AllTracks`, and `AllGroups`.

The control also handles events, such as for mouse actions and painting.

Timeline Manipulators

Manipulators allow you to manipulate timeline objects in a timeline control. These function somewhat like control adapters, in that they add capabilities to a control. However, they do not derive from `ControlAdapter` and the control they operate on, `D2dTimelineControl`, is not an `AdaptableControl`.

The manipulators all take a `D2dTimelineControl` parameter to attach them to that control, as for `D2dMoveManipulator`:

```
public D2dMoveManipulator(D2dTimelineControl owner)
```

This manipulator is created this way, for example, where `m_timelineControl` holds a `D2dTimelineControl`:

```
D2dMoveManipulator moveManipulator = new D2dMoveManipulator(m_timelineControl);
```

This table describes the Direct2D manipulators:

Manipulator	Description
<code>D2dMoveManipulator</code>	A timeline manipulator for moving <code>IEvent</code> objects, such as markers, keys, and intervals.
<code>D2dScaleManipulator</code>	Creates the scale manipulator, a horizontal bar on top of the scale bar. It has handles at each end, bracketing the selection set. It has two distinct modes: <ol style="list-style-type: none">1. Individual interval scaling occurs when the left or right border of an interval is scaled. In this case, the entire selection of intervals is scaled "in place". That is, their starting locations do not change. Only intervals are affected.2. If the handles on the scaling manipulator are dragged, the portion of the timeline that is bracketed by the first selected object to the last selected object is scaled in time. Everything selected is scaled--keys, markers, and intervals.
<code>D2dScrubberManipulator</code>	Creates the scrubber manipulator, a vertical bar that can slide left and right by grabbing the handle on top.

D2dSelectionManipulator	A timeline manipulator for implementing the selection logic. It should probably be attached first to the timeline control. The attachment is permanent and there must be one timeline control per ITimelineDocument .
D2dSnapManipulator	A timeline manipulator intended to be used by other manipulators to allow for snapping selected objects to non-selected events and give a visual indication of this.
D2dSplitManipulator	A timeline manipulator for splitting intervals into two. When hovering the mouse over an interval, if the user holds down a modifier key (Alt by default), the cursor changes to indicate where a split will take place. On the <code>MouseDown</code> event, the interval is split.

The order in which these manipulators are attached to the `D2dTimelineControl` matters. The order here determines the order of receiving Paint events and is the reverse order of receiving picking events. For example, a custom control that is drawn on top of everything else and that can be clicked on should come last in this list, so that it is drawn last and is picked first.

This is the recommended order in which manipulators should be constructed.

1. D2dSelectionManipulator
2. D2dMoveManipulator
3. D2dScaleManipulator
4. D2dSplitManipulator
5. D2dSnapManipulator
6. D2dScrubberManipulator

Putting It All Together: Using the Timeline Classes

Follow these steps to create a timeline with these classes.

1. Construct the `D2dDefaultTimelineRenderer`. Its constructor takes no parameters, so it is easy:

```
timelineRenderer = new D2dDefaultTimelineRenderer();
```

2. Create the `D2dTimelineControl`, as in this statement:

```
m_timelineControl = new D2dTimelineControl(null, m_renderer, new TimelineConstraints(),
false);
```

Recall that the first argument is an `ITimelineDocument`, which can be `null`. The second argument is a `D2dTimelineRenderer`; if `null`, a new `D2dDefaultTimelineRenderer` is constructed and used. The third argument is a `TimelineConstraints`, which can be constructed in-line. For more information, see [TimelineConstraints Class](#).

3. Set the `D2dTimelineControl`'s `TimelineDocument` property to the `ITimelineDocument`, if there is one.
4. Attach manipulators to the control by constructing them with the `D2dTimelineControl`, as in:

```
D2dMoveManipulator moveManipulator = new D2dMoveManipulator(m_timelineControl);
```

Construct manipulators in the recommended order. For more details, see [Timeline Manipulators](#).

5. Initialize the `D2dDefaultTimelineRenderer` with the `D2dGraphics` object from the `D2dTimelineControl`, as in this code. Here `renderer` contains the `renderer`, `document` is the `ITimelineDocument`, which has a `TimelineControl` property containing the timeline control:

```
renderer.Init(document.TimelineControl.D2dGraphics);
```

The `D2dGraphics` object is obtained from the `D2dGraphics` property of the control.

The `TimelineEditor` component and `TimelineDocument` class of the [ATF Timeline Editor Sample](#) perform these steps. For more information on how these controls are used, see [Timeline Editor Programming Discussion](#).

Obsolete Classes

The classes discussed here are the Direct2D versions. There is an older set of classes with the same kind of functionality that use GDI drawing instead. For example, `D2dTimelineRenderer` is the Direct2D version of the GDI `TimelineRenderer`. As elsewhere, the Direct2D version of the classes have the prefix "D2d".

The GDI classes are obsolete. You should use the much faster classes in the `Sce.Atf.Controls.Timelines.Direct2D` namespace.

Topics in this section

Links on this page to other pages

[ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [General Timeline Classes](#), [Timeline Editor Programming Discussion](#)

General Timeline Classes

Like the interfaces discussed in [Timeline Interfaces](#), these classes are UI-platform-agnostic and reside in the `Sce.Atf.Controls.Timelines` namespace and `Atf.Gui` assembly. These classes provide information needed for rendering and displaying timelines.

Timeline Constraint Classes

`TimelineConstraints` is an abstract class with methods that test whether certain constraints on timeline objects are met:

- `IsStartValid`: Test if a start value would be valid for the given event. It is used by `D2dMoveManipulator`.
- `IsLengthValid`: Test if a length would be valid for the given interval. It is used by `D2dMoveManipulator`.
- `IsIntervalValid`: Test if the interval would be valid if it shared a track with another interval. Any modification to the interval's start and length should not invalidate a previous test against a different interval, during a paste operation, for example. It is used by `D2dMoveManipulator`, `D2dScaleManipulator`, and `D2dTimelineControl`.

Contents

- [Timeline Constraint Classes](#)
- [EventMovedEventArgs Class](#)
- [Ghosting Information](#)
- [Hit Information](#)
- [Timeline Information Classes](#)
 - [TimelinePath Class](#)
 - [TimelineLayout Class](#)
 - [TimelineReferenceOptions Class](#)

A `TimelineConstraints` object is a parameter for the `D2dTimelineControl` constructor.

`DefaultTimelineConstraints` is ATF's default implementation of `TimelineConstraints`. It ensures that intervals don't overlap by either clipping the new interval or repositioning it to the right.

EventMovedEventArgs Class

`EventMovedEventArgs` provides event arguments describing an event — interval, key, or marker — that moved in the timeline. The parameters indicate the event, and its new starting position and track.

Ghosting Information

When timeline events move, they are "ghosted" to indicate they are being moved.

The `GhostType` enumeration indicates the type of motion:

- `Move`: moving.
- `ResizeLeft`: resizing left.
- `ResizeRight`: resizing right.

`GhostInfo` provides information for drawing ghosted timeline objects. This object wraps object information: the ghosted object, its target to snap to, and its start, length, and bounds.

The manipulators use `GhostType` and `GhostInfo` during their operation to display the right appearance for objects.

Hit Information

Hit information tells what objects were hit during manipulation of a timeline.

The `HitType` enumeration indicates the timeline object hit: key, marker, left or right resize part of an interval, and so on.

Hit information is used by the manipulators as well as the `D2dTimelineControl` during pick operations.

`HitRecord` encapsulates results of a hit testing operation and is created by manipulators. It contains the hit object and the `HitType`. It also contains the `TimelinePath` of the object. For more information, see [TimelinePath Class](#).

Timeline Information Classes

TimelinePath Class

TimelinePath derives from `AdaptablePath`, which represents a path in a tree or graph with the capability of adapting the last element of the path (the `Last` property) to a requested type. A `TimelinePath` is a sequence of `ITimelineObject` objects that define its position in the timeline: group, track, and event. The class defines several `+` operators to concatenate paths.

TimelineLayout Class

TimelineLayout associates a timeline object's `TimelinePath` with a bounding rectangle in screen space. A timeline object can appear multiple times within a `D2dTimelineControl` due to referenced sub-documents, including the possibility of the same timeline sub-document being referenced multiple times.

The class provides methods to add path-rectangle pairs, to obtain the rectangle for a given path, and to set a path's rectangle.

TimelineReferenceOptions Class

TimelineReferenceOptions contains options for `ITimelineReference` for user interfaces, and so on. Its `Expanded` property indicates whether or not the referenced timeline should have all its groups displayed on their own rows in the timeline control.

Topics in this section

Links on this page to other pages

[Authoring Tools Framework](#), [Timeline Interfaces](#)

Development, Debugging, and Testing

This section discusses techniques for developing, debugging, and testing applications developed using ATF.

- [Debugging the DOM with Visual Studio](#)
 - [Visual Studio Debugger Display Attributes and Other Features](#)
 - [Using the DomRecorder Component](#)
 - [General ATF Scripting Support](#)
 - [Scripting Applications with Python](#)
 - [ATF Test Code](#)
-

Debugging the DOM with Visual Studio

The `DomNode` class has the `_DebugInfo` property that gathers information about the node's contents and formats it so the information is easier to access. This illustration shows the property for a `DomNode`:

Name	Value	Type
this	DomNode hash code (unique per AppDomain) (0x30fe379, http://sony.com/gametech/circuits/1_0:groupType)	Sce.Atf.Dom.DomNode
_DebugInfo	{Additional debug info}	Sce.Atf.Dom.DomNode.I
AttributeChangedListeners	Count = 18	System.Collections.Generic.List`1
AttributeChangingListeners	Count = 0	System.Collections.Generic.List`1
Attributes	Count = 12	System.Collections.Generic.List`1
{name}	"Group"	Sce.Atf.Dom.DomNode.I
AttributeInfo	{name}	Sce.Atf.Dom.AttributeInfo
Value	"Group"	object (string)
{label}	"Group"	Sce.Atf.Dom.DomNode.I
{x}	"Group"	Sce.Atf.Dom.DomNode.I
{y}	"Group"	Sce.Atf.Dom.DomNode.I
{visible}	256	Sce.Atf.Dom.DomNode.I
{expanded}	160	Sce.Atf.Dom.DomNode.I
{showExpandedGroupPins}	true	Sce.Atf.Dom.DomNode.I
{autosize}	true	Sce.Atf.Dom.DomNode.I
{width}	0	Sce.Atf.Dom.DomNode.I
{height}	0	Sce.Atf.Dom.DomNode.I
{minwidth}	0	Sce.Atf.Dom.DomNode.I
{minheight}	0	Sce.Atf.Dom.DomNode.I
Raw View	0	Sce.Atf.Dom.DomNode.I
ChildInsertedListeners	Count = 18	System.Collections.Generic.List`1
ChildInsertingListeners	Count = 1	System.Collections.Generic.List`1
[0]	DomNodeAdapter	Sce.Atf.Dom.DomNode.I
Method	CircuitEditorSample.MasteringValidator on 0x1cafc69, http://sony.com/	System.Reflection.MethodInfo
Target	{Void DomNodeChildInserting(System.Object, Sce.Atf.Dom.ChildEventArgs)}	Sce.Atf.Dom.DomNode.I
Raw View	{CircuitEditorSample.MasteringValidator or 0x1cafc69, http://sony.com/}	object (CircuitEditorSam
ChildRemovedListeners	Count = 19	System.Collections.Generic.List`1
ChildRemovingListeners	Count = 1	System.Collections.Generic.List`1
Children	Count = 8	System.Collections.Generic.List`1
{input}	0x2322c20, http://sony.com/gametech/circuits/1_0:groupPinType	Sce.Atf.Dom.DomNode.C
{input}	0xd2d1c1b9, http://sony.com/gametech/circuits/1_0:groupPinType	Sce.Atf.Dom.DomNode.C
Child	0xd2d1c1b9, http://sony.com/gametech/circuits/1_0:groupPinType	Sce.Atf.Dom.DomNode.C
ChildInfo	{input}	Sce.Atf.Dom.ChildInfo
{output}	0x2df0fd93, http://sony.com/gametech/circuits/1_0:groupPinType	Sce.Atf.Dom.DomNode.C
{output}	0x13562fa, http://sony.com/gametech/circuits/1_0:groupPinType	Sce.Atf.Dom.DomNode.C

Contents

- DomNode Event Listeners
- DomNode Attributes
- Child DomNodes
- DomNode Extensions
- Enhancing Visual Studio

Note that the value of the `DomNode` itself is a hash code along with the `DomNodeType`. Hash codes are the equivalent of unique IDs for the current debugging session; they are unique to the `AppDomain`, the application domain that is an isolated environment where applications execute.

This `_DebugInfo` property contains several categories of information, shown in the figure. You can also open nodes under `_DebugInfo` in the debugger to get more information on the displayed data.

DomNode Event Listeners

You can find out what objects receive any of the six `DomNode` events listed under `_DebugInfo`:

- `AttributeChanged` under `AttributeChangedListeners`.
- `AttributeChanging` under `AttributeChangingListeners`
- `ChildInserted` under `ChildInsertedListeners`
- `ChildInserting` under `ChildInsertingListeners`
- `ChildRemoved` under `ChildRemovedListeners`
- `ChildRemoving` under `ChildRemovingListeners`

Listener objects are listed in the order in which they receive events.

DomNode Attributes

The **Attributes** node lists all the attributes' names and values:

- `AttributeInfo`'s string value for the name.
- `GetAttribute()`'s string value for the value.

The open "{name}" node in the figure also shows the name and value. If there is no "local" value, then the default value is displayed.

Child DomNodes

The **Children** node displays the `DomNode` children's names and values, if any:

- `ChildInfo`'s string value for the name.
- `Child DomNode`'s string value for the value.

DomNode Extensions

All extension defined for the `DomNode`'s type are also listed, as shown in this figure:

Name	Value	Type
>this	{0x1f4122e, http://sony.com/gametech/circuits/1_0:groupType}	Sce.Atf.Dom.DomNode
_DebugInfo	{Additional debug info}	Sce.Atf.Dom.DomNode.E
AttributeChangedListeners	Count = 18	System.Collections.Gene
AttributeChangingListeners	Count = 0	System.Collections.Gene
Attributes	Count = 12	System.Collections.Gene
ChildInsertedListeners	Count = 18	System.Collections.Gene
ChildInsertingListeners	Count = 1	System.Collections.Gene
ChildRemovedListeners	Count = 19	System.Collections.Gene
ChildRemovingListeners	Count = 1	System.Collections.Gene
Children	Count = 12	System.Collections.Gene
Extensions	Count = 4	System.Collections.Gene
[0]	{CircuitEditorSample.Module on 0x1f4122e, http://sony.com/gametech/ object (CircuitEditorSam}	object (CircuitEditorSam}
[1]	{CircuitEditorSample.CircuitEditingContext on 0x1f4122e, http://sony.co object (CircuitEditorSam}	object (CircuitEditorSam}
[2]	{CircuitEditorSample.Group on 0x1f4122e, http://sony.com/gametech/c object (CircuitEditorSam}	object (CircuitEditorSam}
[3]	{Sce.Atf.Controls.Adaptable.Graphs.ViewingContext on 0x1f4122e, http:/ object (Sce.Atf.Controls./	object (Sce.Atf.Controls./
Raw View		

extensions / adapters / decorators
(are almost always DomNodeAdapters)

The value has three parts:

- The extension name, which is nearly always a DOM adapter.
- The `HashCode` of the `DomNode`.
- The `DomNodeType` of the `DomNode`.

Enhancing Visual Studio

The information under `_DebugInfo` is provided by marking the classes used with special attributes. You can use these techniques to enhance the display of your own classes and data. For details, see [Visual Studio Debugger Display Attributes and Other Features](#).

Topics in this section

Links on this page to other pages

[Visual Studio Debugger Display Attributes and Other Features](#)

Visual Studio Debugger Display Attributes and Other Features

You can enhance debugging in Visual Studio by using debugger display attributes and other facilities in C#. This section shows how the `DomNode` information display, described in [Debugging the DOM with Visual Studio](#), was implemented using these features.

Overriding `ToString()` Method

If you override this method for an object, its return value is displayed in Visual Studio in the object's Value column.

The `DomNode` class takes advantage of this to provide more information about a node:

```
public override string ToString()
{
    if (m_type != null)
        return string.Format("0x{0:x}, {1}", GetHashCode(), m_type);

    return base.ToString();
}
```

This override method makes the Visual Studio debugger display the `DomNode`'s hash code and `DomNodeType`:

- `GetHashCode()` returns the object's hash code, generated by `System.GetHashCode()`.
- The `m_type` field contains the `DomNode`'s `DomNodeType`.

Defining Properties to Display Information

You can define properties as convenient holders for all the information you want to display about a class. The `DomNode` class does this with its `_DebugInfo` property:

```
private DomNodeDebugger _DebugInfo
{
    get { return new DomNodeDebugger(this); }
}
```

The method simply returns a newly constructed `DomNodeDebugger` object. This property begins with `_` so it appears first in the list of `DomNode`'s properties. This property adds no additional capability to the class, nor does it change the class's operation.

The `DomNodeDebugger` class is simply a package for all the information that's useful to see while debugging the DOM. It consists of a set of properties containing this useful data:

```
// The properties of this class are designed to appear in the Visual Studio debugger in
// a useful way. For example, IList<> is more useful than IEnumerable<> in the debugger view.
private class DomNodeDebugger
{
    public DomNodeDebugger(DomNode node)
    {
        m_node = node;
    }

    public IList<AttributeDebugger> Attributes
    {
        get { return m_node.Type.Attributes.Select(attrInfo => new AttributeDebugger(attrInfo, m_node
            .GetAttribute(attrInfo))).ToList(); }
    }

    public IList<ChildDebugger> Children
    {
        get { return m_node.Children.Select(child => new ChildDebugger(child.ChildInfo, child))
            .ToList(); }
    }

    public IList<object> Extensions
    {
        get
```

```
{  
    var extensions = new List<object>();  
    for( int i = m_node.Type.FirstExtensionIndex; i < m_node.Type.FieldCount; i++)  
        extensions.Add(m_node.m_data[i]);  
    return extensions;  
}  
}  
  
public IList<ListenerDebugger> AttributeChangingListeners  
{  
    get { return m_node.GetAttributeChangingHandlers().Select(listener => new  
ListenerDebugger(listener)).ToList(); }  
}  
  
public IList<ListenerDebugger> AttributeChangedListeners  
{  
    get { return m_node.GetAttributeChangedHandlers().Select(listener => new ListenerDebugger  
(listener)).ToList(); }  
}  
  
public IList<ListenerDebugger> ChildInsertingListeners  
{  
    get { return m_node.GetChildInsertingHandlers().Select(listener => new ListenerDebugger(listener  
)).ToList(); }  
}  
  
public IList<ListenerDebugger> ChildInsertedListeners  
{  
    get { return m_node.GetChildInsertedHandlers().Select(listener => new ListenerDebugger(listener  
)).ToList(); }  
}  
  
public IList<ListenerDebugger> ChildRemovingListeners  
{  
    get { return m_node.GetChildRemovingHandlers().Select(listener => new ListenerDebugger(listener  
)).ToList(); }  
}  
  
public IList<ListenerDebugger> ChildRemovedListeners  
{  
    get { return m_node.GetChildRemovedHandlers().Select(listener => new ListenerDebugger(listener  
)).ToList(); }  
}  
  
public override string ToString()  
{  
    return "Additional debug info";  
}
```

```

    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly DomNode m_node;
}

```

`DomNodeDebugger` consists of a set of properties containing `DomNode` information. Note that the property getters return an `IList<>`, because that displays much more nicely in the debugger than an `{[IEnumerable<>]}`. Each property returns an `IList<>` of another special debugging class that is a data container.

Simply having the `_DebugInfo` property that displays the contents of the `DomNodeDebugger` class is a major improvement over the default view of `DomNode` in the debugger. However, this display can be enhanced even further with debugger display attributes. For details, see [Debugger Display Attributes](#).

Note that `DomNodeDebugger` overrides `ToString()` to create an explanatory object label, which appears as the `DomNodeDebugger` object's value, as described in [Overriding ToString\(\) Method](#).

Debugger Display Attributes

Visual Studio has a set of attribute classes whose purpose is to format information for displaying in the debugger. For a description of these classes, see the MSDN article [Enhancing Debugging with the Debugger Display Attributes](#).

The `DomNodeDebugger` class described in [Defining Properties to Display Information](#) uses these attributes to better present information.

DebuggerDisplayAttribute Class

`DebuggerDisplayAttribute` controls what is displayed for the attributed object. Its constructor's argument is a string that is displayed in the value column for instances of the attributed type. Thus, it functions similarly to `ToString()`. If you have overridden `ToString()`, you do not need to use `DebuggerDisplayAttribute`. If you use both, the `DebuggerDisplayAttribute` attribute takes precedence over `ToString()`.

In addition, `DebuggerDisplayAttribute` has a `Name` property whose value is displayed in the Name column.

For instance, `DomNodeDebugger.Attributes` returns an `IList<AttributeDebugger>`, and the container class `AttributeDebugger` is attributed:

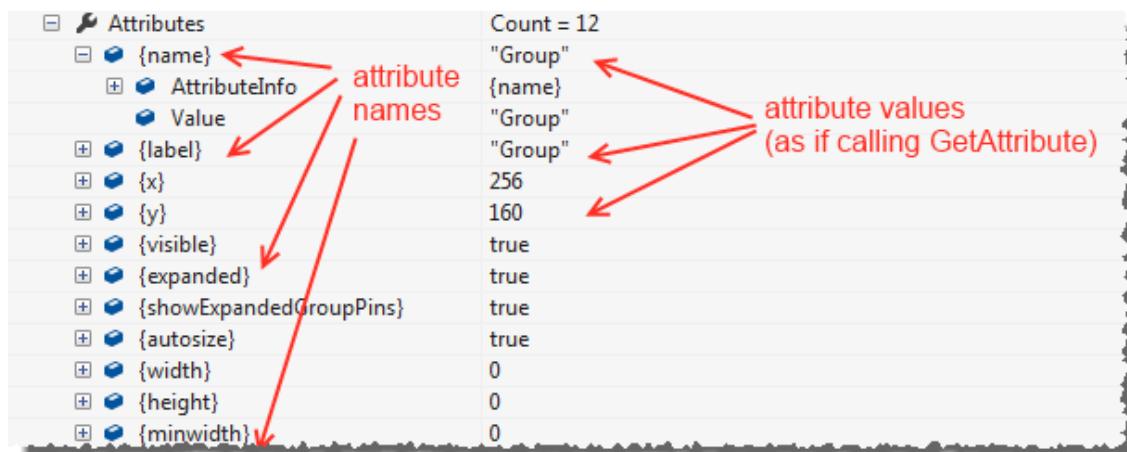
```

[DebuggerDisplay("{Value}", Name = "{AttributeInfo}")]
private class AttributeDebugger
{
    public AttributeDebugger(AttributeInfo info, object value)
    {
        AttributeInfo = info;
        Value = value;
    }

    public readonly AttributeInfo AttributeInfo;
    public readonly object Value;
}

```

`AttributeDebugger` is a container for `AttributeInfo` for the `DomNode`. Setting the `Name` property indicates that the string value of `AttributeInfo` appears in the Name column. Making the constructor's parameter `{Value}` means that the attribute's value appears in the Value column. This illustration shows the attribute display produced by this class using this attribute:



The opened "name" node shows that the contents of the Name and Value columns do indeed match `AttributeInfo` and `Value`.

DebuggerBrowsableAttribute Class

`DebuggerBrowsableAttribute` specifies how a field or property is displayed in the debugger. Its constructor takes one of the `DebuggerBrowsableState` enumeration values:

- `Never`: Don't display this attributed member in the debugger.
- `RootHidden`: The member is not shown, but its constituent objects are displayed if it is an array or collection.
- `Collapsed`: The member is shown but not expanded in the display.

Here's an example of how the `ChildDebugger` class that displays `DOMNode` child information uses attributes:

```
[DebuggerDisplay("{Child}", Name = "{ChildInfo}")]
private class ChildDebugger
{
    public ChildDebugger(ChildInfo info, DomNode child)
    {
        ChildInfo = info;
        Child = child;
    }

    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    public readonly ChildInfo ChildInfo; //is visible inside the child; no need to show it at top-level.
    [DebuggerBrowsable(DebuggerBrowsableState.RootHidden)]
    public readonly DomNode Child; //no need to show "Child" property; go straight to DomNode members.
}
```

First, `ChildDebugger` uses the `DebuggerDisplay` attribute similarly to `AttributeDebugger` to show `ChildInfo` for the name and the child as the value.

Next, `DebuggerBrowsableAttribute` marks the fields for the values used to restrict their display.

To see what this attribute does, here's the `DOMNodeDebugger.Children` node display with the attributes set:

Children	Count = 0x0000001c
{module}	{0x3717a0e, http://sony.com/gametech/circuits/1_0:buttonType}
_DebugInfo	{Additional debug info}
Ancestry	{Sce.Atf.Dom.DomNode.get_Ancestry}
ChildInfo	{module}
base	{module}
IsList	true
m_isList	true
m_rules	null
m_type	{http://sony.com/gametech/circuits/1_0:moduleType}
Rules	{Sce.Atf.EmptyEnumerable<Sce.Atf.Dom.ChildRule>.Enumerable}
Type	{http://sony.com/gametech/circuits/1_0:moduleType}
Children	{Sce.Atf.Dom.DomNode.get_Children}
LevelSubtree	{Sce.Atf.Dom.DomNode.get_LevelSubtree}
Lineage	{Sce.Atf.Dom.DomNode.get_Lineage}
m_childInfo	{module}
m_data	{object[0x00000007]}
m_eventHandlers	null
m_parent	{0x15b0aa4, http://sony.com/gametech/circuits/1_0:circuitDocumentType}
m_type	{http://sony.com/gametech/circuits/1_0:buttonType}
Parent	{0x15b0aa4, http://sony.com/gametech/circuits/1_0:circuitDocumentType}
Subtree	{Sce.Atf.Dom.DomNode.get_Subtree}
Type	{http://sony.com/gametech/circuits/1_0:buttonType}
Static members	
{module}	{0x12c8c3e, http://sony.com/gametech/circuits/1_0:andType}
{module}	{0x2275ee, http://sony.com/gametech/circuits/1_0:lightType}
{module}	{0x10a-220-414-10-000000000000}

Here is the display without `DebuggerBrowsableAttribute`:

Children	Count = 0x0000001c
{module}	{0xe6edf, http://sony.com/gametech/circuits/1_0:buttonType}
Child	{0xe6edf, http://sony.com/gametech/circuits/1_0:buttonType}
_DebugInfo	{Additional debug info}
Ancestry	{Sce.Atf.Dom.DomNode.get_Ancestry}
ChildInfo	{module}
Children	{Sce.Atf.Dom.DomNode.get_Children}
LevelSubtree	{Sce.Atf.Dom.DomNode.get_LevelSubtree}
Lineage	{Sce.Atf.Dom.DomNode.get_Lineage}
m_childInfo	{module}
m_data	{object[0x00000007]}
m_eventHandlers	null
m_parent	{0x142fce8, http://sony.com/gametech/circuits/1_0:circuitDocumentType}
m_type	{http://sony.com/gametech/circuits/1_0:buttonType}
Parent	{0x142fce8, http://sony.com/gametech/circuits/1_0:circuitDocumentType}
Subtree	{Sce.Atf.Dom.DomNode.get_Subtree}
Type	{http://sony.com/gametech/circuits/1_0:buttonType}
Static members	
ChildInfo	{module}
base	{module}
IsList	true
m_isList	true
m_rules	null
m_type	{http://sony.com/gametech/circuits/1_0:moduleType}
Rules	{Sce.Atf.EmptyEnumerable<Sce.Atf.Dom.ChildRule>.Enumerable}
Type	{http://sony.com/gametech/circuits/1_0:moduleType}
{module}	{0x2897d57, http://sony.com/gametech/circuits/1_0:andType}
{module}	{0x22b0865, http://sony.com/gametech/circuits/1_0:lightType}

Note the differences under the opened "module" node. Without `DebuggerBrowsableAttribute` on the `Child` field, an extra node `Child` appears, rather than just showing the child node information. Without `DebuggerBrowsableAttribute` on the `ChildInfo` field, there's also an additional `ChildInfo` node, which is redundant, because that node is already in the child's displayed information.

Contents

- Overriding `ToString()` Method
- Defining Properties to Display Information
- Debugger Display Attributes
 - `DebuggerDisplayAttribute` Class
 - `DebuggerBrowsableAttribute` Class

Topics in this section

Links on this page to other pages

[Authoring Tools Framework](#), [Debugging the DOM with Visual Studio](#)

Using the DomRecorder Component

The `DomRecorder` component records DOM events on the active context and displays them in a list. These logged events can also be retrieved programmatically.

Importing the DomRecorder Component

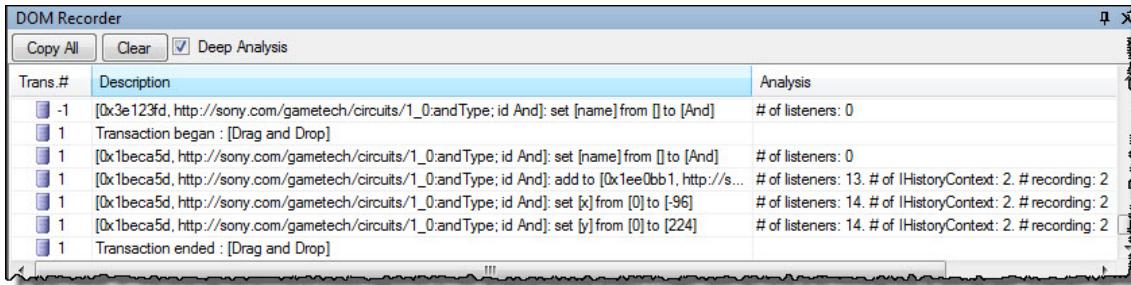
Because it's a MEF component, it's simple to use `DomRecorder`. Add it to the `TypeCatalog` with your other MEF components. For information on how to do this, see [How MEF is Used in ATF](#). The [ATF Circuit Editor Sample](#) uses this component.

Contents

- Importing the DomRecorder Component
- DomRecorder User Interface
- Events Recorded
- DomRecorder API
- Event Deep Analysis
- DomRecorder Example

DomRecorder User Interface

DomRecorder adds a DOM Recorder window to the application, as seen in this figure:



The control buttons do the following:

- Copy All: Copy the text of the selected events to the clipboard.
- Clear: Clear the list of events.
- Deep Analysis: Check to get more event information that may indicate problems.

The window contains an ordered list of DOM events in a `TreeListView`. The columns are:

- Trans.#: Transaction number, which is -1 if the event is not part of a transaction.
- Description: Event description, showing information depending on the event. For details, see [Events Recorded](#).
- Analysis: Description of the event, which is formatted when the Deep Analysis box is checked. For an explanation, see [Event Deep Analysis](#).

Events may be individually selected; press Shift to select a range of events.

Events Recorded

Changes to all `DomNode` trees and transaction events are recorded. Appropriate information is recorded for the events. The event examples are from the [ATF Circuit Editor Sample](#).

Event	Log format
<code>DomNode.DiagnosticAttributeChanged</code>	<code>[<Modified DomNode>; id <DomNode ID or hash code>]: set [<Attribute name>] from [<Old value>] to [<New value>]. For example: [0x3717a0e, http://sony.com/gametech/circuits/1_0:buttonType; id Button_3]: set [name] from [] to [Button_3]</code>

DomNode.DiagnosticChildInserted	[<Child DomNode>; id <Child DomNode ID or hash code>]: add to [<Parent DomNode>; id <Parent DomNode ID or hash code>] at index <index>. For example: [0x3717a0e, http://sony.com/gametech/circuits/1_0:buttonType ; id Button_3]: add to [0x15b0aa4, http://sony.com/gametech/circuits/1_0:circuitDocumentType ; id 15B0AA4] at index 0
DomNode.DiagnosticChildRemoved	[<Child DomNode>; id <Child DomNode ID or hash code>]: remove from [<Parent DomNode>; id <Parent DomNode ID or hash code>] at index <index>. For example: [0x1a23c51, http://sony.com/gametech/circuits/1_0:subCircuitInstanceType ; id MasterInstance_2]: remove from [0x15b0aa4, http://sony.com/gametech/circuits/1_0:circuitDocumentType id 15B0AA4] at index 6
IValidationContext.Beginning	Transaction began : [<Transaction name>]. For example: Transaction began : [Drag and Drop]
IValidationContext.Cancelled	Transaction cancelled : [<Transaction name>]. For example: Transaction cancelled : [Delete]
IValidationContext.Ended	Transaction ended : [<Transaction name>]. For example: Transaction ended : [Drag Items]

DomRecorder API

You can access the log created by a `DomRecorder` object through its properties and methods:

- `IEnumerable<string> GetLogEvents(int maxEvents)`: Get the most recent `maxEvents` log events, from oldest to newest.
- `int NumLogEvents`: Get the number of events in the log.
- `void Clear()`: Clear all the log data and refresh the Control.
- `IDocumentRegistry DocumentRegistry`: Get or set the optional document registry. If present, when a document is closed, all the DOM event data is cleared to prevent memory leaks.
- `IContextRegistry ContextRegistry`: Get or set the context registry to be used to automatically find the root `DomNode` and validation context. This is optional and can be `null`.
- `IValidationContext ValidationContext`: Gets or sets the validation context that is used to record when transactions begin and end. This is optional and can be `null`. It is set automatically if the context registry raises the `ActiveContextChanged` event, and the new context can be adapted to `IValidationContext`.

Event Deep Analysis

Each event can have zero or more listeners, that is, objects that subscribe to the event and have event handlers that are called when the event occurs. For instance, DOM adapters defined for application data types often listen to DOM change events. DOM adapters may well be the bulk of listeners.

When the Deep Analysis box is checked, `DomRecorder` analyzes the listeners for the `DomNode` events. It reports the following information and possible problems on a list of event handlers for a DOM change that has just taken place:

- "# of listeners": Number of event listeners. If this number decreases, it could indicate that changes are not being observed that should be. If this number is higher than expected, perhaps unwanted listeners are hurting performance.
- "DUPLICATE LISTENER: <event handler method name> on <type of listener>". Are there any duplicate listeners? That is, is the same event handler being called from two different objects, indicating duplicate effort is occurring? This could happen if there were two instances of the same listener. This reports only the last duplicate found; there may be more than one.
- "# of IHistoricContext": Number of listeners that implement `IHistoricContext`. If there are no such listeners, the change is not being recorded for undo/redo.
- "# recording": Number of listeners deriving from the ATF class `HistoryContext` that are recording changes. If recording is not enabled, then the change is not automatically added to an undo/redo command. This isn't necessarily an error. For example, the change could be temporary, such as when dragging a circuit element on a canvas. Or perhaps client code manually adds a command to `CommandHistory`.

Deep analysis can be time-consuming, but can be useful when debugging event handling.

Here is a sample analysis message:

```
# of listeners: 13. DUPLICATE LISTENER: DomNode_ChildInserted on Sce.Atf.Dom.Observer. # of IHistoricContext : 2. # recording: 2
```

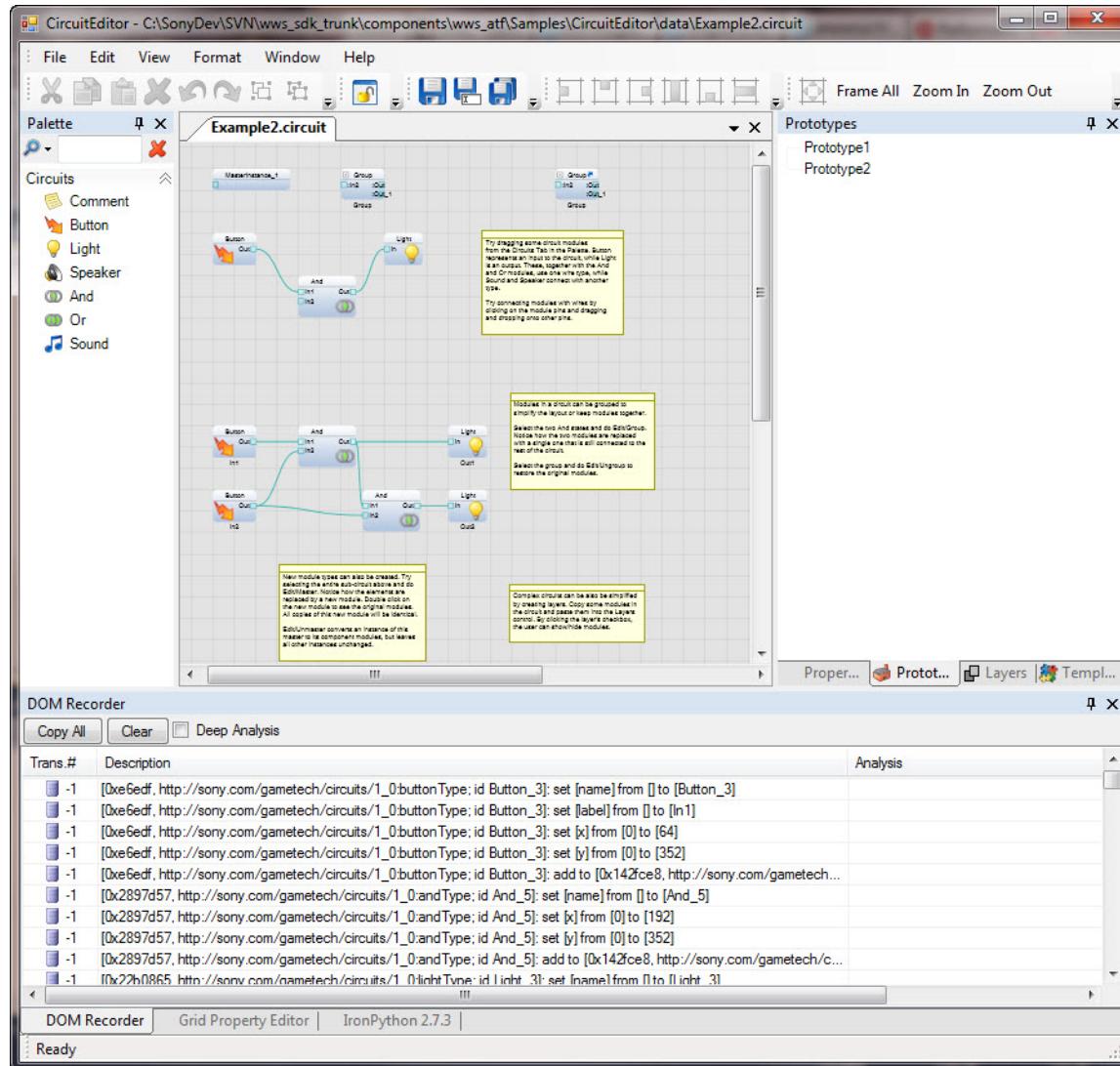
Because it is time-consuming, Deep Analysis is disabled by default. If you need to analyze events that occur during application start up, you can modify the `m_analysisEnabled` field to set its original value to `true`:

```
private bool m_analysisEnabled = true;
```

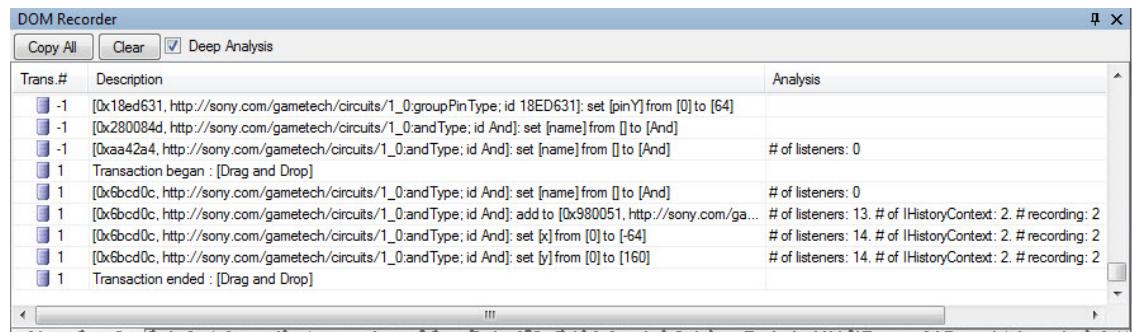
In general, you do not need to do this. Most listeners are probably DOM adapters, and they do not start listening until after they have all been initialized, which does not occur until the application has fully initialized.

DomRecorder Example

The [ATF Circuit Editor Sample](#) imports `DomRecorder`. The illustration shows an opened circuit file and the DOM events recorded and displayed in the DOM Recorder window:



None of these events are part of a transaction, and no transaction events are recorded; the `Trans.#` column is all -1. After an And gate is dragged onto the circuit, a set of events enclosed by a transaction is recorded:



Note that the transaction number is now 1. Four events are bounded by a transaction named "Drag and Drop" that all refer to the same added `DOMNode`:

1. The DomNode's "name" attribute is set to "And": [0x6bcd0c, http://sony.com/gametech/circuits/1_0:andType; id And]: set [name] from [] to [And]. # of listeners: 0
2. The DomNode is added as a child to another DomNode of the "circuitDocumentType": [0x6bcd0c, http://sony.com/gametech/circuits/1_0:andType; id And]: add to [0x980051, http://sony.com/gametech/circuits/1_0:circuitDocumentType; id 980051] at index 12. # of listeners: 13. # of IHistoricContext: 2. # recording: 2
3. The DomNode's "x" attribute is set to "-64" to reflect the dragged And gate's canvas x-coordinate: [0x6bcd0c, http://sony.com/gametech/circuits/1_0:andType; id And]: set [x] from [0] to [-64]. # of listeners: 14. # of IHistoricContext: 2. # recording: 2
4. The DomNode's "y" attribute is set to "160" to reflect the dragged And gate's canvas y-coordinate: [0x6bcd0c, http://sony.com/gametech/circuits/1_0:andType; id And]: set [y] from [0] to [160]. # of listeners: 14. # of IHistoricContext: 2. # recording: 2

Note that there are no listeners until the DomNode is actually added.

Topics in this section

Links on this page to other pages

[ATF Circuit Editor Sample](#), [Authoring Tools Framework](#), [How MEF is Used in ATF](#)

General ATF Scripting Support

ATF provides facilities to script your applications. Scripts allow access to C# objects in the application's classes. Scripts have many uses. , For example, you can create suites of tests for applications.

This section describes the general scripting facilities. As far as a specific scripting language goes, ATF provides support for Python. For details on Python scripting facilities, see [Scripting Applications with Python](#).

Support Classes

ATF's general scripting support comes in the following classes:

- **ScriptingService**: Base class for any scripting service.
- **ScriptConsole**: Scripting console window.
- **AtfScriptVariables**: Set script variables for common ATF services.
- **AutomationService**: Use .NET remoting service for scripts.

Contents

- [Support Classes](#)
- [Scripting Components](#)
- [ScriptingService Class](#)
- [ScriptConsole Component](#)
- [AtfScriptVariables Component](#)
- [AutomationService Component](#)

Scripting Components

Nearly all the samples support scripting, for Python in particular. Thus, most of the samples have the following entries in their MEF TypeCatalog:

```
typeof(PythonService), // scripting service for automated tests
typeof(ScriptConsole), // provides a dockable command console for entering
Python commands
typeof(AtfScriptVariables), // exposes common ATF services as script variables
typeof(AutomationService) // provides facilities to run an automated script using
the .NET remoting service
```

ScriptingService Class

ScriptingService is an abstract base class for services (MEF components, for example) that can expose C# objects to a scripting language. This is the fundamental scripting class that specific scripting languages services derive from.

ScriptingService offers the following methods for working with scripts:

- **LoadAssembly()**: Load the specified assembly into script domain. This adds all the namespaces in the assembly to the script domain, so that you can import types (such as classes) from these namespaces. Many samples use this to load the application's assembly, as in:

```
m_scriptingService.LoadAssembly(GetType().Assembly);
```
- **ImportType()**: Imports the given type from the given namespace.
- **ImportAllTypes()**: Import all the types from the given namespace. Nearly all the samples use this to import the sample's types. For instance, this statement is part of the [ATF Circuit Editor Sample](#) `Editor` component's initialization:

```
m_scriptingService.ImportAllTypes("CircuitEditorSample");
```
- **TryGetVariable()**: Try to get the given variable from the scripting domain. This allows obtaining the variable's value.
- **SetVariable()**: Make the given C# variable accessible to script. Note that this does not set the value of the variable.
- **RemoveVariable()**: Remove the given C# variable from the script domain.
- **ExecuteStatement()**: Execute a single script statement. The statement does not need a carriage return at the end. For example, the [ATF State Chart Editor Sample](#) calls this method in its `Editor` class to import types from a class:

```
m_scriptingService.ExecuteStatement("from Sce.Atf.Controls.Adaptable.Graphs import *");
```

- `ExecuteStatements()`: Execute multiple script statements. Each statement should end with a carriage return.
- `ExecuteFile()`: Execute the statements in the given file.
- `SetEngine()`: Set the scripting engine for a given language. You must set this engine prior to using any scripts. The engine is a `Microsoft.Scripting.Hosting.ScriptEngine` object. This class represents a language in the Hosting API and is the Hosting API counterpart for `Microsoft.Scripting.Hosting.ScriptEngine.LanguageContext`.
- `LoadDefaultAssemblies()`: Load the initial assemblies into the given `Microsoft.Scripting.Hosting.ScriptRuntime`. This class represents a Dynamic Language Runtime in the Hosting API. It is the Hosting API counterpart for `Microsoft.Scripting.Runtime.ScriptDomainManager`. `SetEngine` calls `LoadDefaultAssemblies()` to load assemblies for the script engine runtime.

ScriptConsole Component

ScriptConsole provides a dockable command console for entering script commands. It uses a `ConsoleTextBox` for a text box for a console. `ConsoleTextBox` derives from `System.Windows.Forms.TextBox` and also implements `IConsoleTextBox`. `ScriptConsole` is the control host client for the `ConsoleTextBox`.

The `IConsoleTextBox` contains useful properties and methods:

- `Prompt`: Get or set the command prompt. The default prompt is "`>>>`".
- `CommandHandler`: Set the command handler delegate that is called when a command is entered.
- `Clear()`: Clear the console window.
- `EnterCommand()`: Enter a command programmatically. The result is equivalent to manually typing the command and pressing the `Enter` key. The command doesn't need to terminate with a newline.
- `Write()`: Write the specified string to the console window.
- `WriteLine()`: Write the specified string to the console window, appending a newline character.
- `Control`: Get the underlying control for the console, which is a `ConsoleTextBox`.

ScriptConsole needs a `ScriptingService`, which it either imports or is specified in its constructor when MEF is not used. ScriptConsole has a private `ProcessCommand()` method to process console commands. `ProcessCommand()` uses the `ScriptingService` to process the command. `IConsoleTextBox.CommandHandler` is set to `ProcessCommand()`.

AtfScriptVariables Component

AtfScriptVariables exposes common ATF services as script variables, so that the imported `ScriptingService` can easily use these ATF services.

In addition to the `ScriptingService`, AtfScriptVariables imports numerous common ATF services, such as the Control Host Service:

```
[Import(AllowDefault = true)]
private IControlHostService m_controlHostService = null;
```

These services are all imported with `AllowDefault = true` in case the service is not used by the application.

A variable is associated with each service using the `ScriptingService.SetVariable()` method:

```
if (m_controlHostService != null)
    m_scriptingService.SetVariable("atfControls", m_controlHostService);
```

script variables and services are listed in this table:

Script variable	Service
atfControls	IControlHostService
atfCommands	ICommandService
atfSelect	StandardSelectionCommands
atfFile	StandardFileCommands
atfFileExit	StandardFileExitCommand
atfEdit	StandardEditCommands
atfHistory	StandardEditHistoryCommands

atfContextReg	IContextRegistry
atfDocReg	IDocumentRegistry
atfDocService	IDocumentService
atfPropertyEditor	PropertyEditor

AutomationService Component

AutomationService provides facilities to run an automated script using the .NET remoting service. You do not need this component to run scripts locally.

The constructor for AutomationService checks whether AutomationService command line arguments were provided for the application:

- -automation: Use AutomationService.
- -port: Use this port number.

AutomationService tries to import LiveConnectService, which is used for communication if available.

Topics in this section

Links on this page to other pages

[ATF Circuit Editor Sample](#), [ATF State Chart Editor Sample](#), [Scripting Applications with Python](#)

Scripting Applications with Python

Python scripting support is furnished by the `BasicPythonService` and `PythonService` components. You can use either one.

BasicPythonService Component

`BasicPythonService` provides the Python scripting engine and imports many common .NET and ATF types into the Python namespace. `BasicPythonService` is derived from `ScriptingService`, so it provides a scripting service for Python.

Contents

- [BasicPythonService Component](#)
- [PythonService Component](#)
- [Using Python Scripts](#)
- [Writing Python Scripts](#)

You can use `BasicPythonService` by itself to execute Python scripts. However, consider using `ScriptConsole` and `AtfScriptVariables` as additional MEF components to provide a console and set script variables for common ATF services. For information on these components, see [General ATF Scripting Support](#).

`BasicPythonService` overrides `ImportAllTypes()` and `ImportType()` to provide the right forms for Python.

`BasicPythonService`'s constructor is:

```
[ImportingConstructor]
public BasicPythonService()
{
    ScriptEngine engine = CreateEngine();
    SetEngine(engine);
    Initialize();
}
```

`CreateEngine()` gets the `ScriptEngine` by calling `IronPython.Hosting.Python.CreateEngine()`. `ScriptingService.SetEngine()` then sets the `ScriptEngine`.

`Initialize()` initializes the Python engine with some common assembly imports. It does so by creating a string with a list of Python import statements and then calls `ScriptingService.ExecuteStatements()` to perform the import. The following namespaces are always imported:

- `System`
- `System.Drawing`
- `System.Collections.Generic`
- `System.Collections.ObjectModel`
- `System.Windows.Forms`
- `System.Text`
- `System.IO`
- `System.Xml.Schema`
- `System.Xml.XPath`
- `System.Xml.Serialization`
- `Sce.Atf`
- `Sce.Atf.Applications`
- `Sce.Atf.VectorMath`
- `Sce.Atf.Adaptation`
- `Sce.Atf.Dom`

`Initialize()` also looks through the list of all assemblies loaded in the current `AppDomain` and conditionally imports additional namespaces based on that list. `Initialize()` examines the full assembly name and does the following:

If full assembly name begins with...	Then...
"Atf." or "Scea."	Load the assembly by calling <code>ScriptingService.LoadAssembly()</code>
"Atf.Gui.WinForms"	Import namespaces "Sce.Atf.Controls" and "Sce.Atf.Controls.Adaptable"

"Scea.Core"	Import namespace "SceaEditorsHostInternal"
"Scea.Dom"	Import namespace "Scea.Dom"

PythonService Component

PythonService provides a dockable command console for entering Python commands. If no command console is needed, use the BasicPythonService MEF component.

PythonService derives from BasicPythonService, so if you use PythonService, you don't need to import BasicPythonService too. You can also use BasicPythonService without PythonService. However, all the samples import PythonService for the convenience of having a console.

PythonService tries to import ScriptConsole, and if none was imported, it instantiates a new ScriptConsole.

For greatest flexibility, you should also import:

- ScriptConsole
- AtfScriptVariables
- AutomationService

All the samples that use MEF import all of these, as noted in [Scripting Components](#).

Using Python Scripts

To use Python scripts with your application, do the following:

1. Import the scripting components into your application. As previously noted, it is easiest to import all the scripting components described in [Scripting Components](#).
2. Import ScriptingService wherever you want to expose variables, as in this code from the [ATF Simple DOM Editor Sample](#):

```
[Import(AllowDefault = true)]
private ScriptingService m_scriptingService = null;
```

3. Call the appropriate ScriptingService methods. For example, the [ATF Simple DOM Editor Sample](#) sample does this in its Editor component:

```
if (m_scriptingService != null)
{
    // load this assembly into script domain.
    m_scriptingService.LoadAssembly(GetType().Assembly);
    m_scriptingService.ImportAllTypes("SimpleDomEditorSample");
    m_scriptingService.SetVariable("editor", this);
}
```

Calling LoadAssembly() here adds the namespaces in the assembly to the script domain. One of these namespaces in SimpleDomEditorSample, and the next line imports all its types. Finally, the variable editor is set to correspond to this, the Editor class, so all its methods and properties are exposed to the script using this variable.

4. Write Python scripts using the available variables.

Writing Python Scripts

Topics in this section

Links on this page to other pages

[ATF Simple DOM Editor Sample](#), [General ATF Scripting Support](#)

ATF Code Samples Discussions

ATF samples show how to use ATF to accomplish various tasks using ATF technology. This section overviews them and also discusses in depth how their programming employs ATF.

These samples' complexity varies greatly. Here's a suggested order for studying them, simplest first, most complicated last:

- File Explorer
- Target Manager
- Code Editor
- Tree List Editor
- Model Viewer
- Using Dom
- Property Editing
- Simple DOM Editor and its variants:
 - Simple DOM Editor
 - Simple DOM No XML Editor
 - WinForms and WPF Apps
- DOM Tree Editor
- Diagram Editor
- Timeline Editor
- FSM Editor
- State Chart Editor
- Circuit Editor

These topics discuss in some detail how the samples' programming uses ATF.

- [Circuit Editor Programming Discussion](#): Learn how ATF handles graphs, and provides editors for kinds of graphs, such as circuits.
- [Code Editor Programming Discussion](#): Shows how to interface third party software to an ATF application: the ActiproSoftware SyntaxEditor.
- [Diagram Editor Programming Discussion](#): Very simply combines the Circuit Editor, Fsm Editor, and State Chart Editor components into one application, with the abilities of all three, showing the power of components.
- [DOM Tree Editor Programming Discussion](#): Shows how to edit DOM data using a tree control and display properties in a variety of value editors.
- [File Explorer Programming Discussion](#): Discusses the [ATF File Explorer Sample](#) using list and tree controls with adapters.
- [FSM Editor Programming Discussion](#): Tells you about how the [ATF FSM Editor Sample](#) edits simple graphs for state machines, using DOM adapters for contexts and validation.
- [Model Viewer Programming Discussion](#): Shows how the [ATF Model Viewer Sample](#) is written, discussing how ATGI and Collada model data is handled, using rendering components, and using a `DesignControl` as a canvas for rendering.
- [Property Editing Programming Discussion](#): Description of the [ATF Property Editing Sample](#)'s programming, including customizing property editors, setting up the editing context, creating application data, and using the property editor components.
- [Simple DOM Editor Programming Discussion](#): Programming the [ATF Simple DOM Editor Sample](#), creating a palette, using DOM adapters and contexts, editing application data, and searching `DomNodes`.
- [Simple DOM No XML Editor Programming Discussion](#): Programming the [ATF Simple DOM No XML Editor Sample](#), which is very similar to [ATF Simple DOM Editor Sample](#), except that it doesn't use XML for either its data model or persisting application data.
- [State Chart Editor Programming Discussion](#): Shows using ATF graph and other classes to create a statechart editor, using DOM adapters, documents, contexts, and validators.
- [Target Manager Programming Discussion](#): Description of how a Target Manager is implemented using ATF components to manage target devices, such as Vita or PS3 consoles. A Target Manager is used in other tools, such as the StateMachine tool.
- [Timeline Editor Programming Discussion](#): Discusses how to create a fairly full-featured timeline editor using the ATF timeline facilities, such as the timeline renderer and the timeline control and its manipulators.
- [Tree List Editor Programming Discussion](#): Demonstrates how to use the ATF tree controls `TreeListView` and its enhancement, `TreeListViewEditor`. `TreeListView` uses `TreeListViewAdapter`, which adapts `TreeListView` to display data in a tree.
- [Using Dom Programming Discussion](#): Shows how to use the various parts of the ATF DOM: an XML Schema, a schema metadata class file generated by DomGen, DOM adapters for the data types, a schema loader, and saving application data to an XML file.
- [WinForms and WPF Apps Programming Discussion](#): Overview of programming in both [ATF Win Forms App Sample](#) and [ATF Wpf App Sample](#), which are simplified versions of the [ATF Simple DOM Editor Sample](#).

Circuit Editor Programming Discussion

The [ATF Circuit Editor Sample](#) allows constructing circuits by dragging circuit items onto a canvas and connecting their pins. You can edit the circuit, including grouping items, as well as creating reusable prototypes and templates from groups of elements. The editor allows collections of items to be added to layers, and the visibility of of layers and their individual items can be toggled.

This sample makes heavy use of the graph interfaces and classes in the `Sce.Atf.Controls.Adaptable.Graphs` namespace, especially the circuit related ones.

- For all topics related to graph support in ATF, see [Graphs in ATF](#).
- For a general description of circuits, see [Circuit Graphs](#).
- For details on circuit support, see [Circuit Graph Support](#).

Programming Overview

Circuit Editor shows how to use the graph interfaces and classes in the `Sce.Atf.Controls.Adaptable.Graphs` namespace. A circuit is simply a generalization of a graph, and this sample implements the `IGraph` interface with appropriate parameters for circuits.

Graph support requires the ATF DOM, and this application defines its data model using an XML schema, which makes it easier to support the DOM and also makes it easy to persist data in XML.

Much of the function of Circuit Editor comes from the DOM adapters defined for most of the application data types. ATF provides a particularly rich set of DOM adapters for circuit graphs, and Circuit Editor needs to do very little in some cases to derive from these classes.

The `Editor` class handles circuit editing and is the document client for circuit documents. This client looks very similar to the document client for the [ATF Simple DOM Editor Sample](#). `Editor` is also the control host client for a `D2dAdaptableControl` control. Circuits are displayed using the `D2dAdaptableControl`, employing Direct2D for GPU-accelerated rendering. A set of control adapters is created for each `D2dAdaptableControl` that perform many functions, including rendering, view changes, and selection.

The `CircuitEditingContext` is the editing context for containers, such as circuits and groups. Other contexts handle layering and prototypes.

Circuit Editor contains several windows, mainly to handle ways of grouping and reusing controls: Prototypes, Templates, and Layers. Templates and Layers can have folders, which are handled as types with their own DOM adapters. Circuit Editor also provides a Mastering facility for reusable items. Some of these facilities also need their own contexts and command clients.

Contents

- Programming Overview
- Graphs and Circuits
- Circuit Editor Data Model
 - Main Types
 - Relationships Between Data Types
 - Circuit XML Data
- DOM Adapters
 - Functions of DOM Adapters
- Circuit Documents
- Circuit Document Display and Editing
 - Circuit Document Client
 - Circuit Document Display and Control Adapters
- Context Classes
 - CircuitEditingContext Class
 - LayeringContext and PrototypingContext Classes
 - TemplatingContext Class
- Reusable Circuit Facilities and Windows
 - Prototype Handling
 - Template Handling
 - Master (Subcircuit) Handling
 - Layer Handling
- Miscellaneous Classes
 - CircuitValidator Class
 - GroupingCommands Component
 - ModulePlugin Component
 - PrintableDocument Class

Graphs and Circuits

A circuit is a graph, and so this sample uses the graphical classes in the `Sce.Atf.Controls.Adaptable.Graphs` namespace. These classes include ones specifically tailored to circuit graphs, and these classes do most of the work of the sample application. In fact, about half the classes in Circuit Editor are derived from classes of the same name in `Sce.Atf.Controls.Adaptable.Graphs`, such as `Circuit`, `CircuitEditingContext`, and `Pin`. In this discussion, the class in Circuit Editor is referred to by its name, such as `Circuit`; the class it derives from in `Sce.Atf.Controls.Adaptable.Graphs` is referred to as the base class, to avoid having to specify the namespace each time.

In the most general form, ATF supports graphs using nodes, edges, and routes in its `IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>` interface:

```
public interface IGraph<out TNode, out TEdge, out TEdgeRoute>
    where TNode : class, IGraphNode
    where TEdge : class, IGraphEdge<TNode, TEdgeRoute>
    where TEdgeRoute : class, IEdgeRoute
```

where the elements are:

- `IGraphNode`: Interface for a node in a graph; nodes are connected by edges.
- `IGraphEdge<TNode, TEdgeRoute>`: Interface for routed edges that connect nodes and have a defined source and destination route from and to the nodes.
- `IEdgeRoute`: Interface for edge routes, which act as sources and destinations for graph edges.

The general `Sce.Atf.Controls.Adaptable.Graphs.Circuit` class is specific to circuits and uses this variant of the `IGraph` interface:

```
public abstract class Circuit : DomNodeAdapter, IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram
, ICircuitContainer
```

The parameters in the `IGraph<Element, Wire, ICircuitPin>` interface are:

- `Element`: Adapts `DomNode` to a circuit element with pins. Implements `ICircuitElement`, which implements `IGraphNode`.
- `Wire`: Adapts `DomNode` to a connection in a circuit. Implements `IGraphEdge<Element, ICircuitPin>`.

- **ICircuitPin**: Interface for pins, which are the sources and destinations for wires between Elements.

Finally, the Circuit Editor's **Circuit** class, derived from the above, uses its own **IGraph** variant:

```
public class Circuit : Sce.Atf.Controls.Adaptable.Graphs.Circuit, IGraph<Module, Connection, ICircuitPin>
```

The parameters in the **IGraph<Module, Connection, ICircuitPin>** interface are:

- **Module**: Adapter for circuit modules. Derives from **Element**.
- **Connection**: Adapter for connections in a circuit. Derives from **Wire**.
- **ICircuitPin**: Interface for pins.

For more information about **IGraph** and other graph interfaces, see [ATF Graph Interfaces](#).

Circuit Editor Data Model

Circuit Editor defines its data model with an XML Schema in the **Circuit.xsd** type definition file. The DomGen tool is used to generate a **Schema** class containing metadata classes for the types, as in many of the samples, like [ATF Simple DOM Editor Sample](#). Circuit Editor also defines a **SchemaLoader** class that uses these metadata classes in a variety of ways, to define DOM adapters, for instance. For general information about data modeling in graphs, see [Graph Data Model](#).

Main Types

Most of Circuit Editor's types have associated classes that are DOM adapters. Thus each item in a circuit is represented by a **DomNode** in a tree of **DomNodes** in the application data. The key circuit item classes are:

- **Module ("moduleType")**: Represents all the circuit modules found on the palette, such as AND gates. Derives from **Sce.Atf.Controls.Adaptable.Graphs.Element**.
- **Connection ("connectionType")**: A connection between two module pins. Derives from **Sce.Atf.Controls.Adaptable.Graphs.Wire**.
- **Pin ("pinType")**: A pin on a module. Derives from **Sce.Atf.Controls.Adaptable.Graphs.Pin**.
- **Circuit ("circuitType")**: A circuit, containing modules, connections among them, and annotations. Derives from **Sce.Atf.Controls.Adaptable.Graphs.Circuit**.
- **Group ("groupType")**: Collection of modules, connections between them, input and output pins (group pins), and annotations. Derives from **Sce.Atf.Controls.Adaptable.Graphs.Group**.

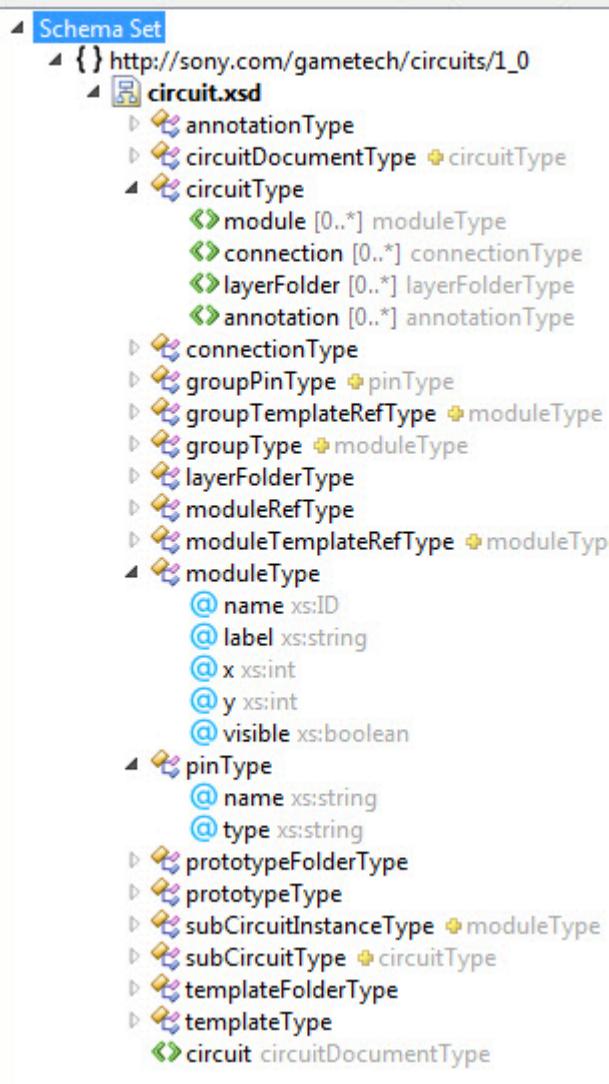
Note that these classes all derive from classes in the **Sce.Atf.Controls.Adaptable.Graphs** namespace.

Circuit and **Group** both behave as circuit containers, and each can be displayed separately in a window.

Relationships Between Data Types

All the sample's data types are shown in this figure from the Visual Studio XML Schema Explorer, which shows that some types are extensions of others:

Type here to search



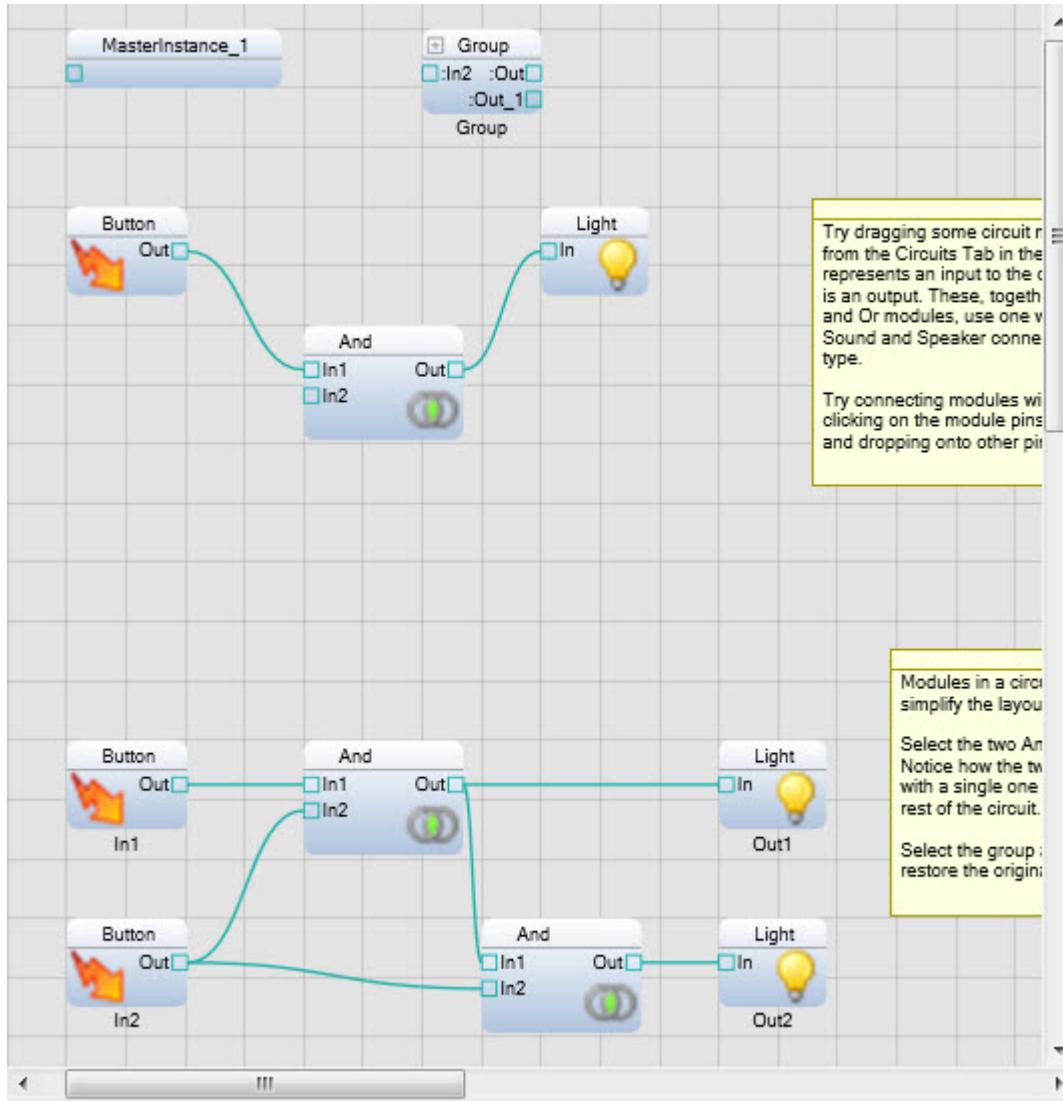
Several types are based on the "circuitType", "pinType", and "moduleType" types, whose nodes are open to show their attributes.

In this list, the second level entries are listed under the type they extend:

- annotationType: comment on the circuit canvas.
- circuitType: circuit, containing modules, connections among them, and annotations.
 - circuitDocumentType: document holding subcircuits (masters), prototype folders, and template folders.
 - subCircuitType: circuit that defines a master module type, with custom input and output pins and can be referenced by instances, which are all equivalent.
- connectionType: connects the output pin on the output module to the input pin of the input module.
- layerFolderType: hierarchy containing layers.
- moduleRefType: reference to a module in the circuit, used to build lists of module references in a layer.
- moduleType: module with name, label, and position, and can be referenced by connections.
 - groupTemplateRefType: reference instance to a group.
 - groupType: collection of modules, internal connections between them, input and output pins (group pins), and annotations.
 - moduleTemplateRefType: reference instance to a module.
 - subCircuitInstanceType: module whose type is defined by a subcircuit (master).
- pinType: pin with name and type that determines connection type.
 - groupPinType: pin on a group, preserving the internal pin/module which is connected to the outside circuit.
- prototypeFolderType: hierarchy containing prototypes.
- prototypeType: set of modules and connections among them that can be copied and pasted into a circuit.
- templateFolderType: hierarchy containing templates.
- templateType: module that can be referenced in a circuit.

Circuit XML Data

Examining the XML in a circuit file saved from the application illustrates the circuit data types. Here is a circuit graph in Circuit Editor:



Here's its .circuit file, with some data omitted (...) to clarify the data hierarchy:

```

<circuit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs=
"http://www.w3.org/2001/XMLSchema" xmlns="http://sony.com/gametech/circuits/1_0">
  <module xsi:type="buttonType" name="Button_3" label="In1" x="64" y="352" />
  <module xsi:type="andType" name="And_5" x="192" y="352" />
  ...
  <module xsi:type="groupType" name="Group_1" label="Group" x="768" y="128"
showExpandedGroupPins="true">
    <input name=":In2" type="boolean" module="And_9" pin="1" />
    <output name=":Out" type="boolean" module="Button_9" />
    <output name=":Out_1" type="boolean" module="And_9" index="1" pinY="64" />
    <module xsi:type="buttonType" name="Button_9" />
    <module xsi:type="andType" name="And_9" x="128" y="64" />
    <module xsi:type="lightType" name="Light_9" x="256" />
    <connection outputModule="And_9" inputModule="Light_9" />
    <connection outputModule="Button_9" inputModule="And_9" />
  </module>
  <connection outputModule="Button_3" inputModule="And_5" />
  <connection outputModule="And_5" inputModule="Light_3" />
  ...
  <layerFolder name="Layer1">
    <layerFolder name="Layer2">
      <moduleRef ref="Button_3" />
      <moduleRef ref="And_5" />
      ...
    </layerFolder>
    <moduleRef ref="Light_7" />
    <moduleRef ref="And_7" />
    <moduleRef ref="Button_7" />
  </layerFolder>

```

```
<annotation text="Try dragging some circuit modules from the Circuits Tab in the  
Palette."  
...  
<annotation text="Complex circuits can be also be simplified by creating layers. Copy  
some  
...  
<subCircuit name="MasterInstance_1">  
    <module xsi:type="lightType" name="Light_2" x="272" y="16" />  
    <module xsi:type="andType" name="And_2" x="144" y="80" />  
...  
</subCircuit>  
<prototypeFolder>  
    <prototype name="Prototypel">  
        <module xsi:type="buttonType" name="Button_5" label="In1" x="64" y="352" />  
        <module xsi:type="andType" name="And_3" x="192" y="352" />  
        ...  
        <connection outputModule="Button_5" inputModule="And_3" />  
        <connection outputModule="And_3" inputModule="Light_5" />  
        ...  
    </prototype>  
    <prototype name="Prototype2">  
        <module xsi:type="lightType" name="Light_8" x="320" y="64" />  
        <module xsi:type="andType" name="And_8" x="192" y="128" />  
        ...  
        <connection outputModule="And_8" inputModule="Light_8" />  
        <connection outputModule="Button_8" inputModule="And_8" />  
    </prototype>  
</prototypeFolder>  
<templateFolder name="_TemplateRoot_">  
    <template guid="852d657f-c7ef-4474-933f-743737be1b59" label="Group">  
        <module xsi:type="groupType" name="Group" label="Group" x="96" y="96"  
showExpandedGroupPins="true">  
            <input name=":In2" type="boolean" module="And_1" pin="1" />  
            <output name=":Out" type="boolean" module="Button_1" />  
            ...  
            <module xsi:type="buttonType" name="Button_1" />  
            <module xsi:type="andType" name="And_1" x="128" y="64" />  
            ...  
            <connection outputModule="Button_1" inputModule="And_1" />  
            <connection outputModule="And_1" inputModule="Light_1" />  
        </module>
```

```

</template>
</templateFolder>
</circuit>

```

The main container is the "circuitDocument" type. Its name is "circuit" and it contains all the other items. In this case, the circuitDocument contains these type items:

- module
- connection
- layerFolder
- annotation
- subCircuit
- prototypeFolder
- templateFolder

DOM Adapters

As previously mentioned, nearly all the types in Circuit Editor have DOM adapters, and most of these adapters are defined in `SchemaLoader.RegisterCircuitExtensions()`. The following table lists the type, its DOM adapter, interfaces the adapter implements, the base class the adapter derives from in `Sce.Atf.Controls.Adaptable.Graphs` (unless indicated otherwise), and any interfaces this base class implements plus the class the base derives from, if any.

Type	DOM Adapter Class	Implements	Derives from	Base class derives from, implements
annotationType	Annotation		Annotation	IAnnotation
circuitType	Circuit	IGraph<Module, Connection, ICircuitPin>	Circuit	IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram, ICircuitContainer
circuitType	CircuitEditing-Context	IEditableGraphContainer<Module, Connection, ICircuitPin>	CircuitEditingContext	EditingContext, IEnumerableContext, INamingContext, IInstancingContext, IObservableContext, IColoringContext, IEditableGraphContainer<Element, Wire, ICircuitPin>
connectionType	Connection	IGraphEdge<Module, ICircuitPin>	Wire	IGraphEdge<Element, ICircuitPin>
groupPinType	GroupPin	ICircuitGroupPin<Module>	GroupPin	Pin, ICircuitGroupPin<Element>
groupTemplateRefType	GroupInstance	ICircuitGroupType<Module, Connection, ICircuitPin>, IReference<Module>, IReference<DomNode>, IReference<Sce.Atf.Controls.Adaptable.Graphs.Group>	CircuitEditorSample.Module	
groupType	Group	ICircuitGroupType<Module, Connection, ICircuitPin>, IGraph<Module, Connection, ICircuitPin>	Group	Element, ICircuitGroupType<Element, Wire, ICircuitPin>, IGraph<Element, Wire, ICircuitPin>, IAnnotatedDiagram, ICircuitContainer

layerFolderType	LayerFolder		LayerFolder	
moduleRefType	ModuleRef		ElementRef	
moduleTemplateRefType	ModuleInstance	IReference<Module>, IReference<DomNode>	CircuitEditorSample.Module	
moduleType	Module		Element	ICircuitElement, IVisible
pinType	Pin		Pin	ICircuitPin
prototypeFolderType	PrototypeFolder		PrototypeFolder	
prototypeType	Prototype		Prototype	
subCircuitType	SubCircuit		SubCircuit	Circuit, ICircuitElementType
subCircuitInstanceType	SubCircuit-Instance		SubCircuitInstance	Element
templateFolderType	TemplateFolder		Sce.Atf.Dom.TemplateFolder	
templateType	Template		Sce.Atf.Dom.Template	IReference<DomNode>

All the derived from classes are in `Sce.Atf.Controls.Adaptable.Graphs`, except as noted. All these base classes derive directly from `DomNodeAdapter`, except for `CircuitEditingContext`, which derives from `EditingContext`, which ultimately derives from `DomNodeAdapter`. For more information on the DOM adapter base classes, see [Circuit DOM Adapters](#).

Functions of DOM Adapters

What do these DOM adapters do?

Override Base Class Properties

DOM adapters typically implement properties containing information about the object.

For instance, the `Annotation` DOM adapter derives from `Sce.Atf.Controls.Adaptable.Graphs.Annotation`, which defines these properties:

```
// required DOM attributes info
protected abstract AttributeInfo TextAttribute { get; }
protected abstract AttributeInfo XAttribute { get; }
protected abstract AttributeInfo YAttribute { get; }
protected abstract AttributeInfo WidthAttribute { get; }
protected abstract AttributeInfo HeightAttribute { get; }
protected abstract AttributeInfo BackColorAttribute { get; }
```

These properties provide the kind of information needed for an annotation, such as its location and text. These properties are all overridden in `Annotation`, typically using the data in the DOM metadata classes defined in `Schema`, as in:

```
protected override AttributeInfo TextAttribute
{
    get { return Schema.annotationType.textAttribute; }
}
```

Nearly all Circuit Editor's DOM adapters override base class properties in this way. Half the adapters, such as `Annotation`, `LayerFolder`, `Module`, and `Pin`, do nothing more than this.

Call OnNodeSet() Method

Every DOM adapter has an `OnNodeSet()` method. Most of the DOM adapters in Circuit Editor use their base class's `OnNodeSet()`, but some of the DOM adapters provide their own, such as `Group`:

```

protected override void OnNodeSet()
{
    m_modules = new DomNodeListAdapter<Module>(DomNode, Schema.groupType.moduleChild);
    m_connections = new DomNodeListAdapter<Connection>(DomNode, Schema.groupType.connectionChild
);
    m_annotations = new DomNodeListAdapter<Annotation>(DomNode, Schema.groupType.annotationChild
);
    m_inputs = new DomNodeListAdapter<GroupPin>(DomNode, Schema.groupType.inputChild);
    m_outputs = new DomNodeListAdapter<GroupPin>(DomNode, Schema.groupType.outputChild);
    m_thisModule = DomNode.Cast<Module>();

    base.OnNodeSet();
}

```

The types used in this method are all the types that can be children of a "groupType" DomNode. This OnNodeSet() method saves a DomNodeListAdapter list of all the DomNode's child objects. The Circuit DOM adapter does the same for its child objects in its OnNodeSet() method.

Implement Interfaces

For example, the Circuit DOM adapter implements IGraph:

```

#region IGraph Members

/// <summary>
/// Gets all visible nodes in the circuit</summary>
///<remarks>IGraph.Nodes is called during circuit rendering, and picking(in reverse order).
</remarks>
IEnumerable<Module> IGraph<Module, Connection, ICircuitPin>.Nodes
{
    get
    {
        return m_modules.Where(x => x.Visible);
    }
}

/// <summary>
/// Gets all connections between visible nodes in the circuit</summary>
IEnumerable<Connection> IGraph<Module, Connection, ICircuitPin>.Edges
{
    get
    {
        return m_connections.Where(x => x.InputElement.Visible && x.OutputElement.Visible);
    }
}
...

```

Circuit Documents

Circuit Editor's document class is `CircuitDocument`, which merely overrides the `SubCircuitChildInfo` property in its derived class `Sce.Atf.Controls.Adaptable.Graphs.CircuitDocument`. This base class, in turn, derives from `DomDocument`, which implements `IDocument`. This base class is also a DOM adapter, and it implements an `OnNodeSet()` method to initialize the editor type and other information.

Several samples implement `IDocument` through `DomDocument`, including [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), and [ATF State Chart Editor Sample](#). For details on how the ATF Simple DOM Editor Sample uses `DomDocument`, see [EventSequenceDocument Class](#), which is the document class in that sample.

Circuit Document Display and Editing

The `Editor` class handles circuit editing and is the document client for circuit documents. `Editor` is also the control host client for a `D2dAdaptableControl` control, which displays circuit documents.

Circuit Document Client

Circuit Editor's implementation of `IDocumentClient` has a lot of similarities to the document client of [ATF Simple DOM Editor Sample](#), which is discussed in [Document Handling](#).

The implementations of `IDocumentClient` are very similar in the two samples. For example, here are the `Circuit Editor Info` property and `CanOpen()` method:

```
public DocumentClientInfo Info
{
    get { return EditorInfo; }
}

/// <summary>
/// Document editor information for circuit editor</summary>
public static DocumentClientInfo EditorInfo =
    new DocumentClientInfo(Localizer.Localize("Circuit"), ".circuit", null, null);

/// <summary>
/// Returns whether the client can open or create a document at the given URI</summary>
/// <param name="uri">Document URI</param>
/// <returns>True iff the client can open or create a document at the given URI</returns>
public bool CanOpen(Uri uri)
{
    return EditorInfo.IsCompatibleUri(uri);
}
```

And the same property and method for Simple DOM Editor:

```
public DocumentClientInfo Info
{
    get { return DocumentClientInfo; }
}

/// <summary>
/// Information about the document client</summary>
public static DocumentClientInfo DocumentClientInfo = new DocumentClientInfo(
    Localizer.Localize("Event Sequence"),
    new string[] { ".xml", ".esq" },
    Sce.Atf.Resources.DocumentImage,
    Sce.Atf.Resources.FolderImage,
    true);

/// <summary>
/// Returns whether the client can open or create a document at the given URI</summary>
/// <param name="uri">Document URI</param>
/// <returns>True iff the client can open or create a document at the given URI</returns>
public bool CanOpen(Uri uri)
{
    return DocumentClientInfo.IsCompatibleUri(uri);
}
```

The `IDocumentClient.Open()` method also has many commonalities in the two samples. For instance, both use `DomXmlReader` to read the file and create a tree of `DomNodes`, because both samples use the ATF DOM for data persistence. Both set up an editing context derived from the `EditingContext` class. A notable difference is that Circuit Editor's `Open()` method creates a `D2dAdaptableControl` to display circuits, as discussed in [Circuit Document Display and Control Adapters](#).

The `IDocumentClient.Show()`, `IDocumentClient.Save()`, and `IDocumentClient.Close()` methods are all very similar in the two samples. Both samples' `Save()` methods use `DomXmlWriter` to write the document to XML, although Circuit Editor does this through a private class `CircuitWriter` derived from `DomXmlWriter`.

Circuit Document Display and Control Adapters

Circuits are displayed using a `D2dAdaptableControl` employing Direct2D for GPU-accelerated rendering. A set of control adapters is created for each `D2dAdaptableControl`.

Control adapters reside in several namespaces. Some are not circuit or even graph specific, such as `HoverAdapter`.

Although a typical circuit control has more than a dozen control adapters, each adapter has well defined, unique responsibilities. The adapters' code is relatively short and easy to follow and extend. Divide and conquer is a good strategy here. Note that:

- Viewing related operations (pan and zoom) use `ITransformAdapter`, implemented in `TransformAdapter`.

- Mouse click and drag selection use `ISelectionAdapter`, implemented in `SelectionAdapter`.
- Rendering, picking and selection is delegated to a circuit rendering class, such as `D2dCircuitRenderer`, which is a parameter in the `D2dGraphAdapter` constructor. For information on circuit renderers, see [Circuit Rendering Classes](#).

Client applications can mostly use the standard control adapters provided in ATF, occasionally extending a few adapters. Circuit Editor uses the adapters as is.

A `D2dAdaptableControl` is used as a canvas for displaying circuits in:

- A circuit document.
- A circuit displayed in a tab after double-clicking on a group or master instance.
- A circuit displayed when hovering over a master instance.

The first two `D2dAdaptableControl` controls are in tab windows, are fully editable, and are created by the `CreateCircuitControl()` method. The `D2dAdaptableControl` for the hover is created in `Editor`'s constructor:

```
m_d2dHoverControl = new D2dAdaptableControl();
m_d2dHoverControl.Dock = DockStyle.Fill;
var xformAdapter = new TransformAdapter();
xformAdapter.EnforceConstraints = false; //to allow the canvas to be panned to view negative
coordinates
m_d2dHoverControl.Adapt(xformAdapter, new D2dGraphAdapter<Module, Connection, ICircuitPin>(m_circuitRenderer
, xformAdapter));
m_d2dHoverControl.DrawingD2d += new EventHandler(m_d2dHoverControl_DrawingD2d);
```

Here the `D2dAdaptableControl` is created and configured, adapters are created and configured, and then `AdaptableControl.Adapt()` sets the `D2dAdaptableControl`'s adapters. This process is also performed in `CreateCircuitControl()` with many more adapters.

`Editor`'s constructor also creates two objects that are used later in the `D2dAdaptableControl`'s adapters created in `CreateCircuitControl()` to render circuits:

- `D2dCircuitRenderer`: Graph renderer that draws graph nodes as circuit elements, and edges as wires. Elements have zero or more output pins where wires originate, and zero or more input pins where wires end. Input pins are on the left side and output pins are on the right of elements. For more information, see [D2dCircuitRenderer Class](#).
- `D2dSubCircuitRenderer`: Subgraph (group or master) renderer that draws subnodes as circuit elements, and subedges as wires. Also draws virtual representations of group pins for editing. For more details, see [D2dSubCircuitRenderer Class](#).

The rest of this section explores the `CreateCircuitControl()` method, because `D2dAdaptableControl` is central to the sample's operation.

For general information on adaptation, including adapting controls, see [Adaptation in ATF](#) and [Adaptable Controls](#).

Control Adapter Creation and Configuration

After the `D2dAdaptableControl` is created and configured, its adapters are created. These adapters all derive from `ControlAdapter`, the base class for control adapters.

```

internal D2dAdaptableControl CreateCircuitControl(DomNode circuitNode)
{
    var control = new D2dAdaptableControl();
    control.SuspendLayout();
    control.BackColor = SystemColors.ControlLight;
    control.AllowDrop = true;

    var transformAdapter = new TransformAdapter();
    transformAdapter.EnforceConstraints = false; //to allow the canvas to be panned to view
negative coordinates
    transformAdapter.UniformScale = true;
    transformAdapter.MinScale = new PointF(0.25f, 0.25f);
    transformAdapter.MaxScale = new PointF(4, 4);
    var viewingAdapter = new ViewingAdapter(transformAdapter);
    viewingAdapter.MarginSize = new Size(25, 25);
    var canvasAdapter = new CanvasAdapter();
    ((ILayoutConstraint) canvasAdapter).Enabled = false; //to allow negative coordinates for
circuit elements within groups

    var autoTranslateAdapter = new AutoTranslateAdapter(transformAdapter);
    var mouseTransformManipulator = new MouseTransformManipulator(transformAdapter);
    var mouseWheelManipulator = new MouseWheelManipulator(transformAdapter);
    var scrollbarAdapter = new ScrollbarAdapter(transformAdapter, canvasAdapter);

    var hoverAdapter = new HoverAdapter();
    hoverAdapter.HoverStarted += control_HoverStarted;
    hoverAdapter.HoverStopped += control_HoverStopped;

    var annotationAdaptor = new D2dAnnotationAdapter(m_theme); // display annotations under
diagram

```

These adapters allow the control to be used in all sorts of ways:

- **TransformAdapter**: implement `ITransformAdapter` for scaling and translating the control's contents, and is also needed by several other adapters. This adapter is configured by setting its properties. `EnforceConstraints` is false to allow the canvas to be panned to view negative coordinates. `UniformScale` is set true to guarantee that the canvas is scaled the same on both the x- and y-axis. `MinScale` and `MaxScale` enforce a minimum and maximum scaling. `MarginSize` indicates the margin used when framing a selection.
- **ViewingAdapter**: implement methods in `IViewingContext` to frame the control's items and ensure their visibility.
- **CanvasAdapter**: allow setting the canvas bounds and visible window size. Setting `Enabled` false allows negative coordinates for circuit elements within groups.
- **MouseTransformManipulator**: convert mouse drags into translation and scaling the canvas using the `TransformAdapter`. This allows panning items on the canvas with Alt+Left button mouse dragging and scaling with Alt+Right button mouse dragging.
- **MouseWheelManipulator**: convert mouse wheel rotation into scaling using the `TransformAdapter`.
- **ScrollbarAdapter**: add horizontal and vertical scrollbars to the adapted control, using the `TransformAdapter` and `CanvasAdapter`.
- **HoverAdapter**: add hover events over pickable items. This allows seeing the items inside a master element in a window when hovering over the master. The adapter subscribes to `HoverStarted` and `HoverStopped` events.
- **D2dAnnotationAdapter**: display and edit annotations on the control.

In addition, the following adapters are used and instantiated in the call to `AdaptableControl.Adapt()`:

- **RectangleDragSelector**: allow user to drag-select rectangular regions on the adapted control to modify the selection.
- **KeyboardIOGraphNavigator**: navigate an "input-output" graph using the arrow keys. This kind of graph has inputs on only one side of a node and outputs on another side, like a circuit.
- **LabelEditAdapter**: edit element labels in place.
- **SelectionAdapter**: add mouse click and drag selection of canvas items. The context must be convertible to `ISelectionContext`.
- **DragDropAdapter**: add drag and drop support for dragging circuit elements onto the canvas from the palette, prototype window, or template window.
- **ContextMenuAdapter**: support a context menu on right button mouse-up on circuit elements.

Circuit Display

`CreateCircuitControl()`'s `DomNode` parameter indicates what kind of circuit is being displayed: `Circuit` for an open circuit document, or `Group` if a group was double-clicked on. The control functions slightly differently in these cases, defining different additional adapters. For a circuit:

```

if (circuitNode.Is<Circuit>())
{
    var circuitAdapter = new D2dGraphAdapter<Module, Connection, ICircuitPin>(m_circuitRenderer,
, transformAdapter);
    var circuitModuleEditAdapter = new D2dGraphNodeEditAdapter<Module, Connection, ICircuitPin>(
)
    m_circuitRenderer, circuitAdapter, transformAdapter);
circuitModuleEditAdapter.DraggingSubNodes = false;

var circuitConnectionEditAdapter =
    new D2dGraphEdgeEditAdapter<Module, Connection, ICircuitPin>(m_circuitRenderer, circuitAdapter
, transformAdapter);

control.Adapt(
    hoverAdapter,
    scrollbarAdapter,
    autoTranslateAdapter,
    new RectangleDragSelector(),
    transformAdapter,
    viewingAdapter,
    canvasAdapter,
    mouseTransformManipulator,
    mouseWheelManipulator,
    new KeyboardIOGraphNavigator<Module, Connection, ICircuitPin>(),
    new D2dGridAdapter(),
    annotationAdaptor,
    circuitAdapter,
    circuitModuleEditAdapter,
    circuitConnectionEditAdapter,
    new LabelEditAdapter(),
    new SelectionAdapter(),
    new DragDropAdapter(m_statusService),
    new ContextMenuAdapter(m_commandService, m_contextMenuCommandProviders)
);
}

```

The additional adapters are:

- **D2dGraphAdapter:** provide support for drawing, viewing, and hit testing on the canvas.
- **D2dGraphNodeEditAdapter:** add node dragging to move circuit elements. This adapter is instantiated with a **D2dCircuitRenderer**. Pressing the Shift key constrains dragging to be parallel to the x- or y-axis. Its **DraggingSubNodes** property is set **false** to prevent moving subnodes.
- **D2dGraphEdgeEditAdapter:** add graph edge editing capabilities to make and move connections between pins in the circuit. This adapter is created with a **D2dCircuitRenderer**.

Group Display

This portion of code runs when creating a control for a Group:

```

else if (circuitNode.Is<Group>())
{
    var circuitAdapter = new D2dSubgraphAdapter<Module, Connection, ICircuitPin>(m_subGraphRenderer
    ,
                                         transformAdapter
);
    var circuitModuleEditAdapter = new D2dGraphNodeEditAdapter<Module, Connection, ICircuitPin>(
        m_subGraphRenderer, circuitAdapter, transformAdapter);
    circuitModuleEditAdapter.DraggingSubNodes = false;

    var circuitConnectionEditAdapter =
        new D2dGraphEdgeEditAdapter<Module, Connection, ICircuitPin>(m_subGraphRenderer, circuitAdapter
, transformAdapter);

    var groupPinEditor = new GroupPinEditor(transformAdapter);
    groupPinEditor.GetPinOffset = m_subGraphRenderer.GetPinOffset;

    canvasAdapter.UpdateTranslateMinMax = groupPinEditor.UpdateTranslateMinMax;

    control.Adapt(
        hoverAdapter,
        scrollbarAdapter,
        autoTranslateAdapter,
        new RectangleDragSelector(),
        transformAdapter,
        viewingAdapter,
        canvasAdapter,
        mouseTransformManipulator,
        mouseWheelManipulator,
        new KeyboardIOGraphNavigator<Module, Connection, ICircuitPin>(),
        new D2dGridAdapter(),
        annotationAdaptor,
        circuitAdapter,
        circuitModuleEditAdapter,
        circuitConnectionEditAdapter,
        new LabelEditAdapter(),
        groupPinEditor,
        new SelectionAdapter(),
        new DragDropAdapter(m_statusService),
        new ContextMenuAdapter(m_commandService, m_contextMenuCommandProviders)
    );
}
else throw new NotImplementedException(
    "graph node type is not supported!");

```

Additional adapters for a Group are:

- **D2dSubgraphAdapter**: reference and render a subgraph, that is, a group diagram. Also provides hit testing and viewing support on the canvas.
- **D2dGraphNodeEditAdapter**: add node dragging capabilities to move circuit elements. This object is instantiated with a D2dSubCircuitRenderer. Pressing the Shift key constrains dragging to be parallel to the x- or y-axis. DraggingSubNodes is false to prevent moving subnodes.
- **D2dGraphEdgeEditAdapter**: add graph edge editing capabilities to make and move connections between pins in the circuit. This adapter is instantiated with a D2dSubCircuitRenderer.
- **GroupPinEditor**: add floating group pin location and label editing capabilities to a subgraph control. The GetPinOffset property is set to a callback from D2dSubCircuitRenderer to compute group pin y offset. In addition, CanvasAdapter's UpdateTranslateMinMax property is initialized to a callback from GroupPinEditor for updating the translation minimum and maximum values.

Completion

CreateCircuitControl() finishes up with this:

```

control.ResumeLayout();

control.DoubleClick += new EventHandler(control_DoubleClick);
control.MouseDown += new MouseEventHandler(control_MouseDown);
return control;
}

```

The DoubleClick event handler displays a window when a group or master is double-clicked. It first ascertains whether the hit element is a subcircuit (master) or group.

If the clicked on item is a master, a control is created as well as a new circuit document. It creates an editing context in which the schema loader is specified, to enable copying and pasting to other applications. It creates a `ControlInfo` for the new control and registers it with the `CircuitControlRegistry`. For information about this component, see [CircuitControlRegistry Component](#).

If the clicked on item is a group, the process is similar, but a document is not created.

In both cases, `CreateCircuitControl()` itself is called to create a control to display the circuit elements.

The `MouseDown` handler does different things, depending on which part of a circuit element was clicked:

- Group expander button: toggle the group contents' visibility.
- Pin: allow dragging a connection to another pin without having to keep the mouse button pressed.
- Group pin visibility icon (): toggle the group pin's visibility.

All these mouse down actions are performed as transactions, in an `ITransactionContext` adapted from the `DomNode` tree's root `DomNode`, ultimately provided by the `CircuitEditingContext`, described in [CircuitEditingContext Class](#).

Context Classes

Circuit Editor uses several context classes, most of which are derived from `Sce.Atf.Controls.Adaptable.Graphs` classes.

An editing context is usually associated with a container, so editing is constrained to items in the container and currently selected items. The current active context is usually associated with the current active window in the user interface. For instance, when the circuit canvas is the active window, the `CircuitEditingContext` is the active context.

CircuitEditingContext Class

`CircuitEditingContext` derives from `Sce.Atf.Controls.Adaptable.Graphs.CircuitEditingContext`, which provides all its function. The base class cannot be used directly, because it is abstract. This base class itself derives from `EditingContext`, which implements `ISelectionContext` to enable selecting circuit elements. This base class also implements several interfaces, including `IInstancingContext` to allow editing selected items. For information on the base `CircuitEditingContext` class, see [CircuitEditingContext Class in Circuit Graph Support](#).

A circuit and group both behave as containers, and each can be displayed and edited separately in a graph window. Therefore, both "circuitType" and "groupType" have the DOM adapter `CircuitEditingContext` defined in `SchemaLoader`.

`CircuitEditingContext` contains the property `WireType` that must be overridden, which Circuit Editor does in the usual fashion:

```

protected override DomNodeType WireType
{
    get { return Schema.connectionType.Type; }
}

```

`CircuitEditingContext` implements `IEditableGraphContainer<Module, Connection, ICircuitPin>` to allow editing an expanded group by dragging items in and out of the group. The methods in this implementation simply call methods in the base `CircuitEditingContext`'s `IEditableGraphContainer<Element, Wire, ICircuitPin>` implementation.

For example, here is the `CircuitEditingContext` implementation of `IEditableGraphContainer.CanMove()`

```

bool IEditableGraphContainer<Module, Connection, ICircuitPin>.CanMove(object newParent,
IEnumerable<object> movingObjects)
{
    if (newParent.Is<IReference<Module>>())
        return false;
    var editableGraphContainer =
        DomNode.Cast<CircuitEditingContext>() as IEditableGraphContainer<Element, Wire, ICircuitPin>;
    return editableGraphContainer.CanMove(newParent, movingObjects);
}

```

The `editableGraphContainer` variable is an instance of the base class, `Sce.Atf.Controls.Adaptable.Graphs.CircuitEditingContext`, because this base class implements `IEditableGraphContainer<Element, Wire, ICircuitPin>`. Thus the next line's call `editableGraphContainer.CanMove(newParent, movingObjects)` is to the `CanMove()` method in the base class.

The other methods that `CircuitEditingContext` implements in `IEditableGraphContainer<Module, Connection, ICircuitPin>` also use the `IEditableGraphContainer<Element, Wire, ICircuitPin>` methods in the base class.

`IEditableGraphContainer` also derives from `IEditableGraph`, which governs making graph connections:

```

public interface IEditableGraphContainer<in TNode, TEdge, in TEdgeRoute>:
    IEditableGraph<TNode, TEdge, TEdgeRoute>
    where TNode : class, IGraphNode
    where TEdge : class, IGraphEdge<TNode, TEdgeRoute>
    where TEdgeRoute : class, IEdgeRoute

```

`CircuitEditingContext` also implements this interface, again by calling the corresponding methods in the base class. For example, here is `IEditableGraph.CanConnect()`:

```

public bool CanConnect(Module fromNode, ICircuitPin fromRoute, Module toNode, ICircuitPin toRoute)
{
    var editableGraphContainer =
        DomNode.Cast<CircuitEditingContext>() as IEditableGraphContainer<Element, Wire, ICircuitPin>;
    return editableGraphContainer.CanConnect(fromNode, fromRoute, toNode, toRoute);
}

```

This method's code is very similar to `IEditableGraphContainer.CanMove()` shown previously: the `editableGraphContainer` variable is created exactly the same way. As a result, this method calls the `IEditableGraph.CanConnect()` method in the base `CircuitEditingContext` class.

`CircuitEditingContext` is the DOM adapter defined for both the "circuit" and "group" types. Circuit Editor allows you to open multiple separate windows to view a group, in addition to the main document (circuit) window. Each opened graph window has an associated editing context with its own undo history. However, `GlobalHistoryContext` is also defined as a DOM adapter for the "circuit" type. `GlobalHistoryContext` is an adapter that implements a single global history on a DOM node tree containing multiple local `HistoryContexts`. This adapter tracks all other `HistoryContexts` in the subtree rooted at `DomNode` and passes their transactions to a global `HistoryContext`, which must be on the same `DomNode` as this adapter. As a result, there is only one undo/redo history stack for all circuits, including groups.

LayeringContext and PrototypingContext Classes

Both classes derive from classes of the same name in `Sce.Atf.Controls.Adaptable.Graphs`. These classes simply override properties in the base class to get type metadata from the `Schema` class. For details on what these base context classes do, see the sections [LayeringContext Class](#) and [PrototypingContext Class](#) in [Circuit Graph Support](#). For details on how prototyping works in Circuit Editor, see [Prototype Handling](#).

TemplatingContext Class

`TemplatingContext` derives from `Sce.Atf.Dom.TemplatingContext`. It is the editing context for the templates library, and is the context bound to the `TemplateLister` component when a circuit document becomes the active context. For details on working with templates in Circuit Editor, see [Template Handling](#).

Reusable Circuit Facilities and Windows

Circuit Editor has several windows that handle ways of grouping and reusing controls, such as the Prototypes window. Mastering also provides re-usability, although masters are not listed in a window. The Layering window resembles the Templates window, but provides a mechanism for layering circuits and controlling their visibility.

Prototype Handling

You create a prototype by copying circuit items and pasting them into the Prototypes window. The prototype can then be dragged back onto the canvas.

Circuit Editor does very little itself to implement prototyping. As with many of the other classes in Circuit Editor, its prototype classes all derive from classes with the same name in the `Sce.Atf.Controls.Adaptable.Graphs` namespace. These classes are all DOM adapters and override properties in the base class to get type metadata from the Schema class:

- `Prototype`: Circuit prototype, which contains `Elements` and `Wires` that can be copied into a circuit.
- `PrototypeFolder`: Folder of prototypes.
- `PrototypingContext`: Editing context for prototypes in the circuit document. For a discussion of this context's base class, see [PrototypingContext Class](#).

DOM adapters, such as `Prototype`, are discussed in [DOM Adapters](#).

The `PrototypeLister` component completes the prototype implementation, providing a tree control in which to list prototypes. Circuit Editor simply includes `PrototypeLister` in its MEF TypeCatalog.

Template Handling

Templates are similar to prototypes in that both allow saving parts of a circuit and organizing these items in a window with a tree control, from which the item can be dragged back onto the circuit canvas. The two differ in what can be copied and how it is copied: a template can be made only from module or group using a menu item. Note that promoting a module or group to a template changes the selected item into a template. In addition, changing one instance of a template changes them all.

Like prototypes, most of the template classes derive from classes by the same name in another namespace: `Sce.Atf.Dom` in this case. These classes are all DOM adapters.

The `TemplateLister` component provides a tree control in which to list templates. Circuit Editor simply includes `TemplateLister` in its MEF TypeCatalog.

Template and TemplateFolder Classes

`Template` adapts a template, containing information about the group or module in the template. The `Template` and `TemplateFolder` DOM adapters are discussed briefly in [DOM Adapters](#). The `Template` base class contains several abstract properties that describe the template and must be overridden. For instance, the `Name` property:

```
public override string Name
{
    get { return (string)DomNode.GetAttribute(Schema.templateType.labelAttribute); }
    set { DomNode.SetAttribute(Schema.templateType.labelAttribute, value); }
}
```

In the usual fashion of such overrides, the information is obtained from the data model's metadata classes in the `Schema` class.

There is also a `Guid` property with a GUID identifying the template:

```
public override Guid Guid
{
    get { return new Guid((string)DomNode.GetAttribute(Schema.templateType.guidAttribute)); }
    set { DomNode.SetAttribute(Schema.templateType.guidAttribute, value.ToString()); }
}
```

`TemplateFolder` has a similar, simple implementation, overriding the properties in the base `TemplateFolder` class, getting information from the metadata classes.

GroupInstance and ModuleInstance Classes

A template instance references a module or group, so the data model defines two types: "moduleTemplateRefType" and "groupTemplateRefType". If a circuit contains a template instance, the `DomNode` for the instance in the application data is a module with a "moduleTemplateRefType" or "groupTemplateRefType" type.

`GroupInstance` and `ModuleInstance` represent instances of a template. They are actually implemented in Circuit Editor and both derive from `Module`. These classes are DOM adapters for the template data types:

```
Schema.moduleTemplateRefType.Type.Define(new ExtensionInfo<ModuleInstance>());
Schema.groupTemplateRefType.Type.Define(new ExtensionInfo<GroupInstance>());
```

Both classes implement `IReference<Module>` and `IReference<DomNode>`. `IReference` has a method and a property:

- `CanReference()`: Can the given `DomNode` be referenced, that is, is it of the right type?
- `Target`: Actual target `Module` of the reference.

`TemplatingContext` uses `GroupInstance` or `ModuleInstance` to create a template instance for a group or module.

Keep in mind that these instances reference a module or group. The information for the circuit items in a template module instance resides in the template itself, not in the template instance.

ProxyGroup Class

`ProxyGroup` delegates communications with the targeted group on behalf of a `GroupInstance`. `ProxyGroup`'s constructor is

```
public ProxyGroup(GroupInstance owner, Group target)
```

and `ProxyGroup` is constructed by `GroupInstance` in its `ProxyGroup` property:

```
m_proxyGroup = new ProxyGroup(this, Target.As<Group>());
```

so the `ProxyGroup` is constructed using the underlying `Group` the group template instance refers to. `ProxyGroup` handles functions for a group template instance that are different from a regular group.

TemplatingCommands Component

`TemplatingCommands` is a command client that provides three commands:

- Add Template Folder: Add a template folder to the Templates window.
- Promote To Template Library: Make the selected module a template, copying it to the Templates window. The selected item is copied and through the `TemplatingContext`'s `Insert()` method, added to the list of templates. The original item is replaced by a template instance.
- Demote To Copy Instance: Make the selected template instance a module that is no longer a template.

The base `TemplatingCommands` sets up the commands, implements `ICommandClient`, and implements the command to add a template folder.

The `TemplatingCommands` class in Circuit Editor implements the promote and demote commands, which is the bulk of the work for templating.

TemplatingContext Class

`TemplatingContext` provides the editing context for the Templates window. The base class's definition is:

```
public abstract class TemplatingContext : SelectionContext,
    IInstancingContext,
    ITemplatingContext,
    IObservableContext,
    INamingContext
```

`SelectionContext` is a DOM adapter that implements `ISelectionContext` to allow selecting templates in the Templates window. The other interfaces perform a similar role to the interfaces implemented for `PrototypingContext`. For a discussion of these, see [PrototypingContext Class in Circuit Graph Support](#).

`ITemplatingContext`, similarly to `IPrototypingContext`, helps present a tree view of the templates and creates `IDataObject` instances from template items. `ITemplatingContext` also checks to see if a reference can reference the specified target item and actually get the reference. `ITemplatingContext`'s implementation is split between the `TemplatingContext` class and its base. The main work is done by the non-base `TemplatingContext`, which implements the referencing methods, `CanReference()` and `CreateReference()`. `CreateReference()` creates a new `GroupInstance` or `ModuleInstance` as a template instance for a group or module.

`TemplatingContext` also gets notified if a template is removed and turns any template references into instances in that case.

Master (Subcircuit) Handling

Mastering is similar to templating, in that it creates a master from selected circuit items, that is, a subcircuit, creating a new type of circuit element. Any changes to this master element affect all instances of it. Mastering differs from templating in that any selection of items in the circuit can be converted to a master, not just a module or group. The master appears as a module displaying whatever pins were not internally connected. There is no Mastering window.

Mastering is implemented almost entirely in Circuit Editor, except for the `SubCircuit` and `SubCircuitInstance` classes, which are DOM adapters derived from classes of the same name in `Sce.Atf.Controls.Adaptable.Graphs`.

SubCircuit Class

`SubCircuit` adapts a `DomNode` to a mastered subcircuit. Like many DOM adapters in Circuit Editor, it simply overrides the base class's properties, getting information from the `Schema` class's metadata classes.

SubCircuitInstance Class

`SubCircuitInstance` adapts a `DomNode` to an instance of a mastered subcircuit. Like `SubCircuit`, it simply overrides the base class's properties, getting information from the `Schema` class's metadata classes.

Note that `SubCircuitInstance` derives from `Element` — not `Circuit` as the `SubCircuit` DOM adapter does. A `SubCircuitInstance` represents a single element in a circuit, whereas a `SubCircuit` represents one or more elements in a circuit. These types have adapters defined in this way:

```
Schema.subCircuitType.Type.Define(new ExtensionInfo<SubCircuit>());
Schema.subCircuitInstanceType.Type.Define(new ExtensionInfo<SubCircuitInstance>());
```

The `Editor` component uses `SubCircuitInstance` when creating an adaptable control for a master subcircuit instance, either when the master is double-clicked or hovered over.

MasteringCommands Component

`MasteringCommands` is a command client implementing these commands under the Edit menu:

- Master: Create a custom module type from the selection.
- Unmaster: Expand the last selected custom module into the original circuit elements, no longer a master instance.

The mastering process entails copying all the selected circuit elements to a master copy, removing the original selected items corresponding `DomNodes`, setting up the pins correctly for the new master, and then creating a `DomNode` for the master module, represented by a `SubCircuitInstance` `DomNode`. The canvas is updated accordingly. Unmastering is a simpler process, simply removing the existing master `DomNode` and creating a new list of `DomNodes` from copies of the master's items.

MasteringValidator Class

`MasteringValidator` tracks changes to subcircuits and subcircuit instances in the document and throws an `InvalidOperationException` for certain errors. `MasteringValidator` is a DOM adapter that is defined for the circuit document type in `SchemaLoader`:

```
Schema.circuitDocumentType.Type.Define(new ExtensionInfo<MasteringValidator>()); // validates sub-circuits
```

This allows the validator to track changes throughout the entire circuit document, that is, whenever the circuit changes. The `OnNodeSet()` subscribes to changes to the `DomNode` tree:

```
protected override void OnNodeSet()
{
    base.OnNodeSet();
    DomNode.ChildInserting += DomNodeChildInserting;
    DomNode.ChildRemoving += DomNodeOnChildRemoving;
}
```

These event handlers show examples of the kind of circuit checking that can be done. For instance:

```

private void DomNodeChildInserting(object sender, ChildEventArgs e)
{
    if (Validating)
    {
        // inserting an instance of a sub-circuit into itself?
        SubCircuitInstance subCircuitInstance = e.Child.As<SubCircuitInstance>();
        SubCircuit subCircuit = e.Parent.As<SubCircuit>();
        if (subCircuitInstance != null &&
            subCircuit != null &&
            subCircuitInstance.SubCircuit == subCircuit)
        {
            throw new InvalidTransactionException(
                "Can't use a sub-circuit inside itself".Localize());
        }
    }
}

```

Attempting to insert a `SubCircuitInstance` that is an instance of the `SubCircuit` into the `SubCircuit` fails.

Layer Handling

You can copy and paste circuit items into a layer to control their visibility. The layering classes in Circuit Editor all derive from classes of the same name (except for `ModuleRef`) in the `Sce.Atf.Controls.Adaptable.Graphs` namespace that provide their functionality. These classes are all DOM adapters:

- `LayerFolder`: Adapter for layer folders.
- `LayeringCommands`: Component for layering commands. For details, see [LayeringCommands Component](#).
- `LayeringContext`: Context for layering. For details on this context, see [LayeringContext Class](#).
- `ModuleRef`: Derives from `Sce.Atf.Controls.Adaptable.Graphs.ElementRef` and is used within layer folders to represent circuit modules that belong to that layer.

Circuit Editor also imports the `LayerLister` component, which uses the `LayeringContext` to present a tree view of the layers with a `TreeControl`.

Miscellaneous Classes

These classes provide a variety of capabilities.

CircuitValidator Class

`CircuitValidator` derives from the DOM adapter `Sce.Atf.Controls.Adaptable.Graphs.CircuitValidator` and overrides its properties with data it gets from the metadata classes in Schema. For details on the base class, see [CircuitValidator Class](#).

GroupingCommands Component

The `GroupingCommands` component is the command client for grouping commands and derives from `Sce.Atf.Controls.Adaptable.Graphs.GroupingCommands`. For more information on this base component, see [GroupingCommands Component](#).

ModulePlugin Component

`ModulePlugin` is a palette client component that creates the circuit items palette. In some ways, it functions like other palette clients, such as the one for [ATF Simple DOM Editor Sample](#) described in [Using a Palette](#).

The component's `IInitializable.Initialize()` method sets up palette data using its `DefineModuleType()` method, which performs several functions, including adding an item to the palette. It also adds type metadata information by calling the `NamedMetadata.SetTag()` method for the types on the palette.

PrintableDocument Class

`PrintableDocument` is an implementation of `IPrintableDocument` to print the circuit canvas. This implementation uses a private class `CircuitPrintDocument`, which derives from `CanvasPrintDocument`. `CanvasPrintDocument` is an abstract base class for canvas printing. `PrintableDocument` overrides its methods to get "page" bounds for the various `PrintRange` values, as well as the `Render()` method to actually render a circuit to the page.

`PrintableDocument` is defined as the DOM adapter for the circuit type, because the document represents a circuit.

Topics in this section

Links on this page to other topics

[Adaptable Controls](#), [Adaptation in ATF](#), [ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [Circuit Graph Support](#), [Graph Data Model](#), [Graphs in ATF](#), [Simple DOM Editor Programming Discussion](#), [Types of Graphs](#)

Code Editor Programming Discussion

The [ATF Code Editor Sample](#) shows how to interface third party software to an ATF application: the ActiproSoftware SyntaxEditor, which is provided in a set of DLLs. The Code Editor uses this software to access an editing control with language syntax sensitive editing for plain text, C#, Lua, Python, XML, and other format files with appropriate extensions. Code Editor handles text files in these formats, so it does not store application data using a data model. Its data model could be considered to be the formats for the file types, and these are specified in proprietary XML files, such as `CSharpDefinition.xml`.

Code Editor uses only a fraction of the capabilities ActiproSoftware SyntaxEditor provides.

Code Editor uses interfaces and classes in the `Sce.Atf.Controls.SyntaxEditorControl` namespace, which comprises the `Atf.SyntaxEditorControl` assembly. The items in this namespace provide a layer between Code Editor and SyntaxEditor.

Contents

- [Programming Overview](#)
- [Code Editor Document Handling](#)
 - [CodeDocument Class](#)
 - [Editor Component](#)
- [SourceControlContext Component](#)
- [Sce.Atf.Controls.SyntaxEditorControl Namespace](#)
 - [ISyntaxEditorControl Interface](#)
 - [SyntaxEditorControl Class](#)
- [ATF Facilities](#)

Programming Overview

The Code Editor application is basically a container for the ActiproSoftware SyntaxEditor editing control. The Code Editor itself uses some standard ATF facilities for handling documents in its `CodeDocument` class and `Editor` component. Code Editor also provides source control with its own simple `SourceControlContext` component, but mainly with the ATF components `SourceControlCommands` and `PerforceService`.

The `Sce.Atf.Controls.SyntaxEditorControl` namespace contains interfaces and classes Code Editor uses to communicate with the editing control. The most important is the `ISyntaxEditorControl` interface, which describes a framework for a syntax editor and could be used for syntax editors other than ActiproSoftware's. `SyntaxEditorControl` implements `ISyntaxEditorControl` by using ActiproSoftware's `SyntaxEditor` class. Other items in `Sce.Atf.Controls.SyntaxEditorControl` play a minor supporting role.

Code Editor Document Handling

Code Editor handles documents in a similar way as other samples by implementing `IDocument` and `IDocumentClient`. For a description of the general process, see [Implementing a Document and Its Client](#) in the section [Documents in ATF](#).

CodeDocument Class

`CodeDocument` is the document class, implementing `IDocument`. Its constructor sheds light on interfacing with the editing control:

```

public CodeDocument(Uri uri)
{
    if (uri == null)
        throw new ArgumentNullException("uri");

    m_uri = uri;

    string filePath = uri.LocalPath;
    string fileName = Path.GetFileName(filePath);

    m_type = GetDocumentType(fileName);

    m_editor = TextEditorFactory.CreateSyntaxHighlightingEditor();

    Languages lang = GetDocumentLanguage(fileName);
    m_editor.SetLanguage(lang);
    Control ctrl = (Control)m_editor;
    ctrl.Tag = this;

    m_editor.EditorTextChanged += editor_EditorTextChanged;

    m_controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center);
    // tell ControlHostService this control should be considered a document in the menu,
    // and using the full path of the document for menu text to avoid adding a number in the end
    // in control header, which is not desirable for documents that have the same name
    // but located at different directories.
    m_controlInfo.IsDocument = true;
}

```

This constructor does some initial housekeeping with the give URI. Next, the constructor gets an editing control object using the factory method `TextEditorFactory.CreateSyntaxHighlightingEditor()` and saves it in the field `m_editor`. This editing control implements `ISyntaxEditorControl`, which is the interface to access an editing control's capabilities. For information on this interface, see [ISyntaxEditorControl Interface](#). The constructor sets the editing control's language, gleaned from the URI file's extension by the `GetDocumentLanguage()` method. The constructor finishes up by setting the `Tag` property in the control, subscribing to the `EditorTextChanged` event and setting up a new `ControlInfo`.

`CodeDocument` provides a few simple properties, such as `Control`, which gets the editing control. It also provides `Read()` and `Write()` methods, which are straightforward, because Code Editor only works with text files. Here's `Read()`, for example:

```

public void Read()
{
    string filePath = m_uri.LocalPath;
    if (File.Exists(filePath))
    {
        using (StreamReader stream = new StreamReader(filePath, Encoding.UTF8))
        {
            m_editor.Text = stream.ReadToEnd();
            m_editor.Dirty = false;
        }
    }
}

```

The editing control's `Text` property is set to the text read from the file.

`CodeDocument` also implements `IDocument` and `IResource`. These simple interfaces handle the `Dirty` and `Uri` properties and their related events, such as `DirtyChanged` and `UriChanged`.

`CodeDocument` also contains a couple of utility methods `GetDocumentType()` and `GetDocumentLanguage()` that provide their information by examining the file's extension, which indicates the document type.

Editor Component

The `Editor` component provides Code Editor's document client. It is also the control host client and command client for the main editing control, so it implements both `IControlHostClient` and `ICommandClient`. It implements `ICommandClient` for the standard editing commands:

```

public class Editor : IControlHostClient, IInitializable, ICommandClient

```

For information about how ATF handles controls and control host clients, see [Using Controls in ATF](#) in the section [Controls in ATF](#). To learn about commands and their clients, see [Using Commands in ATF](#) in [Commands in ATF](#).

Document Client

`Editor` contains the simple class `DocumentClient` that is the document client for `Code Editor`:

```
private class DocumentClient : IDocumentClient
```

Its constructor takes an instance of `Editor`:

```
public DocumentClient(Editor editor, string extension)
{
    m_editor = editor;
    string fileType = CodeDocument.GetDocumentType(extension);
    m_info = new DocumentClientInfo(fileType, extension, null, null);
}
```

The constructor saves the `Editor` instance in the field `m_editor`. Do not confuse this with the `m_editor` field in the `CodeDocument` class, which holds an instance of the `ActiproSoftware SyntaxEditor` editing control.

This constructor also creates a `DocumentClientInfo` based on the document's type, indicated by its extension.

The document client uses the `IControlHostService` object passed to the `Editor` constructor to do some of its work. For example, `Open()` creates a `CodeDocument` object, reads the document, and then registers the `ActiproSoftware SyntaxEditor` editing control held in the `CodeDocument.Control` property with the Control Host Service:

```
public IDocument Open(Uri uri)
{
    CodeDocument doc = new CodeDocument(uri);
    doc.Read();

    m_editor.m_controlHostService.RegisterControl(
        doc.Control,
        doc.ControlInfo,
        m_editor);

    return doc;
}
```

The other `IDocumentClient` methods are even simpler than `Open()`.

`Editor`'s constructor creates a `DocumentClient` object for every file type that `Code Editor` supports for later use:

```
// create a document client for each file type
m_txtDocumentClient = new DocumentClient(this, ".txt");
...
m_cgDocumentClient = new DocumentClient(this, ".cg");
```

It is typical for an ATF application to create a document client for each document type it handles.

Control Host Client

This client handles the editing control. The `Activate()` method, called when the control becomes active, informs the Document Registry and Command Service of the active document and control host client:

```

public void Activate(Control control)
{
    if (control.Tag is CodeDocument)
    {
        IDocument doc = (IDocument)control.Tag;
        m_documentRegistry.ActiveDocument = doc;
        m_commandService.SetActiveClient(this);
    }
}

```

`Close()` gets the active document, if any, and asks the Document Service to close it, which prompts the user to save the document if it was modified:

```

public bool Close(Control control)
{
    CodeDocument document = control.Tag as CodeDocument;
    if (document != null)
        return m_documentService.Close(document);

    return true;
}

```

Command Client

The `Editor` component's `IInitializable.Initialize()` is called after `Editor` is constructed and finishes initialization that can't be done in the constructor. In this case, it uses the Command Service component to register the Edit commands:

```

// register commands
m_commandService.RegisterCommand(CommandInfo.EditUndo, this);
...
m_commandService.RegisterCommand(CommandInfo.EditDelete, this);

m_commandService.RegisterCommand(
    Command.FindReplace,
    StandardMenu.Edit,
    StandardCommandGroup.EditOther,
    "Find and Replace...",
    "Find and replace text",
    Keys.None,
    Resources.FindImage,
    CommandVisibility.Menu,
    this);

m_commandService.RegisterCommand(
    Command.Goto,
    StandardMenu.Edit,
    StandardCommandGroup.EditOther,
    "Go to...",
    "Go to line",
    Keys.None,
    null,
    CommandVisibility.Menu,
    this);

```

In addition to standard Edit commands, it registers a couple of commands for search and going to a line in the file.

Code Editor imports the `StandardFileCommands` component to create the actual Edit menu items for the edit commands.

The command client relies on the editing control to actually perform editing commands. Here's what `ICommandClient.DoCommand()` does:

```

public void DoCommand(object commandTag)
{
    CodeDocument activeDocument = m_documentRegistry.ActiveDocument as CodeDocument;
    if (commandTag is StandardCommand)
    {
        switch ((StandardCommand)commandTag)
        {
            case StandardCommand.EditUndo:
                activeDocument.Editor.Undo();
                break;
            ...
            case StandardCommand.EditDelete:
                activeDocument.Editor.Delete();
                break;
        }
    }
    else if (commandTag is Command)
    {
        switch ((Command)commandTag)
        {
            case Command.FindReplace:
                activeDocument.Editor.ShowFindReplaceForm();
                break;

            case Command.Goto:
                activeDocument.Editor.ShowGoToLineForm();
                break;
        }
    }
}

```

The variable `activeDocument` is adapted to a `CodeDocument` object, so the value `activeDocument.Editor` is the editing control. The editing control methods invoked here, such as `Undo()`, are in the `ISyntaxEditorControl` interface, which the editing control implements. For details on this interface, see [ISyntaxEditorControl Interface](#).

SourceControlContext Component

`SourceControlContext` implements `ISourceControlContext`, which gets an enumeration of the `IResources` under source control with its `Resources` property. A source control facility can examine the URIs in this property and adapt the `IResource` to `IDocument` to track a document's dirty flag.

`Code Editor` also imports the `SourceControlCommands` and `PerforceService` components, which actually do the source control management work.

`SourceControlCommands` registers the commands for source control and also provides the command client to perform them. In addition, it implements `IContextMenuCommandProvider` to provide a context menu with the source control commands. Its command client relies on a `SourceControlService` component to do the actual source control, which allows using different source control providers.

`SourceControlService` is an abstract component that implements `ISourceControlService`, the interface for source control services. `PerforceService` derives from `SourceControlService` and uses the Perforce Client for its source control provider.

Sce.Atf.Controls.SyntaxEditorControl Namespace

The interfaces and classes in `Sce.Atf.Controls.SyntaxEditorControl` comprise the layer between `Code Editor` and the `ActiproSoftware SyntaxEditor`. The most important are `ISyntaxEditorControl` and its implementer, `SyntaxEditorControl`.

ISyntaxEditorControl Interface

`ISyntaxEditorControl` is an interface for syntax aware editing controls in general, not just the `ActiproSoftware SyntaxEditor`. This interface contains properties, methods, and events to do various things:

- Get the actual text editing control.
- Get or set text in the control.
- Get or set information about the text, such as the number of lines.
- Get or set user interface information, such as splitter locations.
- Notify when key press and change events occur.
- Determine if editing commands are doable and do them.
- Enable facilities, such as word wrapping.

- Perform selection operations.
- Do search and replace operations.

There are also interfaces for search and replace: `ISyntaxEditorFindReplaceOptions`, `ISyntaxEditorFindReplaceResult`, and `ISyntaxEditorFindReplaceResultSet`.

SyntaxEditorControl Class

`SyntaxEditorControl` derives from `SyntaxEditor`, the actual ActiproSoftware `SyntaxEditor` editing control, defined in the ActiproSoftware DLLs. `SyntaxEditorControl` implements `ISyntaxEditorControl` by using the properties and methods of `SyntaxEditor`. Code Editor also uses some ATF facilities; for more information, see [ATF Facilities](#). Code Editor could use another editing control package by implementing `ISyntaxEditorControl` for that editing control's API.

`SyntaxEditorControl`'s constructor sets default values for various `ISyntaxEditorControl` properties and subscribes to the events in `ISyntaxEditorControl`.

The `SetLanguage()` method sets the control to one of the built-in languages. This is where the XML language definition files, such as `PythonDefinition.xml` and `LuaDefinition.xml`, are used to set `SyntaxEditor`'s language. These files have a proprietary format.

`SyntaxEditorControl` also contains classes to implement `ISyntaxEditorFindReplaceOptions`, `ISyntaxEditorFindReplaceResult`, and `ISyntaxEditorFindReplaceResultSet`.

`SyntaxEditorControl` is an internal class to abide by `SyntaxEditor`'s licensing terms.

ATF Facilities

The namespace `Sce.Atf.Controls.SyntaxEditorControl` contains other ATF facilities that are primarily used in `SyntaxEditorControl`.

Breakpoints

Code Editor provides breakpoint facilities, as would be expected. An interface and classes provide breakpoint support:

- `IBreakpoint`: Properties describing a breakpoint, such as `Enabled` and `LineNumber`.
- `BreakpointEventArgs`: Breakpoint event argument information for the `BreakpointChanging` event, providing a constructor and properties, such as `IsSet` and `LineNumber`. The `BreakpointChanging` event is not implemented in Code Editor.
- `BreakpointIndicator`: Breakpoint indicator, implementing `IBreakpoint`. Its `DrawGlyph()` method draws the indicator.

Event Arguments

Several classes besides `BreakpointEventArgs` provide event arguments for events defined in `ISourceControlContext`.

- `EditorTextChangedEventArgs`: Event arguments for the `EditorTextChanged` event, describing the nature of the text change.
- `MouseHoverOverTokenEventArgs`: Event arguments for the `MouseHoveringOverToken` event. The `MouseHoveringOverToken` event is not implemented in Code Editor.
- `ShowContextMenuEventArgs`: Event arguments for the `ShowContextMenu` event. The event is raised when the context menu should be displayed by right-clicking the `SyntaxEditor` control.

Language Related

These provide support for languages in Code Editor:

- `Languages`: Enumeration of languages supported in Code Editor.
- `LuaDynamicSyntaxLanguage`: Adds folding for code blocks in the Lua language.

Support Entities

These miscellaneous items generally support the Code Editor:

- `SyntaxEditorRegions`: Enumeration of regions in the `SyntaxEditorControl` user interface for hit testing. These include items such as splitters, scroll bars, and margins.
- `Token`: A struct for a text token representing a word in a document. Besides the constructor, it holds token properties, such as the lexeme, the base unit of meaning of some word that can have a variety of forms (run with the forms runs, ran, running, etc.).
- `TextEditorFactory`: Factory with methods to produce editing objects:
 - `CreateSyntaxHighlightingEditor()`: Provide a `SyntaxEditorControl` object. Used by the `CodeDocument` constructor.
 - `CreateSyntaxEditorFindReplaceOptions()`: Get a `SyntaxEditorFindReplaceOptions` object.

Topics in this section

Links on this page to other pages

[ATF Code Editor Sample](#), [Authoring Tools Framework](#), [Commands in ATF](#), [Controls in ATF](#), [Documents in ATF](#), [Implementing a Document and Its Client](#), [Using Commands in ATF](#), [Using Controls in ATF](#)

Diagram Editor Programming Discussion

The [ATF Diagram Editor Sample](#) serves as a great example of how powerful MEF component composition can be. This sample simply puts the MEF components used in the other graph samples in its `TypeCatalog` to produce an application that has nearly all the capabilities of the individual graph samples:

- [ATF Circuit Editor Sample](#), described in [Circuit Editor Programming Discussion](#).
- [ATF FSM Editor Sample](#), described in [FSM Editor Programming Discussion](#).
- [ATF State Chart Editor Sample](#), described in [State Chart Editor Programming Discussion](#).

Thus Diagram Editor can edit circuits, simple state machines, and statecharts.

Contents

- [Programming Overview](#)
- [Common Components](#)
- [Graph Components](#)

Programming Overview

Diagram Editor provides all of its capabilities by adding graph components from the graph samples to its MEF `TypeCatalog`. Everything else that is needed is included by referencing it, such as XML Schema files from `SchemaLoader` classes. The `TypeCatalog` includes (almost) a superset of the components in the three graph editor samples, so Diagram Editor provides all the capabilities those components do.

Common Components

The only file in this sample of any substance is `Program.cs`. Its `Main()` function sets up the MEF components in a typical fashion. The MEF components it uses are a superset of the components used in the other three graph samples, with the few exceptions noted.

Some of these components are the ones most samples include: `CommandService`, `ControlHostService`, and `WindowLayoutService`. `PaletteService` is used in about half the samples and is needed here, because all the graph samples use palettes and have palette clients. A few are ones that are common to the graph samples, but not the other samples, such as `PrototypeLister`.

Graph Components

Diagram Editor must also include the key graph components in the graph samples. Here are the editor components listed for these samples:

```
// Editors
typeof(StatechartEditorSample.Editor),           // sample statechart editor
typeof(StatechartEditorSample.SchemaLoader),       // loads statechart schema and extends types
typeof(StatechartEditorSample.PaletteClient),      // component which adds palette items

typeof(CircuitEditorSample.Editor),                // sample circuit editor
typeof(CircuitEditorSample.SchemaLoader),          // loads circuit schema and extends types
typeof(GroupingCommands),                         // circuit group/ungroup commands
typeof(CircuitControlRegistry),                   // circuit controls management
typeof(MasteringCommands),                        // circuit master/unmaster commands
typeof(CircuitEditorSample.ModulePlugin),          // component that defines circuit module types
typeof(LayeringCommands),                         // "Add Layer" context menu command for the Layer
Lister

typeof(FsmEditorSample.Editor),                   // editor which manages FSM documents and
controls
typeof(FsmEditorSample.PaletteClient),            // component which adds palette items
typeof(FsmEditorSample.SchemaLoader),              // loads FSM schema and extends types
```

There are three components that are common to all the samples:

- `SchemaLoader`: Schema loader, derived from `XmlSchemaTypeLoader`, that loads the schema file describing the sample's data model. All the graph samples use the ATF DOM and define their data model with an XML Schema.
- `Editor`: Editor that opens and closes documents and manages the document editing controls. This is both a document and control host client, implementing `IDocumentClient` and `IControlHostClient`. These components perform in a similar way in the different samples. For more details of what they do, see [Editor Component](#) in [Fsm Editor](#) and [Circuit Document Display and Editing](#) in

Circuit Editor.

- **PaletteClient:** Palette client that populates the palette with the items that can be dragged onto a canvas. The Circuit Editor sample's palette client class is called `ModulePlugin`. Note that the application handles multiple palettes smoothly, combining them in one pane, with individual palette items separated by headers. In addition, you can only drag items onto a canvas of the proper type. You can't drag circuit items onto a statechart canvas, for example. For more details on palettes, see [Using a Palette](#).

Note that though Diagram Editor itself has only a few files associated directly with it, it depends on classes and files in other samples. For instance, the `SchemaLoader` classes all need to load their XML Schema `.xsd` files. `SchemaLoader` also depends on the type metadata classes in the `Schema` class for each sample. Diagram Editor is built out of much more than appears in its project.

For Fsm Editor and State Chart Editor, these three components above are the only ones needed, because they are the only components implemented in those samples.

Circuit Editor, however, has several more listed that provide important capabilities. `GroupingCommands`, for example, is used for commands to group and ungroup modules and the connections between them. `LayerLister` is also used only by Circuit Editor, even though it is not grouped with the other components.

Some of the components in Circuit Editor are not listed in Diagram Editor, such as `GraphViewCommands` or any of the template related components like `TemplateLister`. This means that Diagram Editor doesn't have the commands to increase or decrease the magnification of the graph using zoom presets, because `GraphViewCommands` provides that. Nor does it have templates. These absences also demonstrate how capabilities can be easily removed from (or added to) applications by encapsulating them in components.

Topics in this section

Links on this page to other pages

[ATF Circuit Editor Sample](#), [ATF Diagram Editor Sample](#), [ATF FSM Editor Sample](#), [ATF State Chart Editor Sample](#), [Circuit Editor Programming Discussion](#), [FSM Editor Programming Discussion](#), [Simple DOM Editor Programming Discussion](#), [State Chart Editor Programming Discussion](#)

DOM Tree Editor Programming Discussion

The [ATF DOM Tree Editor Sample](#) is an editor for UI elements, which are hierarchical and displayed as a tree in the main editing window. The data is handled by the ATF DOM. The application's data types' attributes are designed with enough variety to demonstrate the various value editors and controls ATF offers. For more information on value editors, see [Property Editing in ATF](#).

The sample also illustrates the `CurveEditor` component, which allows you to draw curves in a window. For more information on using it, see [CurveEditor Component](#).

Programming Overview

This sample defines its data model in an XML Schema and uses the ATF DOM to handle application data, including persisting data. Containment relationships are an important part of the data model and affect how the document can be edited.

The data model also includes specifying property descriptors for type attributes, so they can be viewed and edited in property editors. Descriptors are specified in both annotations in the XML Schema and in descriptor constructors in the schema loader. Using annotations requires special support from the schema loader to parse the annotations.

This sample's data types have diverse attribute types, so a plethora of value editors are used to display them in the property editors.

Over half the classes in this sample are DOM adapters, most for the various data types. DOM adapters are also used for documents and data verification.

The sample uses its `TreeLister` and `TreeView` component and class to display and edit its data. The `EditingContext` class provides the context for editing data, as in drag and drop operations from the UI element palette. The palette's implementation closely resembles other samples.

Contents

- [Programming Overview](#)
- [DOM Tree Editor Data Model](#)
 - [DOM Tree Editor Data Definition](#)
 - [Schema Loader](#)
 - [DOM Tree Editor DOM Adapters](#)
- [Palette Implementation](#)
- [Document Handling](#)
- [Tree Handling: TreeLister and TreeView](#)
- [Context Handling](#)
- [Validation in DOM Tree Editor](#)
- [Property Descriptors in DOM Tree Editor](#)
 - [Specifying Property Descriptors](#)

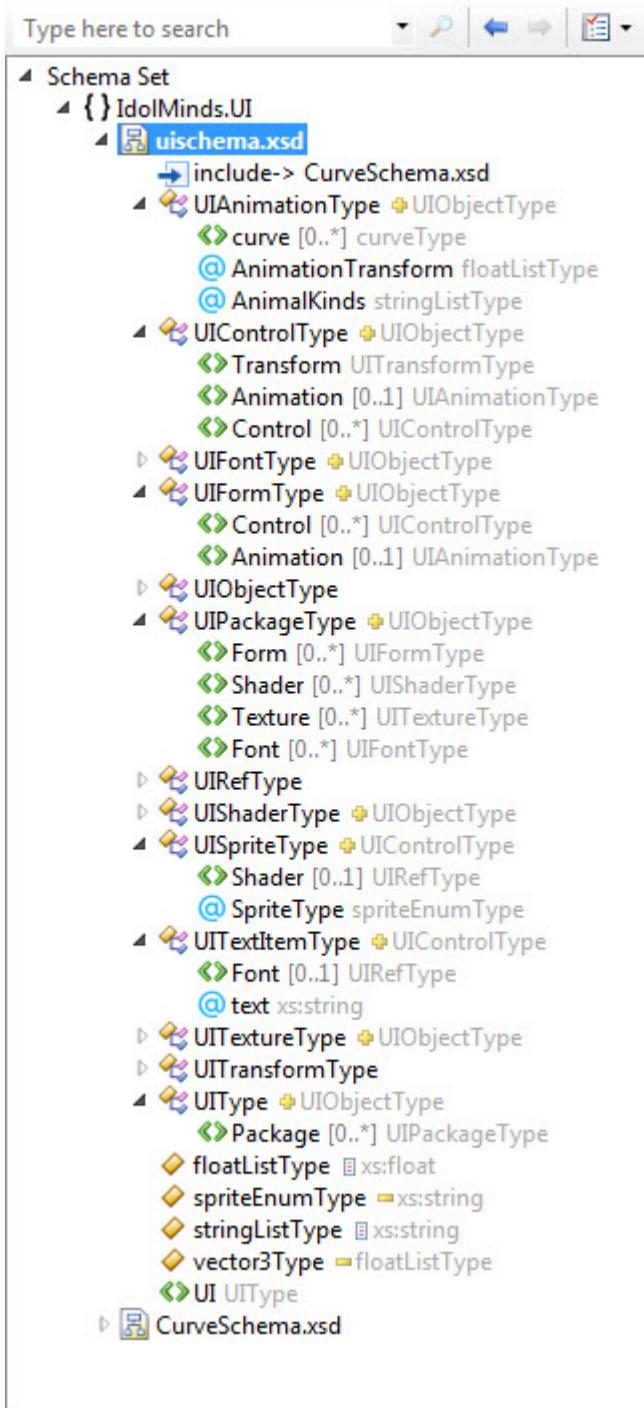
DOM Tree Editor Data Model

The data model uses the ATF DOM. It also defines property descriptors for its data types' attributes, so the attributes appear properly in property editors. The `GenSchemaDef.bat` command file runs DomGen to generate the `UISchema.cs` file containing the `UISchema` class with its metadata classes.

DOM Tree Editor Data Definition

DOM Tree Editor defines its types in an XML Schema type definition file `UISchema.xsd` for UI element types. This file includes `CurveSchema.xsd`, which defines the control point and curve types for curves in the `CurveEditor` window.

The base type is "UIObjectType" representing UI elements; most of the other types are based on "UIObjectType", including the root type "UIType". The root type is the type of the root `DomNode` in the tree. Most of the objects, such as animations and textures, are of "UIObjectType". A couple types, "UISpriteType" and "UITextItem Type", are of type "UIControlType" — also based on "UIObjectType". This figure from the Visual Studio XML Schema Explorer shows the type base relations:



This figure also shows the containment relationships: all the object types that can contain objects of other types have their nodes opened to show their child types. For instance, the root type "UIType" can contain an object of "UIPackageType", which can contain objects of types "UIFontType", "UIShaderType", "UITextureType", and "UIFontType". These relationships are enforced by the DOM Tree Editor when you drag objects from the palette onto other objects, as described in [Context Handling](#). The first object you can drag onto the editing window is a package, for example, because the root type "UIType" only contains "UIPackageType".

Property Descriptor Annotations

Some of the attributes in the schema definition have `scea.dom.editors.attribute` annotations to define property descriptors:

```

<!--The values here are ignored, but need enumeration type for UISpriteType-->
<xssimpleType name="spriteEnumType">
  <xssrestriction base="xs:string">
    <xsenumeration value="xxxx"/>
  </xssrestriction>
</xssimpleType>

<xsscomplexType name="UISpriteType">
  <xssannotation>
    <xssappinfo>
      <scea.dom.editors.attribute name="SpriteType" displayName="Sprite type" description="Sprite type"
        editor=
"Sce.Atf.Controls.PropertyEditing.LongEnumEditor,Atf.Gui.WinForms:Pixie,Ghost,Gollum,Robot,Skeleton,Big
bird"
        converter="Sce.Atf.Controls.PropertyEditing.EnumTypeConverter" />
    </xssappinfo>
  </xssannotation>
  <xsscomplexContent>
    <xssextension base="UIControlType">
      <xsssequence>
        <xsselement name="Shader" type="UIRefType" minOccurs="0" maxOccurs="1"/>
      </xsssequence>
      <xssattribute name="SpriteType" type="spriteEnumType" />
    </xssextension>
  </xsscomplexContent>
</xsscomplexType>

```

This annotation defines a property descriptor for the "SpriteType" attribute of the type "UISpriteType". It specifies `LongEnumEditor` as the value editor for this attribute, which has the type "spriteEnumType", shown just before the "UISpriteType" definition. This attribute must have an enumeration type to be able to use `LongEnumEditor` as its value editor. The "SpriteType" attribute has a stub enumeration definition; the actual enumeration values are listed as parameters with the value editor specification. `LongEnumEditor` implements `IAnnotatedParams`, so it takes a list of enumeration value parameters. Note also that the value editor's path name is fully qualified, and its assembly "Atf.Gui.WinForms" is also spelled out. Specifying the wrong path or assembly results in the schema not being loaded, because the value editor can't be located. The annotation also gives the value converter `EnumTypeConverter` that converts the parameters to the enumeration's values, and this converter is required for `LongEnumEditor`.

For more details on specifying attribute property descriptors in annotations, see the ATF Programmer's Guide: Document Object Model (DOM), which can be downloaded at [ATF Documentation](#).

Schema Loader

This application's schema loader is similar to other samples. As usual, it derives from `XmlSchemaTypeLoader` to do most of the schema loading work.

Its `OnSchemaSetLoaded()` method performs these typical functions after the schema is loaded:

- Initialize `UISchema`'s meta data classes with `UISchema.Initialize(typeCollection)`.
- Define DOM adapters, as described in [DOM Tree Editor DOM Adapters](#).
- Tag the UI types that appear in the object palette with information in a `NodeTypePaletteItem` object. Palette objects types are set the same way as in the [ATF Simple DOM Editor Sample](#), described in [Add Palette Items](#).
- Tag types to define property descriptors for type attributes. For details, see [Type Tag Property Descriptors](#).

Type Tag Property Descriptors

The schema loader also sets tags to a `PropertyDescriptorCollection` value on types to define their property descriptors. The descriptors defined for "UIControlType" have a number of interesting features:

```

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()),
        new ChildAttributePropertyDescriptor(
            Localizer.Localize("Rotation"),
            UISchema.UITransformType.RotateAttribute,
            UISchema.UIControlType.TransformChild,
            null,
            Localizer.Localize("Item rotation"),
            false,
            new NumericTupleEditor(typeof(float), new string[] { "X", "Y", "Z" }),
            new FloatArrayConverter()),
        new ChildAttributePropertyDescriptor(
            Localizer.Localize("Scale"),
            UISchema.UITransformType.ScaleAttribute,
            UISchema.UIControlType.TransformChild,
            null,
            Localizer.Localize("Item scale"),
            false,
            new UniformArrayEditor<float>())
    }));

```

The `PropertyDescriptorCollection` contains `ChildAttributePropertyDescriptors` for several attributes. Note that `ChildAttributePropertyDescriptor` is used, because these descriptors are for attributes of child nodes of "UIControlType". A "UIControlType" object can have children of types "UITransformType", "UIAnimationType", and "UIControlType". The descriptors here describe the attributes of "UITransformType": "RotateAttribute", "ScaleAttribute", and "TranslateAttribute". Property descriptors for type attributes would use an `AttributePropertyDescriptor` constructor instead.

Note that the `ChildAttributePropertyDescriptor` constructors contain the same kind of information as the property descriptor annotations described in [Property Descriptor Annotations](#), because they are both specifying the same thing.

"TranslateAttribute" and "RotateAttribute" both use a `NumericTupleEditor` value editor, because they both are of type "vector3Type", a vector of 3 `float` values. The `NumericTupleEditor` constructor takes two values for the data type and the names of the numeric fields:

```
public NumericTupleEditor(Type numericType, string[] names)
```

This descriptor definition also contains the associated value converter `FloatArrayConverter`, required by `NumericTupleEditor`.

Although the attribute "ScaleAttribute" also has the type "vector3Type", it uses a different value editor `UniformArrayEditor`. This editor also handles an array of numbers, but only one value is exposed in the value editing control and all the array's numbers are set to this value. This forces the scaling of all three dimensions to be identical for this property. This demonstrates that while an attribute must use a value editor appropriate for its value type, it is not limited to only one choice.

For further discussion of these two ways of specifying property descriptors and what is required, see [Property Descriptors in DOM Tree Editor](#).

DOM Tree Editor DOM Adapters

The classes whose names begin with "UI" — almost half the classes in DOM Tree Editor — are DOM adapters. Most are very simple, merely providing properties for their attributes. For instance, the base class `UIObject` only describes a `Name` property:

```

/// <summary>
/// Base for all UI DomNode adapters, with a name attribute</summary>
public abstract class UIObject : DomNodeAdapter
{
    /// <summary>
    /// Gets or sets the UI object's name</summary>
    public string Name
    {
        get { return GetAttribute<string>(UISchema.UIObjectType.nameAttribute); }
        set { SetAttribute(UISchema.UIObjectType.nameAttribute, value); }
    }
}

```

The "Name" attribute is common to all types. The class UIControl is also fairly simple, deriving from UIObject:

```

public class UIControl : UIObject
{
    /// <summary>
    /// Performs initialization when the adapter is connected to the diagram annotation's DomNode
</summary>
    protected override void OnNodeSet()
    {
        DomNode transform = DomNode.GetChild(UISchema.UIControlType.TransformChild);
        if (transform == null)
        {
            transform = new DomNode(UISchema.UITransformType.Type);
            transform.SetAttribute(UISchema.UITransformType.ScaleAttribute, new float[] { 1.0f,
1.0f, 1.0f });
            DomNode.SetChild(UISchema.UIControlType.TransformChild, transform);
        }
    }

    /// <summary>
    /// Gets the control's transform</summary>
    public UITransform Transform
    {
        get { return GetChild<UITransform>(UISchema.UIControlType.TransformChild); }
    }

    /// <summary>
    /// Gets the list of all child controls in the control</summary>
    public IList<UIControl> Controls
    {
        get { return GetChildList<UIControl>(UISchema.UIControlType.ControlChild); }
    }
}

```

According to its definition, "UIControlType" can have an unlimited number of "UIControlType" children, so its `Controls` property returns a list of these children:

```

<xss:complexType name="UIControlType" abstract="true">
    <xss:complexContent>
        <xss:extension base="UIObjectType">
            <xss:sequence>
                <xss:element name="Transform" type="UITransformType" minOccurs="1" maxOccurs="1"/>
                <xss:element name="Animation" type="UIAnimationType" minOccurs="0" maxOccurs="1"/>
                <xss:element name="Control" type="UIControlType" minOccurs="0" maxOccurs="unbounded"/>
            </xss:sequence>
        </xss:extension>
    </xss:complexContent>
</xss:complexType>

```

It has only one "UITransformType" child, which its `Transform` property returns.

DOM adapters do not necessarily define properties for all the type's attributes' values. If the value is not used programmatically, then no property is needed. For example, though "UITextItemType" has a "text" attribute, there is no property to obtain its value in the UITextItem DOM adapter.

Most of the DOM adapters don't even have `OnNodeSet()` methods, although `UIControl` does. Its main task is to set the scale attribute's values for the transform associated with the control object, if the transform is not already set:

```
transform.SetAttribute(UISchema.UITransformType.ScaleAttribute, new float[] { 1.0f, 1.0f, 1.0f } ) ;
```

Because of this, whenever a control object, `Sprite` or `Text`, is dragged onto the editor window, it already has its scale values set.

The other two DOM adapters are `ControlPoint` and `Curve`. These adapters are primarily devoted to implementing `IControlPoint` and `ICurve` respectively. `IControlPoint` is used by `ICurve`, and the `CurveEditor` component manipulates `ICurve` objects, which represent curves. Inside DOM Tree Editor, `Curve` is used by `UIAnimation` whose `OnNodeSet()` method creates a set of curves that are displayed in the Curve Editor window when an Animation object is selected.

The `SchemaLoader` defines DOM adapters for their corresponding types:

```
// register adapters to define the UI object model
UISchema.UIPackageType.Type.Define(new ExtensionInfo<UIPackage>());
UISchema.UIFormType.Type.Define(new ExtensionInfo<UIForm>());
UISchema.UIShaderType.Type.Define(new ExtensionInfo<UIShader>());
UISchema.UITextureType.Type.Define(new ExtensionInfo<UITexture>());
UISchema.UIFontType.Type.Define(new ExtensionInfo<UIFont>());
UISchema.UISpriteType.Type.Define(new ExtensionInfo<UISprite>());
UISchema.UITextItemType.Type.Define(new ExtensionInfo<UITextItem>());
UISchema.UIRefType.Type.Define(new ExtensionInfo<UIRef>());
UISchema.UIAnimationType.Type.Define(new ExtensionInfo<UIAnimation>());
UISchema.curveType.Type.Define(new ExtensionInfo<Curve>());
UISchema.controlPointType.Type.Define(new ExtensionInfo<ControlPoint>());
```

Palette Implementation

DOM Tree Editor creates and uses a palette with its `PaletteClient` class and palette data set up, described in Schema Loader. Palette implementation is very similar to the [ATF Simple DOM Editor Sample](#). For details on setting up a palette, see [Using a Palette in Simple DOM Editor Programming Discussion](#).

Document Handling

This sample's document handling is straightforward and in the fashion of several other samples. Its `Document` class derives from `DomDocument`, a DOM adapter that implements `IDocument`. Its `Editor` component implements `IDocumentClient` to handle opening, saving, and closing documents, using XML to persist documents.

`Document` is defined as the DOM adapter for "UIType", the type of the `DomNode` tree:

```
UISchema.UIType.Type.Define(new ExtensionInfo<Document>());
```

For general information about creating documents, see [Implementing a Document and Its Client](#). More specifically, the [ATF Simple DOM Editor Sample](#) handles documents much the same as this sample. For details, see [Document Handling in Simple DOM Editor Programming Discussion](#).

Tree Handling: TreeLister and TreeView

The editing window holding the current document displays its contents as a tree using the `TreeLister` component, which derives from `TreeControlEditor`, the base class for tree editors. Only one document can be open at a time.

`TreeView` is the DOM adapter for the root `DomNode` in the tree of application data. Similarly to other samples such as [ATF File Explorer Sample](#) and [ATF Circuit Editor Sample](#), it implements `ITreeView`, `IItemView`, and `IObservableContext` so the data in the tree can be displayed by `TreeLister`:

- `ITreeView`: Define a view on hierarchical data so it can be displayed in a tree.
- `IItemView`: Provide display information about an item in an `ItemInfo` object. `ItemInfo` holds information on the appearance and behavior of an item in a list or tree control.
- `IObservableContext`: Its events trigger refreshing the editor window display.

`TreeView` is defined as the DOM adapter for "UIType", the type of the `DomNode` tree root:

```
UISchema.UIType.Type.Define(new ExtensionInfo<TreeView>());
```

For information on how other samples and classes use these interfaces to display data in a tree, see [File Explorer Programming Discussion](#) and [Circuit Graph Support](#).

`TreeLister` is the control host for the tree control, implementing `IControlHostClient`. Its `Activate()` method sets the active context to the `TreeView` property, which is the `ITreeView` object displayed by the editor. Its `Close()` method is invoked when the application ends and closes the open document, if any. The `OnLastHitChanged()` method performs the important function of setting the last object in the tree a palette item was dragged over and the mouse button released, which is the potential insertion parent for the palette item.

The `TreeControlEditor` handles drag and drop operations in the tree, in this case, dragging and dropping objects from the palette onto their new parent objects.

Context Handling

The `EditingContext` class derives from `Sce.Atf.Dom.EditingContext`, which is used as an editing context in several samples, such as [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), and [ATF State Chart Editor Sample](#). For more information on this useful context, see [Sce.Atf.Dom.EditingContext Class](#).

`EditingContext` is defined as the DOM adapter for "UIType", the type of the `DomNode` tree:

```
UISchema.UIType.Type.Define(new ExtensionInfo<EditingContext>());
```

`EditingContext`'s main job is implementing `IInstancingContext` to create UI element instances when items are dragged from the palette onto the tree control. Its methods are called during instancing operations — dragging and dropping of palette items onto the tree. For a discussion of instancing, see [Instancing In ATF](#).

`EditingContext` must enforce the parenting relationships discussed in [DOM Tree Editor Data Definition](#). Several types of objects have child objects, but only of certain types.

Here, the key method is `IInstancingContext.CanInsert()`, which is called when a palette item is dragged over an object in the tree and the mouse button released. `CanInsert()` determines whether the item can be inserted at the insertion point, which is the last object the item was dragged over and the mouse button released. This insertion point is set by `SetInsertionParent()`, which was called by `TreeLister.OnLastHitChanged()`. `CanInsert()` checks whether the insertion object is suitable by calling `CanParent()`:

```
private bool CanParent(DomNode parent, DomNodeType childType)
{
    return GetChildInfo(parent, childType) != null;
}
```

`GetChildInfo()` checks if the type of the potentially inserted item is one of the child types under the parent. If not, it returns `null`, so the insertion fails:

```
private ChildInfo GetChildInfo(DomNode parent, DomNodeType childType)
{
    foreach (ChildInfo childInfo in parent.Type.Children)
        if (childInfo.Type.IsAssignableFrom(childType))
            return childInfo;
    return null;
}
```

In addition, the control objects, `Sprite` and `Text`, can take a reference to a `Shader` or `Font` object, respectively, when the right kind of object is dragged to the empty reference "slot". The `EmptyRef` class represents the empty slot for a reference before it is set. The `CanReference()` method checks the validity of these kind of insertions by checking that the right kind of object is dropped over its parent:

```

private bool CanReference(EmptyRef emptyRef, DomNodeType childType)
{
    return
        // dropping shader on sprite?
        ((childType == UISchema.UIShaderType.Type) &&
        emptyRef.ChildInfo.IsEquivalent(UISchema.UISpriteType.ShaderChild)) ||

        // dropping font on text item?
        ((childType == UISchema.UIFontType.Type) &&
        emptyRef.ChildInfo.IsEquivalent(UISchema.UITextItemType.FontChild));
}

```

Finally, the `IInstancingContext.Insert()` method performs the actual insertion: creating new `DomNodes`, initializing their extensions (DOM adapters), and adding them as children to the parent `DomNode`. It must also handle references appropriately for Sprite and Text parents.

Validation in DOM Tree Editor

This sample uses several validators to ensure the integrity of the application data by examining the `DomNode` tree in response to various events.

The `Validator` class is particular to DOM Tree Editor and checks that resources that are referenced are available in the same package, responding to `DomNodes` being inserted in the tree. `Validator` derives from `Sce.Atf.Dom.Validator`, the abstract base class for validators that need to track all validation events in a subtree.

DOM Tree Editor employs a couple other general purpose validators to maintain `DomNode` tree integrity, and these are all defined for the type of the root `DomNode` "UIType", so the entire tree is checked:

```

UISchema.UIType.Type.Define(new ExtensionInfo<Validator>()); // makes sure referenced resources
are in package
                                                // this must be first so unique
naming can work on copied resources
UISchema.UIType.Type.Define(new ExtensionInfo<ReferenceValidator>()); // prevents dangling
references
UISchema.UIType.Type.Define(new ExtensionInfo<UniqueIdValidator>()); // makes sure ref targets
have unique ids

```

The other validators do the following:

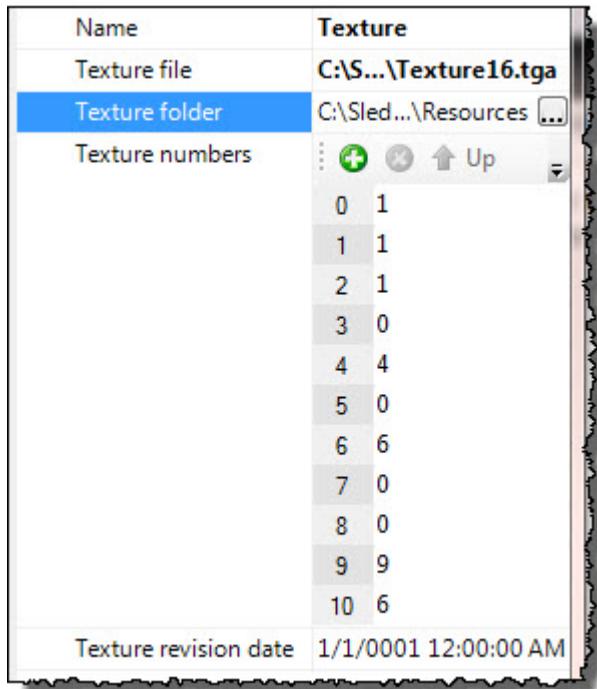
- `ReferenceValidator`: Track references and reference holders to ensure integrity of references within the DOM data.
- `UniqueIdValidator`: Ensure that every `DomNode` in a subtree has a unique ID.

For a general discussion of validators, see the ATF Programmer's Guide: Document Object Model (DOM).

Property Descriptors in DOM Tree Editor

To display property attributes of application data in property editors, you must specify a property descriptor for every attribute you want to view and edit. Property descriptors hold such information as the property metadata attribute, the user interface name, and a value editor to display the value in its associated value editing control. For details on specifying properties, see [Property Editing in ATF](#).

DOM Tree Editor offers a rich variety of types in its data model to demonstrate the various value editors and their controls. For instance, this figure shows texture properties, its attributes taking on several different types:



This shows several value editors:

- Texture file: `FileUriEditor` to select a URI for the texture file.
- Texture folder: `FolderBrowserDialogUITypeEditor` to select texture folder.
- Texture numbers: `ArrayEditor` to specify an array of numbers.
- Texture revision date: `DateTimeEditor` to set the revision date.

For a description and illustrations of all ATF value editors, see [Value Editors and Value Editing Controls](#).

Specifying Property Descriptors

As previously mentioned, you can describe property descriptors two ways in ATF: by an XML Schema annotation or by a property descriptor constructor, usually in the schema loader. DOM Tree Editor employs both methods.

XML Schema Annotation Property Descriptors

Here is the type definition for the "UIShaderType" from the type definition file `UISchema.xsd`:

```
<xs:complexType name="UIShaderType">
  <xs:annotation>
    <xs:appinfo>
      <scea.dom.editors.attribute name="ShaderID" displayName="Shader ID" description="Shader ID"
        editor="Sce.Atf.Controls.PropertyEditing.BoundedIntEditor,Atf.Gui.WinForms:10,1000"
        converter="Sce.Atf.Controls.PropertyEditing.BoundedIntConverter" />
    </xs:appinfo>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="UIObjectType">
      <xs:attribute name="FxFile" type="xs:string" use="required"/>
      <xs:attribute name="ShaderID" type="xs:int" use="required"/>
      <xs:attribute name="ShaderParam" type="xs:int" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The annotation for the Shader ID attribute specifies:

- Name
- Display name and description
- Value editor `BoundedIntEditor`
- Value converter `BoundedIntConverter`

The paths for the value editor `BoundedIntEditor` and converter `BoundedIntConverter` are fully qualified, plus the assembly for the

editor. The editor is also followed by parameters indicating its lower and upper limits.

Note that this annotation only specifies the property descriptor for one of the three attributes of this property. The rest are specified in property descriptor constructors, described in the next section.

To process annotations in the XML Schema, you must provide override the base `XmlSchemaTypeLoader.ParseAnnotations()` method, which is described in [ParseAnnotations\(\) Method](#).

Property Descriptor Constructors

The remaining property descriptors for the "UIShaderType" type's attributes are in `SchemaLoader.OnSchemaSetLoaded()`:

```
UISchema.UIShaderType.Type.SetTag(  
    new PropertyDescriptorCollection(  
        new PropertyDescriptor[] {  
            new AttributePropertyDescriptor(  
                Localizer.Localize("Shader"),  
                UISchema.UIShaderType.FxFileAttribute,  
                null,  
                Localizer.Localize("Shader file path"),  
                false,  
                new FileUriEditor("Fx Files (*.fx)|*.fx")),  
            new AttributePropertyDescriptor(  
                Localizer.Localize("Shader param"),  
                UISchema.UIShaderType.ShaderParamAttribute,  
                null,  
                Localizer.Localize("Shader param"),  
                false,  
                new NumericEditor(typeof(Int32)))  
        }));
```

The call to `NamedMetadata.SetTag()` sets a tag for the `UIShaderType` attribute consisting of a `PropertyDescriptorCollection` containing an array of `AttributePropertyDescriptor`s. The constructor for each `AttributePropertyDescriptor` contains much the same information as the annotation:

```
public AttributePropertyDescriptor(  
    string name,  
    AttributeInfo attribute,  
    string category,  
    string description,  
    bool isReadOnly,  
    object editor)
```

Constructors are provided for the value editors `FileUriEditor` and `NumericEditor` used to edit the attributes in property editors.

ParseAnnotations() Method

`ParseAnnotations()` parses annotations in XML schemas to create property descriptors from information in annotations. This method is overridden here, because the base method does not call `PropertyDescriptor.ParseXml()`, which processes `<scea.dom.editors.attribute>` annotations for property descriptors. This override does call `ParseXml()`:

```

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);
            if (propertyDescriptors != null && propertyDescriptors.Count > 0)
            {
                // Property descriptor annotation found. Add any descriptors already set for this
                type.
                PropertyDescriptorCollection propertyDescriptorsAlreadySet =
                    nodeType.GetTag<PropertyDescriptorCollection>();
                if (propertyDescriptorsAlreadySet != null)
                    foreach (PropertyDescriptor desc in propertyDescriptorsAlreadySet)
                        propertyDescriptors.Add(desc);
                // Set all property descriptors
                nodeType.SetTag<PropertyDescriptorCollection>(propertyDescriptors);
            }
        }
    }
}

```

The method `XmlSchemaTypeLoader.Load()` loads the schema from the type definition file. In `Load()`, `OnSchemaSetLoaded()` is called first, followed by `ParseAnnotations()`. `OnSchemaSetLoaded()` is where property descriptor constructors are processed, and property descriptor annotations get handled in `ParseAnnotations()`.

The inner loop in `ParseAnnotations()` must do special processing to ensure that property descriptors specified by both an annotation and a property descriptor constructor are both added to the `PropertyDescriptorCollection` collection for the property. Otherwise, the property descriptors set in `OnSchemaSetLoaded()` would be overwritten by the ones `ParseAnnotations()` sets. If a property descriptor was found in the annotation, `ParseAnnotations()` gets any property descriptors already found for the type and adds them to the `PropertyDescriptorCollection` for the type.

For more information on parsing annotations, see the ATF Programmer's Guide: Document Object Model (DOM).

Topics in this section

Links on this page to other topics

[ATF Circuit Editor Sample](#), [ATF Documentation](#), [ATF DOM Tree Editor Sample](#), [ATF File Explorer Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [Circuit Graph Support](#), [Context Classes](#), [File Explorer Programming Discussion](#), [Implementing a Document and Its Client](#), [Instancing In ATF](#), [Property Editing in ATF](#), [Property Editor Components](#), [Simple DOM Editor Programming Discussion](#), [Value Editors and Value Editing Controls](#)

File Explorer Programming Discussion

The [ATF File Explorer Sample](#) shows how to adapt controls to data. The folder view in the main window's left panel uses a `TreeControl` with a `TreeControlAdapter`, and the file panel on the right uses a `System.Windows.Forms.ListView` with a `ListViewAdapter`.

The sample also shows how to use MEF to add an arbitrary number of components to specify file information to display in a `ListView`.

Programming Overview

This page discusses the custom components added, such as `FolderViewer` and `FileViewer`, which control operations in the two panels of the main window. These components operate similarly, both creating controls and adapters that adapt the data to the control's view. The components each define a private view class, such as `FileTreeView`, that implements interfaces for the adapters to provide capabilities like viewing and selection.

The sample defines an `IFileDataExtension` interface that provides file and folder information. The sample defines several components that implement and export this interface. `FileViewer` imports these components and iterates through them, so that it is easy to add an additional file information provider component.

Topics

- [Programming Overview](#)
- [File Explorer Components](#)
- [FolderViewer Component](#)
 - [TreeControlAdapter Class](#)
 - [FileTreeView Class](#)
- [FileViewer Component](#)
 - [ListViewAdapter Class](#)
 - [FileListView Class](#)
- [IFileDataExtension Provider Components](#)
 - [IFileDataExtension Interface](#)
 - [IFileDataExtension Exporter Components](#)

File Explorer Components

File Explorer follows the standard WinForms initialization pattern discussed in [WinForms Application](#). In addition to standard ATF components, it adds the following components of its own to the MEF TypeCatalog:

- `FolderViewer`: manage a `TreeControl` to display a folder hierarchy of the computer's "C:" drive. For discussion, see [FolderViewer Component](#).
- `FileViewer`: manage a `ListView` to display the selected folder's contents. For details, see [FileViewer Component](#).
- File extension components implementing `IFileDataExtension`, which provides information about files and folders. For the details, see [IFileDataExtension Exporter Components](#). There are three extensions defined:
 - `NameDataExtension`: extension in `FileDataExtensions.cs` to display the folder or file name.
 - `SizeDataExtension`: extension in `FileDataExtensions.cs` to display the file's size.
 - `CreationTimeDataExtension`: extension in `FileDataExtensions.cs` to display folder or file creation time.

FolderViewer Component

The `FolderViewer` component's constructor creates a `TreeControl` to display the hierarchical data of the "C:" drive's contents as a tree:

```
[ImportingConstructor]
public FolderViewer(MainForm mainForm, FileViewer fileViewer)
{
    m_mainForm = mainForm;
    m_fileViewer = fileViewer;

    m_treeControl = new TreeControl();
    m_treeControl.Text = "Folder Viewer";
    m_treeControl.ImageList = ResourceUtil.GetImageList16();
    m_treeControl.SelectionMode = SelectionMode.One;
    m_treeControl.Dock = DockStyle.Fill;
    m_treeControl.Text = "Folder Viewer";
    m_treeControl.Width = 256;

    m_treeControlAdapter = new TreeControlAdapter(m_treeControl);
    m_fileTreeView = new FileTreeView();
    m_fileTreeView.SelectionChanged += new EventHandler(fileTreeView_SelectionChanged);
    m_treeControlAdapter.TreeView = m_fileTreeView;
}
```

The `FileViewer` parameter imported is a `FileViewer` object, discussed in [FileViewer Component](#).

The constructor sets various properties of the `new TreeControl`. For example, the `ImageList` property is set to a list of image resources, so that an image can be used for folders in the `TreeControl`. The `SelectionMode` property is set so that only one item can be selected at a time.

After setting the `TreeControl`'s properties, the constructor creates a `TreeControlAdapter` for that tree, which adapts the `TreeControl` to a data context that implements `ITreeView`. Finally, a new `FileTreeView` is created, which adapts the "C:" drive's contents to a tree of selectable items.

The component's `IInitializable.Initialize()` method sets the left panel of the main window to the newly created `TreeControl`:

```
void IInitializable.Initialize()
{
    m_mainForm.SplitContainer.Panel1.Controls.Add(m_treeControl);
}
```

The handler for the `SelectionChanged` event sets the `Path` property of the imported `FileViewer` component to the path of the selected folder:

```
private void fileTreeView_SelectionChanged(object sender, EventArgs e)
{
    Sce.Atf.Path<object> lastPath = m_fileTreeView.LastSelected as Sce.Atf.Path<object>;
    if (lastPath != null)
    {
        FileSystemInfo info = lastPath.Last as FileSystemInfo;
        if (info is DirectoryInfo)
            m_fileViewer.Path = info.FullName;
    }
}
```

`FileViewer`'s `Path` determines what is displayed in the right panel of the main window, which is discussed in [FileViewer Component](#).

TreeControlAdapter Class

`FolderViewer` uses this `TreeControlAdapter` constructor:

```
public TreeControlAdapter(TreeControl treeControl)
    : this(treeControl, null)
{ }
```

This constructor invokes the actual constructor used:

```

public TreeControlAdapter(TreeControl treeControl, IEqualityComparer<object> comparer)
{
    m_treeControl = treeControl;
    m_itemToNodeMap = new Multimap<object, TreeControl.Node>(comparer);

    m_treeControl.MouseDown += treeControl_MouseDown;
    m_treeControl.MouseUp += treeControl_MouseUp;
    m_treeControl.DragOver += treeControl_DragOver;
    m_treeControl.DragDrop += treeControl_DragDrop;

    m_treeControl.NodeExpandedChanged += treeControl_NodeExpandedChanged;
    m_treeControl.NodeSelectedChanged += treeControl_NodeSelectedChanged;
    m_treeControl.SelectionChanging += treeControl_SelectionChanging;
    m_treeControl.SelectionChanged += treeControl_SelectionChanged;
}

```

This code subscribes to event handlers for mouse operations, drag and drop, node expansion/contraction, and selection changes. This sample application uses all these events, except the drag and drop ones.

`TreeControlAdapter` has a `TreeView` property, shown here with its comments:

```

/// <summary>
/// Gets or sets the tree displayed in the control. When setting, consider having the
/// ITreeView object also implement IItemView, IObservableContext, IValidationContext,
/// ISelectionContext, IInstancingContext, and IHierarchicalInsertionContext.</summary>
public ITreeView TreeView
...

```

Recall that `FolderViewer`'s constructor sets this `TreeView` property to the newly created `FileTreeView`, which implements `ITreeView`:

```

m_fileTreeView = new FileTreeView();
m_fileTreeView.SelectionChanged += new EventHandler(fileTreeView_SelectionChanged);
m_treeControlAdapter.TreeView = m_fileTreeView;

```

It also subscribes the `FileTreeView` to the `SelectionChanged` event, whose handler was shown previously.

FileTreeView Class

`FileTreeView` is a private class that implements `ITreeView` for `TreeControlAdapter`:

```

private class FileTreeView : ITreeView, IItemView, ISelectionContext

```

Its constructor sets up a new `Selection` object:

```

public FileTreeView()
{
    m_selection = new Selection<object>();
    m_selection.Changed += new EventHandler(selection_Changed);

    // suppress compiler warning
    if (SelectionChanging == null) return;
}

```

Selected items in the tree represent folders, and `Selection` provides what's needed to select these items. `FileTreeView` works mainly with `System.IO.DirectoryInfo` objects, which encapsulate information for a directory.

`FileTreeView` implements several interfaces.

ITreeView Interface

`ITreeView` generalizes a tree to a root with child objects:

- `Root`: property for the root object of the tree view.
- `GetChildren()`: get children of the given parent.

The `ITreeView` implementation works with file system objects, as seen in the next code example. `Root` simply returns directory information for the "C:" drive, because `m_path` is initialized to "C:\\". `GetChildren()` gets the files and folders in a parent folder.

```
public object Root
{
    get { return new DirectoryInfo(m_path); }
}

public IEnumerable<object> GetChildren(object parent)
{
    IEnumerable<object> result = null;
    DirectoryInfo directoryInfo = parent as DirectoryInfo;
    if (directoryInfo != null)
        result = GetSubDirectories(directoryInfo); //may return null

    if (result == null)
        return Enumerable<object>.Instance;
    return result;
}
```

The private method `GetSubDirectories()` gets the contents of a directory, as an array of `DirectoryInfo` objects.

IItemView Interface

`IItemView` continues the work with the file system. Its `GetInfo()` method fills out an `ItemInfo` with information from a `DirectoryInfo`: the folder name, including whether it's a leaf, that is, whether it contains no files or folders.

```
public void GetInfo(object item, Sce.Atf.Applications.ItemInfo info)
{
    DirectoryInfo directoryInfo = item as DirectoryInfo;
    if (directoryInfo != null)
    {
        info.Label = directoryInfo.Name;
        info.ImageIndex = info.GetImageList().Images.IndexOfKey(Resources.ComputerImage);
        DirectoryInfo[] directories = GetSubDirectories(directoryInfo);

        info.IsLeaf =
            directories != null &&
            directories.Length == 0;
    }
}
```

`ItemInfo`'s `ImageIndex` property is set to display a computer icon for each item in the tree view, using standard ATF image resources from the `Resources` class.

ISelectionContext Interface

`ISelectionContext` is the general interface for selections. It's very easy to implement here, because the constructor created a `Selection` object:

```
m_selection = new Selection<object>();
```

This interface implementation uses the `Selection` class's methods to provide everything `ISelectionContext` needs. For instance, `GetSelection<T>()` does this:

```
public IEnumerable<T> GetSelection<T>()
    where T : class
{
    return m_selection.AsIEnumerable<T>();
}
```

This interface also includes the `SelectionChanged` event, which is raised after the selection changes. This is important, because changing the selected item in the tree representation of the "C:" drive determines what's displayed in the right panel of the main window. As previously mentioned, this event's handler places the selected folder's path in `FileViewer`'s `Path` property, which governs which folder's contents appear in `FileViewer`'s `ListView` in the main window's right panel.

FileViewer Component

FileViewer displays the contents of a folder using a ListView control and parallels FolderViewer in its operation. FileViewer's constructor does basically the same things as FolderViewer's constructor: it creates a control, configures it, and then creates an adapter for the control. FileViewer creates a ListView rather than a TreeControl:

```
[ ImportingConstructor ]
public FileViewer(MainForm mainForm)
{
    m_mainForm = mainForm;

    // create a standard WinForms ListView control
    m_listView = new ListView();
    m_listView.Dock = DockStyle.Fill;
    m_listView.Text = "File Viewer";
    m_listView.BackColor = SystemColors.Window;
    m_listView.SmallImageList = ResourceUtil.GetImageList16();
    m_listView.AllowColumnReorder = true;

    // create an adapter to drive the ListView control
    m_listViewAdapter = new ListViewAdapter(m_listView);

    m_fileListView = new FileListView();
}
```

The new ListView's SmallImageList property is set to display small icons for files and folders in the ListView.

After the ListView's properties are set, a ListViewAdapter for that list is created. A ListViewAdapter adapts a ListView to a data context that implements IListView. Finally, a new FileListView is created, which adapts a directory to an observable list of items, that is, the folders and files in the selected folder in the TreeControl.

The component's IInitializable.Initialize() method accomplishes the following:

- Makes a list of all the IFileDataExtension providers, which were imported into the m_extensions field:

```
[ ImportMany ] // gets all file data extensions
private IEnumerable<Lazy<IFileDataExtension>> m_extensions = null;
```
- Sets the ListViewAdapter's ListView property to the new FileListView.
- Adds the newly created ListView to the right panel of the main window.

```
void IInitializable.Initialize()
{
    // pass all file data extensions to adapter
    List<IFileDataExtension> list = new List<IFileDataExtension>();
    foreach (Lazy<IFileDataExtension> extension in m_extensions)
        list.Add(extension.Value);

    m_fileListView.FileDataExtensions = list.ToArray();

    // set the adapter's ListView to an adapter that returns directory contents
    m_listViewAdapter.ListView = m_fileListView;

    m_mainForm.SplitContainer.Panel2.Controls.Add(m_listView);

    SettingsServices.RegisterSettings(
        m_settingsService,
        this,
        new BoundPropertyDescriptor(this, () => ListViewSettings, "ListViewSettings", null, null
    );
}
```

FileViewer's Path property gets or sets the path to the folder whose contents are displayed in the ListView:

```

public string Path
{
    get { return m_fileListView.Path; }
    set
    {
        m_fileListView.Path = value;
        m_mainForm.Text = value;
    }
}

```

Note that this property simply accesses the `Path` property of the `FileListView` object. As discussed previously, `FileViewer`'s `Path` property is set to the currently selected file path in the tree view when the selection changes by the `FolderViewer`'s `SelectionChanged` event handler.

ListViewAdapter Class

`FileViewer` uses this `ListViewAdapter` constructor:

```

public ListViewAdapter(ListView listView)
{
    m_control = listView;
    m_control.View = View.Details;
    m_control.FullRowSelect = true;
    m_control.HideSelection = false;

    // default to allow sorting
    m_allowSorting = true;
    m_control.ListViewItemSorter = new ListViewItemSorter(m_control);

    m_control.AfterLabelEdit += control_AfterLabelEdit;
    m_control.ColumnWidthChanged += control_ColumnWidthChanged;
    m_control.MouseDown += control_MouseDown;
    m_control.MouseUp += control_MouseUp;
    m_control.DragOver += control_DragOver;
    m_control.DragDrop += control_DragDrop;
}

```

This code subscribes to event handlers for label and column changes, mouse operations, and drag and drop. This sample uses all these events, except the drag and drop ones.

`ListViewAdapter` has a `ListView` property, which gets or sets the `IListView` object for the list data. `FileViewer`'s `IIInitializable.Initialize()` method sets this property to the `FileListView` object, which implements `IListView`:

```

// set the adapter's ListView to an adapter that returns directory contents
m_listViewAdapter.ListView = m_fileListView;

```

`FileListView` is analogous to the `FileTreeView` that `TreeControlAdapter` uses and serves much the same function: to handle data in the view, a list view in this case — rather than a tree view.

FileListView Class

`FileListView` is a private class that implements `IListView` for the `ListViewAdapter`:

```

private class FileListView : IListView, IItemView, IObservableContext

```

This class does not implement selection, but it does implement several other interfaces. Although it implements `IObservableContext`, which contains events, it does not raise these events.

IListView Interface

`IListView` is the main interface used with `ListView` controls and abstracts an enumeration of objects that can be used as tags, one per row, in a list control, along with corresponding user-readable column names at the top.

The `ColumnNames` property provides an array of strings for the column names:

```

public string[] ColumnNames
{
    get
    {
        string[] result = new string[m_fileDataExtensions.Length];
        for (int i = 0; i < result.Length; i++)
            result[i] = m_fileDataExtensions[i].ColumnName;
        return result;
    }
}

```

`ColumnNames` enumerates the `IFileDialogExtension` providers to get the `ColumnName` property for each one. Recall how `m_fileDataExtensions` was set up to contain all the `IFileDialogExtension` providers:

```

List<IFileDialogExtension> list = new List<IFileDialogExtension>();
foreach (Lazy<IFileDialogExtension> extension in m_extensions)
    list.Add(extension.Value);

m_fileListView.FileDataExtensions = list.ToArray();

```

And `m_extensions` is the field into which all the `IFileDialogExtension` providers are imported:

```

[ImportMany] // gets all file data extensions
private IEnumerable<Lazy<IFileDialogExtension>> m_extensions = null;

```

For further explanation of how `IFileDialogExtension` is used, see [IFileDialog Extension Components](#).

The `IListView.Items` property contains all the items that the `ListView` displays. That is, it displays all the files and folders in the folder selected in the `TreeControl` in the left panel of the main window. Therefore, `Items` uses the `m_path` field, which, as previously noted, contains the currently selected file path in the tree view. `Items` uses `DirectoryInfo` methods to get lists of all the directories and files in the path specified by `m_path`:

```

public IEnumerable<object> Items
{
    get
    {
        if (m_path == null ||
            !Directory.Exists(m_path))
        {
            return EmptyEnumerable<object>.Instance;
        }

        DirectoryInfo directory = new DirectoryInfo(m_path);
        DirectoryInfo[] subDirectories = null;
        try
        {
            subDirectories = directory.GetDirectories();
        }
        catch
        {
        }
        if (subDirectories == null)
            subDirectories = new DirectoryInfo[0];

        FileInfo[] files = null;
        try
        {
            files = directory.GetFiles();
        }
        catch
        {
        }
        if (files == null)
            files = new FileInfo[0];

        List<object> children = new List<object>(subDirectories.Length + files.Length);
        children.AddRange(subDirectories);
        children.AddRange(files);
        return children;
    }
}

```

ItemView Interface

IItemView gets information about individual file or folder items to display in the view. This interface was also implemented by FileTreeView to display information about folders in the tree view.

GetInfo() fills out a given ItemInfo object to display item's information in the ListView:

```

public void GetInfo(object item, ItemInfo info)
{
    // set the first column info (name)
    FileSystemInfo fileInfo = item as FileSystemInfo;
    if (fileInfo is DirectoryInfo)
    {
        info.Label = fileInfo.Name;
        info.ImageIndex = info.GetImageList().Images.IndexOfKey(Resources.FolderImage);
    }
    else if (fileInfo is FileInfo)
    {
        info.Label = fileInfo.Name;
        info.ImageIndex = info.GetImageList().Images.IndexOfKey(Resources.DocumentImage);
        info.IsLeaf = true;
    }

    // set the 2nd and 3rd columns info (size and creation time)
    info.Properties = new string[m_fileDataExtensions.Length-1];
    for (int i = 0; i < info.Properties.Length; i++)
        info.Properties[i] = m_fileDataExtensions[i+1].GetValue(fileInfo);
}

```

`GetInfo()` casts the item to `FileSystemInfo` and determines whether it specifies information for a folder or file. It obtains the item's name and the appropriate image, depending on whether its a folder or file. Similarly to the implementation of `IListView.ColumnNames`, this property iterates the `IFileDataExtension` providers in `m_fileDataExtensions` to get file or folder information and then add it to the `ItemInfo`'s `Properties` array.

IFileDataExtension Provider Components

This sample creates the `IFileDataExtension` interface to define components that provide file information for a given `FileSystemInfo` object. These components get the folder and file information displayed in `FileViewer`'s `ListView`. As will be seen, you can easily add similar components to display additional information about folders and files.

IFileDataExtension Interface

`IFileDataExtension` defines two items in its interface:

- `string ColumnName` property: get a string for the column name in the `ListView` for the file information. This information is obtained for `FileListView`'s `IListView.ColumnNames` property, as described in [FileListView Class](#).
- `string GetValue(FileSystemInfo fileInfo)` method: get a string value describing file information, such as a size or creation time, from a given `FileSystemInfo`. `FileListView`'s `IItemView.GetInfo()` method uses this to get the information displayed in the `ListView`.

IFileDataExtension Exporter Components

The file `FileDialogExtension.cs` contains several components that export `IFileDataExtension`. These components are imported by `FileViewer`, as previously described.

Note that `FileViewer` imports these components using `[ImportMany]`, so that an arbitrary number of these components can be imported. In addition, `FileViewer` iterates through the list of imported `IFileDataExtension` providers, so that it is easy to add an additional provider.

All these components operate pretty much the same. For example, here's the `CreationTimeDataExtension` component that obtains the creation time:

```
[Export(typeof(IFileDataExtension))]
public class CreationTimeDataExtension : IFileDataExtension
{
    /// <summary>
    /// Gets the name of the column</summary>
    public string ColumnName
    {
        get { return "Creation Time"; }
    }

    /// <summary>
    /// Gets the value for the column and given file system item</summary>
    /// <param name="fileSystemInfo">Info describing the file or directory</param>
    /// <returns>Value for the column and given file system item</returns>
    public string GetValue(FileSystemInfo fileInfo)
    {
        return fileInfo.CreationTime.ToString(CultureInfo.CurrentCulture);
    }
}
```

`ColumnName` simply provides a string for the name to be used as the column label in the `ListView`.

`GetValue()` extracts the creation time from the given `FileSystemInfo`, returning it as a string.

Topics in this section

Links on this page to other topics

[ATF File Explorer Sample, Authoring Tools Framework, WinForms Application](#)

FSM Editor Programming Discussion

The [ATF FSM Editor Sample](#) shows how to use the ATF graph facilities to create a finite state machine (FSM) editor, so it has some similarities to the [ATF Circuit Editor Sample](#), which is described in [Circuit Editor Programming Discussion](#). You can drag states from a palette onto a canvas and then connect them with transitions. You can also drag comment windows onto the canvas to enter explanatory text. You can copy items and paste them onto the Prototypes window for later reuse by dragging them back onto the canvas.

For information on a fully functional state machine tool based on ATF, see the [StateMachine Home Page](#).

Programming Overview

FSM Editor is a graph editor, so it employs the various ATF graph handling mechanisms; for details see [Graphs in ATF](#). ATF does not offer special support for the simple graphs that FSM Editor uses, so see [General Graph Support](#) as well.

The data model is quite simple with no extension of data types and only a few data types to describe the limited number of state machine items.

Most of the classes in FSM Editor are DOM adapters, and they handle graph adaptation, documents, contexts, and graph validation.

The `Editor` component provides state machine display and editing, setting up the `D2dAdaptableControl` in which the state machine appears. It sets up control adapters for the `D2dAdaptableControl` in a very similar fashion to the [ATF Circuit Editor Sample](#). You can also print state machines from the canvas, handled by the `PrintableDocument` class.

Contexts in FSM Editor handle editing contexts in the canvas as well as the Prototypes window. Prototypes are handled almost identically to the [ATF Circuit Editor Sample](#). The `ViewingContext` provides viewing functions for the finite state machine on the canvas and also enables printing it.

Validation of the state machine is primarily to ensure than transition arrows between the same pairs of states don't overlap and is handled by `TransitionRouter`.

Contents

- [Programming Overview](#)
- [Graphs and State Machines](#)
- [FSM Editor Data Model](#)
- [FSM Editor DOM Adapters](#)
 - [FsmType DOM Adapters](#)
 - [FSM Editor Types DOM Adapters](#)
- [Fsm Class Graph Adapter](#)
- [FSM Editor Documents](#)
 - [Document Class](#)
 - [Editor Component](#)
 - [PrintableDocument Class](#)
- [FSM Editor Contexts](#)
 - [EditingContext Class](#)
 - [PrototypingContext Class](#)
 - [ViewingContext Class](#)
- [Validating State Machines](#)
 - [UniqueIdValidator Class](#)
 - [ReferenceValidator Class](#)
 - [TransitionRouter Class](#)
- [Palette Operations](#)

Graphs and State Machines

A state machine is a graph, and so this sample uses the graphical classes in the `Sce.Atf.Controls.Adaptable.Graphs` namespace.

In the most general form, ATF supports graphs using nodes, edges, and routes in its `IGraph<IGraphNode, IGraphEdge<IGraphNode, IEdgeRoute>, IEdgeRoute>` interface:

```
public interface IGraph<out TNode, out TEdge, out TEdgeRoute>
    where TNode : class, IGraphNode
    where TEdge : class, IGraphEdge<TNode, TEdgeRoute>
    where TEdgeRoute : class, IEdgeRoute
```

where the elements are:

- `IGraphNode`: Interface for a node in a graph; nodes are connected by edges.
- `IGraphEdge<TNode, TEdgeRoute>`: Interface for routed edges that connect nodes and have a defined source and destination route from and to the nodes.
- `IEdgeRoute`: Interface for edge routes, which act as sources and destinations for graph edges.

The FSM Editor's `Fsm` class, derived from `DomNodeAdapter`, uses its own `IGraph` variant:

```
public class Fsm : DomNodeAdapter, IGraph<State, Transition, NumberedRoute>, IAnnotatedDiagram
```

The parameters in the `IGraph<State, Transition, NumberedRoute>` interface are:

- `State`: DOM adapter for states.
- `Transition`: DOM adapter for transitions in the state machine.
- `NumberedRoute`: Route for directed graph edges, so that multiple edges do not overlap on the canvas.

For more information about `IGraph` and other graph interfaces, see [ATF Graph Interfaces](#). For details on the `Fsm` class, see [Fsm Class Graph Adapter](#).

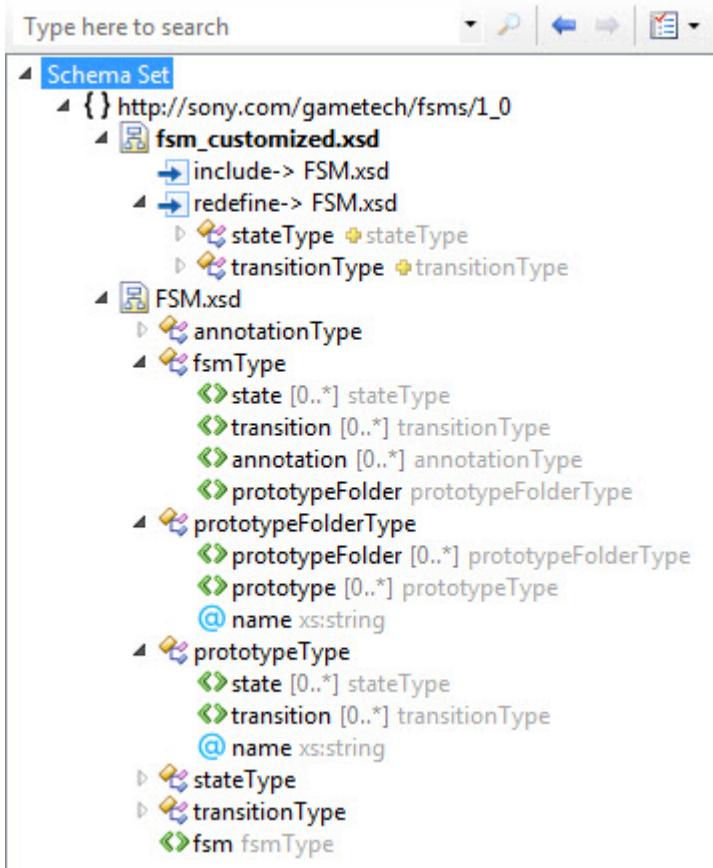
FSM Editor Data Model

Like most of the samples, this one defines a data model using an XML Schema. The main design is in the file `FSM.xsd`, which defines these types:

- `stateType`: a state in the state machine.
- `transitionType`: a transition from one state to another.
- `annotationType`: a comment.
- `prototypeType`: a set of states and transitions.
- `prototypeFolderType`: a collection of prototypes and prototype folders.
- `fsmType`: the state machine itself, consisting of all its states, transitions, annotations, and prototype folders.

`FSM.xsd` is augmented by the file `FSM_customized.xsd`, which adds extra attributes for state and transition types using a `redefine` element.

This schema is simple; no types are based on other types. Some types can have child types. In this figure from the Visual Studio XML Schema Explorer of the schema files, the parent types' nodes are opened to show their child types:



FSM Editor uses the `GenSchemaDef.bat` command file for DomGen to create the `Schema` class (containing type metadata classes) from the XML Schema files.

FSM Editor has a `SchemaLoader` class derived from `xmlSchemaTypeLoader` as usual. Its overridden `OnSchemaSetLoaded()` also performs the typical tasks of associating data with the `Schema` class's metadata classes:

- Defining DOM adapters on types. For details, see [FSM Editor DOM Adapters](#).
- Adding `NodeTypePaletteItem` objects for palette items. This implementation is very similar to others in the samples, such as the one in [ATF Simple DOM Editor Sample](#). For details, see [Using a Palette](#).
- Defining property descriptors for type attributes. Again, these descriptors are defined very similarly to ones in other samples. For an example, see [Create Property Descriptors](#).

FSM Editor DOM Adapters

Many of the classes in FSM Editor are DOM adapters, and they perform a wide variety of functions, including handling basic types, providing contexts, and displaying and editing state machines.

FsmType DOM Adapters

Most of the DOM adapters are defined for the "fsmType" type, which represents the whole state machine. The root element of the `DomNode` tree is of "fsmType". Such adapters fall in these general categories:

- Graph handler. For more information, see [Fsm Class Graph Adapter](#).
- Document handling. For more information, see [FSM Editor Documents](#).
- Context handling. For details, see [FSM Editor Contexts](#).
- Validation. For details, see [Validating State Machines](#).

FSM Editor Types DOM Adapters

Each of the types in the data model, such as "stateType" and "transitionType", has a DOM adapter defined for it.

Annotation DOM Adapter

The adapter `Annotation` implements `IAnnotation`, the interface for an annotation on a diagram. It is similar to the `Sce.Atf.Controls.Adaptable.Graphs.Annotation` class used by [ATF Circuit Editor Sample](#).

Annotation defines properties for the annotation attributes that are saved in the application data:

- Text: Get or set annotation text as a string.
- Location: Get or set annotation center point as a Point. The rectangle size does not persist; the rectangle is sized to fit the data.

Only the Text property is in IAnnotation. Note that Text implements a setter, even though this is not in IAnnotation.

IAnnotation also has a Bounds property to set annotation bounds and a SetTextSize() method to set the size of the annotation's text, measured using the annotation font. The Bounds property gets and sets Location for its own get and set.

Its OnNodeSet() method sets the comment text to the default value "Comment", so that appears in the comment when you drag a Comment item from the palette onto the canvas:

```
protected override void OnNodeSet()
{
    base.OnNodeSet();
    if (string.IsNullOrEmpty(Text))
        Text = "Comment";
}
```

The text is set only when there is no text already set.

Prototype and PrototypeFolder DOM Adapters

These adapters simply get and set attributes of the Schema metadata classes for these two types. For instance, Prototype gets and sets the prototype name with the DomNode.GetAttribute() and DomNode.SetAttribute() methods. Prototype's States and Transitions properties gets a list of states and transitions, using the DomNodeAdapter.GetChildList() method.

PrototypeFolder folder does something similar with its Folders and Prototypes properties.

These DOM adapters are used in the prototyping context. For details, see [PrototypingContext Class](#).

State DOM Adapter

State implements IGraphNode, a simple interface with the properties Name and Bounds for the name and bounding rectangle of the state. In addition, State has properties whose values are attributes of "stateType", so they are accessed with the DomNode.GetAttribute() and DomNode.SetAttribute() methods:

- Position: Get or set center Point of state.
- Size: Get or set diameter of state circle.
- Hidden: Get or set visibility of state.

Transition DOM Adapter

A transition connects states directionally, so it is a graph edge, and thus Transition implements IGraphEdge<State, NumberedRoute>, the interface for routed edges in a graph. This interface in turn implements IGraphEdge<State>.

Transition also implements the application data properties FromState, ToState, and Label, which simply get and set the appropriate attributes for "stateType". IGraphEdge<State> contains the properties FromNode, ToNode, and Label. These properties simply get the value of the FromState, ToState, and Label properties.

IGraphEdge<State, NumberedRoute> has properties for the routes' "from" and "to" states FromRoute and ToRoute, which both get the same NumberedRoute object.

Fsm Class Graph Adapter

The Fsm class is a DOM adapter for the entire state machine and adapts the state machine to a graph. It is defined on "fsmType", which is the type of the root DomNode of the tree representing the state machine. As previously mentioned, it implements IGraph<State, Transition, NumberedRoute> and IAnnotatedDiagram.

IGraph's properties, Nodes and Edges, get lists of these items. Fsm has properties that get an IList of various state machine entities: States, Transitions, and Annotations.

IAnnotatedDiagram simply gets a list of annotations, which Fsm provides with its Annotations property.

For more information about IGraph and other graph interfaces, see [ATF Graph Interfaces](#).

FSM Editor Documents

FSM Editor implements both document and document client interfaces. For a general discussion of documents, see [Implementing a Document and Its Client](#).

Document Class

Document, a DOM adapter defined for the "fsmType", derives from DomDocument that implements IDocument.

Document is a minimal class, simply providing the document Type and handling URI and dirty changed events.

Editor Component

Editor is both the document client and control host client for the control that holds the state machine graph, so it implements IDocumentClient and IControlHostClient.

Document Client

Much of what Document does as a document client is common to other samples and is well described in [Implementing a Document and Its Client](#).

The main method of interest here is Open(), which creates a D2dAdaptableControl to display the state machine graph. The process of creating and setting up control adapters for the D2dAdaptableControl has much in common with the [ATF Circuit Editor Sample](#) and is discussed in [Circuit Document Display and Control Adapters](#).

Adapters

A few control adapters deserve special mention:

```
var fsmAdapter = // adapt control to allow binding to graph data
    new D2dGraphAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, transformAdapter);

var fsmStateEditAdapter = // adapt control to allow state editing
    new D2dGraphNodeEditAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, fsmAdapter, transformAdapter
);
    
var fsmTransitionEditAdapter = // adapt control to allow transition
    new D2dGraphEdgeEditAdapter<State, Transition, NumberedRoute>(m_fsmRenderer, fsmAdapter, transformAdapter
);

var mouseLayoutManipulator = new MouseLayoutManipulator(transformAdapter);
```

Note that several of these adapters take a renderer m_fsmRenderer, which is constructed in this way:

```
m_theme = new D2dDiagramTheme();
m_fsmRenderer = new D2dDigraphRenderer<State, Transition>(m_theme);
```

D2dDigraphRenderer is a general graph rendering class; for more information, see [Direct2D Renderers](#). It takes a D2dDiagramTheme, a diagram rendering theme class for Direct2D rendering. Its constructor configures a variety of graphic objects to set the theme for drawing the graph:

```

public D2dDiagramTheme(string fontFamilyName, float fontSize)
{
    m_d2dTextFormat = D2dFactory.CreateTextFormat(fontFamilyName, fontSize);
    m_fillBrush = D2dFactory.CreateSolidBrush(SystemColors.Window);
    m_textBrush = D2dFactory.CreateSolidBrush(SystemColors.WindowText);
    m_outlineBrush = D2dFactory.CreateSolidBrush(SystemColors.ControlDark);
    ...
    int fontHeight = (int)TextFormat.FontHeight;
    m_rowSpacing = fontHeight + PinMargin;
    m_pinOffset = (fontHeight - m_pinSize) / 2;

    D2dGradientStop[] gradstops =
    {
        new D2dGradientStop(Color.White, 0),
        new D2dGradientStop(Color.LightSteelBlue, 1.0f),
    };
    m_fillLinearGradientBrush = D2dFactory.CreateLinearGradientBrush(gradstops);
    StrokeWidth = 2;
}

```

Getting back to the control adapters, `D2dGraphAdapter` adapts a control to displaying a graph. For more information on this adapter, see [D2dGraphAdapter Class and Control Adapters](#).

`D2dGraphNodeEditAdapter` adds graph node dragging capabilities to an adapted control. In FSM Editor, this allows you to reposition states on the canvas by dragging them. For more details on this adapter, see [D2dGraphNodeEditAdapter Class and Control Adapters](#).

`D2dGraphEdgeEditAdapter` provides graph edge dragging capabilities to an adapted control. For more details on this adapter, see [D2dGraphEdgeEditAdapter Class and Control Adapters](#).

The [ATF Circuit Editor Sample](#) also uses these adapters, and its usage is described in [Circuit Document Display and Control Adapters](#).

`MouseLayoutManipulator` marks a state as selected. For more information, see [Control Adapters](#).

`HoverAdapter` is used to display information about states when the cursor hovers over them. For a discussion of how to use this adapter, see [Hover Adapter](#).

Finishing Up

After calling `control.Adapt()` to set all the control adapters, `Open()` performs a few housekeeping tasks:

```

// associate the control with the viewing context; other adapters use this
// adapter for viewing, layout and calculating bounds.
ViewingContext viewingContext = node.Cast<ViewingContext>();
viewingContext.Control = control;

// set document URI
document = node.As<Document>();
ControlInfo controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center);

//Set IsDocument to true to prevent exception in command service if two files with the
// same name, but in different directories, are opened.
controlInfo.IsDocument = true;

document.ControlInfo = controlInfo;
document.Uri = uri;

// now that the data is complete, initialize the rest of the extensions to the Dom data;
// this is needed for adapters such as validators, which may not be referenced anywhere
// but still need to be initialized.
node.InitializeExtensions();

// set control's context to main editing context
EditingContext editingContext = node.Cast<EditingContext>();
control.Context = editingContext;

```

These final tasks consist of adapting the root `DomNode` to various objects. The root `DomNode` is of type "fsmType", and so it can be adapted to `ViewingContext`, `Document`, and `EditingContext`, because all of these DOM adapters are defined on "fsmType".

The adapted objects are used to store information. For example, the adapted `viewingContext`'s `Control` property is set to the

D2dAdaptableControl. The adapted Document has its ControlInfo and Uri properties set. And the D2dAdaptableControl's Context property is set to the adapted EditingContext.

The last thing the Open() method does is to register the D2dAdaptableControl it just constructed with the control host service:

```
m_controlHostService.RegisterControl(control, controlInfo, this);
```

Control Host Client

This client also behaves very similarly to other samples' control host clients. Its Activate() method sets the active context and document. Close() tries to close the active document and gives the user a chance to save it if it has changed.

For further discussion, see [Creating Control Clients](#).

PrintableDocument Class

PrintableDocument confers the ability to print the canvas with its implementation of IPrintableDocument. The IPrintableDocument.GetPrintDocument() gets a PrintDocument that can be printed and works with the standard Windows print dialogs.

This PrintDocument object is actually a FsmPrintDocument deriving from CanvasPrintDocument, the abstract base class for printing a canvas. FsmPrintDocument overrides methods to get "page" bounds for the various PrintRange values, as well as the Render() method to actually render the canvas to a printed page:

```
protected override void Render(RectangleF sourceBounds, Matrix transform, Graphics g)
{
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.InterpolationMode = InterpolationMode.HighQualityBicubic;

    g.Transform = transform;
    sourceBounds.Inflate(1, 1); // allow for pen widths
    g.SetClip(sourceBounds);

    foreach (IPrintingAdapter printingAdapter in m_viewingContext.Control.AsAll<IPrintingAdapter>())
        printingAdapter.Print(this, g);
}
```

The printing operation is done by the ViewingContext (in m_viewingContext) that is adapted to an IPrintingAdapter, which has a Print() method to render the canvas to a GDI Graphics object. For the discussion of FSM Editor's ViewingContext, see [ViewingContext Class](#).

FSM Editor Contexts

FMS Editor context classes provide several different kinds of capabilities.

EditingContext Class

This context governs editing the state machine on the canvas. Like several samples, the EditingContext derives from Sce.Atf.Dom.EditingContext:

```
public class EditingContext : Sce.Atf.Dom.EditingContext,
    IEnumerableContext,
    IObservableContext,
    INamingContext,
    IInstancingContext,
    IEditableGraph<State, Transition, NumberedRoute>
```

There may be several of these EditingContexts in a document, and PrototypingContext also derives from EditingContext. For more information on this useful context, see [Sce.Atf.Dom.EditingContext Class](#).

There are several interfaces implemented that many other editing contexts in the samples implement:

- IEnumerableContext: Enumerate items in the state machine.

- **IObservableContext**: Events for states being added or removed. This is required so the state machine can be refreshed after it changes.
- **INamingContext**: Name states.
- **IIInstancingContext**: Handle copy and paste of selected state and transition items.

For an example of how circuit graphs use interfaces like this, see [CircuitEditingContext Class](#).

IIInstancingContext requires the most work to implement. Some of the methods in this interface are very similar to other **IIInstancingContext** implementations. Both `CanCopy()` and `CanDelete()`, for instance, simply check that `Selection.Count > 0`, that is, that something is selected. `Copy()` iterates through the selection, copying whichever `DomNodes` are adapted to `State`, `Transition`, and `Annotation`. `Insert()` uses a `DragDropAdapter` to find the insertion location for the object. It then copies the list of inserted objects and adapts it to an enumeration of `DomNodes`, which are then adapted to an appropriate list, such as `List<State>`, and added to the list of such items in the state machine.

EditingContext also uses the interface `IEEditableGraph<State, Transition, NumberedRoute>` to edit transitions between states. This interface includes methods to determine whether connections can be made or undone, and to make or break the connection:

`CanConnect`, `Connect`, `CanDisconnect`, and `Disconnect`. Making a transition means adding a "transitionType" object, which adds a `DomNode` of that type to the `DomNode` tree:

```
Transition IEEditableGraph<State, Transition, NumberedRoute>.Connect(
    State fromNode, NumberedRoute fromRoute, State toNode, NumberedRoute toRoute, Transition existingEdge
)
{
    DomNode domNode = new DomNode(Schema.transitionType.Type);
    Transition transition = domNode.As<Transition>();

    transition.FromState = fromNode as State;
    transition.ToState = toNode as State;
    // we set the route after the logical operation completes

    if (existingEdge != null)
        transition.Label = existingEdge.Label;

    m_fsm.Transitions.Add(transition);
    return transition;
}
```

Note that the new `Transition` object, a `DomNode` adapted to `Transition`, has its `FromState` and `ToState` properties set to the appropriate `DomNodes` adapted to `State`.

PrototypingContext Class

This context is for editing the Prototypes window's prototype list, which is created by the `PrototypeLister` component. This component requires an `IPrototypingContext` to be viewed and edited by the user, so `PrototypingContext` implements `IPrototypingContext`.

Like `EditingContext`, `PrototypingContext` derives from `Sce.Atf.Dom.EditingContext`. `PrototypingContext` also implements several of the interfaces that `EditingContext` does, such as `IObservableContext`, and they serve much the same purposes, although in the context of a prototype list.

`IPrototypingContext` itself requires `ITreeView` and `IItemView`, so `PrototypingContext` implements them as well. For more information on these interfaces, see [ITreeView Interface](#) and [IItemView Interface](#).

Because `PrototypingContext` is a separate context from `EditingContext` and also that the `MultipleHistoryContext` DOM adapter for multiple history contexts is defined for the root type "fsmType", the undo/redo history stacks for these two contexts is distinct. That is, you can undo and redo on the main canvas and the Prototypes window separately.

This context is almost identical to `Sce.Atf.Controls.Adaptable.Graphs.PrototypingContext`. For more information, see [PrototypingContext Class](#).

ViewingContext Class

`ViewingContext` is a DOM adapter that provides viewing functions for the finite state machine on the canvas. It contains a reference to the `D2dAdaptableControl` so it can update the control's canvas bounds during validation. `PrintableDocument` also requires a `ViewingContext`. It implements `ILayoutContext` and `IViewingContext`:

```
public class ViewingContext : Validator, ILayoutContext, IVViewingContext
```

`IViewingContext` allows framing items in the canvas and ensuring their visibility.

`ILayoutContext` works with the bounds of canvas items. `GetBounds()`, for instance, attempts to adapt the item as either a `State` or `Annotation` and, if successful, returns the `Bounds` property:

```
BoundsSpecified ILayoutContext.GetBounds(object item, out Rectangle bounds)
{
    State state = Adapters.As<State>(item);
    if (state != null)
    {
        bounds = state.Bounds;
        return BoundsSpecified.All;
    }

    Annotation annotation = Adapters.As<Annotation>(item);
    if (annotation != null)
    {
        bounds = annotation.Bounds;
        return BoundsSpecified.All;
    }

    bounds = new Rectangle();
    return BoundsSpecified.None;
}
```

A `Transition` is not bounded, per se, because a transition's location is determined by its "from" and "to" states. Drawing it requires no bounds information.

Both the `IViewingContext` and `ILayoutContext` interfaces are discussed further in [Context Interfaces](#).

Validating State Machines

The `SchemaLoader` defines a few validating DOM adapters on the "fsmType" type:

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

UniqueIdValidator Class

The `UniqueIdValidator` DOM adapter ensures that every DOM node in the subtree has a unique ID. The "stateType" has an ID ("xs:ID") that "transitionType" references for the "from" and "to" states, as seen in the XML Schema definition:

```
<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:attribute name="size" type="xs:int" />
    <xs:attribute name="hidden" type="xs:boolean" />
    <xs:attribute name="start" type="xs:boolean" />
</xs:complexType>
...
<xs:complexType name="transitionType">
    <xs:attribute name="label" type="xs:string" />
    <xs:attribute name="source" type="xs:IDREF" use="required" />
    <xs:attribute name="destination" type="xs:IDREF" use="required" />
</xs:complexType>
```

These IDs must therefore be checked for uniqueness.

ReferenceValidator Class

As just seen, the type "transitionType" references IDs for the "from" and "to" states in the "transitionType", so these references must be

validated whenever the graph changes. ReferenceValidator tracks references and reference holders to ensure reference integrity within DOM data. Checks are only made within validations, which are signaled by IValidationContexts. This adapter should be defined on the DOM's root node type, and it is: on "fsmType". This adapter does the following:

1. Track all DomNode references in the subtree.
2. Raise notifications if external DomNode references are added or removed.
3. After validating, raise notification events if referents have been removed, leaving dangling references.

TransitionRouter Class

This validator is summoned when making transitions. Its purpose is to track changes to transitions and update their routing, so that the transition arrows don't overlap on the canvas.

When a transition is drawn and the user releases the mouse, OnEnded() is called, which calls RouteTransitions():

```
private void RouteTransitions()
{
    Dictionary<Pair<State, State>, int> routesByPair = new Dictionary<Pair<State, State>, int>();
    foreach (Transition transition in m_fsm.Transitions)
    {
        Pair<State, State> states = new Pair<State, State>(transition.FromState, transition.ToState);
        int routes;
        if (!routesByPair.TryGetValue(states, out routes))
        {
            // start routes at 0 if there are no connections in the opposite direction,
            // otherwise, start at 1
            if (routesByPair.ContainsKey(new Pair<State, State>(transition.ToState, transition.FromState)))
                routes = 1;
            routesByPair.Add(states, routes);
        }

        transition.Route = routes++;
        routesByPair[states] = routes;
    }
}
```

This method creates a dictionary of all the transition from-to state pairs as the key, with the number of routes between this pair as the value. The dictionary is created while iterating through all the transitions in the state machine. Each transition's Route property is set each loop iteration so it is set to a unique value:

```
transition.Route = routes++;
```

This allows the NumberedRoutes to be set up so the transition arrows don't overlap.

Palette Operations

FSM Editor creates and uses a palette with its PaletteClient class and palette data set up, described in [FSM Editor Data Model](#). Palette implementation is very similar to the [ATF Simple DOM Editor Sample](#). For details on setting up a palette, see [Using a Palette in Simple DOM Editor Programming Discussion](#).

Topics in this section

Links on this page to other pages

[ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Graph Interfaces](#), [ATF Simple DOM Editor Sample](#), [Authoring Tools Framework](#), [Circuit Editor Programming Discussion](#), [Circuit Graph Support](#), [Context Classes](#), [Context Interfaces](#), [Control Adapters](#), [Creating Control Clients](#), [File Explorer Programming Discussion](#), [General Graph Support](#), [Graphs in ATF](#), [Implementing a Document and Its Client](#), [Simple DOM Editor Programming Discussion](#), [State Chart Editor Programming Discussion](#)

Model Viewer Programming Discussion

Games contain graphic models of game objects, such as characters and vehicles. The [ATF Model Viewer Sample](#) renders 3D models of objects in ATGI and Collada format files, with `.atgi` and `.dae` extensions. The model can be rendered in variety of ways, as a wireframe, for instance.

Besides rendering model data, this sample illustrates handling documents, commands, and controls in ATF. Although this application does not have its own data model or edit data, it still makes use of the ATF DOM by treating the model data as a tree of DOM nodes.

Programming Overview

The page briefly touches on ATGI and Collada, providing references for each. While largely routine, the MEF initialization for Model Viewer uses the Component Model API to get an exported component.

Although Model Viewer does not edit data, it makes use of the ATGI and Collada data models expressed in their XML Schema. It uses the `AtgiResolver` and `ColladaResolver` components, which implement `IResourceResolver` to both load the schema files and to load model files, creating `DomNode` trees from their data.

Model viewer treats the loaded data as a document. Its `ModelDocument` implements `IDocument` and its document client class `ModelViewer` implements `IDocumentClient`.

`RenderView` registers a `DesignControl` to display a 3D scene from the active document. The `RenderCommands` component works with `RenderView` to change the rendering mode. `RenderView` builds a scene graph from the model's `DomNode` tree. A `DesignControl` traverses the scene graph to render the model: each `IRenderObject` object in the scene graph is traversed to create a `TraverseNode` instance and add it to the traverse list. This results in the `Traverse()` method in the DOM adapters `RenderTransform` and `RenderPrimitive` being called for each `IRenderObject`. `RenderTransform` transforms model objects properly, and `RenderPrimitives` does the lowest level rendering using OpenGL. OpenGL is not discussed here, because it is distinct from ATF and well-documented elsewhere.

Topics

- [Programming Overview](#)
- [ATGI and Collada File Formats](#)
- [MEF Initialization](#)
- [Model Data Handling](#)
 - [Resource Resolvers](#)
 - [Loading Schema File](#)
 - [IResourceResolver.Resolve\(\) Method](#)
 - [ATGI and Collada Schema Classes](#)
- [Document Handling](#)
 - [ModelDocument Class](#)
 - [ModelViewer Class](#)
 - [ModelViewer Initialization](#)
 - [ModelViewer CanOpen\(\) and Open\(\) Methods](#)
- [Rendering Components](#)
 - [RenderView Component](#)
 - [RenderCommands Component](#)
- [DesignControl Operation](#)
- [Rendering DOM Adapters](#)
 - [RenderTransform DOM Adapter](#)
 - [RenderPrimitives DOM Adapter](#)

ATGI and Collada File Formats

ATGI and Collada offer formats for exchanging digital assets among graphics software applications. ATGI is a Sony intermediate file format for game art data. COLLADA (from collaborative design activity) defines an open standard format.

Both provide XML schema defining the formats' types. Collada model files have the extension `.dae` (digital asset exchange). ATGI model files have the `.atgi` extension.

For details on Collada, see the [Collada page](#). For more information about ATGI, see the [ATGI project page](#). There is a [MayaToAtgi SDK component](#) for a Maya exporter for the ATGI file format.

MEF Initialization

Model Viewer creates a MEF TypeCatalog as many samples do and uses many of the common components, such as ControlHostService and StandardFileCommands. It uses several standard components to open model files:

- AtgiResolver: resource resolver for ATGI files.
- ColladaResolver: resolves COLLADA resource files.

Model Viewer also adds several components of its own:

- ModelViewer: document client for 3D model files, opening model files.
- RenderView: registers a DesignControl to display a 3D scene from a model document.
- RenderCommands: provides user commands related to change the view of the model.

After the usual CompositionContainer, CompositionBatch, and MainForm creation and handling, the sample application determines which menu items appear under the File menu, just Open in this case. This must be set before component initialization, which immediately follows:

```
StandardFileCommands stdfile = container.GetExportedValue<StandardFileCommands>();
stdfile.RegisterCommands = StandardFileCommands.CommandRegister.FileOpen;

// Initialize components
foreach (IInitializable initializable in container.GetExportedValues<IInitializable>())
    initializable.Initialize();
```

The ExportProvider.GetExportedValue<T>() method gets the exported value for the StandardFileCommands component and sets its RegisterCommands property, which contains a mask for the File menu items that are used. This property must be set before StandardFileCommands is initialized.

After this, the sample directly initializes the components by calling Initialize() for every component in the CompositionContainer object, but it could just as well have used the IInitializable.InitializeAll() extension method like this, which is what most samples do:

```
container.InitializeAll();
```

The advantage of this latter method is that it first instantiates the components, because components are loaded in a lazy fashion otherwise, only created when required to satisfy another component's import.

Model Data Handling

Model Viewer can view two different kinds of model data: from ATGI and from Collada. Although these two data models are different, they can both be specified in the XML Schema Definition Language (XSD), also known as XML Schemas. The data model defines a set of data types that describe the data in a model file. ATF includes two type definition files in this format, "atgi.xsd" and "collada.xsd".

The ATF DOM can use XML Schemas for its data models, and many of the ATF DOM's facilities are used in the Model Viewer sample. For details, see the ATF Programmer's Guide: Document Object Model (DOM), downloadable at [ATF Documentation](#).

Resource Resolvers

How does the Model Viewer use the XML Schema for ATGI and Collada? It imports the available IResourceResolver objects, the AtgiResolver and ColladaResolver components. These resolvers do two things:

- Load the schema file for model data.
- Provide a Resolve() method to load a model file from a given URI.

Loading Schema File

The Initialize() method for each resolver component loads the model schema. Here is Initialize() for AtgiResolver:

```

public void Initialize()
{
    if (m_initialized)
        return;

    m_initialized = true;

    Assembly assembly = Assembly.GetExecutingAssembly();
    m_loader.SchemaResolver = new ResourceStreamResolver(assembly, assembly.GetName().Name +
    "/schemas");
    m_loader.Load("atgi.xsd");
}

```

The `m_loader` field is initialized this way:

```

private AtgiSchemaTypeLoader m_loader = new AtgiSchemaTypeLoader();

```

`AtgiSchemaTypeLoader` derives from `XmlSchemaTypeLoader`, which can load schema files with its `Load()` method.

`AtgiSchemaTypeLoader` uses this `load` method to load the ATGI schema file "atgi.xsd" after resolving its URI.

`ColladaResolver` uses the `ColladaSchemaTypeLoader`, also derived from `XmlSchemaTypeLoader`, to load the Collada schema.

Both ATGI and Collada schema files are loaded when the `AtgiSchemaTypeLoader` and `ColladaResolver` components are initialized while the application starts up. These schema definitions provide the information needed to parse and view model files governed by the data models defined in the schemas.

IResourceResolver.Resolve() Method

Once the schema describing model files has been loaded into the application, Model Viewer can load model files.

The `IResourceResolver.Resolve()` method loads a file from a given URI and produces an object implementing `IResource`, a resource with a type and unique URI.

The `DomNode` class represents an object of one of the types described in the data model for ATGI or Collada. The `Resolve()` method creates a tree of `DomNode` objects representing all the data in the model file. `Resolve()` returns a `DomNode` that is the root of the `DomNode` tree.

The returned `DomNode` is adapted to `IResource`, as this code for `ColladaResolver.Resolve()` demonstrates:

```

public IResource Resolve(Uri uri)
{
    string fileName = PathUtil.GetCanonicalPath(uri);
    if (!fileName.EndsWith(".dae"))
        return null;

    DomNode domNode = null;
    try
    {
        using (Stream stream = File.OpenRead(fileName))
        {
            var persister = new ColladaXmlPersister(m_loader);
            domNode = persister.Read(stream, uri);
        }
    }
    catch (IOException e)
    {
        Outputs.WriteLine(OutputMessageType.Warning, "Could not load resource: " + e.Message);
    }

    IResource resource = Adapters.As<IResource>(domNode);
    if (resource != null)
        resource.Uri = uri;

    return resource;
}

```

`Resolve()` does the following:

1. Checks that the file extension is "dae".
2. Creates a `DomNode`.
3. Reads file data with a `Stream`.
4. Creates a `ColladaXmlPersister`, which derives from `DomXmlReader`, using the `ColladaSchemaTypeLoader` object. This loader has the information from the Collada schema.
5. Call `ColladaXmlPersister.Read()` to convert the stream data into a tree of `DomNode` objects, with a `DomNode` at the root. The `ColladaXmlPersister` has the information to do this from the `ColladaSchemaTypeLoader` that loaded the Collada schema.
6. Adapts the root `DomNode` to an `IResource` and returns it.

The `AtgiResolver.Resolve()` method performs similarly.

Loading the model file's data creates a tree of `DomNode` objects that the application can work with to render the model file's data.

ATGI and Collada Schema Classes

In addition to the type definition files, each model provides a `Schema` class, generated by the ATF DomGen utility from the XML schema definitions for the ATGI and Collada data models. DomGen creates a metadata class, such as `DomNodeType`, for every type in the schema definition file and provides a convenient way to reference the types in the two different data models. In particular, each `DomNode` has a `DomNodeType` metadata object associated with it, describing the node's type.

These metadata classes in the `Schema` classes are used when the application starts up by the document client component, `ModelViewer`, as part of its initialization. For details, see [ModelViewer Class](#).

Document Handling

Model Viewer implements `IDocument` and `IDocumentClient` to handle documents in a class and a component. For general information about documents, see [Documents in ATF](#).

ModelDocument Class

In Model Viewer, the `ModelDocument` class implements `IDocument` and holds the model data in a file. `ModelDocument`'s constructor takes parameters for a `DomNode` and a URI:

```
public ModelDocument(DomNode node, Uri ur);
```

The `DomNode` here is the root of a tree of `DomNode` objects, because model data in the document is treated as a tree of DOM nodes, just as in the ATF DOM. This is discussed in [Model Data Handling](#).

The `ModelDocument` constructor sets the `RootNode` property to this root `DomNode`, and this property is referenced wherever the model data is needed.

Model Viewer only reads documents and does not edit them, so these properties are set accordingly:

```
public bool IsReadOnly
{
    get { return true; }
}
...
public bool Dirty
{
    get { return false; }
    set { throw new InvalidOperationException(); }
}
```

`IDocument` derives from `IResource`, so it also defines the `Type` property to indicate the document type:

```
public string Type
{
    get { return "3D Model"; }
}
```

The ModelViewer component is a document client, implementing `IDocumentClient`. Its constructor creates a `DocumentClientInfo`, obtained from the `IDocumentClient.Info` property. This determines the document type and extensions the application handles:

```
public ModelViewer()
{
    string[] exts = { ".atgi", ".dae" };
    m_info = new DocumentClientInfo("3D Model", exts, null, null, false);
}
...
DocumentClientInfo IDocumentClient.Info
{
    get { return m_info; }
}
```

ModelViewer Initialization

The ModelViewer component's initialization defines DOM extensions used for various `DomNode` types:

```
void IInitializable.Initialize()
{
    // Register ATGI and Collada nodes
    Register<RenderTransform>(Sce.Atf.Atgi.Schema.nodeType.Type);
    Register<RenderPrimitives>(Sce.Atf.Atgi.Schema.vertexArray_primitives.Type);

    Register<RenderTransform>(Sce.Atf.Collada.Schema.nodeType);
    Register<RenderPrimitives>(Sce.Atf.Collada.Schema.polylist.Type);
    Register<RenderPrimitives>(Sce.Atf.Collada.Schema.triangles.Type);
    Register<RenderPrimitives>(Sce.Atf.Collada.Schema.trifans.Type);
    Register<RenderPrimitives>(Sce.Atf.Collada.Schema.tristrips.Type);

    if (m_scriptingService != null)
        m_scriptingService.SetVariable("viewer", this);
}
...
private static void Register<T>(DomNodeType nodeType) where T : new()
{
    nodeType.Define(new ExtensionInfo<T>());
}
```

Defining a DOM extension on a type allows `DomNode` objects of this type to be adapted to other objects, whose API is more suitable to the task at hand. The classes `here`, `RenderTransform` and `RenderPrimitives`, are known as DOM adapters, and are discussed in [Rendering DOM Adapters](#).

In most of the samples, DOM extensions are defined in the application's Schema loader. However, Model Viewer has no Schema loader, since it doesn't define its own data model. Instead, it uses the schemas defined for ATGI and Collada data. In particular, it uses metadata classes like `Sce.Atf.Atgi.Schema.nodeType.Type`, as shown in the `IInitializable.Initialize()` above. These metadata classes are in the Schema classes generated by DomGen, as discussed in [ATGI and Collada Schema Classes](#).

ModelViewer CanOpen() and Open() Methods

Model Viewer only views model data; it doesn't change it, so the document client doesn't need to save data to a file or even close the file. Only open functions are needed; all the other methods in `IDocumentClient` do nothing.

`CanOpen()` simply checks that the file extension is appropriate:

```
bool IDocumentClient.CanOpen(Uri uri)
{
    return m_info.IsCompatibleUri(uri);
}
```

The `Open()` method creates a `ModelDocument`:

```

IDocument IDocumentClient.Open(Uri uri)
{
    foreach (IResourceResolver resolver in m_resolvers)
    {
        DomResource res = resolver.Resolve(uri) as DomResource;
        if (res != null)
        {
            return new ModelDocument(res.DomNode, uri);
        }
    }

    return null;
}

```

This `Open()` method, though also brief, requires some explanation.

First, the `IResourceResolver` objects are iterated from the field `m_resolvers`, which is imported from all components exporting `IResourceResolver`:

```

[ImportMany]
private IEnumerable<IResourceResolver> m_resolvers;

```

Model Viewer has two such `IResourceResolver` exporters: `AtgiResolver` and `ColladaResolver`, discussed in the [Resource Resolvers](#) section.

The loop attempts to resolve the URI using all the `IResourceResolver` objects it has. It calls `Resolve()` for each resolver and sees if it gets a non-null result, indicating resolution succeeded. `Resolve()` creates a tree of `DomNode` objects from the data in the model file. This tree is used to render the model into a 3D drawing.

`Open()` constructs a new `ModelDocument` using the URI and the `DomNode` tree's root, and then returns the `ModelDocument`.

Rendering Components

The resource resolver components `AtgiResolver` and `ColladaResolver` load a model file and create a tree of `DomNode` objects. Each `DomNode` represents an object that can be rendered. During the rendering process, these `DomNode` objects are adapted to various interfaces, such as `IRenderObject`, through which rendering ultimately occurs.

Two Model Viewer components facilitate model rendering.

RenderView Component

`RenderView` registers a `DesignControl` that is used to display a 3D scene from the active document.

Several classes are used to render objects. `SceneNode` objects hold all the render objects and constraints that are associated with an object being rendered. `SceneNode` objects are arranged in a graph that determines which objects to render, and in what order nodes are traversed during rendering. `Scene` derives from `SceneNode` and holds the root `SceneNode` of the scene graph.

`SceneGraphBuilder` builds a scene graph from a root source object, typically a `DomNode`. In the Model Viewer application, the scene graph is built from the `DomNode` tree. A `SceneNode` object holds a reference to its underlying source object, the `DomNode`.

The `RenderView` constructor creates two objects:

```

public RenderView()
{
    m_scene = new Scene();
    m_designControl = new DesignControl(m_scene);
}

```

`DesignControl` extends `CanvasControl3D` to provide scene graph rendering and picking, using the scene graph provided in its constructor's `Scene` parameter. `DesignControl` is the canvas on which model objects are drawn.

The `IInitializable.Initialize()` method for the `RenderView` component registers the `DesignControl` and subscribes to the active document changed event:

```

void IInitializeable.Initialize()
{
    ControlInfo cinfo = new ControlInfo("3D View", "3d viewer", StandardControlGroup.CenterPermanent);
    m_controlHostService.RegisterControl(m_designControl, cinfo, null);

    m_documentRegistry.ActiveDocumentChanged += (sender, e) =>
    {
        ClearRenderGraph(m_context);
        m_context = null;

        ModelDocument doc = m_documentRegistry.GetActiveDocument<ModelDocument>();
        if (doc != null)
        {
            m_context = doc.RootNode;
            SceneGraphBuilder builder = new SceneGraphBuilder(typeof(IRenderThumbnail));
            builder.Build(doc.RootNode, m_scene);
            Fit();
        }
    };
}

```

A lambda expression is used for the event handler. In this expression, `ClearRenderGraph()` clears the `DesignControl` canvas, because a model is going to be rendered for the new active document. The handler creates a `ModelDocument` for the active document.

If the new document is valid, the lambda expression constructs a `SceneGraphBuilder` that builds a scene graph from the specified type of `IRenderObject` objects. In this case, objects to render must implement `IRenderThumbnail`, which is an interface for objects that can generate thumbnails. The `IRenderThumbnail` interface is empty, so this is not much of a restriction.

`IRenderObject` is an interface for renderable objects. `IRenderObject` extends `IBuildSceneNode`, which enables scene graph building for a DOM object. The nodes in the model's `DomNode` tree are adapted to `IRenderObject` in the process of rendering.

The `SceneGraphBuilder.Build()` method creates a scene graph from the `DomNode` tree. This scene graph is used for rendering a document's model in the `DesignControl` window as long as that document is active.

The `Fit()` method fits the rendered object in the `DesignControl` window. It changes the settings of `DesignControl`'s Camera to fit the object.

RenderCommands Component

The `RenderCommands` component provides user commands related to the `RenderView` component to change rendering mode.

Its `IInitializeable.Initialize()` method registers commands with the `CommandService` component, which it imported into the `m_commandService` field:

```

public virtual void Initialize()
{
    m_commandService.RegisterCommand(
        Command.Fit,
        StandardMenu.View,
        CommandGroup,
        "Fit",
        "Fit All",
        Keys.F,
        null,
        CommandVisibility.Menu,
        this);
    m_commandService.RegisterCommand(
        Command.RenderSmooth,
        StandardMenu.View,
        CommandGroup,
        "Smooth",
        "Smooth shading",
        Keys.None,
        Resources.SmoothImage,
        CommandVisibility.All,
        this);
    ...
    m_commandService.RegisterCommand(
        Command.RenderCycle,
        StandardMenu.View,
        CommandGroup,
        "CycleRenderModes",
        "Cycle render modes",
        Keys.Space,
        null,
        CommandVisibility.Menu,
        this);
}

```

Commands are triggered by a menu item and/or a tool button, depending on the setting of the `CommandVisibility` parameter. Icons for menu items and tool buttons are specified in the `icon` parameter, which references items like `Resources.SmoothImage`. All these image resources are in the `Resources` class, which provides several sizes of each image so the right size can be used for a menu item or tool button.

`RenderCommands` implements `ICommandClient`, whose methods determine whether commands can be performed and performs them. `CanDoCommand()` returns true when the `DesignControl` exists.

The `DoCommand()` method performs all the registered commands:

```

public void DoCommand(object commandTag)
{
    if (commandTag is Command)
    {
        DesignControl control = m_renderView.ViewControl;
        switch ((Command)commandTag)
        {
            case Command.Fit:
                m_renderView.Fit();
                break;

            case Command.RenderSmooth:
                control.RenderState.RenderMode &= ~RenderMode.Wireframe;
                control.RenderState.RenderMode |= (RenderMode.Smooth | RenderMode.SolidColor |
                    RenderMode.Lit | RenderMode.CullBackFace | RenderMode.Textured);
                control.Invalidate();
                break;
        }
    }
}

```

In the `Command.Fit` case, the `Fit()` method fits the rendered object in the canvas.

Most of the other commands, such as `Command.RenderSmooth`, change how the model is rendered. `RenderState` is a platform-independent representation of a GPU render state. It uses the `RenderMode` enum, which corresponds to various rendering modes, such as wireframe. The commands here set a new `RenderState` for the control and then call `Invalidate()` to trigger repainting the model object in the `DesignControl`, which is discussed in [DesignControl Operation](#).

Finally, the `UpdateCommand()` method updates the UI appearance based on the current rendering state. This results in the appropriate menu item being checked and tool button highlighted.

DesignControl Operation

After the scene graph is built, it is traversed by `DesignControl` to render the model. The actual rendering is done by methods called in the `DOM adapters`, `RenderTransform` and `RenderPrimitives`, discussed in [Rendering DOM Adapters](#).

`DesignControl`'s constructor sets up several objects to assist in the rendering process:

```
public DesignControl(Scene scene)
{
    m_scene = scene;

    m_renderAction = new RenderAction(RenderStateGuardian);
    m_pickAction = new PickAction(RenderStateGuardian);

    // default render states. These correspond to the state of the toggles on the toolbar,
    // like wireframe on/off, lighting on/off, backface culling on/off, and textures on/off.
    m_renderState.RenderMode = RenderMode.Smooth | RenderMode.CullBackFace | RenderMode.Textured
    | RenderMode.Lit | RenderMode.Alpha;
    m_renderState.WireframeColor = new Vec4F(0, 0.015f, 0.376f, 1.0f);
    m_renderState.SolidColor = new Vec4F(1, 1, 1, 1.0f);
}
```

A `RenderAction` is created and default values for a `RenderState` object `m_renderState` are set. `RenderAction` implements `IRenderAction`, which features methods to build a traverse list from the scene graph and dispatch the list for rendering.

When a `Paint` event occurs for the `DesignControl`, such as when the view is invalidated, it calls its `Render()` method:

```
protected override void OnPaint(PaintEventArgs e)
{
    Render(m_renderAction, false, false);
    m_invalidate = false;
}
```

`Render()` calls `RenderAction.Dispatch()`, which calls `BuildTraverseList()` to build a traverse list of `TraverseNode` objects. `TraverseNode` is a class for encapsulating the rendering state for each `IRenderObject` object. `BuildTraverseList()` calls the `IRenderObject.Traverse()` method for each `IRenderObject` object in the scene graph to create a `TraverseNode` instance and add it to the traverse list, if the object is to be rendered. This results in the `Traverse()` method in `RenderTransform` and `RenderPrimitive` being called for each `IRenderObject`, as seen in [Rendering DOM Adapters](#).

After creating the traverse list, `RenderAction.Dispatch()` calls `RenderPass()` to render the traverse list. This results in the `Render()` method in `RenderPrimitive` being called for each `IRenderObject`, so each node is rendered, which is commented on in [RenderPrimitives DOM Adapter](#).

`DesignControl` uses a `Camera` object with default settings to display rendered objects.

Rendering DOM Adapters

Recall that the `ModelViewer` component's initialization defines DOM extensions for some `DomNode` types, as shown in [ModelViewer Initialization](#). DOM extensions for ATGI types are seen here:

```
void IInitializeable.Initialize()
{
    // Register ATGI and Collada nodes
    Register<RenderTransform>(Sce.Atf.Atgi.Schema.nodeType.Type);
    Register<RenderPrimitives>(Sce.Atf.Atgi.Schema.vertexArray_primitives.Type);
    ...
}
```

The result of these definitions is to allow `DomNode` objects to be adapted to another class. From the second definition above, for instance, a `DomNode` of type `Sce.Atf.Atgi.Schema.vertexArray_primitives.Type` is adapted to `RenderPrimitives`, a DOM adapter class. This means that all the methods and properties of `RenderPrimitives` can be used on this type's `DomNode`.

The Model Viewer application defines two DOM adapters, `RenderTransform` and `RenderPrimitives`. Note that:

- Both these DOM adapters derive from `RenderObject` and thus implement `IRenderObject`. Thus, a `DomNode` object of any type for which these DOM extensions are defined implements `IRenderObject`. This is a prerequisite for being rendered, as mentioned in [Rendering Components](#).
- Both implement `IRenderThumbnail`, so `SceneGraphBuilder` can build a scene graph using any `DomNode` of types for which DOM extensions are defined. For details on `SceneGraphBuilder` and `IRenderThumbnail`, see [RenderView Component](#).

RenderTransform DOM Adapter

This DOM adapter is used for "node" types in both ATGI and Collada that contain other nodes.

`RenderTransform` implements `ISetsLocalTransform`:

```
public class RenderTransform : RenderObject, IRenderThumbnail, ISetsLocalTransform
```

`ISetsLocalTransform` is an interface for `IRenderObject` objects that sets the local transform matrix (the transform from the parent to this render object) by calling `IRenderAction.PushMatrix()` in the `Traverse()` method.

The `Traverse()` method has this call:

```
action.PushMatrix(m_node.Transform, true);
```

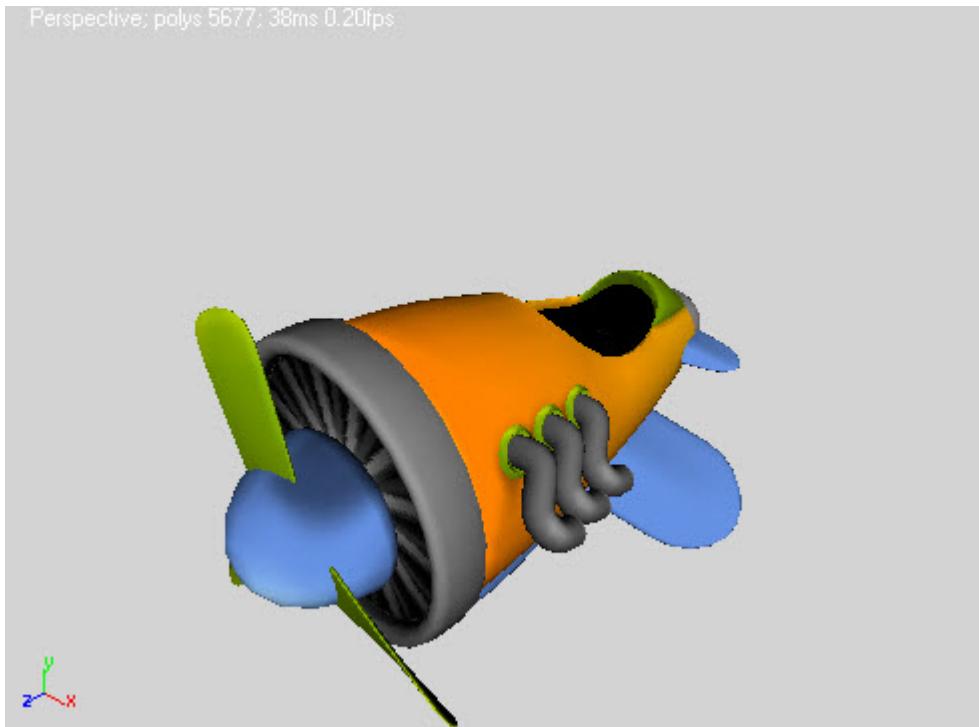
The `m_node` field is set from the original `DomNode`, adapted to `ITransformable`, an interface for objects that maintain 3D transformation information:

```
m_node = this.Cast<ITransformable>();
```

`m_node` is a node with transformation information obtained from its `Transform` property, which contains a local transformation matrix. This transformation information comes from the original `DomNode`.

This node basically specifies that any of its child node renderable objects??? should be transformed using the given matrix.

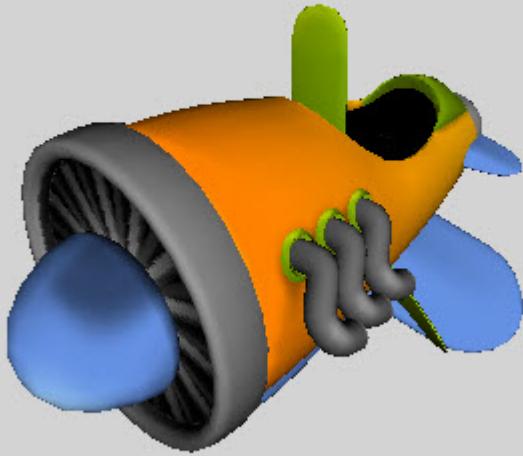
To demonstrate what this means, this figure shows a model drawn using the transformations performed by this DOM adapter:



Now, suppose that this line is removed from the `ModelViewer` component defining the DOM extensions, so the transform doesn't occur:

```
Register<RenderTransform>(Sce.Atf.Collada.Schema.node.Type);
```

Here is the resulting model:



The propellor is now in the cockpit, which is very bad news for the pilot! This group of renderable objects was not transformed properly.

This change to the transformation matrix is undone in `PostTraverse()`, which is called after post visiting the `SceneNode` specified by the graph path, so the original transformation matrix state is restored:

```
action.PopMatrix();
```

RenderPrimitives DOM Adapter

This adapter actually does the work of rendering the model, using OpenGL. This class also derives from `RenderObject` and implements `IRenderThumbnail`, but not `ISetsLocalTransform` because it does not change the transformation matrix:

```
public class RenderPrimitives : RenderObject, IRenderPick, IRenderThumbnail
```

The `RenderPrimitives.Traverse()` method is simpler than the version for `RenderTransform`. It mainly calls the base `Traverse()` method that creates a `TraverseNode` instance and adds it to the traverse list.

The `Init()` method initializes the render object, which happens once, when `SceneGraphBuilder` builds a scene graph in its `Build()` method, as described in [RenderView Component](#).

The `RenderPrimitives.Render()` method is called to render the object.

Both `Init()` and `RenderPrimitives.Render()` actually use OpenGL to perform the rendering. OpenGL's operation is beyond the scope of this discussion.

Topics in this section

[Links on this page to other topics](#)

[ATF Documentation](#), [ATF Model Viewer Sample](#), [Authoring Tools Framework](#), [Documents in ATF](#)

Property Editing Programming Discussion



Data items in an application can have various attributes or properties. The [ATF Property Editing Sample](#) shows a `ListView` of various drinks and displays their nutritional properties in property editor controls. This sample demonstrates how to edit properties of data that is not in a DOM. Other samples, such as the [ATF Timeline Editor Sample](#), show how to edit properties of DOM application data.

Note that property in the context of this sample refers to an attribute of application data — not to be confused with a C# property.

For general information on editing properties, see [Property Editing in ATF](#).

Programming Overview

The Property Editing sample customizes the standard components `PropertyEditor` and `GridPropertyEditor` to slightly modify how these editors display properties. The `ListEditingContext` class provides a selection context and a `ListView` control to display drink items, implementing `ISelectionContext`, `IListView`, and `IItemView` to do so. This sample creates its own selectable data with the `ListItem` class; it does not use an XML schema for a data model. The sample demonstrates specifying the `NestedCollectionEditor` and `EmbeddedCollectionEditor` value editors for subitem properties, depending on their data type. Otherwise, default value editors for the data type are used.

Customizing Property Editor Components

`PropertyEditingSample`, like many other samples, uses two property editors: a two-column `PropertyGrid` and the spreadsheet style `GridControl`, which can display properties for several selected items. These are provided in two components, `CustomPropertyEditor` and `CustomGridPropertyEditor`, which are added to the MEF type catalog in `Program.cs`. This sample also adds the `PropertyEditingCommands` component, which adds a context menu of property editing commands when right-clicking properties in the property editors.

Topics

- Programming Overview
- Customizing Property Editor Components
 - `CustomPropertyEditor` Component
 - `CustomGridPropertyEditor` Component
- Editing Context
 - `ISelectionContext`
 - `IListView`
 - `IItemView`
- Creating Application Data
 - `ListItem` Class Fields
 - `SubList` and `SubItem` Classes
 - `Drink` and `DrinkEbc` Classes
- Property Editor Display
 - Custom Grid Property Editor Display
 - Custom Property Editor Display
 - `EmbeddedCollectionEditor` in `CustomGridPropertyEditor`
 - Adding Collection Items

CustomPropertyEditor Component

`PropertyEditingSample` derives `CustomPropertyEditor` from the `PropertyEditor` component. `CustomPropertyEditor` is also a MEF component. This demonstrates that components can also serve as base classes for new components.

`PropertyEditingSample` makes only slight modifications to `PropertyEditor`. First, it overrides the `PropertyEditor.Configure()` method that is called in the base constructor:

```

protected override void Configure(
    out Sce.Atf.Controls.PropertyEditing.PropertyGrid propertyGrid,
    out ControlInfo controlInfo)
{
    propertyGrid = new CustomPropertyGrid();
    controlInfo = new ControlInfo(
        "Custom Property Editor".Localize(),
        "Edits selected object properties".Localize(),
        StandardControlGroup.Left);
}

```

This method uses the `CustomPropertyGrid` class for the main property editor and sets up some information for this control in the `ControlInfo`, such as the "Custom Property Editor" text that appears on the tab for the window holding the editor.

The `CustomPropertyGrid` class is a slight variant of the ATF class `PropertyGrid`:

```

private class CustomPropertyGrid : Sce.Atf.Controls.PropertyEditing.PropertyGrid
{
    public CustomPropertyGrid()
        : base(PropertyGridMode.PropertySorting | PropertyGridMode.DisplayDescriptions, new
CustomPropertyGridView())
    {
    }
}

```

This constructor sets up the `PropertyGrid`'s `PropertyGridMode` for both property sorting and displaying property descriptions in the bottom description area. In addition, it uses a new class `CustomPropertyGridView` for the property grid.

`CustomPropertyGridView` is a private class, also defined in `CustomPropertyEditor.cs`:

```

private class CustomPropertyGridView : PropertyGridView

```

It overrides only two methods of `PropertyGridView`: `DrawCategoryRow()` and `DrawPropertyRow()`. The definition of these methods is almost exactly the same as in the base class `PropertyGridView`.

`DrawCategoryRow()` draws the category row in the property grid. The override method simply changes the row's y padding and the brushes used to draw the row's background and text.

`DrawPropertyRow()` draws the property row with properties and simply adds code between the comments below to make the background red (other changes are minor):

```

Brush nameBrush = SystemBrushes.ControlText;
if (property == m_selectedProperty)
{
    g.FillRectangle(SystemBrushes.Highlight, x, y, middle - x, height);
    nameBrush = SystemBrushes.HighlightText;
}

//-----
// customized section for this sample app -- for a particular property name, make
// the background color of the property name be red.
else if (property.Descriptor.Name.Equals("CupSize"))
{
    g.FillRectangle(s_specialBackgroundBrush, x, y, middle - x, height);
}
//-----

```

These changes demonstrate how you can customize ATF classes by simply overriding their methods.

CustomGridPropertyEditor Component

`CustomGridPropertyEditor` similarly customizes the `GridPropertyEditor` component. It also overrides the `GridPropertyEditor.Configure()` method to create a `CustomGridControl` instead of a `GridControl`:

```

protected override void Configure(out GridControl gridControl, out ControlInfo controlInfo)
{
    gridControl = new CustomGridControl();
    controlInfo = new ControlInfo(
        "Custom Grid Property Editor".Localize(),
        "Edits selected object properties".Localize(),
        StandardControlGroup.Bottom);
}

private class CustomGridControl : GridControl
{
    public CustomGridControl()
        : base(PropertyGridMode.PropertySorting, new CustomGridView())
    {
    }
}

```

CustomGridControl derives from GridControl to allow property sorting and use CustomGridView as its underlying control.

CustomGridView derives from GridView and overrides the FillRowBackground() and DrawValue() methods. These methods are practically the same as in GridView.

FillRowBackground(), which draws the background of the entire row, adds these lines to its implementation:

```

Drink drink = SelectedObjects[row] as Drink;
if (drink != null && drink.CupSize == ListItem.Size.Power)
    defaultBrush = s_powerSizeBackgroundBrush;

```

When the selected row is a Drink object (discussed in [Creating Application Data](#)) with a ListItem.Size.Power cup size, the red brush s_powerSizeBackgroundBrush is used for the cell's background.

The DrawValue() method draws values in the row, and it adds these lines:

```

Drink drink = obj as Drink;
if (drink != null && drink.CupSize == ListItem.Size.Power)
    defaultBrush = s_powerSizeValueBrush;

```

When the selected row is a Drink object with a ListItem.Size.Power cup size, the light colored brush s_powerSizeValueBrush is used for the cell's text. This matches up with the other change to make text stand out against a red background; otherwise, text is dark.

Editing Context

In PropertyEditingSample, the ListEditingContext class provides a selection context and a ListView control displaying drink items. The two property editors display a collection of properties common to all objects selected in the ListView. Because the property editors are designed to display common properties of selected items, handling selections is key to the process. For information on how property editors discover what to display from a context, see [Selection Property Editing Context](#).

The ListEditingContext constructor creates and configures a ListView in which multiple drinks can be selected.

ListEditingContext also creates an AdaptableSelection object that handles selection in the ListView and subscribes to selection change events:

```

m_selection = new AdaptableSelection<object>();

m_selection.Changing += new EventHandler(selection_Changing);
m_selection.Changed += new EventHandler(selection_Changed);

```

Handling selection events is essential to displaying selected items' properties in property editors.

ListEditingContext implements several interfaces to handle selections, discussed in the following sections.

ISelectionContext

ISelectionContext is the interface for selection contexts. The Selection property gets and sets the current selection, and is implemented by simply getting and setting the AdaptableSelection m_selection field:

```
IEnumerable<object> ISelectionContext.Selection
{
    get { return m_selection; }
    set { m_selection.SetRange(value); }
}
```

The other properties and methods in this interface get information about the selection, such as the count of selected objects and the last selected object. These methods are also implemented using `m_selection`, as in the `LastSelected` property:

```
public object LastSelected
{
    get { return m_selection.LastSelected; }
}
```

The other important thing in this interface are selection related events:

```
public event EventHandler SelectionChanging;
public event EventHandler SelectionChanged;
```

`m_selection` subscribes to `EventHandler` functions that simply raise these events, as for `selectionChanging()`:

```
private void selection_Changing(object sender, EventArgs e)
{
    SelectionChanging.Raise(this, e);
}
```

Raising events hooks into the event handling system that ultimately generates lists of property descriptors the property editors use to display the common properties of selected items.

IListView

`IListView` defines the `Items` property that supplies the list of selected items, which are in the field `m_items`:

```
public IEnumerable<object> Items
{
    get { return Adapters.AsIEnumerable<object> (m_items); }
}
```

The `ListEditingContext` constructor calls `InitializeList()`, which initializes `m_items`, creating a `List<ListItem>` object:

```
private void InitializeList()
{
    m_items = new List<ListItem> ();
    m_listViewAdapter = new ListViewAdapter(m_listView);
    m_listViewAdapter.AllowSorting = true;

    ListItem listItem1 = new Drink("Caribbean Passion", "Sun-kissed flavors of passion fruit");
    listItem1.CupSize = ListItem.Size.Original;
    listItem1.Calories = 360;
    ...
    ListItem listItem2 = new DrinkEbc("Super Yumberrry", "Yumberrry juice blend");
    listItem2.CupSize = ListItem.Size.Power;
    listItem2.Calories = 490;
    listItem2.Nutrition = "4.5 Fruit Servings";
    ...
    m_items.Add(listItem1);
    m_items.Add(listItem2);

    m_listViewAdapter.ListView = this;
}
```

This method places all the drink data in the List `m_items` that is seen in the application's `ListView` and the two property editors. `ListItem` is the class that provides the custom type information for `Drink` objects used in this application. `ListItem` is discussed in [Creating](#)

Application Data

`ListViewAdapter` adapts a `ListView` control to an `IListView`. The `ListView` control is passed in `ListViewAdapter`'s constructor, and the `IListView` implementation provided by `ListEditingContext` is assigned to the `ListViewAdapter`'s `ListView` property.

In addition, the `IListView.ColumnNames` property gets a list of the column labels on the `ListView`:

```
public string[] ColumnNames
{
    get { return new string[] { Localizer.Localize("Name"), Localizer.Localize("Description") }; }
}
```

IItemView

This interface gets the display information for the "Name" and "Description" items displayed in the `ListView` columns with the `GetInfo()` method:

```
public void GetInfo(object item, ItemInfo info)
{
    ListItem listItem = Adapters.As<ListItem>(item);
    info.Label = listItem.Name;
    info.Properties = new object[] { listItem.Description };
}
```

Creating Application Data

The application needs to define data and its properties so that it is displayed in the `ListView` and property editors as expected. A property can be in a category and also have a parent property. The two-column property editor `CustomPropertyEditor` with the window labeled "Custom Property Editor" shows categories and child properties; the spreadsheet property editor `CustomGridPropertyEditor` labeled "Custom Grid Property Editor" does not show categories or children.

The `ListItem` class is used for selectable data items. It can represent a drink item in the `ListView` control visible in an application window. `ListItem` is also used for the data that represents drink properties.

`ListItem` derives from `ItemBase`, also defined in `ListItem.cs`. `ItemBase`, in turn, derives from `System.ComponentModel.CustomTypeDescriptor`, which is a simple default implementation of the `System.ComponentModel.ICustomTypeDescriptor` interface. `ICustomTypeDescriptor` supplies dynamic custom type information for an object and is useful for describing data items in an application.

When a selection occurs, through a chain of events, a collection of property descriptors is requested by calling the `GetProperties()` method for a selected item that implements `ICustomTypeDescriptor`. This is why the selectable items in the `ListView`, that is the `ListItem` objects, implement `ICustomTypeDescriptor` (through `CustomTypeDescriptor`).

Here is part of the `GetProperties()` method that `ItemBase` implements:

```

public override PropertyDescriptorCollection GetProperties()
{
    PropertyDescriptorCollection props = new PropertyDescriptorCollection(null);
    foreach (FieldInfo field in GetType().GetFields())
    {
        FieldPropertyDescriptor fieldDesc = new FieldPropertyDescriptor(field, GetType());
        bool toAdd = true;
        foreach (Attribute attr in fieldDesc.Attributes)
        {
            if (attr.GetType() == typeof(ChildPropertyAttribute))
            {
                toAdd = false;
                break;
            }
        }

        if (toAdd)
            props.Add(fieldDesc);
    }
    return props;
}

```

This method iterates through all the fields defined in its class, obtained from `GetFields()`, and creates a `FieldPropertyDescriptor` for each field. It then checks all the attributes for the data in the property descriptor. If any of the attributes is `ChildPropertyAttribute`, it does not add the property descriptor to the `PropertyDescriptorCollection` object `props`. This prevents fields that have the attribute `ChildProperty()`, such as the field `VitaminC`, from being displayed in either of the property editors:

```

[ChildProperty("Nutrition")]
[Category("Nutrition\\Vitamins")]
public int VitaminC;

```

`FieldPropertyDescriptor` derives from `PropertyDescriptor` and provides override implementations of `PropertyDescriptor`'s methods and properties.

Note that `GetFields()` gets called on `ListItem` objects, so a `FieldPropertyDescriptor` is created for every field in the `ListItem` class, which includes the fields in its parent class, `ItemBase`.

ListItem Class Fields

`ListItem` has a variety of fields, such as

```

public SubList SubItems = new SubList();
...
public Size CupSize;

```

Because a `FieldPropertyDescriptor` is created for each field, these fields appear as properties in the property editor. In the `CustomPropertyEditor`, `SubItems` and `CupSize` are listed under no category, as shown in this illustration of that property editor:

Misc	
CupSize	Original
Description	Sun-kissed flavors of passion fruit
Name	Caribbean Passion

Nutrition	
Calories	360
SaturatedFat	0
TotalFat	0
TransFat	0

Vitamins	
Nutrition	3.5 Fruit Servings
Biotin	0
Calcium	0
Iron	0
VitaminC	0
VitaminE	0

Note also the uncategorized properties `Description` and `Name`. These are fields in `ItemBase`, which `ListBase` derives from.

Some fields are marked with an attribute to place them in a category:

```
[Category("Nutrition")]
public int Calories;
```

The `Calories` field displays under the "Nutrition" category in the `CustomPropertyEditor`, as seen in the illustration above.

In addition to the "Nutrition" category, `ItemBase` also has a `Nutrition` field:

```
[Category("Nutrition\\Vitamins")]
public string Nutrition;
```

This `Category` attribute places the `Nutrition` property in the "Vitamins" category under the "Nutrition" category, as seen in the `CustomPropertyEditor` illustration above.

`ListBase` defines its own attribute class `ChildPropertyAttribute`:

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = true)]
public class ChildPropertyAttribute : Attribute
{
    /// <summary>
    /// Constructor taking the name of the field within the same class that should be
    /// considered the parent property</summary>
    /// <param name="parentPropertyName">The name of the field of the parent property</param>
    public ChildPropertyAttribute(string parentPropertyName)
    {
        ParentName = parentPropertyName;
    }
    public string ParentName;
}
```

This class itself is marked with the attribute `AttributeUsageAttribute`, which tells how its defined attribute can be used. In this case, the `AttributeTargets.Field` enumeration value means the attribute applies to fields. `AllowMultiple = true` means that the attribute can be used more than once.

This `ChildPropertyAttribute` attribute indicates the property's parent. For instance, the field `TransFat` is a child of the `Calories` field and is in the "Nutrition" category:

```
[ChildProperty("Calories")]
[Category("Nutrition")]
public int TransFat;
```

As seen in the figure of the `CustomPropertyEditor` above, `TransFat` appears in the expected place.

Note: A child property must have the same category as its parent property; if you do not specify a category when the parent has one, the property does not display properly in property editors.

Consider the `VitaminC` property, which is a child of the `Nutrition` property, and appears that way in the property editor, as the figure above shows:

```
[ChildProperty("Nutrition")]
[Category("Nutrition\\Vitamins")]
public int VitaminC;
```

Note that `VitaminC` has exactly the same category as its parent property, `Nutrition`.

SubList and SubItem Classes

`ListItem` has a `SubItems` field that is a `SubList` object:

```
public SubList SubItems = new SubList();
```

`SubList` derives from `System.Collections.CollectionBase`, a base class for a strongly typed collection. `SubList` is a collection of subitem objects, which are themselves `ListItem` objects. `SubList` defines methods to manipulate `ItemBase` (the base class of `ListItem`) objects, such as adding them to the collection.

`SubItem.cs` contains subitem classes, such as `Sub0ItemA`, that all derive from `ListItem` and ultimately from `ItemBase`. Objects in these classes are added to `SubList` objects in the `ListEditingContext.InitializeList()` method, as seen in the next section, [Drink and DrinkEbc Classes](#). Because subitems derive from `ItemBase`, they are selectable items with properties.

Drink and DrinkEbc Classes

`Drink` derives from `ListItem` and represents a drink, which is displayed in the application's `ListView` control. `DrinkEbc` derives from `Drink`.

Both `Drink` and `DrinkEbc` represent drinks and can have a `SubList` field with subitems, as the following code from `ListEditingContext.InitializeList()` indicates:

```

ListItem listItem1 = new Drink("Caribbean Passion", "Sun-kissed flavors of passion fruit");
listItem1.CupSize = ListItem.Size.Original;
listItem1.Calories = 360;
listItem1.Nutrition = "3.5 Fruit Servings";
SubList subList1 = new SubList();
subList1.Add(new Sub0ItemA());
subList1.Add(new Sub0ItemB());
subList1.Add(new Sub0ItemC());
subList1.Add(new Sub0ItemEbc());
listItem1.SubItems = subList1;

ListItem listItem2 = new DrinkEbc("Super Yumberrry", "Yumberrry juice blend");
listItem2.CupSize = ListItem.Size.Power;
listItem2.Calories = 490;
listItem2.Nutrition = "4.5 Fruit Servings";

SubList subList2 = new SubList();
subList2.Add(new Sub0ItemA());
subList2.Add(new Sub0ItemB());
subList2.Add(new Sub0ItemC());
subList2.Add(new Sub0ItemEbc());
listItem2.SubItems = subList2;

m_items.Add(listItem1);
m_items.Add(listItem2);

```

The difference in `Drink` and `DrinkEbc` classes is how their subitems appear in the property editor, which depends on the value editor used for the `SubItems` property.

As noted in [Value Editors](#), every property descriptor specifies the value editor for the property. The `FieldPropertyDescriptor` `GetEditor()` method in `ListItem.cs` does this:

```

public override object GetEditor(Type editorBaseType)
{
    if (m_field.FieldType == typeof(SubList))
    {
        if (UseEmbeddedCollectionEditor(OwnerType))
        {
            if (m_embeddedCollectionEditor == null)
            {
                m_embeddedCollectionEditor = new EmbeddedCollectionEditor();
                m_embeddedCollectionEditor.GetItemInsertersFunc = GetItemInserters;
                m_embeddedCollectionEditor.RemoveItemFunc = RemoveItem;
                m_embeddedCollectionEditor.MoveItemFunc = MoveItem;
            }
            return m_embeddedCollectionEditor;
        }

        if (m_nestedCollectionEditor == null)
        {
            m_nestedCollectionEditor = new NestedCollectionEditor();
            m_nestedCollectionEditor.EditorOpening += NestedCollectionEditor_EditorOpening;
        }
        return m_nestedCollectionEditor;
    }

    return base.GetEditor(editorBaseType);
}

```

The method first determines if the field's type is `SubList`. If so, it then checks the type of the owner of `SubList`, that is, the type of the `ListItem` containing the `SubList`, with `UseEmbeddedCollectionEditor()`:

```

private static bool UseEmbeddedCollectionEditor(Type type)
{
    return typeof(DrinkEbc).IsAssignableFrom(type)
        || typeof(Sub0ItemEbc).IsAssignableFrom(type)
        || typeof(Sub1ItemEbc).IsAssignableFrom(type);
}

```

If the type of the owner of the SubList is any of the ListItem types with "Ebc" in its name, an EmbeddedCollectionEditor value editor is used to display the collection. Otherwise, a NestedCollectionEditor value editor is used. The next section, [Property Editor Display](#), shows these editors.

If the field's type is not SubList, it calls `base.GetEditor()`, that is, `PropertyDescriptor.GetEditor()` to get the default value editor for the property's type.

Property Editor Display

This sample uses two property editors: `CustomGridPropertyEditor` and `CustomPropertyEditor`, discussed in [Customizing Property Editor Components](#).

Custom Grid Property Editor Display

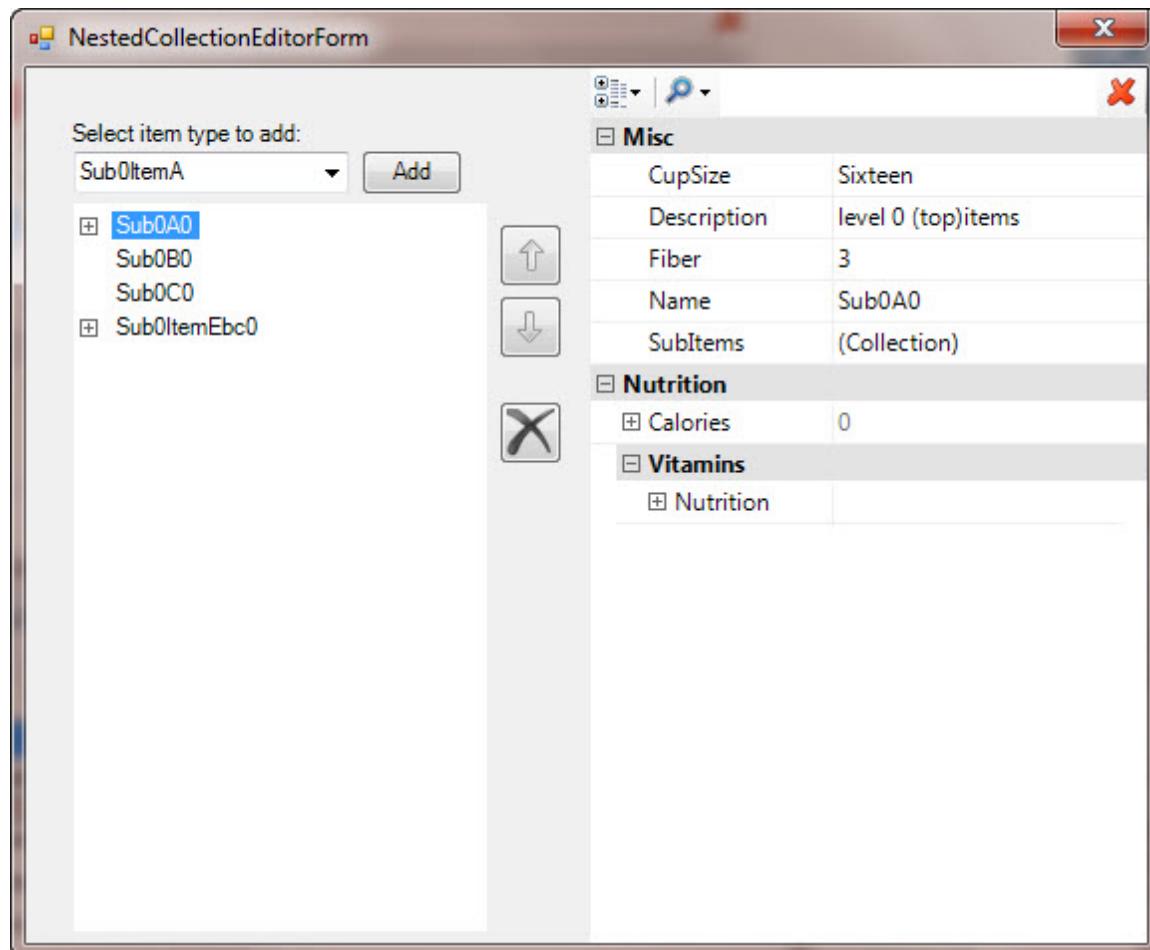
This is the spreadsheet style editor that displays all selected items, one item on each line. This illustration shows the `CustomGridPropertyEditor` when both drink items are selected in the ListView:

SubItems	Calories	Nutrition	CupSize	Name	Description
(Collection)	490	4.5 Fruit Servings	Power	Super Yumberry	Yumberry juice blend
(Collection)	360	3.5 Fruit Servings	Original	Caribbean Passi...	Sun-kissed flavors of passion fruit

Note that the "Super Yumberry" drink is highlighted in red, because its "CupSize" is "Power". This results from `CustomGridPropertyEditor`'s customization, discussed in [CustomGridPropertyEditor Component](#).

This control does not display all the properties, however. None of the child properties are shown, nor are categories.

Clicking the control on the right side inside the "(Collection)" field under the `SubItems` property displays the `NestedCollectionEditor` form:



In this dialog, the left panel lists all the items in the collection. Selecting an item shows all its properties in a two-column `PropertyEditor` in the right panel.

Custom Property Editor Display

CustomPropertyEditor is the two-column property editor that shows only one item at a time. However, it displays all child properties and categories in a tree control that opens to as many levels as necessary to display all the data. This figure shows the "Caribbean Passion" drink, which is selected in the ListView:

Misc	
CupSize	Original
Description	Sun-kissed flavors of passion fruit
Name	Caribbean Passion
SubItems	(Collection) 

Nutrition	
Calories	360

Vitamins	
Nutrition	3.5 Fruit Servings

This is a tree control, and you can open the nodes marked by plus signs to display the child properties:

Misc	
CupSize	Original
Description	Sun-kissed flavors of passion fruit
Name	Caribbean Passion
SubItems	(Collection)

Nutrition	
Calories	360
SaturatedFat	0
TotalFat	0
TransFat	0

Vitamins	
Nutrition	3.5 Fruit Servings
Biotin	0
Calcium	0
Iron	0
VitaminC	0
VitaminE	0

The initialization in `ListEditingContext.InitializeList()` described in Drink and DrinkEbc Classes shows that "Caribbean Passion" is a Drink object, and "Super Yumberry" is a DrinkEbc object. That section also indicated that a property of type SubList, such as `SubItems`, is displayed in different value editors depending on the owner of the SubList. For a Drink object owner, such as "Caribbean Passion", the value editor for a SubList is NestedCollectionEditor. Note that the value for `SubItems` is "Collection" and selecting that item displays an ellipsis button  that, when clicked, displays the NestedCollectionEditor shown in the illustration above.

The `DrinkEbc` item "Super Yumberry" has its `SubItem` property displayed using an `EmbeddedCollectionEditor` value editor, as shown in this illustration, when "Super Yumberry" is selected in the ListView:

<input type="checkbox"/> Misc	CupSize	Power																																																																																																														
Description	Yumberry juice blend																																																																																																															
Name	Super Yumberry																																																																																																															
SubItems	<input type="button" value="Add Sub0ItemA"/> <input type="button" value="X"/> <input type="button" value="Up"/> <input type="button" value="Down"/> [4 items]																																																																																																															
	<table border="1"> <tr> <td>0</td> <td>Fiber</td> <td>3</td> </tr> <tr> <td></td> <td>SubItems</td> <td>(Collection)</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0A1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> <tr> <td></td> <td>Sugar</td> <td>6</td> </tr> <tr> <td></td> <td>SubItems</td> <td>(Collection)</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td>1</td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0B1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> <tr> <td>2</td> <td>Iron</td> <td>9</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0C1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> <tr> <td></td> <td>Fiber</td> <td>3</td> </tr> <tr> <td></td> <td>SubItems</td> <td> <input type="button" value="Add Sub1ItemA"/> <input type="button" value="X"/> <input type="button" value="Up"/> <input type="button" value="Down"/> </td> </tr> <tr> <td></td> <td></td> <td> <table border="1"> <tr> <td>0</td> <td>SubItems</td> <td>(Co...on)</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub1A3</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td>1</td> <td>SubItems</td> <td></td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Su...c1</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0ItemEbc1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> </table> </td> </tr> </table>	0	Fiber	3		SubItems	(Collection)		<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub0A1		Description	level 0 (top)items		Sugar	6		SubItems	(Collection)		<input type="checkbox"/> Calories	0	1	<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub0B1		Description	level 0 (top)items	2	Iron	9		Name	Sub0C1		Description	level 0 (top)items		Fiber	3		SubItems	<input type="button" value="Add Sub1ItemA"/> <input type="button" value="X"/> <input type="button" value="Up"/> <input type="button" value="Down"/>			<table border="1"> <tr> <td>0</td> <td>SubItems</td> <td>(Co...on)</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub1A3</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td>1</td> <td>SubItems</td> <td></td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Su...c1</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0ItemEbc1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> </table>	0	SubItems	(Co...on)		<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub1A3		Description	lev...ms	1	SubItems			<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Su...c1		Description	lev...ms		<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub0ItemEbc1		Description	level 0 (top)items
0	Fiber	3																																																																																																														
	SubItems	(Collection)																																																																																																														
	<input type="checkbox"/> Calories	0																																																																																																														
	<input type="checkbox"/> Nutrition																																																																																																															
	CupSize	Sixteen																																																																																																														
	Name	Sub0A1																																																																																																														
	Description	level 0 (top)items																																																																																																														
	Sugar	6																																																																																																														
	SubItems	(Collection)																																																																																																														
	<input type="checkbox"/> Calories	0																																																																																																														
1	<input type="checkbox"/> Nutrition																																																																																																															
	CupSize	Sixteen																																																																																																														
	Name	Sub0B1																																																																																																														
	Description	level 0 (top)items																																																																																																														
2	Iron	9																																																																																																														
	Name	Sub0C1																																																																																																														
	Description	level 0 (top)items																																																																																																														
	Fiber	3																																																																																																														
	SubItems	<input type="button" value="Add Sub1ItemA"/> <input type="button" value="X"/> <input type="button" value="Up"/> <input type="button" value="Down"/>																																																																																																														
		<table border="1"> <tr> <td>0</td> <td>SubItems</td> <td>(Co...on)</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub1A3</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td>1</td> <td>SubItems</td> <td></td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Su...c1</td> </tr> <tr> <td></td> <td>Description</td> <td>lev...ms</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Calories</td> <td>0</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Nutrition</td> <td></td> </tr> <tr> <td></td> <td>CupSize</td> <td>Sixteen</td> </tr> <tr> <td></td> <td>Name</td> <td>Sub0ItemEbc1</td> </tr> <tr> <td></td> <td>Description</td> <td>level 0 (top)items</td> </tr> </table>	0	SubItems	(Co...on)		<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub1A3		Description	lev...ms	1	SubItems			<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Su...c1		Description	lev...ms		<input type="checkbox"/> Calories	0		<input type="checkbox"/> Nutrition			CupSize	Sixteen		Name	Sub0ItemEbc1		Description	level 0 (top)items																																																											
0	SubItems	(Co...on)																																																																																																														
	<input type="checkbox"/> Calories	0																																																																																																														
	<input type="checkbox"/> Nutrition																																																																																																															
	CupSize	Sixteen																																																																																																														
	Name	Sub1A3																																																																																																														
	Description	lev...ms																																																																																																														
1	SubItems																																																																																																															
	<input type="checkbox"/> Calories	0																																																																																																														
	<input type="checkbox"/> Nutrition																																																																																																															
	CupSize	Sixteen																																																																																																														
	Name	Su...c1																																																																																																														
	Description	lev...ms																																																																																																														
	<input type="checkbox"/> Calories	0																																																																																																														
	<input type="checkbox"/> Nutrition																																																																																																															
	CupSize	Sixteen																																																																																																														
	Name	Sub0ItemEbc1																																																																																																														
	Description	level 0 (top)items																																																																																																														

 |

Nutrition

Calories 490

Vitamins

Nutrition 4.5 Fruit Servings

Now the SubList is embedded in the property editor, displaying all its property information in a tree control `EmbeddedCollectionEditor`. This figure shows 4 subitems, each of which shows its child properties.

Note further that the `SubItems` property is displayed differently for each subitem. The first three are displayed using a `NestedCollectionEditor`; the last by another `EmbeddedCollectionEditor`. The value editor differs because of the subitem's owner types. Here is the code in `ListEditingContext.InitializeList()` that sets up "Super Yumberry":

```
ListItem listItem2 = new DrinkEbc("Super Yumberry", "Yumberry juice blend");
listItem2.CupSize = ListItem.Size.Power;
listItem2.Calories = 490;
listItem2.Nutrition = "4.5 Fruit Servings";

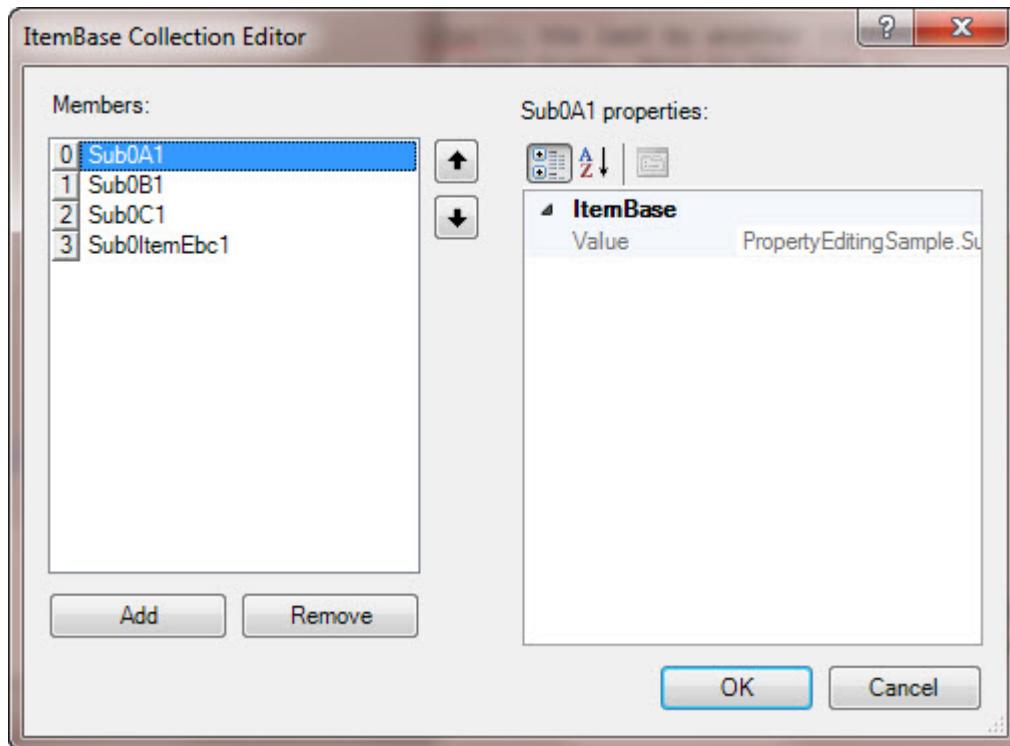
SubList subList2 = new SubList();
subList2.Add(new Sub0ItemA());
subList2.Add(new Sub0ItemB());
subList2.Add(new Sub0ItemC());
subList2.Add(new Sub0ItemEbc());
listItem2.SubItems = subList2;
```

The last subitem is a `Sub0ItemEbc`. As seen in [Drink and DrinkEbc Classes](#), `GetEditor()` returns an `EmbeddedCollectionEditor` when the property owner has "Ebc" in its name, so the last `SubList` is displayed in a `EmbeddedCollectionEditor`.

EmbeddedCollectionEditor in CustomGridPropertyEditor

A two-column property editor like `CustomPropertyEditor` can display the value editing control associated with an `EmbeddedCollectionEditor`. There's no way to display this kind of control in a spreadsheet editor because it displays all properties in a single line.

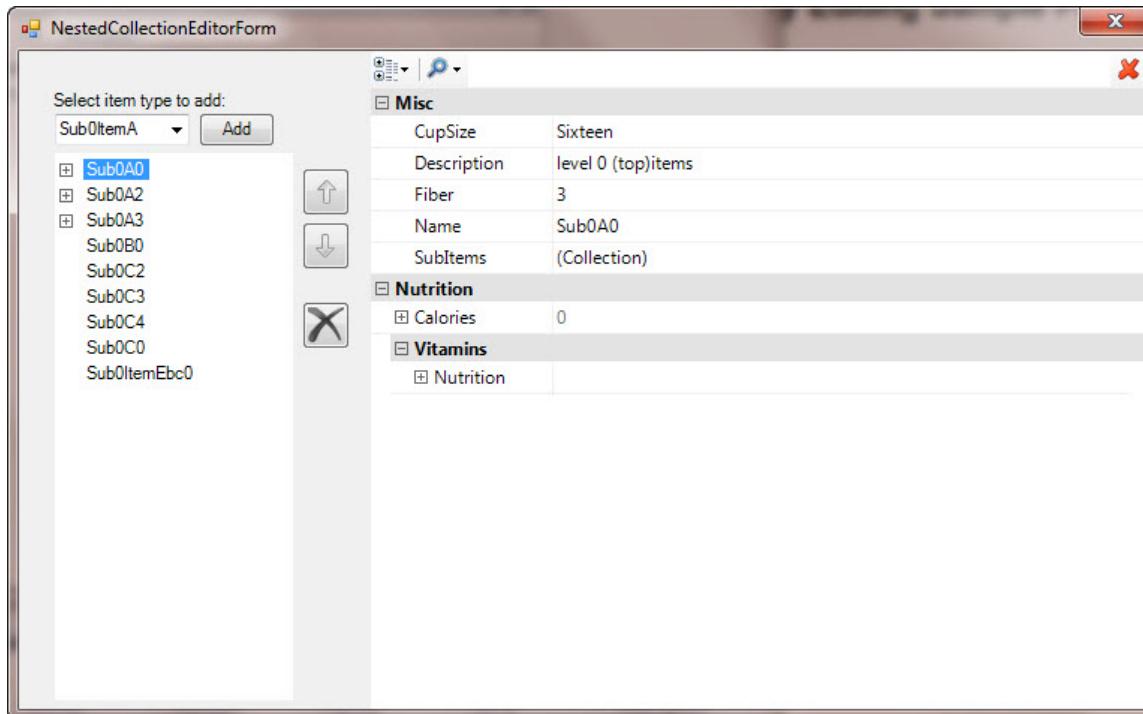
Instead, the `SubItems` property value "(Collections)" field's ellipsis button [...] displays a Collection Editor dialog:



Adding Collection Items

Both the value editing controls for the value editors `EmbeddedCollectionEditor` and `NestedCollectionEditor` have controls to add subitems to the `SubItems` collection.

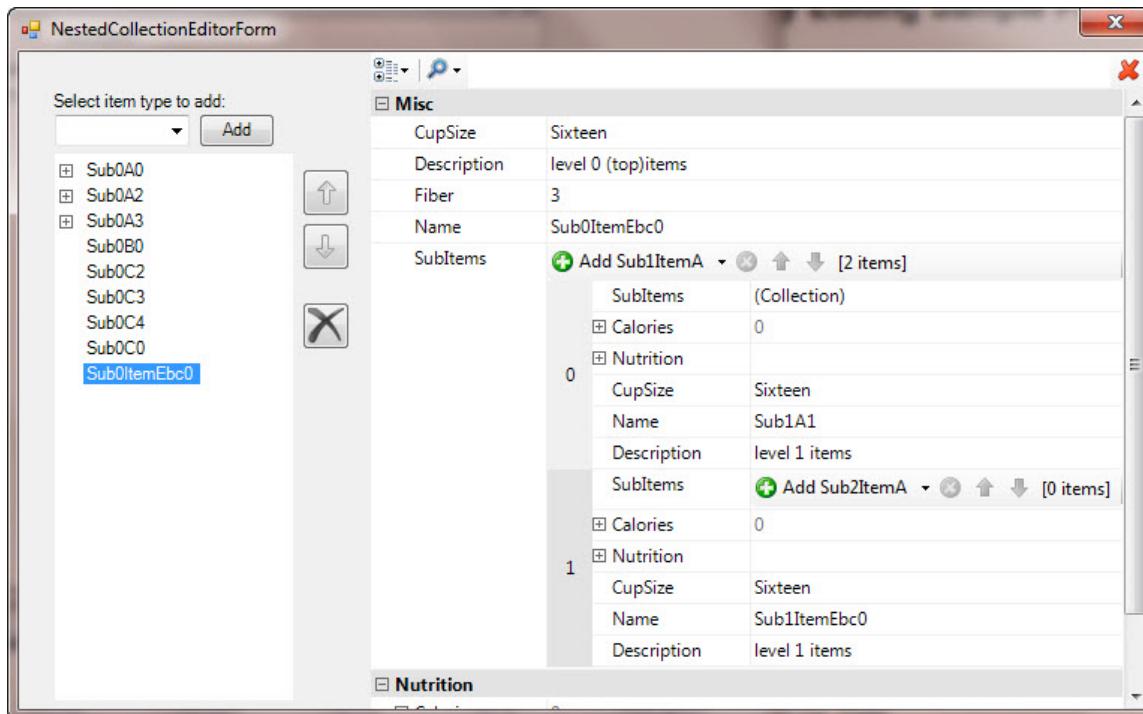
The following illustration shows a `NestedCollectionEditor`:



The left pane lists the subitems in the collection and the top one is selected. The Add button allows adding another subitem after the selected subitem from the adjacent drop down list on the left. Several items have been added to the collection.

This panel also has buttons to remove items and move them around in the list.

The last item "Sub0ItemEbc0" is an "Ebc" item. When it is selected, an `EmbeddedCollectionEditor` displays in the right pane:



The `EmbeddedCollectionEditor` also has an add button to add subitems to the `SubItems` collection, choosing the subitem from the adjacent drop down list on the right.

The action of these buttons is set in the `FieldPropertyDescriptor` class's `GetEditor()` method:

```

public override object GetEditor(Type editorBaseType)
{
    if (m_field.FieldType == typeof(SubList))
    {
        if (UseEmbeddedCollectionEditor(OwnerType))
        {
            if (m_embeddedCollectionEditor == null)
            {
                m_embeddedCollectionEditor = new EmbeddedCollectionEditor();
                m_embeddedCollectionEditor.GetItemInsertersFunc = GetItemInserters;
                m_embeddedCollectionEditor.RemoveItemFunc = RemoveItem;
                m_embeddedCollectionEditor.MoveItemFunc = MoveItem;
            }
            return m_embeddedCollectionEditor;
        }

        if (m_nestedCollectionEditor == null)
        {
            m_nestedCollectionEditor = new NestedCollectionEditor();
            m_nestedCollectionEditor.EditorOpening += NestedCollectionEditor_EditorOpening;
        }
        return m_nestedCollectionEditor;
    }

    return base.GetEditor(editorBaseType);
}

```

The methods set for the new `EmbeddedCollectionEditor`, `GetItemInserters`, `RemoveItem`, and `MoveItem`, are invoked when the corresponding buttons are clicked.

For a `NestedCollectionEditor`, the `NestedCollectionEditor_EditorOpening` method is invoked when the `NestedCollectionEditor` opens, and it sets up its controls to add, remove, and rearrange subitems.

Topics in this section

Links on this page to other topics

[ATF Property Editing Sample](#), [ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [ChildProperty\(\)](#), [Property Editing in ATF](#), [Selection Property Editing Context](#), [Value Editors](#)

Simple DOM Editor Programming Discussion

The [ATF Simple DOM Editor Sample](#) shows defining a data model in an XML Schema, editing them, and saving the edited data in an XML file. Application data is displayed in two `ListView` controls, and its properties can be examined in two property editors.

This application's data consist of sequences of events, which can contain resources. The sequence of events is displayed in the main `ListView` control, and the resources of the selected event are listed in the Resources `ListView` control.

Note that "event" in this context means application data, not an event that is processed by an event handler. Both meanings of the term are used in this discussion, and the meaning of "event" should be clear from the context.

This sample is especially interesting, because this Guide describes converting this particular sample into another ATF application. To walk through this process, see [Creating an Application from an ATF Sample](#).

Programming Overview

Simple DOM Editor uses an XML Schema to define its data model of sequences of events with resources. Its schema loader also defines DOM adapters, palette items, and property descriptors that determine which properties appear in the property editors.

The application's `PaletteClient` along with ATF's `PaletteService` creates a palette of events and resource items.

Simple DOM Editor provides the DOM adapters `Event`, `EventSequence`, and `Resource` for their corresponding types to provide properties that access information in a `DomNode` of that type.

Simple DOM Editor relies heavily on contexts. `EventContext` handles resources belonging to the selected event and manages the Resources window `ListView` editing. This context implements a host of interfaces to handle displaying items, selection, and editing. `EventSequenceContext` does a similar job for a sequence of events in the main window's `ListView`, including handling subselection.

`EventSequenceDocument` extends `DomDocument`, which implements `IDocument`. The `Editor` component is its document client, handling closing and saving documents.

The `EventListEditor` and `ResourceListEditor` components edit event sequences and their associated resources in their `ListView`s, handling drag and drop as well as copy, paste, and delete.

This sample also shows how to search `DomNodes`, using the components `DomNodeNameSearchService` and `DomNodePropertySearchService`. Searching also uses `DomNodeQueryables`, which is defined as a DOM adapter for the root type in `SchemaLoader`, so any node can be searched.

Topics

- [Programming Overview](#)
- [Application Initialization](#)
- [Data Model](#)
 - [Type Definition](#)
 - [Schema Loader Component](#)
- [Using a Palette](#)
 - [Add Palette Items](#)
 - [Implement IPaletteClient](#)
- [DOM Adapters](#)
- [Working With Contexts](#)
 - [EventContext Class](#)
 - [EventSequenceContext Class](#)
- [Document Handling](#)
 - [EventSequenceDocument Class](#)
 - [Editor Component Document Client](#)
 - [Editor Component Control Host Client](#)
- [Editors](#)
 - [EventListEditor Component](#)
 - [ResourceListEditor Component](#)
- [Node Searching](#)
 - [DomNodeQueryables DOM Adapter](#)
 - [DomNodeNameSearchService Component](#)
 - [DomNodePropertySearchService Component](#)
 - [Using a Sub Selection Context](#)

Application Initialization

Simple DOM Editor's initialization is typical for a WinForms application, as described in [WinForms Application](#) in the [ATF Application Basics and Services](#) section. Its MEF TypeCatalog contains components in the application shell framework, such as `SettingsService`, `StatusService`, `CommandService`, and `ControlHostService`. It uses components to handle documents: `DocumentRegistry`, `AutoDocumentService`, `RecentDocumentCommands`, `StandardFileCommands`, `StandardFileExitCommand`, `MainWindowTitleService`, and `TabbedControlSelector`. Editing and context management are handled by `ContextRegistry`, `StandardEditCommands`, `StandardEditHistoryCommands`, and `StandardSelectionCommands`. How these common components function is discussed elsewhere in this Guide, such as [Documents in ATF](#) and [ATF Contexts](#).

The `PropertyEditor`, `GridPropertyEditor`, and `PropertyEditingCommands` components handle property editing. These are well integrated with the ATF DOM, so the application only needs to include these components to be able to edit properties of the application's data. For details on property editing, see [Property Editing in ATF](#).

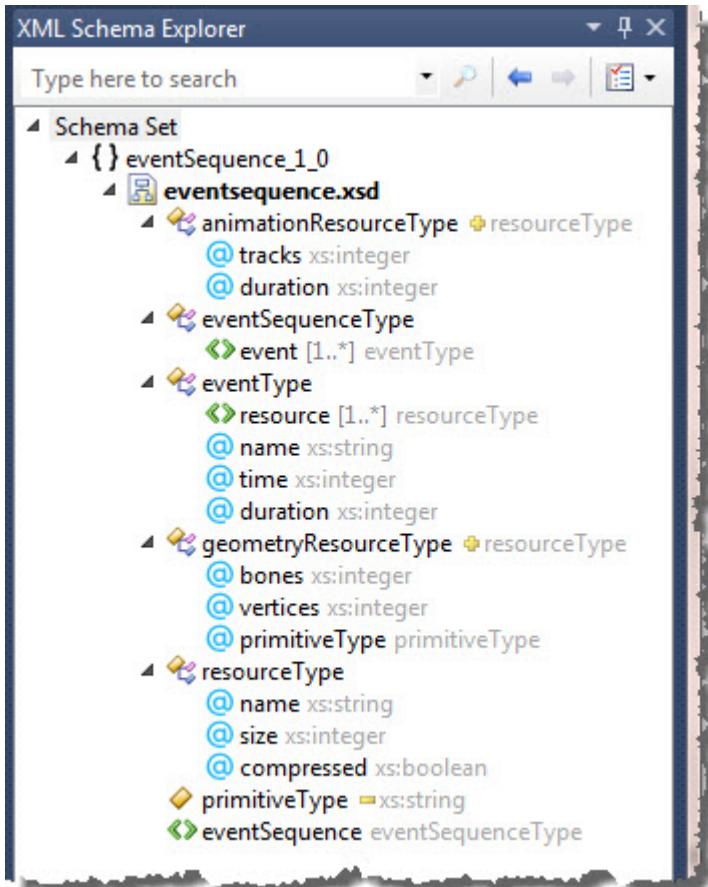
Simple DOM Editor adds a few custom components of its own:

- `PaletteClient`: populate the palette with the basic DOM types, with help from the ATF component `PaletteService`. For details, see [Using a Palette](#).
- `DomNodeNameSearchService` and `DomNodePropertySearchService`: provide a GUI for searching and replacement of `DomNode` names on the currently active document. See [Node Searching](#).
- `Editor`: create and save event sequence documents. See [Document Handling](#).
- `SchemaLoader`: load the XML Schema for the data model as well as define property descriptors and palette items. See [Data Model](#).
- `EventListEditor`: track event sequence contexts and controls that display event sequences. See [Working With Contexts](#).
- `ResourceListEditor`: display and edit resources that belong to the most recently selected event. See [ResourceListEditor Component](#).

Data Model

Type Definition

This application creates and edits event sequences. An event has attributes, such as duration, and can contain one or more resources, such as animations. The sample defines its data model in the XML Schema Definition (XSD) language in the type definition file `eventSequence.xsd`. This figure shows the Visual Studio XML Schema Explorer view of the sample's data definition:



This tree view shows that an "Event", `eventType`, contains a "Resource" type and has "name", "time", and duration attributes. The XML for

this type shows a different view the same thing:

```
<!--Event, with name, start time, duration and a list of resources-->
<xss:complexType name = "eventType">
  <xss:sequence>
    <xss:element name="resource" type="resourceType" maxOccurs="unbounded" />
  </xss:sequence>
  <xss:attribute name="name" type="xs:string" />
  <xss:attribute name="time" type="xs:integer" />
  <xss:attribute name="duration" type="xs:integer" />
</xss:complexType>
```

The data model has a "Resource" type, and the types "Animation" and "Geometry" are based on "Resource". "Resource" types have the attributes "name", "size", and "compressed".

An event sequence is simply a sequence of any number of events, as this type definition shows:

```
<!--Event sequence, a sequence of events-->
<xss:complexType name = "eventSequenceType">
  <xss:sequence>
    <xss:element name="event" type="eventType" maxOccurs="unbounded" />
  </xss:sequence>
</xss:complexType>
```

Note this definition for the root element:

```
<!--Declare the root element of the document-->
<xss:element name="eventSequence" type="eventSequenceType" />
```

The root element is of "EventSequence" type, because a document contains one event sequence. This is very important, because this type has several DOM adapters defined for it, which can apply to the entire document.

Schema Class

The ATF DomGen utility generated a Schema class file from the type definition file. The file GenSchemaDef.bat contains commands for DomGen.

Schema contains subclasses for all the data types and fields for the attributes that are of the appropriate ATF DOM metadata classes.

For example, the "Resource" type has three AttributeInfo fields for its "name", "size", and "compressed" attributes:

```
public static class resourceType
{
  public static DomNodeType Type;
  public static AttributeInfo nameAttribute;
  public static AttributeInfo sizeAttribute;
  public static AttributeInfo compressedAttribute;
}
```

The "Event" type also has AttributeInfo fields for its attributes, plus a ChildInfo field for any resources contained in the object:

```
public static class eventType
{
  public static DomNodeType Type;
  public static AttributeInfo nameAttribute;
  public static AttributeInfo timeAttribute;
  public static AttributeInfo durationAttribute;
  public static ChildInfo resourceChild;
}
```

Even though an event may contain any number of resources, it needs only one ChildInfo object, because in the ATF DOM, a node can have either a single child or a single list of children in the DOM node tree.

Schema Loader Component

The SchemaLoader component loads the XML schema and does any other initialization required for the data model. It derives from XmlSchemaTypeLoader, which performs a substantial part of the schema loading process.

Its constructor, which is automatically called during MEF initialization when the component object is created, resolves the type definition file address and loads the file:

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

SchemaLoader constructors nearly always take this form. Other standard items are the NameSpace and TypeCollection properties:

```
public string NameSpace
{
    get { return m_namespace; }
}
private string m_namespace;
...
public XmlSchemaTypeCollection TypeCollection
{
    get { return m_typeCollection; }
}
private XmlSchemaTypeCollection m_typeCollection;
```

The other work to do is to define the method OnSchemaSetLoaded(), which is called after the schema set has been loaded and the DomNodeType objects have been created. This method accomplishes several things, described in the following sections.

Schema.Initialize()

The schema loader calls Schema.Initialize() to initialize the Schema class. This method was also created by DomGen when it generated the rest of the metadata classes.

Define DOM Extensions

The loader defines DOM extensions for data types, so that methods in the various DOM adapters are called appropriately:

```
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceDocument>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceContext>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<MultipleHistoryContext>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<EventSequence>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
Schema.eventSequenceType.Type.Define(new ExtensionInfo<DomNodeQueryable>());

Schema.eventType.Type.Define(new ExtensionInfo<Event>());
Schema.eventType.Type.Define(new ExtensionInfo<EventContext>());

Schema.resourceType.Type.Define(new ExtensionInfo<Resource>());
```

Note that the types used here are the ones defined in the Schema class.

The first group's definitions all apply to Schema.eventSequenceType, which is the type of the document's root. These DOM adapters thus apply to the entire document and accomplish a number of purposes. A document describes a single sequence of events, so its DomNode tree has only one node of type "EventSequence", the tree root DomNode. This root node can be adapted to all of the adapters listed above. For instance, it's useful to treat the document as a EventSequenceContext. For more details, see [Working With Contexts](#).

The Schema.eventType and Schema.resourceType types also have DOM adapters. For more information on these adapters, see [DOM Adapters](#).

The schema loader enables metadata driven property editing for events and resources by creating an `AdapterCreator`:

```
AdapterCreator<CustomTypeDescriptorNodeAdapter> creator =
    new AdapterCreator<CustomTypeDescriptorNodeAdapter>();
Schema.eventType.Type.AddAdapterCreator(creator);
Schema.resourceType.Type.AddAdapterCreator(creator);
```

If this were not done, the property editors would not show properties from the property descriptors, defined later on.

Palette Type Setup

This step adds tag information to the type that is used later to add these types to the palette: "Event", "Animation", and "Geometry". The palette item information is encapsulated in a `NodeTypePaletteItem` object, a container class:

```
Schema.eventType.Type.SetTag(
    new NodeTypePaletteItem(
        Schema.eventType.Type,
        Localizer.Localize("Event"),
        Localizer.Localize("Event in a sequence"),
        Resources.EventImage));
```

The `NodeTypePaletteItem` contains all the information needed to display and use the palette item: its type, descriptive text, and an icon image to appear on the palette and in the `ListView` controls.

For further details on how this sets up the palette, see [Using a Palette](#).

Create Property Descriptors

Property descriptors for types specify the data that shows up in property editors for the types. For details on how this works, see [Property Editing in ATF](#) and [Property Descriptors](#) in particular.

The `NamedMetadata.SetTag()` method used below creates a `PropertyDescriptorCollection` containing an `AttributePropertyDescriptor` for each attribute of an "Event" type. This information is required for each attribute to appear in a property editor when an event is selected in the main `ListView`. The `AdapterCreator` must also be set up, as described previously in [Metadata Driven Property Editing](#).

```
Schema.eventType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                Schema.eventType.nameAttribute,
                null,
                Localizer.Localize("Event name"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Time"),
                Schema.eventType.timeAttribute,
                null,
                Localizer.Localize("Event starting time"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Duration"),
                Schema.eventType.durationAttribute,
                null,
                Localizer.Localize("Event duration"),
                false),
        }));
});
```

Using a Palette

Simple DOM Editor uses the ATF `PaletteService` component, which manages a palette of objects that can be dragged on to other controls. Simple DOM Editor allows dragging palette items onto the two `ListView` controls: the main `ListView` for event sequences and the

Resources ListView for resources of the selected event. PaletteService's constructor creates a System.Windows.Forms.UserControl for the palette, configures it, and registers it with the ControlHostService component, placing the control on the left side of the main window.

Add Palette Items

Simple DOM Editor adds its own PaletteClient component that imports IPaletteService, which is satisfied by the PaletteService component. The IInitializable.Initialize() method for the PaletteClient component adds items to the palette:

```
void IInitializable.Initialize()
{
    string category = "Events and Resources";
    m_paletteService.AddItem(Schema.eventType.Type, category, this);
    foreach (DomNodeType resourceType in m_schemaLoader.GetNodeTypes(Schema.resourceType.Type))
    {
        NodeTypePaletteItem paletteItem = resourceType.GetTag<NodeTypePaletteItem>();
        if (paletteItem != null)
            m_paletteService.AddItem(resourceType, category, this);
    }
}
```

This method adds items to the palette with the IPaletteService.AddItem() method, defined as:

```
void AddItem(object item, string categoryName, IPaletteClient client);
```

In the first call to AddItem(), the type for an event is added, Schema.eventType.Type, in the "Events and Resources" category.

The foreach loop iterates through all the types for the "Resource" type, Schema.resourceType.Type. The types "Animation" and "Geometry" are both based on the "Resource" type. For each of these "Resource" types, the loop checks that it can get a NodeTypePaletteItem for the type:

```
NodeTypePaletteItem paletteItem = resourceType.GetTag<NodeTypePaletteItem>();
```

Recall that in the SchemaLoader class, this kind of initialization occurs to add information for each "Resource" type for the palette:

```
Schema.animationResourceType.Type.SetTag(
    new NodeTypePaletteItem(
        Schema.animationResourceType.Type,
        Localizer.Localize("Animation"),
        Localizer.Localize("Animation resource"),
        Resources.AnimationImage));
```

This call to GetTag() above simply retrieves the above NodeTypePaletteItem information, if present. After it verifies that the type has an associated NodeTypePaletteItem, Initialize() adds the type to the palette.

Looping in this way guarantees that all resource types are added to the palette and makes adding new resource types to the palette later easier.

Implement IPaletteClient

The IPaletteClient.GetInfo() method gets display information for the item. It retrieves the data from the NodeTypePaletteItem that was set in SchemaLoader:

```
void IPaletteClient.GetInfo(object item, ItemInfo info)
{
    DomNodeType nodeType = (DomNodeType)item;
    NodeTypePaletteItem paletteItem = nodeType.GetTag<NodeTypePaletteItem>();
    if (paletteItem != null)
    {
        info.Label = paletteItem.Name;
        info.Description = paletteItem.Description;
        info.ImageIndex = info.GetImageList().Images.IndexOfKey(paletteItem.ImageName);
    }
}
```

`GetInfo()` simply sets the `ItemInfo` properties from the fields of the type's `NodeTypePaletteItem` object, which it obtains by calling `GetTag()`.

`Convert()` takes a palette item and returns an object that can be inserted into an `IInstancingContext` — a `DomNode` in this case:

```
object IPaletteClient.Convert(object item)
{
    DomNodeType nodeType = (DomNodeType)item;
    DomNode node = new DomNode(nodeType);

    NodeTypePaletteItem paletteItem = nodeType.GetTag<NodeTypePaletteItem>();
    if (paletteItem != null)
    {
        if (nodeType.IdAttribute != null)
            node.SetAttribute(nodeType.IdAttribute, paletteItem.Name); // unique id, for
referencing

        if (nodeType == Schema.eventType.Type)
            node.SetAttribute(Schema.eventType.nameAttribute, paletteItem.Name);
        else if (Schema.resourceType.Type.IsAssignableFrom(nodeType))
            node.SetAttribute(Schema.resourceType.nameAttribute, paletteItem.Name);
    }
    return node;
}
```

This method first casts the item as a `DomNodeType` and creates a `DomNode` of that type. The `item` parameter in `Convert()` is a selected item in the palette. And the items that were added to the palette are types, as in this line:

```
m_paletteService.AddItem(Schema.eventType.Type, category, this);
```

These palette items, like `Schema.eventType`, as defined in the `Schema` class:

```
public static class eventType
{
    public static DomNodeType Type;
    public static AttributeInfo nameAttribute;
    public static AttributeInfo timeAttribute;
    public static AttributeInfo durationAttribute;
    public static ChildInfo resourceChild;
}
```

So these items are already a `DomNodeType`, and the cast succeeds.

Next, the method `GetTag()` is used once again to retrieve a `NodeTypePaletteItem` containing palette item information.

The rest of the code sets attributes of the `DomNode` with the `DomNode.SetAttribute()` method:

```
public void SetAttribute(AttributeInfo attributeInfo, object value);
```

Both the ID and Name attributes are set. The type of item is checked, so that the proper `nameAttribute` can be set.

DOM Adapters

DOM adapters allow a `DomNode` to be dynamically cast to another interface. Simple DOM Editor provides the DOM adapters `Event`, `EventSequence`, and `Resource` for their corresponding types. All these adapters do is provide properties that access information in a `DomNode`. For instance, `Event` has a `Name` property:

```
/// Gets or sets name associated with event, such as a label
public string Name
{
    get { return GetAttribute<string>(Schema.eventType.nameAttribute); }
    set { SetAttribute(Schema.eventType.nameAttribute, value); }
}
```

The `DomNodeAdapter.GetAttribute()` method gets the value of a specified attribute, `Schema.eventType.nameAttribute`, defined in

the Schema class as

```
public static AttributeInfo nameAttribute;
```

Most of the DOM adapter properties simply access attribute values of the type, but the EventSequence DOM adapter's Events property gets all the events in an event sequence:

```
/// Gets list of Events in sequence
public IList<Event> Events
{
    get { return GetChildList<Event>(Schema.eventSequenceType.eventChild); }
}
```

Note that the returned property value is a `IList<Event>`, that is, a list of the DOM adapter `Event` objects associated with an event sequence DOM adapter `EventSequence`.

The DOM adapters in this application provide a simple way to get properties for the event, event sequence, and resource objects, all embodied in `DomNode` objects.

Working With Contexts

Simple DOM Editor relies heavily on contexts. A context provides services for operations in certain situations, hence the name context. ATF provides quite a few interfaces and classes for different types of contexts, and this sample uses several. For information about contexts in general, see [ATF Contexts](#).

This section illustrates how contexts control and facilitate editing operations in the main and Resources ListView controls.

EventContext Class

`EventContext` handles resources belonging to the selected event and manages the Resources ListView's editing. This class adapts the application data to a list and uses contexts to enable data editing as well as undoing and redoing data changes.

`EventContext` has this derivation ancestry: `EditingContext`, `HistoryContext`, `TransactionContext`, and finally `DomNodeAdapter`, so all these context classes are DOM adapters. `EditingContext` is a history context with a selection, providing a basic self-contained editing context. Several samples, such as [ATF FSM Editor Sample](#) and [ATF State Chart Editor Sample](#) have context classes that extend `EditingContext`.

As its declaration shows, `EventContext` also implements several context and other interfaces:

```
public class EventContext : EditingContext,
    IListview,
    IItemView,
    IObservableContext,
    IInstancingContext,
    IEnumerableContext
```

These ancestor classes also implement some useful interfaces, such as `IHistoryContext` and `ITransactionContext`, which allow changes in the context, such as deleting resources in an event, to be undone and redone.

OnNodeSet() Method

As a DOM adapter, `EventContext`'s `OnNodeSet()` method subscribes to several events for a `DomNode`, such as `ChildInserted`:

```
DomNode.ChildInserted += new EventHandler<ChildEventArgs>(DomNode_ChildInserted);
...
private void DomNode_ChildInserted(object sender, ChildEventArgs e)
{
    Resource resource = e.Child.As<Resource>();
    if (resource != null)
    {
        ItemInserted.Raise(this, new ItemInsertedEventArgs<object>(e.Index, resource));
    }
}
```

After adapting the DomNode to the Resource DOM adapter, this handler raises the ItemInserted event, which is defined in the IObservableContext interface.

The other event handlers defined deal with Resource objects, too, in a similar way.

IListView and IItemView Interfaces

IListView offers a couple of properties with information about the Resources ListView. Items gets the list of resources and Columns provides the column headers.

Note that a DomNode for an Event can have a list of "Resource" DomNode children that belong to the Event. Therefore, the Items property gets a list of children:

```
public IEnumerable<object> Items
{
    get { return GetChildList<object>(Schema.eventType.resourceChild); }
}
```

This property value is also returned by the IEnumerableContext.Items property.

ColumnNames simply returns a list of ListView column name strings:

```
public string[] ColumnNames
{
    get { return new string[] { Localizer.Localize("Name"), Localizer.Localize("Size") }; }
}
```

IItemView's GetInfo() method complements IListView by setting an ItemInfo with information for each column in the ListView:

```
public void GetInfo(object item, ItemInfo info)
{
    Resource resource = Adapters.As<Resource>(item);
    info.Label = resource.Name;
    string type = null;
    if (resource.DomNode.Type == Schema.animationResourceType.Type)
        type = Resources.AnimationImage;
    else if (resource.DomNode.Type == Schema.geometryResourceType.Type)
        type = Resources.GeometryImage;

    info.ImageIndex = info.GetImageList().Images.IndexOfKey(type);
    info.Properties = new object[] { resource.Size };
}
```

The resource DomNode is adapted to the Resource DOM adapter. ItemInfo.Label is set to the Name property of the Resource. The resource type is checked to decide which icon to associate with the item. The icon and name go in the "Name" column of the ListView. The "Size" column's value comes from the ItemInfo.Properties value, constructed from the property Resource.Size.

IInstancingContext Interface

IInstancingContext handles instancing in the Resources ListView, that is, working with resource instances in this control, which can be edited using menu items and by drag and drop. You insert resources in the Resources ListView for an event by dragging and dropping a resource from the palette onto the Resources ListView. Such drag and drop events are also handled using the IInstancingContext interface. For a description of how this is done, see the section [Drag and Drop and Instancing](#) in the [Instancing In ATF](#) topic.

The IInstancingContext interface provides methods to check whether items can be copied, inserted (pasted), or deleted, and performs these actions, too.

Here are the copy related methods:

```

public bool CanCopy()
{
    return Selection.Count > 0;
}
...
public object Copy()
{
    IEnumerable<DomNode> resources = Selection.AsIEnumerable<DomNode>();
    List<object> copies = new List<object>(DomNode.Copy(resources));
    return new DataObject(copies.ToArray());
}

```

`CanCopy()` simply verifies that there is at least one selected item. `Copy()` adapts the selection to a collection of `DomNodes`, copies them as a list of objects, and then constructs a `System.Windows.Forms.DataObject` from the copy, which can be placed on the Windows clipboard. The `StandardEditCommands` component actually calls `Copy()` and puts the copied data onto the clipboard.

`CanInsert()` and `Insert()` do the following:

```

public bool CanInsert(object insertingObject)
{
    IDataObject dataObject = (IDataObject)insertingObject;
    object[] items = dataObject.GetData(typeof(object[])) as object[];
    if (items == null)
        return false;

    foreach (object item in items)
        if (!Adapters.Is<Resource>(item))
            return false;

    return true;
}
...
public void Insert(object insertingObject)
{
    IDataObject dataObject = (IDataObject)insertingObject;
    object[] items = dataObject.GetData(typeof(object[])) as object[];
    if (items == null)
        return;

    DomNode[] itemCopies = DomNode.Copy(Adapters.AsIEnumerable<DomNode>(items));
    IList<Resource> resources = this.Cast<Event>().Resources;
    foreach (Resource resource in Adapters.AsIEnumerable<Resource>(itemCopies))
        resources.Add(resource);

    Selection.SetRange(itemCopies);
}

```

The first part of both methods is identical, simply verifying that an insertion can occur.

`StandardEditCommands` calls `CanInsert()` to determine whether the `Insert` menu item is enabled or not. `CanInsert()` casts the inserted data as a `IDataObject` and then casts it to an array of selected objects, each of which is a `DomNode` of type "Resource". The second part of `CanInsert()` checks whether all the selected items can be adapted as `Resource` objects. If so, the method returns true, and false otherwise.

For the insert operation, `StandardEditCommands` gets the clipboard data and calls `Insert()` to insert this data. `Insert()` casts the inserted data as a `IDataObject` and then casts it to an array of selected objects, just as for `CanInsert()`. These objects are adapted to `DomNodes` and then copied.

In this method, `this` represents an `EventArgs` object, which is an adapted `DomNode` of type "Event" representing a selected event in the main `ListView`. Both `EventArgs` and `Event` are DOM adapters for the "Event" type, as shown in these lines from `SchemaLoader` where DOM adapters are defined:

```

Schema.eventType.Type.Define(new ExtensionInfo<Event>());
Schema.eventType.Type.Define(new ExtensionInfo<EventArgs>());

```

Because `Event` is a DOM adapter for a `DomNode` of type "Event", this `DomNode` can be adapted to an `EventArgs` object, and its list of resources can be obtained from the `Resources` property of the `EventArgs` DOM adapter:

```
IList<Resource> resources = this.Cast<Event>().Resources;
```

Finally, each inserted `DomNode` is adapted to a `Resource` object and added to the list of resources for the selected `Event` in the main `ListView`.

The deletion operations are simpler:

```
public bool CanDelete()
{
    return Selection.Count > 0;
}
...
public void Delete()
{
    foreach (DomNode node in Selection.AsIEnumerable<DomNode>())
        node.RemoveFromParent();

    Selection.Clear();
}
```

`CanDelete()` is identical to `CanCopy()`, simply verifying that there's something selected to delete.

`Delete()` iterates all selected items' underlying `DomNodes` and removes them from their parents with `DomNode.RemoveFromParent()`, effectively removing them from the application data. It also clears the selection using the `Selection.Clear()` method.

EventSequenceContext Class

`EventSequenceContext` handles a sequence of events in the main `ListView`, managing editing this list.

`EventSequenceContext` has many similarities to `EventContext`, including its derivation ancestry from `EditingContext`. One of the differences is that `EventSequenceContext`'s constructor creates and configures the `ListView` instance used to display events. It also creates a `GenericSelectionContext` as a subselection context.

`EventSequenceContext` implements all the interfaces that `EventContext` does. Their implementations are almost identical, but `Resource` objects are replaced by `Event` objects in `EventSequenceContext`. The `IInstancingContext` implementations look identical except for that substitution, for example.

IListView Interface

The `IListView` implementation merely has the obvious differences from `EventContext`. The `Items` property returns a child list of `Event` objects, rather than `Resource` objects. `ColumnNames` returns the list of headings seen over the main `ListView` columns.

```
public IEnumerable<object> Items
{
    get { return GetChildList<object>(Schema.eventSequenceType.eventChild); }
}
...
public string[] ColumnNames
{
    get { return new string[] { Localizer.Localize("Name"), Localizer.Localize("Duration") }; }
}
```

ISubSelectionContext Interface and GenericSelectionContext Class

`ISubSelectionContext` is implemented for node search to show which attribute was found in a list of search hits when a hit list entry is selected. The found attribute in the selected node is highlighted in the property editor, which constitutes the subselection. For details, see [Node Searching](#).

`EventSequenceContext` uses `GenericSelectionContext` for a subselection context:

```
m_subSelectionContext = new GenericSelectionContext();
```

`EventSequenceContext` implements the property `ISubSelectionContext.SubSelectionContext`, which returns an

ISelectionContext:

```
public ISelectionContext SubSelectionContext
{
    get { return m_subSelectionContext; }
}
```

GenericSelectionContext is a straightforward implementation of ISelectionContext. Its constructor creates a Selection object and subscribes to an event on it:

```
public GenericSelectionContext()
{
    m_selection = new Selection<object>();
    m_selection.Changed += new EventHandler(selection_Changed);

    // suppress compiler warning
    if (SelectionChanging == null) return;
}
```

The Changed event is raised when a subselection occurs. For details on how this occurs, see [Node Searching](#).

The ISelectionContext implementation simply makes use the Selection object, as in GetSelection<T>() and LastSelected:

```
public IEnumerable<T> GetSelection<T>()
    where T : class
{
    return m_selection.AsIEnumerable<T>();
}
...
public object LastSelected
{
    get { return m_selection.LastSelected; }
}
```

Document Handling

Document handling applications typically implement both IDocument for a document object and IDocumentClient for the document client.

EventSequenceDocument Class

EventSequenceDocument extends DomDocument, which implements IDocument.

The DomDocument class provides the basics of a document: IsReadOnly and Dirty properties, a DirtyChanged event, and the OnDirtyChanged() and OnReloaded event handlers. DomDocument extends DomResource and implements IResource to provide the resource properties Type and Uri to describe a document's type and location.

EventSequenceDocument itself implements overrides for the property Type and the methods OnUriChanged() and OnDirtyChanged() for application-specific behavior. These latter event handlers simply execute the base methods, as well as update ControlInfo for the ListView displaying the new document. ControlInfo holds information about controls hosted by ControlHostService.

EventSequenceDocument also implements ISearchableContext to allow searching for data in DomNode objects. For details on searching, see [Node Searching](#).

Editor Component Document Client

The Editor component is the client for event sequence documents.

DocumentClientInfo Class

Editor uses the DocumentClientInfo class to hold document editor information. The Info property gets the DocumentClientInfo from a static variable:

```

public DocumentClientInfo Info
{
    get { return DocumentClientInfo; }
}

/// <summary>
/// Information about the document client</summary>
public static DocumentClientInfo DocumentClientInfo = new DocumentClientInfo(
    Localizer.Localize("Event Sequence"),
    new string[] { ".xml", ".esq" },
    Sce.Atf.Resources.DocumentImage,
    Sce.Atf.Resources.FolderImage,
    true);

```

Opening a Document

`CanOpen()` simply checks that the filename extension is suitable, using the `DocumentClientInfo`:

```

public bool CanOpen(Uri uri)
{
    return DocumentClientInfo.IsCompatibleUri(uri);
}

```

The `Open()` method reads the document file's data and converts it into a tree of `DomNode` objects. After that, it sets up the context and other information needed for the new document. The initial phase of opening creates a `DomNode` for the tree root and gets the file name:

```

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

```

For an existing document, a `FileStream` is created and read. Because the document is stored as an XML document, you can use an `DomXmlReader` to read data and convert it to a tree of `DomNodes`. `DomXmlReader.Read()` returns the root `DomNode` of the tree it creates. For a new document, a `DomNode` with the `DomNodeType` of the event sequence root element — from the `Schema` class — is created.

```

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.eventSequenceType.Type, Schema.eventSequenceRootElement);
}

```

Next, `Open()` performs a series of ATF DOM set up operations to complete the open process:

```

EventSequenceDocument document = null;
if (node != null)
{
    // Initialize Dom extensions now that the data is complete
    node.InitializeExtensions();

    EventSequenceContext context = node.As<EventSequenceContext>();

    ControlInfo controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center);
    context.ControlInfo = controlInfo;

    // set document URI
    document = node.As<EventSequenceDocument>();
    document.Uri = uri;

    // set document GUIs for search and replace
    document.SearchUI = new DomNodeSearchToolStrip();
    document.ReplaceUI = new DomNodeReplaceToolStrip();
    document.ResultsUI = new DomNodeSearchResultsListView(m_contextRegistry);

    context.ListView.Tag = document;

    // show the ListView control
    m_controlHostService.RegisterControl(context.ListView, controlInfo, this);
}

return document;
}

```

First, an `EventSequenceDocument` object is created.

`InitializeExtensions()` initializes all the DOM adapters defined in the schema loader, shown in [Define DOM Extensions](#).

An `EventSequenceContext` is created by adapting the root `DomNode` to `EventSequenceContext`, which is a DOM adapter for the "EventSequence" type. Recall that the "EventSequence" type is the type of the document's root.

The event sequence document's data is going to be displayed in a `ListView` that is part of the `EventSequenceContext`. A `ControlInfo` is set up for this control and is placed in the `EventSequenceContext`.

`EventSequenceDocument` is also a DOM adapter for the "EventSequence" type, so the tree root `DomNode` can be adapted to `EventSequenceDocument`.

The `EventSequenceDocument` also has some search properties set. For information about searching, see [Node Searching](#).

The document is saved in the context's `ListView` for future reference:

```
context.ListView.Tag = document;
```

Finally, the `ListView` control is registered with the `ControlHostService` component.

Saving a Document

`Save()` uses the capabilities of `DomXmlWriter` to write the `DomNode` tree to an XML file. `DomXmlWriter` gets data model information from the schema so it knows how to write the `DomNode` tree to XML with its `Write()` method. The document is cast back to an `EventSequenceDocument` to get its root `DomNode` for `Write()`.

```

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);
        EventSequenceDocument eventSequenceDocument = (EventSequenceDocument)document;
        writer.Write(eventSequenceDocument.DomNode, stream, uri);
    }
}

```

Closing a Document

Closing the document entails cleaning up the contexts that were used:

```
public void Close(IDocument document)
{
    EventSequenceContext context = Adapters.As<EventSequenceContext>(document);
    m_controlHostService.UnregisterControl(context.ListView);
    context.ControlInfo = null;

    // close all active EditingContexts in the document
    foreach (DomNode node in context.DomNode.Subtree)
        foreach (EditingContext editingContext in node.AsAll<EditingContext>())
            m_contextRegistry.RemoveContext(editingContext);

    // close the document
    m_documentRegistry.Remove(document);
}
```

The ListView control in the EventSequenceContext is unregistered.

Finally, any EditingContexts that were created are removed from the ContextRegistry, and the document is removed from the DocumentRegistry.

Contexts are discussed in [Working With Contexts](#).

Editor Component Control Host Client

Editor also implements IControlHostClient for the ListView that is in the EventSequenceContext. This requires it to handle the control's activation, deactivation, and close events.

The Activate() method retrieves the EventSequenceDocument from the Tag in the ListView it was stored in and sets it as the activate document in the DocumentRegistry component. It gets the EventSequenceContext by adapting the document again and sets it as the active context with the ContextRegistry:

```
void IControlHostClient.Activate(Control control)
{
    EventSequenceDocument document = control.Tag as EventSequenceDocument;
    if (document != null)
    {
        m_documentRegistry.ActiveDocument = document;

        EventSequenceContext context = document.As<EventSequenceContext>();
        m_contextRegistry.ActiveContext = context;
    }
}
```

Close() performs a similar operation to Activate(), but with the aim of closing things out. It uses the IDocumentService, provided by the StandardFileCommands component in this sample, to close the document. If it succeeds, it then removes the context associated with the document from the ContextRegistry.

```
bool IControlHostClient.Close(Control control)
{
    bool closed = true;

    EventSequenceDocument document = control.Tag as EventSequenceDocument;
    if (document != null)
    {
        closed = m_documentService.Close(document);
        if (closed)
            m_contextRegistry.RemoveContext(document);
    }

    return closed;
}
```

Two components handle editing event sequences and their associated resources: `EventListEditor` and `ResourceListEditor`. Events and resources are edited by either dragging and dropping them from the palette onto a `ListView` or deleting them from the `ListView`. You can also edit events and resources with context menus in the `ListView`. Event and resource attributes can also be edited, but that falls under property editing, which is all performed by the `PropertyEditor`, `GridPropertyEditor`, and `PropertyEditingCommands` components.

Document data is held in a tree of `DomNodes`. The editor components freely adapt the document's root `DomNode` to the various DOM adapters for the "EventSequence" type, such as `EventSequenceDocument` and `EventSequenceContext`, depending on what capabilities are needed.

EventListEditor Component

`EventListEditor` edits the event sequence document in a `ListView` on a tab. Several documents can be opened at a time, one to each tab in the central window. An `EventSequenceContext` owns the `ListView` for each document.

The constructor sets up event handling for context changes, that is, tracking which `ListView` is active:

```
m_contextRegistry.ActiveContextChanged += new EventHandler(contextRegistry_ActiveContextChanged);
m_contextRegistry.ContextAdded += new EventHandler<ItemInsertedEventArgs<object>>(contextRegistry_ContextAdded);
);
m_contextRegistry.ContextRemoved += new EventHandler<ItemRemovedEventArgs<object>>(contextRegistry_ContextRemoved);
);
```

The `ActiveContextChanged` event handler, `contextRegistry_ActiveContextChanged`, gets the current `EventSequenceContext` and, if non-null, uses it to obtain the `ListView` settings and apply them to the new `ListView`:

```
private void contextRegistry_ActiveContextChanged(object sender, EventArgs e)
{
    if (m_eventSequenceContext != null)
        m_listViewSettings = m_eventSequenceContext.ListViewAdapter.Settings;

    m_eventSequenceContext = m_contextRegistry.GetMostRecentContext<EventSequenceContext>();
    if (m_eventSequenceContext != null)
        m_eventSequenceContext.ListViewAdapter.Settings = m_listViewSettings;
}
```

When an `EventSequenceContext` is added for a new document, this event's handler subscribes the new context's `ListView` to editing events:

```
private void contextRegistry_ContextAdded(object sender, ItemInsertedEventArgs<object> e)
{
    EventSequenceContext context = Adapters.As<EventSequenceContext>(e.Item);
    if (context != null)
    {
        context.ListView.DragOver += new DragEventHandler(listView_DragOver);
        context.ListView.DragDrop += new DragEventHandler(listView_DragDrop);
        context.ListView.MouseUp += new MouseEventHandler(listView_MouseUp);

        context.ListViewAdapter.LabelEdited +=
            new EventHandler<LabelEditedEventArgs<object>>(listViewAdapter_LabelEdited);
    }
}
```

When an `EventSequenceContext` is removed, all these events are unsubscribed from.

These editing event handlers do the actual editing. For instance, this is the `listView_DragDrop` method that takes care of a `DragDrop` event on the `ListView`:

```

private void listView_DragDrop(object sender, DragEventArgs e)
{
    IInstancingContext context = m_eventSequenceContext;
    if (context.CanInsert(e.Data))
    {
        ITransactionContext transactionContext = context as ITransactionContext;
        TransactionContexts.DoTransaction(transactionContext,
            delegate
            {
                context.Insert(e.Data);
            },
            Localizer.Localize("Drag and Drop"));

        if (m_statusService != null)
            m_statusService.ShowStatus(Localizer.Localize("Drag and Drop"));
    }
}

```

This method can treat the current `EventSequenceContext` as an `IInstancingContext`, because `EventSequenceContext` implements `IInstancingContext`. It uses this `IInstancingContext` to check if the drag and drop editing operation can occur, that is, can the type represented by the dragged icon be inserted into the `ListView`? If so, it uses a `ITransactionContext` to perform the insertion, so that it can be undone if desired. Note that both the `CanInsert()` and `Insert()` methods are invoked on the `IInstancingContext`. The methods actually invoked are in the `IInstancingContext` implementation of `EventSequenceContext`, discussed in [EventSequenceContext Class](#). Also note that the `ITransactionContext` can be used here, because one of the classes `EventSequenceContext` derives from implements `ITransactionContext`, as mentioned in [EventSequenceContext Class](#),

The mouse up event is handled by `listView_MouseUp` and checks for a right-button click to determine whether to display a context menu:

```

private void listView_MouseUp(object sender, MouseEventArgs e)
{
    Control control = sender as Control;
    object target = null; // TODO
    if (e.Button == MouseButtons.Right)
    {
        IEnumerable<object> commands =
            ContextMenuCommandProvider.GetCommands(
                Lazies.GetValues(m_contextMenuCommandProviders), m_eventSequenceContext, target);

        Point screenPoint = control.PointToScreen(new Point(e.X, e.Y));
        m_commandService.RunContextMenu(commands, screenPoint);
    }
}

```

The method uses `ContextMenuCommandProvider.GetCommands()` to get any context commands. This is an extension method for the `IContextMenuCommandProvider` interface for providers of context menu commands. There are two providers implementing `IContextMenuCommandProvider` in this sample: `StandardEditCommands` that implements the standard Edit menu's Cut, Copy, Paste, and Delete commands; and `PropertyEditingCommands` that provides context commands for property editors. In this case, `StandardEditCommands` is the provider; `PropertyEditingCommands` provides context menu items in the property editors. These commands are displayed in a context menu by invoking `RunContextMenu()` on the `CommandService` component.

ResourceListEditor Component

This component performs editing operations on the Resources `ListView` control in the dialog, displaying the resources for the currently selected event. You can also drag resources from the palette onto this `ListView` and edit them.

`ResourceListEditor` is very similar to `EventListEditor` and does pretty much the same things. This editor tracks editing events on a `ListView`, such as drag and drop, and performs the operation when valid.

What differs is that there is only one `ListView` control. Instead of belonging to a context, this `ListView` is owned by the editor itself. This component's `IInitializable.Initialize()` method creates the `ListView` and initializes it: sets its properties, subscribes to editing events, and registers it with the `ControlHostService` component, very similarly to how `EventSequenceContext` sets up its `ListView`. It also uses the `SettingsServices` component to persist its `ListView` settings.

```

void IInitializable.Initialize()
{
    m_resourcesListView = new ListView();
    m_resourcesListView.AllowDrop = true;
    m_resourcesListView.MultiSelect = true;
    m_resourcesListView.AllowColumnReorder = true;
    m_resourcesListView.LabelEdit = true;
    m_resourcesListView.Dock = DockStyle.Fill;

    m_resourcesListView.DragOver += new DragEventHandler(resourcesListView_DragOver);
    m_resourcesListView.DragDrop += new DragEventHandler(resourcesListView_DragDrop);
    m_resourcesListView.MouseUp += new MouseEventHandler(resourcesListView_MouseUp);
    m_resourcesListViewAdapter = new ListViewAdapter(m_resourcesListView);
    m_resourcesListViewAdapter.LabelEdited +=
        new EventHandler<LabelEditedEventArgs<object>>(resourcesListViewAdapter_LabelEdited);

    m_resourcesControlInfo = new ControlInfo(
        Localizer.Localize("Resources"),
        Localizer.Localize("Resources for selected Event"),
        StandardControlGroup.Bottom);

    m_controlHostService.RegisterControl(m_resourcesListView, m_resourcesControlInfo, this);

    if (m_settingsService != null)
    {
        SettingsServices.RegisterSettings(
            m_settingsService,
            this,
            new BoundPropertyDescriptor(this, () => ListViewSettings, "ListViewSettings", "", ""))
    }
}
}

```

Because this ListView shows the resources for the currently selected event, this component must track context changes. The ResourceListEditor constructor subscribes to the ActiveContextChanged event on the ContextRegistry. This handler, contextRegistry_ActiveContextChanged, gets the current m_eventSequenceContext and subscribes to its SelectionChanged event, so that it is informed of event selection changes in the currently active document. It then calls UpdateEvent() to update the ResourceListEditor's ListView. UpdateEvent() determines the last selected event in the current event sequence document:

```

private void UpdateEvent()
{
    Event nextEvent = null;
    if (m_eventSequenceContext != null)
        nextEvent = m_eventSequenceContext.Selection.GetLastSelected<Event>();

    if (m_event != nextEvent)
    {
        // remove last event's editing context in case it was activated
        if (m_event != null)
            m_contextRegistry.RemoveContext(m_event.Cast<EventContext>());

        m_event = nextEvent;

        // get next event's editing context and bind to resources list view
        EventContext eventContext = null;
        if (nextEvent != null)
            eventContext = nextEvent.Cast<EventContext>();

        m_resourcesListViewAdapter.ListView = eventContext;
    }
}

```

In this method, Event refers to event data that is edited by this sample, not a subscribable event.

Because application data is in an ATF DOM, an event is represented by a DomNode. Such a DomNode can be adapted to both Event and EventContext DOM node adapters, as indicated by this definition in the SchemaLoader for the "Event" type:

```
Schema.eventType.Type.Define(new ExtensionInfo<Event>());
Schema.eventType.Type.Define(new ExtensionInfo<EventContext>());
```

Both these adaptations are used in this method. The method gets the last selected event item in the current EventSequenceContext's ListView as an Event in this line:

```
nextEvent = m_eventSequenceContext.Selection.GetLastSelected<Event>();
```

If nextEvent is different from the previous Event, m_event, then m_event is adapted to EventContext and removed from the ContextRegistry:

```
m_contextRegistry.RemoveContext(m_event.Cast<EventContext>());
```

Finally, the last selected event, nextEvent, is adapted to EventContext and used to update the ResourceListEditor's ListView:

```
if (nextEvent != null)
    eventContext = nextEvent.Cast<EventContext>();

m_resourcesListViewAdapter.ListView = eventContext;
```

This line sets the ListView property, the list data in the ListView. This property is a IListView and EventContext implements IListView, so this assignment works.

Node Searching

Simple DOM Editor employs two forms of DomNode searching, provided by two components:

- DomNodeNameSearchService: a component in Simple DOM Editor for searching and replacing DomNode names in the currently active document.
- DomNodePropertySearchService: an ATF component that searches for and replaces DomNode names and attribute values using a regular expression in the currently active document.

Both these components use ATF DomNode searching controls.

Searching also uses the ATF DomNodeQueryable DOM adapter.

DomNodeQueryable DOM Adapter

DomNodeQueryable is defined as a DOM adapter for the "EventSequence" type in SchemaLoader, so any node in the tree can be searched:

```
Schema.eventSequenceType.Type.Define(new ExtensionInfo<DomNodeQueryable>());
```

DomNodeQueryable implements several context interfaces to handle searching, displaying results, and replacing results:

- IQueryableContext: Define a Query() method to enumerate objects that satisfy the conditions of search predicates.
- IQueryableResultContext: Access the results of a query and be notified when these results change.
- IQueryableReplaceContext: Interface for classes in which containing objects can be replaced.

There are three interfaces that relate to these previous three:

- ISearchUI: Interface for a client-defined UI that provides user controls for specifying search criteria, and for triggering a search on that criteria.
- IResultsUI: Interface for a client-defined control that displays search results.
- IReplaceUI: Interface for a client-defined control that allows a user to apply replacement data to search results.

These interfaces all define a Bind() method that binds a control to the data the context represents: the above contexts are DOM adapters for the root DomNode of the document.

DomNodeQueryable uses DomNodeQueryMatch to hold matching properties of a given DomNode.

DomNodeNameSearchService Component

DomNodeNameSearchService uses the DomNodeNameSearchControl for its UI. This control was created in the Forms Designer using

ATF controls. This control's `InitializeComponent()` is, in part:

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    System.ComponentModel.ComponentResourceManager resources = new System.ComponentModel.
    ComponentResourceManager(typeof(DomNodeNameSearchControl));
    this.domNodeSearchTextBox1 = new Sce.Atf.Dom.DomNodeNameSearchTextBox();
    this.domNodeReplaceTextBox1 = new Sce.Atf.Applications.ReplaceTextBox();
    this.domNodeSearchResultsListView1 = new Sce.Atf.Dom.DomNodeSearchResultsListView();
    ...
}
```

These new controls are added to the main control. `DomNodeNameSearchTextBox`, for instance, is a simple `TextBox` for specifying a `DomNode` name search. `DomNodeSearchResultsListView` (derived from `SearchResultsListView`), also used by `DomNodePropertySearchService`, displays search results in a `ListBox`.

When the active context changes, this method is called:

```
private void contextRegistry_ActiveContextChanged(object sender, EventArgs e)
{
    SearchUI.Bind(m_contextRegistry.GetActiveContext<IQueryableContext>());
    ResultsUI.Bind(m_contextRegistry.GetActiveContext<IQueryableResultContext>());
    ReplaceUI.Bind(m_contextRegistry.GetActiveContext<IQueryableReplaceContext>());
}
```

This method binds the controls to the data of the currently active context, so the controls work with the active data.

DomNodePropertySearchService Component

`DomNodePropertySearchService` offers a more powerful search than `DomNodeNameSearchService`. Its constructor creates a `UserControl` and adds ATF controls that perform the `DomNode` search: `DomNodeSearchToolStrip`, `DomNodeReplaceToolStrip`, and `DomNodeSearchResultsListView`. For instance, `DomNodeSearchToolStrip` offers a `ToolStrip` for specifying a search on `DomNodes` for both the name and other attributes, using regular expressions.

```
public DomNodePropertySearchService(
    IContextRegistry contextRegistry,
    IControlHostService controlHostService)
{
    m_contextRegistry = contextRegistry;
    m_controlHostService = controlHostService;

    // define root control
    m_rootControl = new UserControl();
    m_rootControl.Name = "Search and Replace";
    m_rootControl.SuspendLayout();
    m_rootControl.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.
    .Right;

    // Create and add the search input control
    SearchUI = new DomNodeSearchToolStrip();
    SearchUI.Control.Dock = DockStyle.None;
    m_rootControl.Controls.Add(SearchUI.Control);
    SearchUI.UIChanged += UIElement_Changed;

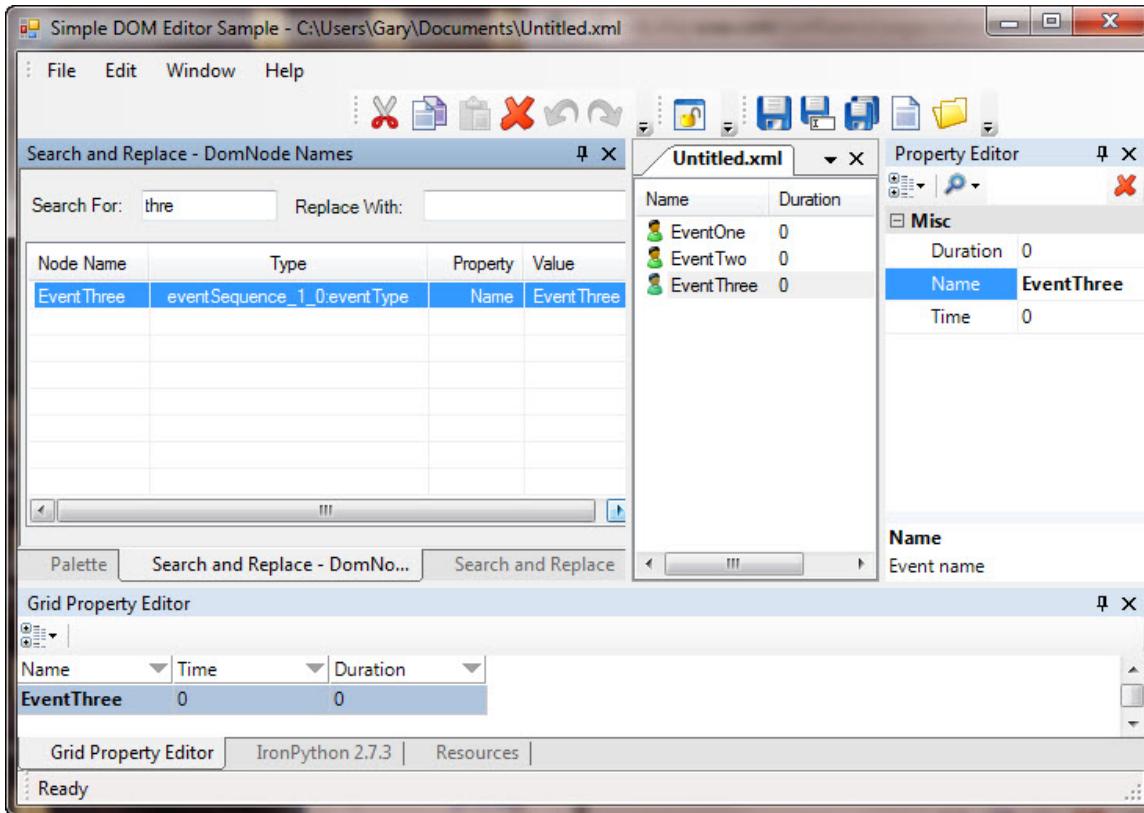
    // Create and add the replace input control
    ReplaceUI = new DomNodeReplaceToolStrip();
    ReplaceUI.Control.Dock = DockStyle.None;
    m_rootControl.Controls.Add(ReplaceUI.Control);
    ReplaceUI.UIChanged += UIElement_Changed;

    // Create and add the results output control
    ResultsUI = new DomNodeSearchResultsListView(m_contextRegistry);
    ResultsUI.Control.Dock = DockStyle.None;
    m_rootControl.Controls.Add(ResultsUI.Control);
    ResultsUI.UIChanged += UIElement_Changed;
    ...
}
```

Using a Sub Selection Context

Search results are displayed in the `ListBox` of a `DomNodeSearchResultsListView` for both search components.

The following illustration shows a name search using `DomNodeNameSearchService`, finding one `DomNode` and displaying it in its `ListView`. When the found node is selected in this `ListView` hit list, the node's properties are displayed in both property editors. In addition, the "Name" value of the event that was found is highlighted in the property editors, which is the subselection.



These search components use the `EventSequenceContext` for a subselection context, so an `EventSequenceContext` must be the active context for these components' controls to be active. This means the main `ListView` must be active, rather than the `Resources ListView`, because the main `ListView` uses an `EventSequenceContext`, but the `Resources ListView` uses `EventContext`.

The `PropertyView` class provides much of the capability of the property editors, as noted in [Property Editor Containers](#). When a search result is selected in `DomNodeSearchResultsListView`, this selection change event results in the following code being executed in `PropertyView`:

```
ISubSelectionContext selectionContext = m_editingContext.As<ISubSelectionContext>();
ISelectionContext subSelectionContext = (selectionContext != null) ? selectionContext.SubSelectionContext
: null;
if (subSelectionContext != null)
    subSelectionContext.SelectionChanged += subSelectionContext_SelectionChanged;
```

As noted previously, an `EventSequenceContext` in `m_editingContext` is the currently active context, and it implements `ISubSelectionContext`. This means that neither `selectionContext` nor `subSelectionContext` is `null`, so the (sub)selection event is subscribed to, with the handler `subSelectionContext_SelectionChanged`. This subselection change event is raised by the `selection_Changed` method in `GenericSelectionContext` when the selection changes in the `DomNodeSearchResultsListView`. For details on `GenericSelectionContext`, see [ISubSelectionContext Interface](#) and [GenericSelectionContext Class](#).

The handler `subSelectionContext_SelectionChanged` invalidates and refreshes the editing controls. It also calls `PropertyView.OnSubSelectionChanged()`, which selects the property that is in the subselection, that is, the property that was found in the search. This results in the attribute standing out in the property editor, as seen in the previous illustration. In the `Property Editor` window, the name of the attribute ("Name") is highlighted; in the `Grid Property Editor` window, the attribute's value is marked bold.

Topics in this section

Links on this page to other topics

Simple DOM No XML Editor Programming Discussion

The [ATF Simple DOM No XML Editor Sample](#) is almost exactly the same as the [ATF Simple DOM Editor Sample](#), except for a few things:

- It doesn't offer `DomNode` searching.
- It doesn't use an XML Schema for its data model.
- It doesn't persist application data in XML.

Other than this, the samples have exactly the same kind of data: sequences of events with the same attributes, including resources. When running, the applications look alike.

The key difference is not using XML, so this sample demonstrates two things different from Simple DOM Editor:

- Defining data types without an XML schema.
- Saving and reading document data from a non-XML file.

Note that these are quite different things: using an XML Schema does not require saving application data to XML, nor vice versa. Using an XML Schema does make it easier to persist data in XML, however.

Topics

- Programming Overview
- Defining Data Model Without an XML Schema
 - Type Definitions
 - Defining DOM Adapters
 - Adding Type Metadata Information
- Persisting Data
 - `EventSequenceDocument` Class `Read()` and `Write()` Methods
 - `Editor` Class `Open()` and `Save()` Methods

It's also worth noting that even though this sample does not use an XML Schema, it still uses the ATF DOM: DOM adapters, for instance. The ATF DOM does not require using an XML Schema.

Other than these differences, both samples' source is very similar, some files being nearly identical. To explore capabilities like using documents, DOM adapters, contexts, a palette, or editing data — common to these two samples — see [Simple DOM Editor Programming Discussion](#).

Programming Overview

Instead of an XML Schema, this sample's `DomTypes` class defines its types. Its data model is exactly the same as [ATF Simple DOM Editor Sample](#), so it defines the same data types using its own metadata classes, and `DomTypes` somewhat resembles the `Schema` class in Simple DOM Editor. Simple DOM No XML Editor uses the same DOM adapters as the Simple DOM Editor sample, and they are defined in almost the same way in `DomTypes`'s constructor. The `DomTypes` constructor also calls `NamedMetadata.SetTag()` to add information to the type metadata objects for palette types and property descriptors.

Simple DOM No XML Editor uses a simple format to store application data in a text file. Its `EventSequenceDocument` class contains methods to read and write these text files, which parse and format strings. The `Editor` class reads and saves event sequence documents using these read and write methods.

Defining Data Model Without an XML Schema

This sample defines a data model without an XML Schema, doing all the work that other samples' schema files, and `Schema` and `SchemaLoader` classes do.

Type Definitions

If you use an XML Schema, you define your data model's types in the XML Schema Definition (XSD) language. You can then use the ATF utility `DomGen` to create a `Schema` class that defines metadata classes for all the types. Without this facility, you have to create all these type definitions some other way.

Although this sample does not use an XML Schema, it does use the ATF DOM and defines metadata classes for all its types and attributes — very similarly to how it is done in the `Schema` class in the Simple DOM Editor sample. And both applications deal with exactly the same event sequence data, so their data model is the same, even though it's realized differently.

Consider how to define the "Event" type. It has the attributes "Name", "Time", and "Duration", and can have any number of "Resource" type children.

Simple DOM No XML Editor creates a `DomTypes` class for all its type definitions. Here's its definition of the "Event" type, starting with a comment containing the XML Schema definition of the type from the Simple DOM Editor sample's XML Schema:

```
//Schema equivalent:  
//<!--Event, with name, start time, duration and a list of resources-->  
//<xss:complexType name = "eventType">  
//  <xss:sequence>  
//    <xss:element name="resource" type="resourceType" maxOccurs="unbounded"/>  
//  </xss:sequence>  
//  <xss:attribute name="name" type="xs:string"/>  
//  <xss:attribute name="time" type="xs:integer"/>  
//  <xss:attribute name="duration" type="xs:integer"/>  
//</xss:complexType>  
/// <summary>  
/// Event type</summary>  
public static class eventType  
{  
    static eventType()  
    {  
        Type.Define(nameAttribute);  
        Type.Define(timeAttribute);  
        Type.Define(durationAttribute);  
        Type.Define(resourceChild);  
        resourceChild.AddRule(new ChildCountRule(1, int.MaxValue));  
    }  
  
    public readonly static DomNodeType Type = new DomNodeType("eventType");  
    public readonly static AttributeInfo nameAttribute =  
        new AttributeInfo("name", new AttributeType(AttributeTypes.String.ToString(), typeof(string)));  
    public readonly static AttributeInfo timeAttribute =  
        new AttributeInfo("time", new AttributeType(AttributeTypes.Int32.ToString(), typeof(int)));  
    public readonly static AttributeInfo durationAttribute =  
        new AttributeInfo("duration", new AttributeType(AttributeTypes.Int32.ToString(), typeof(int)));  
    public readonly static ChildInfo resourceChild = new ChildInfo("resource", Type, true);  
}
```

The initial calls to `DomNodeType.Define()`, like `Type.Define(nameAttribute)`, create attributes for the type, which the XML Schema definition also does.

The attribute metadata class members, such as `nameAttribute`, are explicitly constructed:

```
public readonly static AttributeInfo nameAttribute =  
    new AttributeInfo("name", new AttributeType(AttributeTypes.String.ToString(), typeof(string))));
```

The constructor for `AttributeInfo` here is

```
public AttributeInfo(string name, AttributeType type)
```

and the constructor for `AttributeType` used there is

```
public AttributeType(string name, Type type)
```

`AttributeTypes` is an enumeration for an attribute's primitive types.

This assignment for `nameAttribute` above spells out that the "Event" type's "name" attribute is a `string`, just as the XML Schema definition specified.

Similarly, the "Resource" type children of this type are specified in the `ChildInfo` constructor:

```
public readonly static ChildInfo resourceChild = new ChildInfo("resource", Type, true);
```

As the constructor for `ChildInfo` indicates, the event's resources are in a list of `DomNodes`:

```
public ChildInfo(string name, DomNodeType type, bool isList)
```

Here's the corresponding class for the "Event" type from the Simple DOM Editor sample's Schema file:

```
public static class eventType
{
    public static DomNodeType Type;
    public static AttributeInfo nameAttribute;
    public static AttributeInfo timeAttribute;
    public static AttributeInfo durationAttribute;
    public static ChildInfo resourceChild;
}
```

The `eventType` classes in the two samples have exactly the same fields of the same metadata types: both have a `Type` member that's a `DomNodeType`, and so forth.

Here's how the class for events is initialized in `Schema.Initialize()` in Simple DOM Editor:

```
eventType.Type = typeCollection.GetNodeType("eventType");
eventType.nameAttribute = eventType.Type.GetAttributeInfo("name");
eventType.timeAttribute = eventType.Type.GetAttributeInfo("time");
eventType.durationAttribute = eventType.Type.GetAttributeInfo("duration");
eventType.resourceChild = eventType.Type.GetChildInfo("resource");
```

The fields also end up being initialized to the same values. For instance, this line in `DomTyes`

```
public readonly static DomNodeType Type = new DomNodeType("eventType");
```

produces the same result as this one in `Schema`:

```
eventType.Type = typeCollection.GetNodeType("eventType");
```

Restrictions on types must also be handled. For example, the `eventType` definition in `DomTyes` above added a rule to the `resourceChild`'s `ChildInfo`:

```
resourceChild.AddRule(new ChildCountRule(1, int.MaxValue));
```

This rule's constructor specifies the minimum and maximum child count:

```
public ChildCountRule(int min, int max)
```

Defining DOM Adapters

Simple DOM No XML Editor uses the same DOM adapters as the Simple DOM Editor sample, and they are defined in almost the same way. In Simple DOM Editor, DOM adapters are defined in `SchemaLoader`; in Simple DOM No XML Editor, they are defined in `DomTypes`'s constructor:

```

static DomTypes()
{
    // register extensions
    eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceDocument>());
    eventSequenceType.Type.Define(new ExtensionInfo<EventSequenceContext>());
    eventSequenceType.Type.Define(new ExtensionInfo<MultipleHistoryContext>());
    eventSequenceType.Type.Define(new ExtensionInfo<EventSequence>());
    eventSequenceType.Type.Define(new ExtensionInfo<ReferenceValidator>());
    eventSequenceType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
    eventSequenceType.Type.Define(new ExtensionInfo<DomNodeQueryable>());

    eventType.Type.Define(new ExtensionInfo<Event>());
    eventType.Type.Define(new ExtensionInfo<EventContext>());

    resourceType.Type.Define(new ExtensionInfo<Resource>());

    // Enable metadata driven property editing for events and resources
    AdapterCreator<CustomTypeDescriptorNodeAdapter> creator =
        new AdapterCreator<CustomTypeDescriptorNodeAdapter>();
    eventType.Type.AddAdapterCreator(creator);
    resourceType.Type.AddAdapterCreator(creator);
}

```

The only difference between these definitions and the ones in `SchemaLoader` are the type names, since these types are defined in `DomTypes` rather than `Schema`.

An `AdapterCreator` is used here just as in Simple DOM Editor for the same reason: to make the property editors conform to the data in the property descriptors, defined next.

Adding Type Metadata Information

The `DomTypes` constructor carries out another task that the `SchemaLoader` in Simple DOM Editor did: calling `NamedMetadata.SetTag()` to add information to the type metadata objects. This information is used later for palette types and property descriptors.

This sample adds the same kind of information that Simple DOM Editor did. First, for palette types:

```

eventType.Type.SetTag(
    new NodeTypePaletteItem(
        eventType.Type,
        Localizer.Localize("Event"),
        Localizer.Localize("Event in a sequence"),
        Resources.EventImage));

```

The other use for the tag information is for property descriptors. These are used by property editors to determine which attributes, that is, properties, appear in the property editors for each type.

```

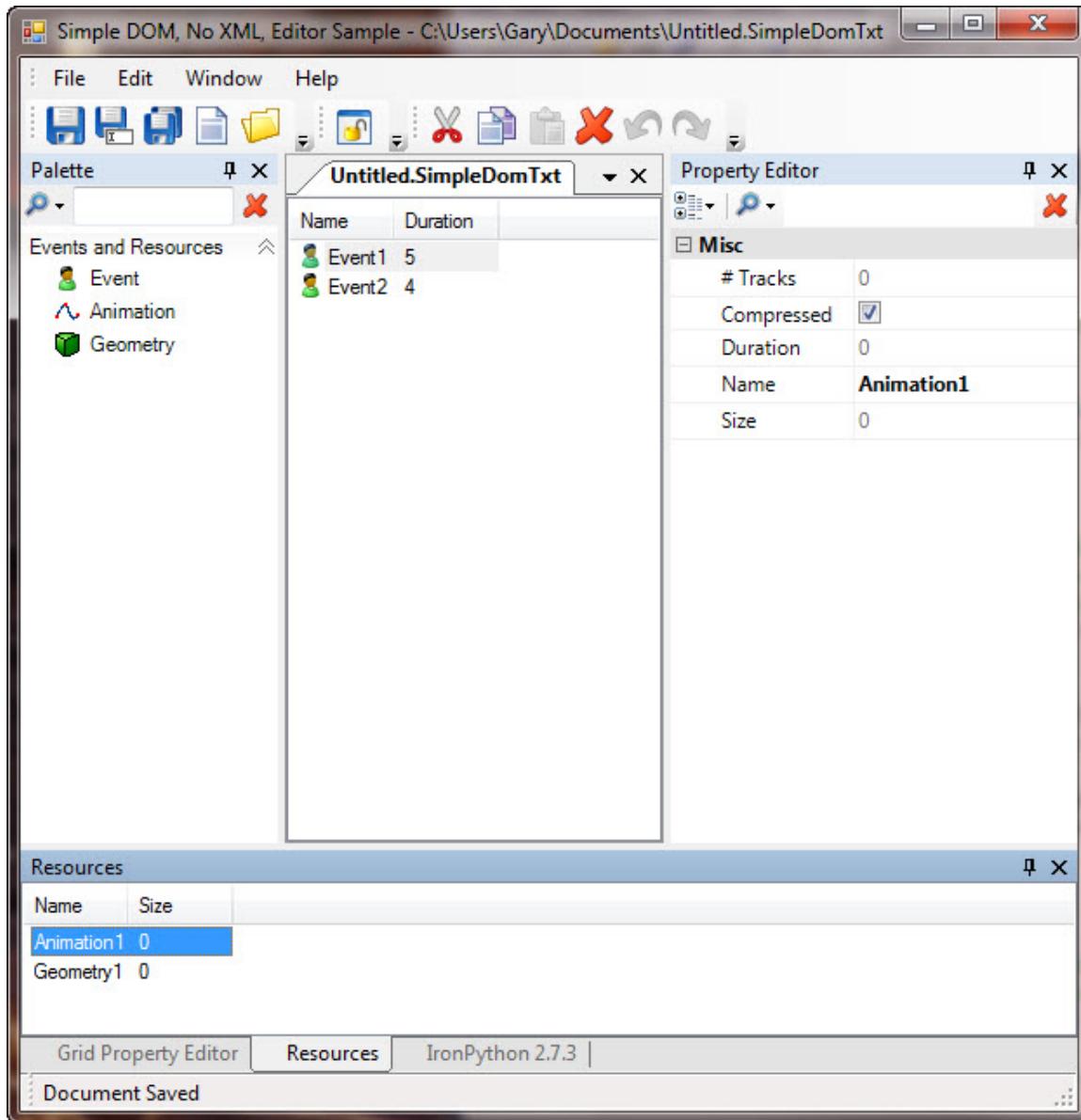
eventType.Type.SetTag(
    new PropertyDescriptorCollection(
        new Sce.Atf.Dom.PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Name"),
                eventType.nameAttribute,
                null,
                Localizer.Localize("Event name"),
                false),
            ...
        }
    )
)

```

Again, these statements are almost identical to the equivalent ones in `SchemaLoader`; only the type names differ.

Persisting Data

Simple DOM No XML Editor uses a very simple format to store application data in a text file with the extension `.SimpleDomTxt`. Consider the following data, with two events, as displayed in the application:



This is the corresponding data in the saved file:

```

eventSequence
event name="Event1", time=5, duration=5
resource type="animationResourceType", name="Animation1", compressed=true
resource type="geometryResourceType", name="Geometry1"
event name="Event2", time=7, duration=4
resource type="animationResourceType", name="Animation"
resource type="geometryResourceType", name="Geometry"

```

The file's first line shows that it's a file with event sequences. Each event is listed on one line with its name and any other attributes, followed by its resources and their attributes, one line per resource.

EventSequenceDocument Class Read() and Write() Methods

The `EventSequenceDocument` class contains methods to read and write `.SimpleDomTxt` files:

- `Read()`: Read the `.SimpleDomTxt` file from a Stream, creating an `EventSequenceDocument`, which is the adapted root of a `DomNode` tree.
- `Write()`: Write an `EventSequenceDocument` to a Stream.

These methods perform a similar function to the `DomXmlReader.Read()` and `DomXmlWriter.Write()` methods, which read and write application data from and to a `DomNode` tree. The `EventSequenceDocument` read and write methods mainly handle text, converting strings back and forth that embody the information in the tree's `DomNodes`. For instance, here's `Write()`:

```

public static void Write(EventSequenceDocument document, Stream stream)
{
    using (StreamWriter writer = new StreamWriter(stream))
    {
        writer.WriteLine("eventSequence");

        DomNode root = document.DomNode;
        foreach (DomNode eventNode in root.GetChildren(DomTypes.eventSequenceType.eventChild))
            WriteEvent(eventNode, writer);
    }
}

```

The method gets the root `DomNode` from the document and iterates through all its children. It writes all the event data for each child event with `WriteEvent()`:

```

private static void WriteEvent(DomNode eventNode, StreamWriter writer)
{
    StringBuilder lineBuilder = new StringBuilder(
        "event name=\"" + eventNode.GetAttribute(DomTypes.eventType.nameAttribute) + "\"");
    WriteAttribute(eventNode, "time", lineBuilder);
    WriteAttribute(eventNode, "duration", lineBuilder);
    writer.WriteLine(lineBuilder.ToString());

    foreach (DomNode resourceNode in eventNode.GetChildren(DomTypes.eventType.resourceChild))
        WriteResource(resourceNode, writer);
}

```

`WriteEvent()` uses a `StringBuilder` to format text for each event, writing a line for the event itself plus all its attributes that it obtains with `DomNode.GetAttribute()`. After that, it gets all the child attributes for the resources and writes a line for each one with `WriteResource()`. `WriteResource()` operates similarly to `WriteEvent()`, obtaining all the "Resource" type attributes and formatting them for output.

The `EventSequenceDocument.Read()` method functions similarly, reading the `.SimpleDomTxt` file, parsing its data, and creating `DomNodes` with the right attributes.

Editor Class Open() and Save() Methods

The `Editor` class for both the Simple DOM Editor and Simple DOM No XML Editor samples create and save event sequence documents in practically the same way.

The `Open()` method of Simple DOM Editor reads application data from a stream this way:

```

DomXmlReader reader = new DomXmlReader(m_schemaLoader);
node = reader.Read(stream, uri);

```

and Simple DOM No XML Editor does this:

```

node = EventSequenceDocument.Read(stream).DomNode;

```

Similarly, for saving a document in the `Save()` method, here's Simple DOM Editor's way:

```

DomXmlWriter writer = new DomXmlWriter(m_schemaLoader.TypeCollection);
EventSequenceDocument eventSequenceDocument = (EventSequenceDocument)document;
writer.Write(eventSequenceDocument.DomNode, stream, uri);

```

and Simple DOM No XML Editor:

```

EventSequenceDocument eventSequenceDocument = (EventSequenceDocument)document;
EventSequenceDocument.Write(eventSequenceDocument, stream);

```

[Links on this page to other topics](#)

[ATF Simple DOM Editor Sample](#), [ATF Simple DOM No XML Editor Sample](#), [Simple DOM Editor Programming Discussion](#)

State Chart Editor Programming Discussion

The [ATF State Chart Editor Sample](#) shows another example of using the ATF graph framework to create a graph editor for a statechart, which is a way of graphically presenting a state machine. It bears a strong resemblance to the [ATF FSM Editor Sample](#) and somewhat to [ATF Circuit Editor Sample](#), which are also graph editors based on ATF graph facilities.

ATF offers support for statecharts in its graph classes. For more details, see [Statechart Graph Support](#).

Statechart concepts are described in the paper [Statecharts: A Visual Formalism for Complex Systems](#) by David Harel. For information on a statechart oriented state machine tool based on ATF, see the [StateMachine Home Page](#).

Programming Overview

State Chart Editor is, in several ways, a little more complex version of the [ATF FSM Editor Sample](#), and the two samples have much in common. You can learn about that sample's internals in [FSM Editor Programming Discussion](#); this topic often references sections in this discussion.

State Chart Editor uses the ATF graph interfaces and classes, with its statechart version of the graph interface `IGraph`. For information about graph support for statecharts, see [Statechart Graph Support](#).

The data model takes into account the containment hierarchy: an ordinary state can contain other states and transitions. A statechart can have pseudo-states, such as the start state, as well as a normal state, and these all extend a base state type.

Like `Fsm` Editor, most of State Chart Editor's classes are DOM adapters, one for each type of object a statechart can contain, for instance. Other DOM adapters handle documents, contexts, and statechart validation.

Contents

- [Programming Overview](#)
- [Statecharts and Graphs](#)
- [State Chart Editor Data Model](#)
- [State Chart Editor DOM Adapters](#)
 - [StatechartDocumentType DOM Adapters](#)
 - [State Chart Editor Types DOM Adapters](#)
- [State Chart Editor Documents](#)
 - [Document Class](#)
 - [Editor Component](#)
 - [PrintableDocument Class](#)
- [State Chart Editor Contexts](#)
 - [EditingContext Class](#)
 - [PrototypingContext Class](#)
 - [ViewingContext Class](#)
- [Validating StateCharts](#)
 - [BoundsValidator Class](#)
 - [StatechartValidator Class](#)
 - [LockingValidator Class](#)

The `Editor` component is the document client and control host client for the control that holds the statechart, a `D2dAdaptableControl`. These clients are implemented very similarly to those in the `Editor` class of the `Fsm` Editor. You can also print statecharts with the `PrintableDocument` class.

The contexts in State Chart Editor also strongly resemble those of `Fsm` Editor. The `EditingContext` class implements interfaces similarly to the corresponding class in `Fsm` Editor; however, it also implements the `IGraph` interface containing properties listing statechart objects.

This sample's validator classes check that parent states' bounds are adjusted when child states change, that only legitimate transitions are added, and that locked states can't be edited.

Statecharts and Graphs

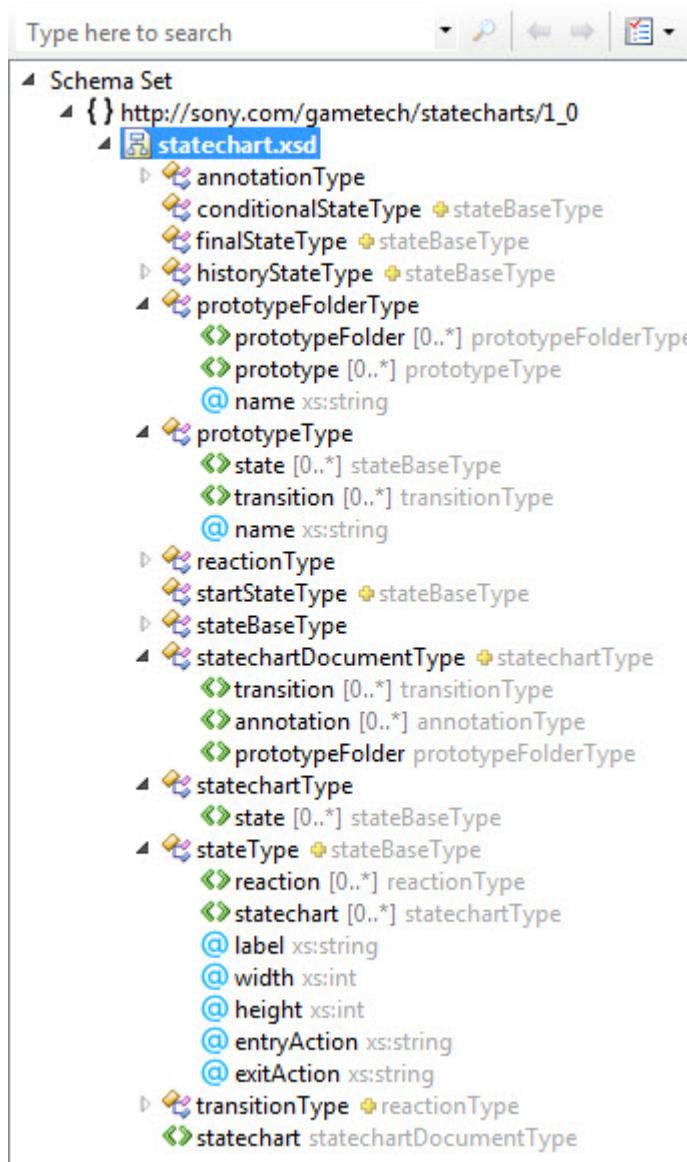
Statecharts specialize the ATF graph model embodied by the `IGraph` interface: `IGraph<IState, IGraphEdge<IState, BoundaryRoute>, BoundaryRoute>`. For a description of this specialized interface, see [Statechart Graph Support](#).

State Chart Editor Data Model

State Chart Editor's data model is defined in the XML Schema file `Statechart.xsd`. Because both are state machine editors, State Chart Editor's data model resembles that of the [ATF FSM Editor Sample](#). Both have state, transition, annotation, prototype, and prototype folder types. Unlike [ATF FSM Editor Sample](#), State Chart Editor has inheritance in its types. Here are the types and types based on them:

- `annotationType`: annotation on a statechart.
- `prototypeFolderType`: prototype folder containing prototypes and other prototype folders.
- `prototypeType`: prototype containing states and transitions that can be reused.
- `reactionType`: event/guard/action triple; a guard is a condition for a state transition.
 - `transitionType`: transition connecting a source state to a destination state.
- `state BaseType`: base state.
 - `conditionalStateType`: conditional pseudo-state.
 - `finalStateType`: final pseudo-state.
 - `historyStateType`: history pseudo-state.
 - `startStateType`: start pseudo-state.
 - `stateType`: normal state (not a pseudo-state) that can contain statecharts.
- `statechartType`: statechart that contains states.
 - `statechartDocumentType`: root statechart, containing all transitions, annotations, and prototypes. It also contains states by virtue of being based on `statechartType`.

The most used parent type is "state BaseType", on which all the other state types are based. This figure from the Visual Studio XML Schema Explorer of the schema file shows types' base types and also has parent types' nodes opened to show their child types:



This differs from the Fsm Editor in its hierarchy. The "stateType" type can contain "reactionType" and "statechartType" types. The "statechartType" type contains only "state BaseType", representing all the various state types, including "stateType" the only non-pseudo-state type. This means that a state can contain states (all the different ones) and transitions, since a transition is a "reactionType". The type

"statechartDocumentType", based on "statechartType", is the root type and can contain "transitionType", "annotationType", and "prototypeFolderType", which allows it to contain everything. Thus, the "stateType" type does not need to contain annotations or prototypes, which simplifies it.

State Chart Editor uses the `GenSchemaDef.bat` command file for DomGen to create the `Schema` class (containing type metadata classes) from `Statechart.xsd`.

State Chart Editor has a `SchemaLoader` class, derived from `XmlSchemaTypeLoader` as usual. Its overridden `OnSchemaSetLoaded()` also performs the typical tasks of adding data to the `Schema` class's metadata class objects:

- Defining DOM adapters on types. For details, see [State Chart Editor DOM Adapters](#).
- Adding `NodeTypePaletteItem` objects for palette items. This implementation follows the pattern of palettes in other samples, such as in [ATF Simple DOM Editor Sample](#), so it is not discussed here. For information on creating palettes, see [Using a Palette](#).
- Defining property descriptors for type attributes. Again, these descriptors are defined very similarly to ones in other samples. For an example in [ATF Simple DOM Editor Sample](#), see [Create Property Descriptors](#).

State Chart Editor DOM Adapters

Like other graph editor samples, most of the classes are DOM adapters. There is a DOM adapter for every type in the data model, for instance; for details, see [State Chart Editor Types DOM Adapters](#).

StatechartDocumentType DOM Adapters

Most of the DOM adapters are defined for the "statechartDocumentType" type, which represents the whole statechart. The root element of the `DomNode` tree is of "statechartDocumentType". These adapters fall in a few categories:

- Document handling. For more information, see [State Chart Editor Documents](#).
- Context handling. For details, see [State Chart Editor Contexts](#).
- Validation. For details, see [Validating StateCharts](#).

State Chart Editor Types DOM Adapters

Each of the types in the data model, such as "stateType" and "transitionType", has a DOM adapter defined for it. Here are their definitions in `SchemaLoader.OnSchemaSetLoaded()`:

```
// register the statechart model interfaces
Schema.prototypeFolderType.Type.Define(new ExtensionInfo<PrototypeFolder>());
Schema.prototypeType.Type.Define(new ExtensionInfo<Prototype>());
Schema.annotationType.Type.Define(new ExtensionInfo<Annotation>());
Schema.statechartType.Type.Define(new ExtensionInfo<Statechart>());
Schema.stateType.Type.Define(new ExtensionInfo<State>());
Schema.conditionalStateType.Type.Define(new ExtensionInfo<ConditionalState>());
Schema.finalStateType.Type.Define(new ExtensionInfo<FinalState>());
Schema.historyStateType.Type.Define(new ExtensionInfo<HistoryState>());
Schema.reactionType.Type.Define(new ExtensionInfo<Reaction>());
Schema.startStateType.Type.Define(new ExtensionInfo<StartState>());
Schema.transitionType.Type.Define(new ExtensionInfo<Transition>());
```

The following table describes the DOM adapter for each type. Most of these DOM adapters are very simple. They have properties that get and set type attribute values, using the `DomNode.GetAttribute()` and `DomNode.SetAttribute()` methods on the attribute metadata class members, defined in the `Schema` class. Some of the DOM adapters use the `DomNodeAdapter.GetChildList()` to get a list of attributes.

Type	DOM Adapter	Derives from, Implements	Properties, Methods	Notes
annotationType	Annotation	DomNodeAdapter, IAnnotation	Bounds, Location, Text, SetTextSize()	Comment on the canvas. Identical to Annotation in ATF FSM Editor Sample . For more details, see Annotation DOM Adapter in FSM Editor Programming Discussion .
conditionalStateType	ConditionalState	StateBase	Type	A conditional pseudostate, which indicates a state with conditions to reduce the number of transitions.

finalStateType	FinalState	StateBase	Type	The final pseudostate, which is the last state that a state machine can be in.
historyStateType	HistoryState	StateBase	Type	A history pseudostate, which may be shallow or deep. History determines which states become active in a destination state's child state machines when the transition is taken to this destination. For a discussion of how history works in the StateMachine application built from ATF, see State Machine History .
prototypeFolderType	PrototypeFolder	DomNodeAdapter	Folders, Name, Prototypes	Folder of prototypes. Identical to PrototypeFolder in ATF FSM Editor Sample . For details, see Prototype and PrototypeFolder DOM Adapters in FSM Editor Programming Discussion .
prototypeType	Prototype	DomNodeAdapter	Name, States, Transitions	Prototype containing states and transitions that can be reused. Almost identical to Prototype in ATF FSM Editor Sample . For details, see Prototype and PrototypeFolder DOM Adapters in FSM Editor Programming Discussion .
reactionType	Reaction	DomNodeAdapter	Action, Event, Guard, ToString()	Adapts DomNodes to a reactions, which are contained in states. For more information, see Reaction DOM Adapter .
startStateType	StartState	StateBase	Type	The start pseudostate, which points to the first state that is active when the state machine starts running.
statechartType	Statechart	DomNodeAdapter	AllStates, Bounds, Parent, States	Statechart that contains states. The AllStates property gets the states in the statechart and all sub-statecharts.
stateType	State	StateBase. IComplexState<StateBase, Transition>	EntryAction, ExitAction, IsPseudoState, IsSimple, Name, Reactions, Size, Statecharts, SubStates, Type	Regular state machine state. For details, see State DOM Adapter . This adapter is not very similar to State in ATF FSM Editor Sample .

transitionType	Transition	Reaction, IGraphEdge<StateBase, BoundaryRoute>	FromPosition, FromState, ToPosition, ToState	Transition from a state to a state. For details, see Transition DOM Adapter . Also, Transition is similar to Transition in ATF FSM Editor Sample ; for details, see Transition DOM Adapter in FSM Editor Programming Discussion .
N/A	StateBase	DomNodeAdapter, IState	Bounds, Indicators, IsPseudoState, Locked, Parent, Position, Size, Type	Base for other state DOM adapters, such as State and ConditionalState. For more information, see StateBase DOM Adapter .

State DOM Adapter

Much of this adapter is devoted to the task of many adapters: setting up type attribute values as gettable and/or settable properties.

State also implements `IComplexState<StateBase, Transition>`, the interface for states in statechart diagrams that are non-pseudo-states. State is the only state in State Chart Editor that is not a pseudo-state. `IComplexState`'s only property is `Text`, which gets the state's interior text, displayed as a popup when the state is hovered over. This is a concatenation of text from `EntryAction`, `ExitAction`, and the `Reactions` associated with the state.

`IComplexState` implements `IHierarchicalGraphNode<StateBase, Transition, BoundaryRoute>`, and its property `SubNodes`, the sequence of nodes that are children of this hierarchical graph node. Because a non-pseudo-state can contain a state machine, this interface provides a way to access that information. `SubNodes` simply gets the property `SubStates`, an enumeration of the states inside the state, also known as substates.

Reaction DOM Adapter

A reaction tells how a state reacts to events. Its properties, all `string` values, are:

- `Event`: event the state sees.
- `Guard`: what triggers a transition from the state.
- `Action`: action associated with an event.

The `ToString()` method returns a `string` containing all these properties' values.

Note that State's `Reactions` property lists all the `Reaction` objects associated with the state.

Transition DOM Adapter

Transition derives from Reaction and adds properties:

- `FromState, ToState`: source and destination states of transition.
- `FromPosition, ToPosition`: the `BoundaryRoute` route position on the perimeter of the state at which this transition begins and ends. Its range is `[0..4]`, and it starts and ends at the top-left, going in a clockwise direction. The range from 0 to 1 is on the right side, from 1 to 2 on the bottom, and so forth.

Transition also implements `IGraphEdge<StateBase, BoundaryRoute>`, which implements `IGraphEdge<StateBase>`. The first interface provides the properties `FromRoute` and `ToRoute` that get the `BoundaryRoute` at each end of the transition; these are the same as `FromPosition` and `ToPosition`. For more details, see [BoundaryRoute Class](#). The second interface provides `FromNode`, `ToNode`, and `Label`: the from and to State and the label on the state displayed to the user.

StateBase DOM Adapter

`StateBase` is the base class for all state type DOM adapters and provides their common properties:

- `Bounds`: Get or set the state's bounding rectangle.
- `IsPseudoState`: Get whether this is a pseudo-state.
- `Locked`: Get or set the locked state, used in locking states. For further discussion, see [LockingValidator Class](#).
- `Parent`: Get or set the state's parent Statechart.
- `Position`: Get or set the state position, which is at the state's upper-left corner.
- `Size`: Get or set the state size.

`StateBase` also implements `IState`, the interface for states in state-transition diagrams, and `IState` implements `IGraphNode`. For more

information, see [IState Interface](#).

State Chart Editor Documents

State Chart Editor implements both document and document client interfaces. For a general discussion of documents, see [Implementing a Document and Its Client](#).

Document Class

Document, a DOM adapter defined for the "statechartDocumentType", derives from DomDocument that implements IDocument. Document is very similar to the Document class in [ATF FSM Editor Sample](#). It implements the same methods in an almost identical way and is very simple.

Where it differs is that State Chart Editor's Document implements IAnnotatedDiagram, for diagrams that contain annotations. Its Annotations property gets the list of annotations on the statechart from the root DomNode, which has the type "statechartDocumentType", because this type contains the type "annotationType":

```
public IList<IAnnotation> Annotations
{
    get { return GetChildList<IAnnotation>(Schema.statechartDocumentType.annotationChild); }
}
```

Editor Component

The Editor component is the document client and control host client for the control that holds the statechart, and so it implements both IDocumentClient and IControlHostClient:

```
public class Editor : IDocumentClient, IControlHostClient, IInitializable
```

These clients also bear a strong resemblance to the clients in [ATF FSM Editor Sample](#). For a discussion of the Fsm Editor clients, see [Editor Component](#).

Graph and Other Control Adapters

The differences from Fsm Editor are primarily in the IDocumentClient.Open() method. Like Fsm Editor, this method creates a D2dAdaptableControl and constructs a set of control adapters for it. One simple addition is to draw a grid on the canvas with D2dGridAdapter:

```
var gridAdapter = new D2dGridAdapter();
```

Another difference is the graph adapter:

```
var statechartAdapter = new StatechartGraphAdapter(m_statechartRenderer, transformAdapter);
```

First, notice the renderer, constructed like this:

```
m_diagramTheme = new D2dDiagramTheme();
D2dGradientStop[] gradStops =
{
    new D2dGradientStop(Color.WhiteSmoke, 0),
    new D2dGradientStop(Color.LightGray, 1)
};
m_diagramTheme.FillGradientBrush = D2dFactory.CreateLinearGradientBrush(gradStops);
m_diagramTheme.FillBrush = D2dFactory.CreateSolidBrush(Color.WhiteSmoke);
m_statechartRenderer = new D2dStatechartRenderer<StateBase, Transition>(m_diagramTheme);
```

D2dStatechartRenderer, which derives from D2dGraphRenderer, is the class to handle rendering and hit testing on statechart graphs. For information about this renderer, see [D2dStatechartRenderer Class](#). D2dStatechartRenderer takes a D2dDiagramTheme parameter to specify its theme. Fsm Editor's renderer also used a D2dDiagramTheme; for more information about D2dDiagramTheme, see [Document Client](#). Here the D2dDiagramTheme is configured by setting its FillGradientBrush and FillBrush properties. The FillGradientBrush draws a gradient, using the specified D2dGradientStop array, between the colors specified in this array.

To return to the adapter, StatechartGraphAdapter is a custom class deriving from D2dGraphAdapter:

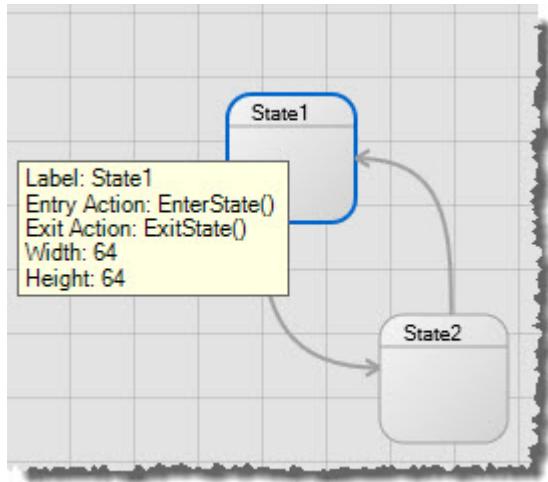
```
private class StatechartGraphAdapter : D2dGraphAdapter<StateBase, Transition, BoundaryRoute>
```

D2dGraphAdapter is a general control adapter to reference and render a graph diagram. All StatechartGraphAdapter does is to override its base class's OnRender() method to draw the statechart the way it wants, using the D2dStatechartRenderer. For information on D2dGraphAdapter, see [D2dGraphAdapter Class](#).

This sample, like Fsm Editor, uses the control adapters D2dGraphNodeEditAdapter and D2dGraphEdgeEditAdapter to allow dragging states and transitions. These control adapters use the renderer D2dStatechartRenderer and the graph adapter StatechartGraphAdapter defined here; these adapters' constructors take a D2dGraphRenderer and D2dGraphAdapter.

Hover Adapter

When the cursor moves over a state, this information window displays:



Both this sample and Fsm Editor use a HoverAdapter to display information when the cursor hovers over canvas items:

```
var hoverAdapter = new HoverAdapter();
hoverAdapter.HoverStarted += control_HoverStarted;
hoverAdapter.HoverStopped += control_HoverStopped;
```

Hovering sheds light on both the hover adapter and property descriptors.

The HoverStarted event handler simply gets the hover form when the cursor is over an object on the canvas:

```
private void control_HoverStarted(object sender, HoverEventArgs<object, object> e)
{
    m_hoverForm = GetHoverForm(e.Object);
}
```

The e.Object object is the object hovered over, which is passed to GetHoverForm().

HoverStopped's handler closes and disposes of this form:

```
private void control_HoverStopped(object sender, EventArgs e)
{
    if (m_hoverForm != null)
    {
        m_hoverForm.Close();
        m_hoverForm.Dispose();
    }
}
```

The GetHoverForm() method creates the form (based on information from the hovered over object) and displays it:

```

private HoverBase GetHoverForm(object hoverItem)
{
    HoverBase result = CreateHoverForm(hoverItem);
    if (result != null)
    {
        Point p = Control.mousePosition;
        result.Location = new Point(p.X - (result.Width + 12), p.Y + 12);
        result.ShowWithoutFocus();
    }
    return result;
}

```

Finally, `CreateHoverForm()` gets a property descriptor associated with the hovered over object, gets its attributes, and formats the text to show in the hover window:

```

// create hover form for primitive state or transition
private static HoverBase CreateHoverForm(object hoverTarget)
{
    // handle states and transitions
    StringBuilder sb = new StringBuilder();
    ICustomTypeDescriptor customTypeDescriptor = Adapters.As<ICustomTypeDescriptor>(hoverTarget);
    if (customTypeDescriptor != null)
    {
        // Get properties interface
        foreach (System.ComponentModel.PropertyDescriptor property in customTypeDescriptor.
GetProperties())
        {
            object value = property.GetValue(hoverTarget);
            if (value != null)
            {
                sb.Append(property.Name);
                sb.Append(": ");
                sb.Append(value.ToString());
                sb.Append("\n");
            }
        }
    }

    HoverBase result = null;
    if (sb.Length > 0) // remove trailing '\n'
    {
        sb.Length = sb.Length - 1;
        result = new HoverLabel(sb.ToString());
    }

    return result;
}

```

The parameter `hoverTarget` is the object hovered over, which is a `State` object of "stateType". `State` derives from `StateBase`, which derives from `DOMNodeAdapter`, which adapts a `DomNode`, so ultimately, this object is a `DomNode` representing a state.

To understand what the `CreateHoverForm()` method is doing, consider this line in `Program.cs`, which nearly all samples, including State chart Editor, have:

```

DOMNodeType.BaseOfAllTypes.AddAdapterCreator(new AdapterCreator<CustomTypeDescriptorNodeAdapter
>());

```

`AddAdapterCreator()` adds an adapter creator onto every type for `CustomTypeDescriptorNodeAdapter`, which implements `ICustomTypeDescriptor`. This means that an adapter can be found for `ICustomTypeDescriptor` for every type in State chart Editor. That is, every `DomNode` can be adapted to `ICustomTypeDescriptor`. This is useful in extracting attribute information from the `DomNode`.

`CreateHoverForm()` first tries to adapt the hovered over object to `ICustomTypeDescriptor`, that is, to determine if the object implements `ICustomTypeDescriptor` and to get an object on which this interface can be used:

```

ICustomTypeDescriptor customTypeDescriptor = Adapters.As<ICustomTypeDescriptor>(hoverTarget);

```

This works because an adapter for `CustomTypeDescriptorNodeAdapter` was added for every type of `DomNode`. So the variable

customTypeDescriptor is not null and the loop iterates on the property collection obtained by customTypeDescriptor.GetProperties(). The SchemaLoader.OnSchemaSetLoaded() method created this property descriptor the "stateType":

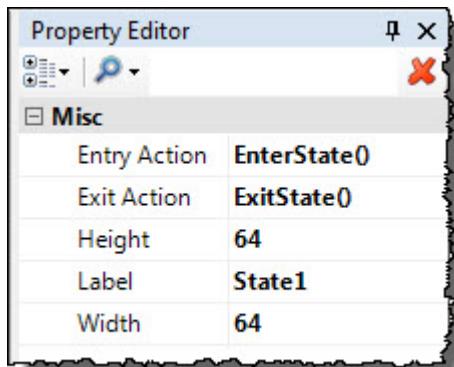
```
Schema.stateType.Type.SetTag(
    new PropertyDescriptorCollection(
        new PropertyDescriptor[] {
            new AttributePropertyDescriptor(
                Localizer.Localize("Label"),
                Schema.stateType.labelAttribute, // 'nameAttribute' is unique id, label is user
                visible name
                null,
                Localizer.Localize("State label"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Entry Action"),
                Schema.stateType.entryActionAttribute,
                null,
                Localizer.Localize("Action taken when state is entered"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Exit Action"),
                Schema.stateType.exitActionAttribute,
                null,
                Localizer.Localize("Action taken when state is exited"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Width"),
                Schema.stateType.widthAttribute,
                null,
                Localizer.Localize("Width of state"),
                false),
            new AttributePropertyDescriptor(
                Localizer.Localize("Height"),
                Schema.stateType.heightAttribute,
                null,
                Localizer.Localize("Height of state"),
                false),
        }));
});
```

The PropertyDescriptor collection for the state object thus includes each of the properties listed above. The value can then be obtained for each property:

```
object value = property.GetValue(hoverTarget);
```

The subsequent code uses a StringBuilder to build the hover text from the property's Name and its value text obtained from value.ToString(). The assembled string is then encapsulated in a HoverLabel object, which CreateHoverForm() returns. GetHoverForm() then displays this value in the hover window by invoking ShowWithoutFocus() on the HoverLabel.

When the state object is selected, these property values are shown in the property editor:



This is the same object that was hovered over in the previous figure, and the property values are the same as shown in the hover window, just as expected, because they come from the same source.

PrintableDocument Class

PrintableDocument provides what's needed to print statecharts. This class is almost the same as PrintableDocument in ATF FSM Editor Sample. For information on this Fsm Editor class, see [PrintableDocument Class](#).

State Chart Editor Contexts

State Chart Editor has the same contexts as Fsm Editor, and their functions and implementations have a lot in common.

EditingContext Class

Like the EditingContext in Fsm Editor, this one derives from Sce.Atf.Dom.EditingContext and implements many of the same interfaces:

```
public class EditingContext : Sce.Atf.Dom.EditingContext,
    IEnumerableContext,
    IObservableContext,
    INamingContext,
    ILockingContext,
    IInstancingContext,
    IGraph<StateBase, Transition, BoundaryRoute>,
    IEditableGraph<StateBase, Transition, BoundaryRoute>
```

The implementations of IEnumerableContext, IObservableContext, INamingContext, and IInstancingContext resemble their Fsm Editor counterparts. Some of the differences are due to the type hierarchy differences, discussed in [State Chart Editor Data Model](#).

State Chart Editor includes the StandardLockCommands component, which creates Lock and Unlock items under the Edit menu. This permits locking states so they can't be changed. Supporting this also requires implementing the ILockingContext, which EditingContext does. StateBase has a Locked property, which is handled by the methods in ILockingContext that test whether an item can be or is locked and also set the locked state. For more information on how locking works, see [LockingValidator Class](#).

Because this sample uses the ATF graph facilities, IGraph must be implemented somewhere. In this case, EditingContext implements IGraph<StateBase, Transition, BoundaryRoute>. For a description of this incarnation of the IGraph interface, see [Statechart Graph Support](#). Its implementation consists simply of creating the properties Nodes and Edges, to get lists of StateBase and Transition objects in the entire statechart.

Like Fsm Editor, State Chart Editor's EditingContext uses IEditableGraph<State, Transition, NumberedRoute> to make and break connections between states, that is, to add and remove transitions. The methods CanConnect, Connect, CanDisconnect, and Disconnect test whether transitions can be created or removed, and make and destroy transitions. Here's the most complicated method:

```
Transition IEditableGraph<StateBase, Transition, BoundaryRoute>.Connect(
    StateBase fromNode, BoundaryRoute fromRoute, StateBase toNode, BoundaryRoute toRoute,
    Transition existingEdge)
{
    DomNode domNode = new DomNode(Schema.transitionType.Type);
    Transition transition = domNode.As<Transition>();

    transition.FromState = fromNode;
    transition.FromPosition = fromRoute.Position;
    transition.ToState = toNode;
    transition.ToPosition = toRoute.Position;

    if (existingEdge != null)
    {
        Transition existingTransition = existingEdge as Transition;
        transition.Event = existingTransition.Event;
        transition.Guard = existingTransition.Guard;
        transition.Action = existingTransition.Action;
    }

    m_transitions.Add(transition);

    return transition;
}
```

In this sequence, a DomNode is created of the "transitionType" and adapted to Transition. Its FromState, FromPosition, ToState, and ToPosition properties are set from the method's parameters. If the transition already existed, it is adapted to Transition and the previous Event, Guard, and Action properties copied over. Finally, the new Transition is added to the transition list m_transitions, created in

the `OnNodeSet()` method for `EditingContext`:

```
m_transitions = new DomNodeListAdapter<Transition>(DomNode, Schema.statechartDocumentType.transitionChild
);
```

PrototypingContext Class

`PrototypingContext` is almost identical to `PrototypingContext` in the `Fsm Editor` and also to `Sce.Atf.Controls.Adaptable.Graphs.PrototypingContext`, differing only in some DOM adapter names. For a discussion of `Fsm Editor`'s `PrototypingContext`, see [PrototypingContext Class](#). For information on `Sce.Atf.Controls.Adaptable.Graphs.PrototypingContext`, see [PrototypingContext Class](#).

ViewingContext Class

Again, this context is almost identical to the `ViewingContext` in `Fsm Editor`. For information on that sample's context, see [ViewingContext Class](#).

Validating StateCharts

`SchemaLoader.OnSchemaSetLoaded()` defines several validating DOM adapters on the "statechartDocumentType" type:

```
Schema.statechartDocumentType.Type.Define(new ExtensionInfo<BoundsValidator>());
Schema.statechartDocumentType.Type.Define(new ExtensionInfo<StatechartValidator>());
...
Schema.statechartDocumentType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
Schema.statechartDocumentType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.statechartDocumentType.Type.Define(new ExtensionInfo<LockingValidator>());
```

All validating DOM adapters are defined on the root element type, so the entire tree of `DomNodes` is checked.

`UniqueIdValidator` and `ReferenceValidator` have pretty much the same aim as in [ATF FSM Editor Sample](#): to make sure that IDs for states are unique and all references to these IDs are valid. This XML Schema from `Statechart.xsd` shows that "stateBaseType" has an ID attribute ("xs:ID") and "transitionType" references this ID ("xs:IDREF"):

```
<!--
A state has a name (id), position, and can be referenced by transitions.
-->
<xss:complexType name="stateBaseType" abstract="true">
  <xss:attribute name="name" type="xs:ID" use="required" />
  <xss:attribute name="x" type="xs:int" use="required" />
  <xss:attribute name="y" type="xs:int" use="required" />
</xss:complexType>
...
<!--
A transition is a reaction, and connects the source state to the destination state.
The positions of the transition ends are determined by the fromPosition and toPosition
values, which are in the range [0..3].
-->
<xss:complexType name="transitionType">
  <xss:complexContent>
    <xss:extension base="reactionType">
      <xss:attribute name="fromState" type="xs:IDREF" use="required" />
      <xss:attribute name="fromPosition" type="xs:float" />
      <xss:attribute name="toState" type="xs:IDREF" use="required" />
      <xss:attribute name="toPosition" type="xs:float" />
    </xss:extension>
  </xss:complexContent>
</xss:complexType>
```

`UniqueIdValidator` guarantees that every `StateBase` element has a unique ID. `ReferenceValidator` makes sure that the references to these IDs are valid as the `DomNode` tree changes by adding and removing states and transitions. For more details on these validator's use in `Fsm Editor`, see [Validating State Machines](#).

BoundsValidator Class

When a child state is moved around inside its container state, the container resizes to accommodate its child state. `BoundsValidator`

derives from `Validator` and tracks changes to states and updates their bounds to be big enough to hold any child states.

More precisely, `BoundsValidator` monitors changes to states with various event handlers, such as `OnAttributeChanged()` and `OnChildInserted()`. These handlers are called as a result of `BoundsValidator` being derived from `Validator` and `Observer`, which track changes to the tree of `DomNodes`.

These event handlers may invalidate the layout by setting `m_layoutInvalid` true to force a redraw when validation is occurring:

```
protected override void OnAttributeChanged(object sender, AttributeEventArgs e)
{
    if (Validating)
        m_layoutInvalid = true;

    base.OnAttributeChanged(sender, e);
}
```

`Validating` is a property of `Validator` that is set true when a validation context changes.

At the end of the change, `OnEnding()` is called. If `m_layoutInvalid` is true, it calls the `Layout()` method for the entire statechart, which calls the `Layout()` method for each `StateBase` in the statechart. This latter `Layout()` method gets the union of its state's child states' bounds and resizes its state's bounds when necessary.

StatechartValidator Class

`StatechartValidator` tracks changes to the statechart and enforce constraints on states and transitions. In its `OnNodeSet()` method, it subscribes to the `ChildInserting` event, so it can check any newly added `DomNode`. The event handler checks for various constraint violations, such as trying to create a transition from a final state, raising an `InvalidOperationException` in such cases.

LockingValidator Class

`LockingValidator` is in the namespace `Sce. Atf. Dom` and not part of State Chart Editor, per se. It derives from `Validator` and tracks locked data in the tree of `DomNodes`. Its purpose is to prevent changing locked `StateBase` objects.

Recall that `EditingContext` implements `ILockingContext`, as described in `EditingContext` Class.

`LockingValidator`'s `OnNodeSet()` method is:

```
protected override void OnNodeSet()
{
    m_lockingContext = this.Cast<ILockingContext>(); // required ILockingContext

    // receive notification before attribute changes, to handle changes to lock state
    DomNode.AttributeChanging += OnAttributeChanging;

    base.OnNodeSet();
}
```

The first line gets an adapter for `LockingValidator` to the interface `ILockingContext`; `EditingContext` is obtained. Because this class is not part of State Chart Editor, it does not know which classes implement `ILockingContext`, so it uses adaptation to find a suitable object. For details on how this adaptation functions, see [Adapting to All Available Interfaces](#). Support for the `ILockingContext` interface provided by `m_lockingContext` is required in several methods to test whether a `DomNode` is locked, so it can't be edited.

`OnNodeSet()` subscribes to `AttributeChanging`, so it can monitor all `DomNode` attribute changes in the tree; `StatechartValidator` is defined on the root type, so the whole `DomNode` tree is checked.

When a change begins, `OnBeginning()` is called:

```
protected override void OnBeginning(object sender, EventArgs e)
{
    m_modified = new HashSet<DomNode>();
}
```

The field `m_modified` holds a set of all `DomNodes` modified during a validation event. When events occur, like adding or removing a `DomNode`, the `DomNode` is added to the set, as in `OnChildInserted()`:

```

protected override void OnChildInserted(object sender, ChildEventArgs e)
{
    if (Validating)
        m_modified.Add(e.Child);
}

```

Finally, `OnEnding()` is called when the change ends:

```

protected override void OnEnding(object sender, EventArgs e)
{
    try
    {
        HashSet<DomNode> knownUnlocked = new HashSet<DomNode>();
        List<DomNode> discoveredUnlocked = new List<DomNode>();
        foreach (DomNode modified in m_modified)
        {
            foreach (DomNode node in modified.Lineage)
            {
                if (knownUnlocked.Contains(node))
                    break;
                if (m_lockingContext.IsLocked(node))
                    throw new InvalidTransactionException("item is locked");
                discoveredUnlocked.Add(node);
            }
            foreach (DomNode node in discoveredUnlocked)
                knownUnlocked.Add(node);

            discoveredUnlocked.Clear();
        }
    }
    finally
    {
        m_modified = null;
    }
}

```

This method iterates through the `DomNode` set in `m_modified` and throws `InvalidTransactionException` if a locked `DomNode` was modified.

Topics in this section

Links on this page to other pages

[Adapting to All Available Interfaces](#), [ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [Authoring Tools Framework](#), [Circuit Graph Support](#), [FSM Editor Programming Discussion](#), [General Graph Support](#), [Implementing a Document and Its Client](#), [Simple DOM Editor Programming Discussion](#), [Statechart Graph Support](#)

Target Manager Programming Discussion

The [ATF Target Manager Sample](#) uses `TargetEnumerationService` and related components to manage targets. Targets are created by applications and are network endpoints, such as IP addresses, PS3 and PS4 DevKits, and Vita DevKits.

Target providers, such as `TcpIpTargetProvider`, discover and report targets of their specific type and these targets' parameters. Once found, you can edit and, in some cases, add and remove, targets, of that type with the provider. `TargetEnumerationService` enumerates all available target providers, combining the targets' information into a list view for displaying and editing.

Target Manager has very little code of its own, and relies on interfaces and classes in the `Sce.Atf.Applications.NetworkTargetServices` namespace for its operation. However, Target Manager does not use all of the interfaces and classes in this namespace. This namespace also supports creating and servicing targets, which Target Manager does not do. This article focuses on the items used by Target Manager in this namespace.

Several applications based on ATF create their own Target Managers similar to the Target Manager sample. These applications include the StateMachine and Scream tools.

Programming Overview

Target Manager simply imports target related components to build the application. These components reside in the `Sce.Atf.Applications.NetworkTargetServices` namespace, which also contains an interface and classes that serve building targets.

Targets are classified by the type of the communication protocol used to communicate with the target. There are two protocols: TCP for x86, PS3, and PS4 targets, and Deci4p for Vita DevKits, and these describe the two types.

Contents

- [Programming Overview](#)
- [Target Interfaces](#)
 - [ITargetConsumer Interface](#)
 - [ITargetProvider Interface](#)
- [Target Consumer Classes](#)
 - [TargetEnumerationService Component](#)
- [Target Provider Classes](#)
 - [TargetInfo Class](#)
 - [TcplpTargetProvider Component](#)
 - [Deci4pTargetProvider Component](#)
- [TargetCommands Component](#)
 - [TargetProviders Property](#)
 - [Command Creation](#)
 - [IContextMenuCommandProvider Implementation](#)
 - [Command Client](#)

The classes and interfaces used can be divided into consumers and providers. Consumers use information from the providers, which compile a list of targets of each type. `ITargetConsumer` handles updating the target list user interface and provides a target list for the consumer. `ITargetProvider` enumerates targets for the particular provider and adds and removes targets.

The consumer is `TargetEnumerationService`, implementing `ITargetConsumer`. This component creates a `ListView` control for the targets and their information, embodied in the `TargetInfo` class.

The providers are `TcpIpTargetProvider` and `Deci4pTargetProvider`, both implementing `ITargetProvider`. These manage the list of targets: editing, and for `TcpIpTargetProvider`, adding, and removing them. TCP targets are added by users; Deci4p are discovered by the provider. These classes inform the user when changes occur so the target display list can be updated.

`TargetCommands` creates the context menu commands on the `ListView` control and is their command client.

Target Interfaces

There is one interface for target consumers, another for providers.

ITargetConsumer Interface

`ITargetConsumer` is for classes that consume network target information, such as `TargetEnumerationService`, which implements this

interface. Such classes should provide a user interface for targets. It consists of the following:

- TargetsChanged(): Updates changed targets for a given provider.
- SelectedTargets: Gets or sets an enumeration of the selected targets in the consumer, if any, as an `IEnumerable<TargetInfo>`. The `TargetInfo` class describes target information. For details, see [TargetInfo Class](#).
- AllTargets: Gets all targets in the consumer as an `IEnumerable<TargetInfo>`

ITargetProvider Interface

`ITargetProvider` provides information about a particular kind of target available on the network, such as a TCP accessible device. Its methods and properties access the providers of this type:

- Name: Get the provider name.
- GetTargets(): Get an enumeration of the `TargetInfo` for all the targets.
- CanCreateNew: Get whether a new target can be created.
- CreateNew(): Create a new target, returning its `TargetInfo`.
- AddTarget(): Add a new target with the given `TargetInfo`.
- Remove(): Remove the target with the given `TargetInfo`.

Target Consumer Classes

TargetEnumerationService Component

The `TargetEnumerationService` component does the following:

- Imports the available target providers.
- Creates a `ListView` with target information.
- Acts as control host client for this `ListView` control.
- Implements `ITargetConsumer`.

Importing Target Providers

`TargetEnumerationService` imports all components implementing `ITargetProvider`, so that it can display their information:

```
[ImportMany]
private IEnumerable<ITargetProvider> m_targetProviders= null;
```

The field `m_targetProviders` is get or set in the `TargetProviders` property.

Providers include the `Deci4pTargetProvider` and `TcpIpTargetProvider` components. More such providers could be added. For the discussion of target providers, see [Target Provider Classes](#).

Creating ListView Control

The component's `IInitializable.Initialize()` method calls `SetUpTargetsView()` to create a targets display control, which can be used as a docked control or a stand alone dialog:

```

void IInitializable.Initialize()
{
    // force creation of the window handles on the GUI thread
    // see
    http://forums.msdn.microsoft.com/en-US/clr/thread/fa033425-0149-4b9a-9c8b-bcd2196d5471/
    var handle = MainForm.Handle;

    var control = SetUpTargetsView();

    if (!ShowAsDialog)
    {
        m_controlHostService.RegisterControl(
            control,
            new ControlInfo(
                "Targets".Localize(),
                "Controls for managing targets.".Localize(),
                StandardControlGroup.Bottom),
            this);
    }

    if (m_settingsService != null)
    {
        m_settingsService.RegisterSettings(this,
            new BoundPropertyDescriptor(this, () => PersistedUISettings, "UI Settings".Localize
(), null, null));
        m_settingsService.RegisterSettings(this,
            new BoundPropertyDescriptor(this, () => PersistedSelectedTargets, "Selected Targets".
Localize(), null, null));
    }
}

```

`SetUpTargetsView()` creates and configures a `DataBoundListView`, which extends `ListView` with data binding and cell editing functionality. If the control is not a dialog, it's registered with the Control Host Service. Settings in the control that persist between sessions are registered with the Settings Service. The properties `PersistedUISettings` and `PersistedSelectedTargets` get and set persistent values.

`TargetEnumerationService` also imports context menu providers for its context menu:

```

[ImportMany]
private IEnumerable<Lazy<IContextMenuCommandProvider>> m_contextMenuCommandProviders;

```

Menu items for the control are provided by the `TargetCommands` component, which implements `IContextMenuCommandProvider`. For details, see [TargetCommands Component](#).

The bulk of the code in `TargetEnumerationService` creates the `DataBoundListView` control and handles its events. For example, the `CellValidating` event is raised when one of the cells of the `DataBoundListView` changes; the `listView_CellValidating()` event handler validates the change.

Control Host Client

This client is very simple. The only method that does anything is `Activate()`, which brings focus to the `DataBoundListView` control so that selected targets are more visible.

ITargetConsumer Implementation

The `ITargetConsumer.TargetsChanged()` method updates the target view in the `DataBoundListView` control. All target providers call `TargetsChanged()` when a target is added, removed, or changed so the target display stays current.

The properties `SelectedTargets` and `AllTargets` get known targets as an `IEnumerable<TargetInfo>`. `SelectedTargets` can also set the target selection.

Target Provider Classes

Target providers manage targets that are using a given type of communications protocol. Deci4p is a proprietary protocol handling the Vita DevKit. Everything else, x86, PS3, and PS4, is handled by TCP.

This section discusses the target provider components and their auxiliary classes.

TargetInfo Class

`TargetInfo` encapsulates target information in its gettable and settable properties, which are all `string` values except for `Scope`:

- `Name`: Target name.
- `Platform`: Type of platform the target can run on, such as as "Ps3" or "Vita".
- `Endpoint`: Network endpoint in string format; an IP address for TCP.
- `Protocol`: Type of protocol the target can use. Currently, this is either "Tcp" or "Deci4p".
- `Scope`: `TargetScope` enumeration indicating how the target is persisted for applications that use Target Manager:
 - `PerApp`: Save target data for the current application. All users on this computer see this target when they run this application. Other applications that use Target Manager do not see this target. This is the default value for TCP targets.
 - `PerUser`: Save target data for the current user. Other applications that use the Target Manager can see this target for this user; other users won't see this target in any application.
 - `AllUsers`: Save the target data for all users and applications, so this target is visible for all users in any application that uses the Target Manager framework.

`TargetInfo` implements `INotifyPropertyChanged`, which contains one event, `PropertyChanged`. It has the method `OnPropertyChanged()` that simply raises this event.

`TargetInfo` has two ATF classes that derive from it, described in the next sections.

TcpIpTargetInfo Class

`TcpIpTargetInfo` describes TCP target information, and is used by `TcpIpTargetProvider`. `TcpIpTargetInfo` contains methods to verify that the IP address provided is valid. Its `Validate()` method checks this, as well as the other properties, and is called by `TargetEnumerationService` when any properties change in a TCP target listed.

`TcpIpTargetInfo` also has derived classes:

- `X86TargetInfo`: X86 target information.
- `Ps3TargetInfo`: PS3 target information.
- `Ps4TargetInfo`: PS4 target information.

These class's constructors set some default values, as for `Ps3TargetInfo`:

```
public Ps3TargetInfo()
    : base()
{
    Name = "Ps3Host";
    Platform = PlatformName;
    Endpoint = "10.89.0.0:1338";
}
```

Deci4pTargetInfo Class

`Deci4pTargetInfo` holds information about a Deci4p target and is used by `Deci4pTargetProvider` in its process of finding active targets of this type.

TcpIpTargetProvider Component

Target Manager imports the `TcpIpTargetProvider` component, which is one of the two providers implementing `ITargetProvider`.

TCP targets are not found; they are added by users. The list of targets is persisted using the Setting Service to save the list between sessions. This requires using the Setting Service to set the saved values when a target changes and to read the saved values when the application starts. The `PersistedTargets` property gets and sets the persisted targets list.

The `ITargetProvider` implementation manipulates the target list. For instance, `GetTargets()` retrieves objects from the targets list in `m_targets`:

```
public IEnumerable<TargetInfo> GetTargets(ITargetConsumer targetConsumer)
{
    foreach (var target in m_targets)
        yield return target;
}
```

`CreateNew()` returns a new `TcpIpTargetInfo`:

```

public virtual TargetInfo CreateNew()
{
    var newTarget = new TcpIpTargetInfo();
    return newTarget;
}

```

AddTarget() adds the given target to the list and calls TargetsChanged() to inform each target consumer so its lists can be updated:

```

public bool AddTarget(TargetInfo target)
{
    if (target is TcpIpTargetInfo && !m_targets.Contains(target))
    {
        m_targets.Add(target);
        foreach (var targetConsumer in TargetConsumers)
            targetConsumer.TargetsChanged(this, m_targets);
        return true;
    }
    return false;
}

```

TcpIpTargetProvider imports as many ITargetConsumer implementers as it can find to populate its TargetConsumers property, which is used in AddTarget():

```

[ImportMany]
protected IEnumerable<ITargetConsumer> TargetConsumers { get; set; }

```

Finally, TcpIpTargetProvider has its own derived classes to match up with TcpIpTargetInfo's derived classes:

- X86TargetProvider: X86 target provider.
- Ps3TargetProvider: PS3 target provider.
- Ps4TargetProvider: PS4 target provider.

These provider classes are simple. For example, Ps3TargetProvider has two gettable string properties and a CreateNew() method that constructs and returns a Ps3TargetInfo:

```

public class Ps3TargetProvider : TcpIpTargetProvider
{
    /// <summary>
    /// Gets the provider's user-readable name</summary>
    public override string Name { get { return "PS3 Target".Localize(); } }

    /// <summary>
    /// Gets the identifier of the provider</summary>
    /// <returns>A string that contains the identifier.</returns>
    public override string Id { get { return @"Sce.Atf.Ps3TcpIpTargetProvider"; } }

    /// <summary>
    /// Creates a new target</summary>
    /// <remarks>Creates and returns a TargetInfo, but does not add it to the watched list</remarks>
    /// <returns>TargetInfo for new target</returns>
    public override TargetInfo CreateNew()
    {
        var newTarget = new Ps3TargetInfo();
        return newTarget;
    }
}

```

Deci4pTargetProvider Component

Target Manager also imports the Deci4pTargetProvider component, one of the two providers implementing ITargetProvider. This supports only the Vita platform.

Unlike TCP targets, you can't create targets — Vita targets can only be discovered. Its IInitializable.Initialize() method starts a background task in another thread to attempt to find these targets with its FindTargets() method, adding them to a list as they are found.

This limitation makes `ITargetProvider`'s implementation rudimentary. `Name` always gets "Vita Target". `CanCreateNew` gets `false`. `CreateNew()` throws an exception. Both `AddTarget()` and `Remove()` do nothing but return `false`. `GetTargets()` does return the list of targets found.

Deci4pTargetProvider also imports as many `ITargetConsumer` implementers as it can find to update the target display of each consumer:

```
[ImportMany(typeof(ITargetConsumer))]
protected IEnumerable<ITargetConsumer> TargetConsumers { get; set; }
```

TargetCommands Component

TargetCommands adds the context menu commands for the targets list control, providing the command client for these commands, so it implements `ICommandClient`:

```
public class TargetCommands : ICommandClient, IContextMenuCommandProvider, IInitializable
```

TargetProviders Property

To create the right commands, TargetCommands needs to know which target providers are available. It does so by importing all exported `ITargetProvider` and accessing them with `TargetProviders`:

```
[ImportMany]
private IEnumerable<ITargetProvider> m_targetProviders = null;

/// <summary>
/// Gets or sets the target providers</summary>
public IEnumerable<ITargetProvider> TargetProviders
{
    get { return m_targetProviders; }
    set { m_targetProviders = value; }
}
```

Command Creation

TargetCommands's `IInitializable.Initialize()` creates the commands, based on available targets:

```

void IInitializable.Initialize()
{
    if (CommandService == null)
        return;

    if (Deci4pTargetProvider.SdkInstalled)
    {
        var cmdInfo = new CommandInfo(
            CommandTag.VitaNeighborhood,
            null,
            null,
            "Edit Vita Target in Neighborhood".Localize(),
            "Edit Vita Target in Neighborhood".Localize());
        cmdInfo.ShortcutsEditable = false;
        CommandService.RegisterCommand(cmdInfo, this);
    }

    foreach (var targetProvider in TargetProviders)
    {
        if (targetProvider.CanCreateNew)
        {
            string addCmdTag = AddNewString.Localize() + targetProvider.Name;
            CommandService.RegisterCommand(
                new CommandInfo(
                    addCmdTag,
                    null,
                    null,
                    addCmdTag,
                    "Creates a new target".Localize()),
                this);

            m_addTargetsCmdTags.Add(addCmdTag);

            string remCmdTag = "Remove ".Localize() + targetProvider.Name;
            CommandService.RegisterCommand(
                new CommandInfo(
                    remCmdTag,
                    null,
                    null,
                    remCmdTag,
                    "Remove selected target".Localize()),
                this);

            m_removeTargetsCmdTags.Add(remCmdTag);
        }
    }
}

```

Deci4p target providers are treated differently than TCP target providers. First, the method checks that a Vita SDK is installed with the `Deci4pTargetProvider.SdkInstalled` property. If present, it adds a command to edit Vita targets.

Next, `Initialize()` iterates through `TargetProviders` to create commands to add and remove targets for each provider. It first checks the `CanCreateNew` for the provider to make sure that new targets can be created; they can't for Vita, because `Deci4pTargetProvider`'s property is this:

```
public bool CanCreateNew { get { return false; } }
```

As a result, Add and Remove commands are created only for `TcpIpTargetProvider`. The general loop is used, so other types of providers can be added easily.

IContextMenuCommandProvider Implementation

This interface provides a method to get commands that apply to a given context:

```

IEnumerable<object> IContextMenuCommandProvider.GetCommands(object context, object selectedTargets)
{
    m_selectedTargets = null;
    if (context.Is<ITargetConsumer>())
    {
        m_targetConsumer = context.Cast<ITargetConsumer>();
        m_selectedTargets = selectedTargets as IEnumerable<TargetInfo>;
        foreach (var cmdTag in m_addTargetsCmdTags)
            yield return cmdTag;

        if (m_selectedTargets != null && m_selectedTargets.Any())
            foreach (var cmdTag in m_removeTargetsCmdTags)
                yield return cmdTag;

        yield return CommandTag.VitaNeighborhood;
    }
}

```

The commands are available only for the target list control provided by `TargetExceptionService`, the only implementer of `ITargetConsumer`, so the method first checks that the given context can be adapted to `ITargetConsumer`. If so, it proceeds and saves this context adapted to `ITargetConsumer` in `m_targetConsumer`, where it is used later to test whether a command can be performed.

Next, the selected objects are adapted to `IEnumerable<TargetInfo>` and saved in the field `m_selectedTargets`, which is used to test whether commands can be performed and to perform commands. Targets are treated as `TargetInfo` objects in Target Manager, so this adaptation works.

Finally, `GetCommands()` iterates through the lists `m_addTargetsCmdTags` and `m_removeTargetsCmdTags` of commands created in `Initialize()`. Remove commands are only returned when there are targets selected, hence the `m_selectedTargets != null && m_selectedTargets.Any()` test.

Command Client

The `ICommandClient` implementation uses the `TargetProviders` property and field values like `m_addTargetsCmdTags`, already set up by `Initialize()` and `IContextMenuCommandProvider.GetCommands()`. For example, here's `DoCommand()`:

```

void ICommandClient.DoCommand(object commandTag)
{
    if (CommandTag.VitaNeighborhood.Equals(commandTag))
    {
        // Invoke "Edit Vita Target in Neighborhood" command merely launches the PSP2
        Neighborhood app that comes with Vita SDK installer,
        // as if you double click "Neighborhood for PlayStation(R)Vita" on your desktop icon.
        // This is intended just for a convenience helper to allow users directly bring up the
        PSP2 app without a detour to desktop first,
        // as PSP2 Neighborhood app can reboot, power off, and do much more for the Vita kit.

        // {BA414141-28C6-7F3C-45FF-14C28C11EE88} is the registered Neighborhood for
        PlayStation(R)Vita Shell extension
        System.Diagnostics.Process.Start("Explorer.exe", @
"/e,/root,::{BA414141-28C6-7F3C-45FF-14C28C11EE88}");
    }
    else if (m_addTargetsCmdTags.Contains(commandTag))
    {
        foreach (var targetProvider in TargetProviders)
        {
            string addCmdTag = AddNewString.Localize() + targetProvider.Name;
            if (addCmdTag.Equals(commandTag))
            {
                targetProvider.AddTarget(targetProvider.CreateNew());
                break;
            }
        }
    }
    else if (m_removeTargetsCmdTags.Contains(commandTag))
    {

        foreach (var item in m_selectedTargets)
            foreach (var provider in TargetProviders)
                provider.Remove(item);
    }
}

```

The commandTag parameter is matched against existing commands in the command lists, and if a match is found, the corresponding command is executed. The Add command logic does additional processing to find the right target provider for the command and uses the target provider's AddTarget() method to add the target. The Remove processing tries to remove each selected target with each target provider's Remove() method, relying on it to do the right thing. Deci4pTargetProvider.Remove() does nothing — Vita targets can't be added or removed, only discovered — so that's not a problem. And TcpIpTargetProvider.Remove() checks that the target is one of the TCP targets, so that works also.

Topics in this section

Links on this page to other pages
[ATF Target Manager Sample](#), [Authoring Tools Framework](#)

Timeline Editor Programming Discussion

A timeline is a graphical representation of a time-sequence. Timelines contain groups that can contain tracks that can contain events, which can be intervals, keys (zero-length intervals) and markers (zero-length events that are on all tracks).

The [ATF Timeline Editor Sample](#) is a relatively full-featured timeline editor that allows creating and editing timelines. This sample is built using ATF's timeline facilities. For details, see [Timelines in ATF](#).

Programming Overview

Timeline Editor shows how to use ATF's timeline facilities, which include interfaces and classes to render, display, and manipulate time lines. General non platform-specific items are in the `Sce.Atf.Controls.Timelines` namespace and `Atf.Gui` assembly. Winforms classes are in the `Sce.Atf.Controls.Timelines.Direct2D` namespace and `Atf.Gui.WinForms` assembly. For a discussion of timeline support, see [Timelines in ATF](#).

Timeline Editor uses an XML Schema to define its data model. The "timelineType" is the root type, and such an object can contain "markerType", "timelineRefType", and "groupType" objects. And "groupType" can contain "trackType". A "trackType" contains "intervalType" and "keyType". The base "eventType" has derived types "intervalType", "keyType", and "markerType". Like many other samples, Timeline Editor has a `Schema` class of metadata type classes generated by DomGen.

The `SchemaLoader` performs the usual tasks of loading the schema and defining DOM adapters for the various types. It also parses annotations, which contain information for palette items and for defining property descriptors, which allow property editors to display object attributes.

Timeline Editor features a set of DOM adapters. DOM adapters defined on the root type have roles as a document adapter, context, or validator. Each timeline object type has a DOM adapter that implements the corresponding interface, described in [Timeline Object Interfaces](#).

There is a `TimelineDocument` DOM adapter object for every timeline open in Timeline Editor. Each document has a `D2dTimelineControl` canvas control and `D2dDefaultTimelineRenderer` to display and render the timeline, as well as a set of manipulators for the timeline.

The `TimelineEditor` component creates `TimelineDocuments`. It also serves as the control host client for the `D2dTimelineControl` and sets up the palette.

The `TimelineCommands` component handles commands on the Edit menu to remove a group, track, or empty groups and tracks, as well as to split an interval.

The `TimelineContext` DOM adapter derives from `Sce.Atf.Dom.EditingContext`, the basic editing context that provides selection and transactions. `TimelineContext` implements interfaces to enumerate timeline objects, provide events for timeline object changes, and name timeline objects. It implements `IInstancingContext` to create instances of timeline when objects are copied and pasted or dragged from the palette onto the timeline; this is the where the bulk of the code in `TimelineContext` resides because of the intricacy of a timeline.

When a property of a timeline object changes, the `TimelineValidator` DOM adapter checks that event attributes are still valid.

Contents

- [Programming Overview](#)
- [Timeline Editor Data Model](#)
 - [Containment Hierarchy](#)
 - [Data Type Derivation](#)
 - [Schema Class](#)
 - [SchemaLoader Class](#)
- [XML Schema Annotations](#)
 - [Annotation Information for Interval Objects](#)
 - [Annotation Information for Key Objects](#)
 - [Annotation Parsing](#)
- [Timeline Editor DOM Adapters](#)
 - [Timeline Object DOM Adapters](#)
- [Timeline Document and Control](#)
 - [TimelineDocument Interfaces](#)
 - [Timeline Control Creation](#)
- [Timeline Document Creation and Editing](#)
 - [Document Client](#)
 - [Creating the Timeline](#)
 - [Other Interfaces](#)
- [TimelineCommands Component](#)
- [TimelineContext Class](#)
 - [EditingContext Class](#)
 - [OnNodeSet\(\) Method](#)
 - [TimelineContext Interfaces](#)
 - [INamingContext Interface](#)
 - [IInstancingContext Interface](#)
- [Validating Timelines](#)

Timeline Editor Data Model

Like many of the samples, the data model for Timeline Editor has two types of hierarchy: containment and data type derivation. This is shown in the Visual Studio XML Schema Explorer view of the schema file `timeline.xsd`, with the nodes for the container types and base type open:

The screenshot shows the XML Schema Explorer interface. At the top, there's a search bar and some navigation icons. Below that, the 'Schema Set' section is expanded, showing the 'timeline' schema. Inside 'timeline', there are several elements: 'eventType', 'groupType', 'intervalType', 'keyType', 'markerType', 'timelineRefType', 'timelineType', and 'trackType'. Each element has its annotations listed below it. For example, 'eventType' has annotations '@start xs:float' and '@description xs:string'. 'groupType' has annotations '@name xs:string' and '@expanded xs:boolean'. 'intervalType', 'keyType', and 'markerType' each have their own annotations. 'timelineRefType' and 'trackType' also have their own annotations. The 'timelineType' element contains three child elements: 'group [0..*] groupType', 'marker [0..*] markerType', and 'timelineRef [0..*] timelineRefType'. Finally, 'trackType' contains 'interval [0..*] intervalType', 'key [0..*] keyType', and '@name xs:string'.

Containment Hierarchy

Containers show how a timeline is built. A "timelineType" contains:

- "markerType" for a marker, a zero-length event on the whole timeline.
- "timelineRefType" for a reference to a timeline.
- "groupType" for a group.
 - "trackType" for a track.
 - "intervalType" for an event spanning some time on a timeline.
 - "keyType" for a zero-length event on a timeline.

The root type is "timelineType".

Data Type Derivation

An "eventType" is an event in a timeline. Rather than using "eventType" directly in a timeline, Timeline Editor uses types derived from "eventType":

- "intervalType"
- "keyType"
- "markerType"

Schema Class

The `GenSchemaDef.bat` command file can be used to run DomGen to generate the `Schema` class, containing metadata classes for the types specified in `timeline.xsd`.

SchemaLoader Class

The `SchemaLoader` derives from `XmlSchemaTypeLoader` and loads the schema file `timeline.xsd`.

`SchemaLoader` also defines DOM adapters for the sample's data types. To learn about these DOM adapters, see [Timeline Editor DOM Adapters](#).

Finally, `SchemaLoader` parses annotations in the XML Schema file. For details on what annotations contain and how they are parsed, see [XML Schema Annotations](#).

XML Schema Annotations

Applications can specify property descriptors in two ways:

1. Creating them explicitly, typically in the schema loader.
2. Specifying them in annotations in the XML Schema.

Nearly all the samples take the first approach, but Timeline Editor uses only annotations to define descriptors. It defines descriptors for attributes of the types that correspond to objects in the timeline.

Annotation Information for Interval Objects

This example shows the type definition for "intervalType":

```
<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:complexContent>
    <xs:extension base="eventType">
      <xs:attribute name="name" type="xs:string" />
      <xs:attribute name="length" type="xs:float"/>
      <xs:attribute name="color" type="xs:integer"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Annotations in this schema specify two things:

- Palette item information.
- Attributes (properties) to display in property editors.

Palette Information

The attributes in the first tag `<scea.dom.editors menuText>` provide information for the Interval palette item:

- `menuText="Interval"`: The tooltip that appears when hovering over the palette item.
- `image="TimelineEditorSample.Resources.interval.png"`: The image file for the Interval icon.
- `category="Timelines"`: The palette in which it appears. This is useful in applications with more than one palette, such as the [ATF Diagram Editor Sample](#).

Property Editor Properties

The subsequent `<scea.dom.editors.attribute>` tags define the property descriptors that specify the information needed to display attributes in a property editor. The "intervalType" type has three attributes/properties: Name, Length, and Color. Note that these three properties are also listed in the attributes at the end of the type definition. Each annotation has three attributes: name, displayName, and description. The Color property has two additional attributes. The editor attribute specifies what value type editor is used for the color. The converter attribute tells what converter to use to convert between application data and color values; this converter goes between an ARGB int and a `System.Drawing.Color` value. For more information on value editors, see [Value Editors and Value Editing Controls](#).

The [ATF DOM Tree Editor Sample](#) also uses annotations for property descriptors. To learn how it handles them, see [XML Schema Annotation Property Descriptors in the DOM Tree Editor Programming Discussion](#).

Annotation Information for Key Objects

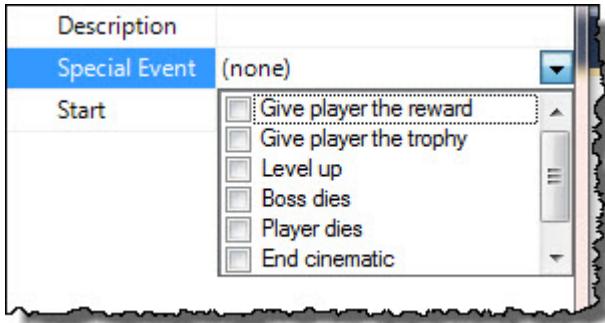
The type definition for "keyType" illustrates a special property descriptor:

```

<xss:complexType name="keyType">
  <xss:annotation>
    <xss:appinfo>
      <scea.dom.editors menuText="Key" description="Key" image=
"TimelineEditorSample.Resources.key.png" category="Timelines" />
    <scea.dom.editors.attribute>
      name="specialEvent"
      displayName="Special Event"
      description="One or more special events will occur at this time"
      editor="Sce.Atf.Controls.PropertyEditing.FlagsUITypeEditor,Atf.Gui.WinForms"
      converter="Sce.Atf.Controls.PropertyEditing.ReadOnlyConverter"/>
    </xss:appinfo>
  </xss:annotation>
  <xss:complexContent>
    <xss:extension base="eventType">
      <xss:attribute name="specialEvent" type="xs:string" />
    </xss:extension>
  </xss:complexContent>
</xss:complexType>

```

The "Special Event" property also has an editor defined for it: `FlagsUITypeEditor` that handles dynamic flag enumerations to select as many flags as desired in a list box. This editor gets additional support in the annotation parser described in [Annotation Parsing](#). The figure shows this value editor in the Property Editor when a key object is selected in the timeline:



Annotation Parsing

You must enhance the standard schema loader to process annotations so that property descriptors are created. Timeline Editor's schema loader provides several types of special processing.

First, it overrides `XmlSchemaTypeLoader.ParseAnnotations()` with its own version, calling the base method. This allows the schema loader to access annotation data and process it.

Second, Timeline Editor's `ParseAnnotations()` calls the `Sce.Atf.Dom.PropertyDescriptor.ParseXml()` method, which processes the `<scea.dom.editors.attribute>` annotation tags that define the property descriptors. The base `ParseAnnotations()` method does not call `ParseXml()`. Here's the first part of the processing that Timeline Editor's `ParseAnnotations()` does:

```

PropertyDescriptorCollection propertyDescriptors = Sce.Atf.Dom.PropertyDescriptor.ParseXml(nodeType
, xmlNodes);

// Customizations
// The flags and enum support from annotation used to be in ATF 2.8.
// Please request this feature from the ATF team if you need it and a ParseXml overload
// can probably be created.
System.ComponentModel.PropertyDescriptor gameFlow = propertyDescriptors["Special Event"];
if (gameFlow != null)
{
    FlagsUITypeEditor editor = (FlagsUITypeEditor)gameFlow.GetEditor(typeof(FlagsUITypeEditor));
    editor.DefineFlags( new string[] {
        "Reward==Give player the reward",
        "Trophy==Give player the trophy",
        "LevelUp==Level up",
        "BossDies==Boss dies",
        "PlayerDies==Player dies",
        "EndCinematic==End cinematic",
        "EndGame==End game",
    });
}

nodeType.SetTag<PropertyDescriptorCollection>(propertyDescriptors);

```

`ParseXml()` returns a collection of property descriptors that it created from the annotations in the XML Schema. The variable `gameFlow` is set to a descriptor with the name "Special Event" from this collection. Recall that this is a descriptor for an attribute in "keyType", as described in [Annotation Information for Key Objects](#). It obtains the editor corresponding to the property descriptor with this statement:

```
FlagsUITypeEditor editor = (FlagsUITypeEditor)gameFlow.GetEditor(typeof(FlagsUITypeEditor));
```

It then calls `FlagsUITypeEditor.DefineFlags()` with a string, which defines the flag names and values for the attribute. Calling `NamedMetadata.SetTag()` puts this information in the metadata class for "keyType", where it can be accessed for property data. Note that these flags are the same ones shown in the previous illustration of the `FlagsUITypeEditor` in the Property Editor for a key. For more information on how property descriptors are set up and used, see [Specifying Property Descriptors](#) in the section [Property Descriptors](#).

In the next processing section, the `ParseAnnotations()` method looks for the `<scea.dom.editors>` tags that specify palette items, as discussed in [Palette Information](#). When it finds these elements, it creates a `NodeTypePaletteItem` with that information: the menu text, description, and image:

```

// 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);
}
}

```

For details on how the [ATF Simple DOM Editor Sample](#) creates and uses a palette, including using `NodeTypePaletteItem`, see [Using a Palette in Simple DOM Editor Programming Discussion](#).

Timeline Editor DOM Adapters

Timeline Editor defines three groups of DOM adapters on the sample's data types:

- DOM adapters for the timeline document, defined on the root type, "timelineType":
 - `TimelineDocument`: The document DOM adapter. For more details, see [TimelineDocument Class](#).
 - `TimelineContext`: Context for the timeline document. For more information, see [TimelineContext Class](#).
 - `MultipleHistoryContext`: DOM adapter for multiple history contexts, so that the undo/redo history stack for each document is distinct.
- Validators on timeline data. For this discussion, see [Validating Timelines](#).

- DOM adapters for objects in a timeline. A DOM adapter is defined for each type of object in a timeline. For a description of these adapters, see [Timeline Object DOM Adapters](#).

Timeline Object DOM Adapters

Most of the DOM adapters are for the types of objects in a timeline, listed in this table:

DOM Adapter	Derives from, Implements	Description
BaseEvent	DomNodeAdapter, IEvent, ICloneable	Base class of adapters for events: intervals, markers, and keys.
Group	DomNodeAdapter, IGroup, ICloneable	Adapter for a group of tracks.
Interval	BaseEvent, IInterval	Adapter for an interval.
Key	BaseEvent, IKey	Adapter for a key.
Marker	BaseEvent, IMarker	Adapter for a marker.
Timeline	DomNodeAdapter, IHierarchicalTimeline, IHierarchicalTimelineList	Adapter for a timeline.
TimelineReference	DomNodeAdapter, ITimelineReference, ICloneable	Adapts a DomNode to a timeline reference, which allows a referenced timeline document to appear within another document, positioned to start at a particular location in that document.
Track	DomNodeAdapter, ITrack, ICloneable	Adapter for a track.

Some of these adapters implement `ICloneable`. Such adapters are for objects than can be moved in the Timeline Editor and show as "ghosts" while they are moving. An object is copied to a "ghost", which moves to show the range of movement and destination in the Timeline Editor. `D2dMoveManipulator`, the manipulator that adds move capabilities to objects, makes a ghost copy using `Clone()`:

```
ICloneable cloneable = ghost.Object as ICloneable;
if (cloneable != null)
    ghostCopy = cloneable.Clone() as ITimelineObject;
```

All the timeline object interfaces derive from `ITimelineObject`, so this adaptation works. For more information about manipulators, see [WinForms Timeline Renderers, Controls, and Manipulators](#).

Some of these DOM adapters also override the base `ToString()` method for ease of debugging and for tooltip support.

Implementing the Object Interface

The bulk of the code in these DOM adapters is devoted to implementing the interface of their object types. For instance, `Group` implements `IGroup`. For details on these various interfaces, see [Timeline Object Interfaces](#).

For example, here is `Group`'s implementation of `IGroup`:

```

/// <summary>
/// Gets and sets the group name</summary>
public string Name
{
    get { return (string)DomNode.GetAttribute(Schema.groupType.nameAttribute); }
    set { DomNode.SetAttribute(Schema.groupType.nameAttribute, value); }
}

/// <summary>
/// Gets and sets whether the group is expanded (i.e., are the tracks it contains
/// visible?)</summary>
public bool Expanded
{
    get { return (bool)DomNode.GetAttribute(Schema.groupType.expandedAttribute); }
    set { DomNode.SetAttribute(Schema.groupType.expandedAttribute, value); }
}

/// <summary>
/// Gets the timeline that contains the group</summary>
public ITimeline Timeline
{
    get { return GetParentAs<Timeline>(); }
}

/// <summary>
/// Creates a new track. Try to use TimelineControl.Create(ITrack) if there is a "source" ITrack.
/// Does not add the track to this group.</summary>
/// <returns>New unparented track</returns>
public ITrack CreateTrack()
{
    return new DomNode(Schema.trackType.Type).As<ITrack>();
}

/// <summary>
/// Gets the list of all tracks in the group</summary>
public IList<ITrack> Tracks
{
    get { return GetChildList<ITrack>(Schema.groupType.trackChild); }
}

```

Note that the `Name` and `Expanded` properties simply use the `DomNode.GetAttribute()` and `DomNode.SetAttribute()` methods to get and set attribute information in the type metadata classes in the `Schema` class. These properties provide easy access to the type's attributes.

The `Timeline` property uses the `DomNodeAdapter.GetParentAs()` method to get the underlying DOM node's parent, adapted as a `Timeline` object.

`CreateTrack()` creates a new `DomNode` of type "trackType", adapted to `ITrack`, which `Track` implements.

The `Tracks` getter uses `DomNodeAdapter.GetChildList()` to get the children of the adapted `DomNode` as an `IList<ITrack>`.

These interactions with the type metadata classes, `DomNodes`, and `DomNode` and `DomNodeAdapter` methods are typical of DOM adapters in this and other ATF samples.

Timeline Class

`Timeline` implements its object interface `ITimeline` in a similar fashion to the other object DOM adapters in this sample.

However, it also implements `IHierarchicalTimeline` and `IHierarchicalTimelineList` to handle timeline references. Recall that "timelineRefType" objects are contained in an object of type "timelineType". Both `IHierarchicalTimeline` and `IHierarchicalTimelineList` have a `References` property that returns the reference timelines, however they return it differently.

```

IEnumerable<ITimelineReference> IHierarchicalTimeline.References
{
    get { return GetChildList<ITimelineReference>(Schema.timelineType.timelineRefChild); }
}
...
public IList<ITimelineReference> IHierarchicalTimelineList.References
{
    get { return GetChildList<ITimelineReference>(Schema.timelineType.timelineRefChild); }
}

```

Both properties get exactly the same value. However, `IHierarchicalTimeline.References` returns it as an `IEnumerable<ITimelineReference>`; `IHierarchicalTimelineList.References` returns it as an `IList<ITimelineReference>`. Users can get either property, depending on what they need.

Timeline Document and Control

Timeline Editor's document class is `TimelineDocument`. Every timeline document is displayed in its own control canvas, a `D2dTimelineControl` that `TimelineDocument` creates. `D2dTimelineControl` derives from `CanvasControl`, a control for viewing and editing a 2D bounded canvas. Timeline Editor can have multiple documents open, with a `TimelineDocument` and `D2dTimelineControl` for each one.

`TimelineDocument` is defined as a DOM adapter for the "timelineType" type, which serves as the timeline document type.

A timeline requires a renderer, canvas control, and manipulators to manipulate timeline objects. For details on these ATF timeline classes, see [WinForms Timeline Renderers, Controls, and Manipulators](#).

TimelineDocument Interfaces

`TimelineDocument` indirectly implements `IDocument`. Here is `TimelineDocument`'s pedigree:

```

public class TimelineDocument : DomDocument, ITimelineDocument, IPrintableDocument, IObservableContext

```

By deriving from `DomDocument`, Timeline Editor gets a basic document class. `DomDocument` provides a simple and versatile implementation of `IDocument` that several samples use for their documents, such as [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), and [ATF Simple DOM Editor Sample](#). In addition, `DomDocument` derives from `DomResource`, which is a DOM adapter and also implements `IResource` to handle URIs for a document's location.

`TimelineDocument` also implements `ITimelineDocument`, whose `Timeline` property gets the `ITimeline` object corresponding to the document.

The `IPrintableDocument` implementation constructs `TimelinePrintDocument`, which derives from `CanvasPrintDocument`. This abstract base class is used for canvas printing, and is also used by [ATF Circuit Editor Sample](#), [ATF FSM Editor Sample](#), and [ATF State Chart Editor Sample](#). `TimelinePrintDocument` overrides methods to get "page" bounds for the various `PrintRange` values, as well as a method to do the actual rendering to the page to be printed.

The `IObservableContext` interface contains events to track items in a context: for items changing, being added or removed, and the collection of items being reloaded. Its implementation has an interesting twist: it subscribes to these interface events in the `TimelineContext` class. For example, here's `ItemInserted`:

```

public event EventHandler<ItemInsertedEventArgs<object>> ItemInserted
{
    add { this.As<TimelineContext>().ItemInserted += value; }
    remove { this.As<TimelineContext>().ItemInserted -= value; }
}

```

For a discussion of how this context operates, see [TimelineContext Class](#).

Timeline Control Creation

A `D2dTimelineControl` is constructed by setting `TimelineDocument`'s `Renderer` property, which contains an instance of `D2dDefaultTimelineRenderer`, the default timeline renderer:

```

public TimelineRenderer Renderer
{
    get { return m_renderer; }
    set
    {
        if (m_renderer != null)
            throw new InvalidOperationException("The timeline renderer can only be set once");
        m_renderer = value;

        // Due to recursion, we need m_timelineControl to be valid before
        m_timelineControl.TimelineDocument is set.
        // So, we pass in 'null' into TimelineControl's constructor.
        m_timelineControl = new TimelineControl(null, m_renderer, new TimelineConstraints(),
false);
        m_timelineControl.TimelineDocument = this;

        m_timelineControl.SetZoomRange(0.1f, 50f, 1f, 100f);
        AttachManipulators();
    }
}

```

A D2dDefaultTimelineRenderer is created when a timeline document is created by TimelineEditor. For details, see [Timeline Document Creation and Editing](#).

The TimelineControl (actually a D2dTimelineControl) is constructed using the given renderer and a TimelineConstraints object. For further information, see [Timeline Constraint Classes](#).

The last parameter of the D2dTimelineControl constructor indicates whether to create default manipulators. It's false here, because the getter calls AttachManipulators() to explicitly attach manipulators to the D2dTimelineControl:

```

protected virtual void AttachManipulators()
{
    // The order here determines the order of receiving Paint events and is the reverse
    // order of receiving picking events. For example, a custom Control that is drawn
    // on top of everything else and that can be clicked on should come last in this
    // list so that it is drawn last and is picked first.
    D2dSelectionManipulator selectionManipulator = new D2dSelectionManipulator(m_timelineControl
);
    D2dMoveManipulator moveManipulator = new D2dMoveManipulator(m_timelineControl);
    D2dScaleManipulator scaleManipulator = new D2dScaleManipulator(m_timelineControl);
    m_splitManipulator = new D2dSplitManipulator(m_timelineControl);
    D2dSnapManipulator snapManipulator = new D2dSnapManipulator(m_timelineControl);
    D2dScrubberManipulator scrubberManipulator = new ScrubberManipulator(m_timelineControl);

    //// Allow the snap manipulator to snap objects to the scrubber.
    snapManipulator.Scrubber = scrubberManipulator;
}

```

Manipulators allow you to manipulate timeline objects in a timeline control, such as moving or scaling timeline objects. The order in which manipulators are attached matters. For a more detailed description of using manipulators, see [Timeline Manipulators](#).

Timeline Document Creation and Editing

TimelineEditor is a MEF component imported into Timeline Editor. It serves several roles, including the document client, and implements the following interfaces:

```

public class TimelineEditor : IDocumentClient, IControlHostClient, IPaletteClient, IInitializable

```

Document Client

The implementation of IDocumentClient is the part that opens and displays timelines. Most of this implementation follows the pattern of other samples, and is described generally in [Implementing a Document Client](#).

The most interesting part of this interface is the Open() method, because it creates a new window for viewing a document:

```

public IDocument Open(Uri uri)
{
    TimelineDocument document = LoadOrCreateDocument(uri, true); //true: this is a master
document

    if (document != null)
    {
        m_controlHostService.RegisterControl(
            document.TimelineControl,
            document.Cast<TimelineContext>().ControlInfo,
            this);

        document.TimelineControl.Frame();
    }

    return document;
}

```

`LoadOrCreateDocument()` creates the document. The `D2dTimelineControl` of the new `TimelineDocument` is registered with the Control Host Service.

Creating the Timeline

The `LoadOrCreateDocument()` method shows a specific example of creating a timeline renderer and control.

The initial phase of `LoadOrCreateDocument()` accomplishes the routine tasks of reading data from a given URI for an existing document into a `DomNode` using a `DomXmlReader` or creating a new `DomNode` for the root node of a new document. This is followed by the heart of the method: these statements result in the creation of the key objects needed for displaying a timeline:

```

D2dTimelineRenderer renderer = CreateTimelineRenderer();
document.Renderer = renderer;
renderer.Init(document.TimelineControl.D2dGraphics);

```

`CreateTimelineRenderer()` is a trivial method that simply constructs a `D2dDefaultTimelineRenderer`:

```

protected virtual D2dTimelineRenderer CreateTimelineRenderer()
{
    return new D2dDefaultTimelineRenderer();
}

```

For further information, see [Timeline Renderers](#).

The next line sets the `TimelineDocument.Renderer` property. This results in creating the `D2dTimelineControl` and attaching manipulators to it, as discussed in [Timeline Control Creation](#).

The last line calls `D2dDefaultTimelineRenderer.Init()`, which calls its base `D2dTimelineRenderer.Init()`. These `Init()` methods set up the various Direct2D graphics objects used to render a timeline and its objects.

After these essential timeline objects are created and set up, a few more tasks remain in creating the document. The timeline context is handled next:

```

string fileName = Path.GetFileName(filePath);
ControlInfo controlInfo = new ControlInfo(fileName, filePath, StandardControlGroup.Center);

//Set IsDocument to true to prevent exception in command service if two files with the
// same name, but in different directories, are opened.
controlInfo.IsDocument = true;

TimelineContext timelineContext = document.Cast<TimelineContext>();
timelineContext.ControlInfo = controlInfo;

```

A `ControlInfo` object is constructed. The `TimelineDocument` is adapted to a `TimelineContext`, which works because `TimelineContext` is a DOM adapter defined on "timelineType", just like `TimelineDocument`. The context's `ControlInfo` property is set to the new `ControlInfo`.

Timeline documents may contain timelines, which is the reason for the "timelineRefType". The next step is to open any of these sub-timeline

documents:

```
IHierarchicalTimeline hierarchical = document.Timeline as IHierarchicalTimeline;
if (hierarchical != null)
{
    ResolveAll(hierarchical, new HashSet<IHierarchicalTimeline>());
}
```

`ResolveAll()` does a depth-first traversal, resolving all referenced sub-documents. If any are found, it calls `LoadOrCreateDocument()` recursively, so that all sub-timelines are found, not matter how deeply embedded they are.

Other Interfaces

IControlHostClient Interface

`TimelineEditor` is the control host client for the `D2dTimelineControl`. The `IControlHostClient` implementation is routine. For a description of what control clients do, see [IControlHostClient interface](#).

IPaletteClient Interface

`Timeline Editor` provides a palette of timeline objects, and it implements this palette similarly to other samples. `IPaletteClient`, in particular, provides methods to get information on palette items and to convert a palette item to an instance that can be inserted into the timeline. Palette information is provided in the XML Schema, as discussed in [Annotation Information for Interval Objects](#) and [Annotation Parsing](#).

`Timeline Editor` also imports the `PaletteService` component to manage the palette. In its constructor, it uses the imported palette service to add items to the palette:

```
paletteService.AddItem(Schema.markerType.Type, "Timelines", this);
paletteService.AddItem(Schema.groupType.Type, "Timelines", this);
paletteService.AddItem(Schema.trackType.Type, "Timelines", this);
paletteService.AddItem(Schema.intervalType.Type, "Timelines", this);
paletteService.AddItem(Schema.keyType.Type, "Timelines", this);
paletteService.AddItem(Schema.timelineRefType.Type, "Timelines", this);
```

`Timeline Editor`'s palette implementation is very similar to the one in [ATF Simple DOM Editor Sample](#). For a description of using a palette in that sample, see [Using a Palette](#).

IInitializable Interface

The `TimelineEditor` component completes its initialization in its `IInitializable.Initialize()` method. `Initialize()` uses the `ScriptingService` and `SettingsService`, which may have not been initialized at the time `TimelineEditor`'s constructor runs.

`Initialize()` sets up information with `ScriptingService` that is useful for debugging.

`Initialize()` creates a `BoundPropertyDescriptor` array with information that is to persist between sessions using the `SettingsService`. For further details on using the setting service to save preference information, see [SettingsService Component](#).

TimelineCommands Component

`TimelineCommands` is the component that handles commands on the Edit menu.

Its `ICommandClient` implementation is the command client for commands to remove a group, track, or empty groups and tracks, as well as to split an interval.

The `ICommandClient.DoCommand()` method does these commands using the properties of the various objects. For example, here's how the command to remove a group is performed:

```

ITimelineObject target = ContextRegistries.GetCommandTarget<ITimelineObject>(m_contextRegistry);
if (target == null)
    return false;

IInterval activeInterval = target as IInterval;
ITrack activeTrack =
    (activeInterval != null) ? activeInterval.Track : Adapters.As<ITrack>(target);
IGroup activeGroup =
    (activeTrack != null) ? activeTrack.Group : Adapters.As<IGroup>(target);
ITimeline activeTimeline =
    (activeGroup != null) ? activeGroup.Timeline : Adapters.As<ITimeline>(target);
ITransactionContext transactionContext = context.TimelineControl.TransactionContext;

switch ((Command)commandTag)
{
    case Command.RemoveGroup:
        if (activeGroup == null)
            return false;

        if (doing)
        {
            transactionContext.DoTransaction(delegate
            {
                activeGroup.Timeline.Groups.Remove(activeGroup);
            },
            "Remove Group");
        }
        return true;
...
}

```

First, the command's target is obtained from the context registry's `GetCommandTarget()` method, which is the last selected `ITimelineObject`. The method then attempts to adapt the selected object to various timeline object interfaces, described in [Timeline Object Interfaces](#). An `ITransactionContext` object is obtained from the `D2dTimelineControl` to use with commands so they go into the history stack and are undoable.

The group is removed in a delegate inside the `TransactionContexts.DoTransaction()` method. The active group is removed from the list of groups in the timeline held in the `ITimeline.Groups` property.

For a description of how command clients work in general, see [Creating Command Clients](#).

TimelineContext Class

`TimelineContext` is a DOM adapter defined for "timelineType", the root type. Here is its definition:

```

public class TimelineContext :
    EditingContext,
    IEnumerableContext,
    IObservableContext,
    INamingContext,
    IInstancingContext
}

```

EditingContext Class

`TimelineContext` derives from `Sce.Atf.Dom.EditingContext`, which is a general purpose editing context used as in several samples. `EditingContext` implements `ISelectionContext` and `ITransactionContext`, so it provides for selection and undo/redo of editing operations. In particular, `EditingContext` defines a `Selection` property that is a `AdaptableSelection<object>`; using this object makes it easy to implement `ISelectionContext`. For more information about this context's capabilities, see [Sce.Atf.Dom.EditingContext Class](#).

OnNodeSet() Method

The `EditingContext` that `TimelineContext` derives from is a DOM adapter, so `TimelineContext` has an `OnNodeSet()` method, called when the `DomNode` is initialized. Because `TimelineContext` is defined on the root type, the `DomNode` is the root of the `DomNode` tree. This method is called when a timeline is created, as when a timeline document is opened.

```

protected override void OnNodeSet()
{
    m_timeline = this.As<Timeline>();
    m_timelineDocument = DomNode.Cast<TimelineDocument>();
    m_timelineControl = m_timelineDocument.TimelineControl;

    DomNode.AttributeChanged += DomNode_AttributeChanged;
    DomNode.ChildInserted += DomNode_ChildInserted;
    DomNode.ChildRemoved += DomNode_ChildRemoved;

    m_timelineControl.DragEnter += timelineControl_DragEnter;
    m_timelineControl.DragOver += timelineControl_DragOver;
    m_timelineControl.DragDrop += timelineControl_DragDrop;
    m_timelineControl.DragLeave += timelineControl_DragLeave;

    base.OnNodeSet();
}

```

This `DomNode` can be adapted to `Timeline` and `TimelineDocument`, because these DOM adapters are all defined on the root type.

The `DomNode` events subscribed to occur whenever any `DomNode` in the tree changes or is inserted or removed anywhere in the tree, since they are subscribed to on the root `DomNode`. The handlers for these events, in turn, call methods that raise events in `IObservableContext`.

Drag events for the `D2dTimelineControl` in `m_timelineControl` allow handling dragging objects from the palette to the timeline. The insertion of these dragged items into the timeline is handled by the `IInstancingContext` interface. For details, see [Drag and Drop Handling](#) and also [IInstancingContext Interface](#).

TimelineContext Interfaces

`TimelineContext` implements interfaces that many other editing contexts in the samples also implement:

- `IEnumerableContext`: Enumerate objects in the timeline.
- `IObservableContext`: Events for timeline objects being added or removed. This is required so the timeline can be redrawn if it changes.
- `INamingContext`: Get or change the name of timeline objects.
- `IInstancingContext`: Handle copy and paste of selected timeline objects, creating new instances.

The implementation of all of these interfaces is simple and very similar to other samples' — except for `IInstancingContext`. For details on that interface's operations, see [IInstancingContext Interface](#).

INamingContext Interface

`INamingContext` is an interface for contexts where objects can be named. Most timeline objects have a `Name` property, which this interface's implementation takes advantage of.

To demonstrate this simple interface, first consider its `GetName()` method, which gets the item's name in the context, or `null` if none:

```

string INamingContext.GetName(object item)
{
    IEvent e = item.As<IEvent>();
    if (e != null)
        return e.Name;

    IGroup group = item.As<IGroup>();
    if (group != null)
        return group.Name;

    ITrack track = item.As<ITrack>();
    if (track != null)
        return track.Name;

    return null;
}

```

`GetName()` first tries to adapt the item to `IEvent`. If that works, it can get the name from the `IEvent.Name` property. This works whether the item is an `IInterval` or `IMarker`, because these both derive from `IEvent`. Failing that, it tries to adapt the item to `IGroup`, and if that works, it returns the `IGroup.Name` property. Finally, it tries to adapt to `ITrack`, the last nameable timeline object, returning the `ITrack.Name` property if successful.

Next, look at the beginning of `CanSetName()`, which returns whether the item can be named:

```
bool INamingContext.CanSetName(object item)
{
    IEvent e = item.As<IEvent>();
    if (e is IKey)
        return false; // Keys.Name currently is always the empty string
    if (e != null)
        return IsEditable(e);
    ...
}
```

Again, the method attempts adaptation to `IEvent`. If the item is an `IKey` — which is based on `IEvent` — the method returns false, because keys are not named, unlike the other events. If it's another kind of `IEvent`, the method returns whatever `TimelineContext.IsEditable()` does:

```
private bool IsEditable(ITimelineObject item)
{
    var path = new TimelinePath(item);
    TimelineDocument document = (TimelineDocument)TimelineEditor.TimelineDocumentRegistry.ActiveDocument;
    if (document != null)
        return document.TimelineControl.IsEditable(path);
    return false;
}
```

`IsEditable()` constructs a `TimelinePath` for the item. It gets the active `TimelineDocument` from the document registry. It uses the document's `TimelineControl` property to get the `D2dTimelineControl`. It can then call this control's `IsEditable()` method, which determines whether the object corresponding to the given path is editable.

Finally, here is `SetName()` that sets the item's name:

```
void INamingContext.SetName(object item, string name)
{
    IEvent e = item.As<IEvent>();
    if (e != null)
    {
        e.Name = name;
        return;
    }
    ...
}
```

`SetName()` acts like the other methods, trying to adapt the item to the various nameable objects, and setting the value of the `Name` property for the object when adaptation succeeds.

IInstancingContext Interface

This interface handles creating new instances of timeline objects when they are copied or cut and then pasted to the timeline, or are dragged from the palette and dropped on the timeline. For a general description of `IInstancingContext`, see [IInstancingContext Interface](#). For a full discussion of instancing in general, see [Instancing In ATF](#).

Note that instancing operations go on the history stack, because `TimelineContext` implements transactions through its base class, `Sce.Atf.Dom.EditingContext`.

There are several ways in which objects may be instanced:

- Copy a timeline object and paste it.
- Drag an object from the palette onto the timeline.

Note that dragging a timeline object from one location on the timeline to another is not handled by instancing — it's handled by the `D2dMoveManipulator`. For more information on manipulators, see [Timeline Manipulators](#).

IInstancingContext Non-Insert Methods

The implementation of `IInstancingContext`'s methods is very timeline specific but fairly simple — except for `Insert()`. For example, here is `CanInsert()`:

```

public bool CanInsert(object insertingObject)
{
    IDataObject dataObject = (IDataObject)insertingObject;
    object[] items = dataObject.GetData(typeof(object[])) as object[];
    return
        items != null &&
        AreTimelineItems(items) &&
        (TimelineControl.TargetGroup == null ||
         TimelineControl.IsEditable(TimelineControl.TargetGroup));
}

```

The first couple of statements get the data to be inserted and convert it to an array of objects. The following must all be true to allow insertion:

- Inserted data is not null.
- Inserted data consists only of ITimeline objects, as determined by `AreTimelineItems()`.
- Either the selected target group (`TimelineControl.TargetGroup`) is null or it is editable.

IInstancingContext.Insert() Method

`Insert()` is called when a paste command occurs and is considerably more complicated than the other `IInstancingContext` methods. A timeline has a hierarchy of objects, so the target insertion location must be suitable for the type of inserted objects. Intervals are pasted onto tracks, for example, and tracks are pasted onto groups. There is a great deal of special case processing in `Insert()`.

`Insert()` begins by verifying there is data to copy and turning it into an array of objects. It makes a copy of the objects to paste as an array of `DomNodes`. It makes a `List<ITimelineObject>` from this `DomNode` array. So far, this is fairly standard for `Insert()` implementations.

It tries to determine where objects should be copied by further processing, such as finding the center position of the copied objects, making a dictionary mapping objects to their tracks, and so on.

`Insert()` now guesses where the user wants to paste, based on this priority, figuring which of these is available as a target:

1. The timeline control's target (currently selected) track.
2. The track in the center of the view.
3. The first visible track.

It then inserts the copy into the timeline, using a private `Insert()` method:

```

foreach (ITimelineObject item in itemCopies)
{
    // Not all items will have a track. The item could be a group, for example.
    ITrack dropTarget;
    copiesToTracks.TryGetValue(item, out dropTarget);
    Insert(item, dropTarget);
}

```

The variable `copiesToTracks` is a dictionary mapping target timeline objects to the tracks they should be placed in. The private `Insert()` method looks at the various types of timeline objects involved in the insertion and determines exactly where the inserted item should go and places it there. As the comment indicates, not all objects are inserted into a track.

Finally, `Insert()` selects the items that were just inserted into the timeline.

Another public `Insert()` method handles inserting items by drag and drop, and this also calls the private `Insert()` method to do the actual insertion.

About half of `TimelineContext` consists of utility methods used by `Insert()` and the other `IInstancingContext` methods. For example, `CreateTrackMappings()` determines which tracks to place timeline objects in. `CenterEvents()` finds the center point of a set of timeline objects.

Drag and Drop Handling

As noted previously in [OnNodeSet\(\) Method](#), handlers for the drag and drop events on the `D2dTimelineControl` call methods in `IInstancingContext`. For example, this is the handler for the `DragDrop` event:

```

private void timelineControl_DragDrop(object sender, DragEventArgs e)
{
    if (CanInsert(e.Data))
    {
        OnDragDrop(e);
    }
}

```

This calls `IInstancingContext.CanInsert`, which was discussed in [IInstancingContext Non-Insert Methods](#). If the insertion is OK, it calls `OnDragDrop()` to drop the item onto the timeline:

```

protected virtual void OnDragDrop(DragEventArgs e)
{
    string name = "Drag and Drop".Localize();
    Selection.Clear();
    this.DoTransaction(() => Insert(e), name);
    m_timelineControl.DragDropObjects = null;
    m_timelineControl.Focus();
}

```

`OnDragDrop()` calls the previously mentioned private `Insert()` method in a transaction to insert the dragged item. It then nullifies the `D2dTimelineControl`'s drag-drop list and gives focus to the `D2dTimelineControl`.

Validating Timelines

In `SchemaLoader`, Timeline Editor defines three validators on the root type "timelineType":

```

Schema.timelineType.Type.Define(new ExtensionInfo<UniqueIdValidator>());
Schema.timelineType.Type.Define(new ExtensionInfo<ReferenceValidator>());
Schema.timelineType.Type.Define(new ExtensionInfo<TimelineValidator>());

```

`UniqueIdValidator` and `ReferenceValidator` make sure that `DomNode` IDs are unique and all references to these IDs are valid. These validators are not really needed in Timeline Editor, because there are no ID attributes or references to them. Many samples use these validators though, such as [ATF FSM Editor Sample](#) and [ATF State Chart Editor Sample](#). Though they are not used, defining the validators does no harm, except for a slight amount of unnecessary checking for `DomNode` IDs.

`TimelineValidator` checks that timeline event attributes are valid. It overrides the `OnAttributeChanged()` method in `Observer`, which is called whenever a `DomNode` attribute changes. In other words, `OnAttributeChanged()` is called whenever a attribute/property of a timeline object changes.

Because these validators are defined on the type of the root `DomNode`, all nodes in the `DomNode` tree are checked, that is, all application data is checked.

Here's `OnAttributeChanged()`:

```

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;
            float constrained = Math.Max(value, 0); // >= 0
            constrained = (float)MathUtil.Snap(constrained, 1.0); // snapped to nearest
integral frame number
            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)
        {
            if (e.AttributeInfo.Equivalent(Schema.intervalType.lengthAttribute))
            {
                float value = (float)e.NewValue;
                float constrained = Math.Max(value, 1); // >= 1
                constrained = (float)MathUtil.Snap(constrained, 1.0); // snapped to nearest
integral frame number
                if (constrained != value)
                    throw new InvalidTransactionException(Localizer.Localize("Timeline intervals
must have an integer length"));
                return;
            }
        }
    }
}

```

Only event timeline objects are checked:

- Timeline events must have a positive integer start time.
- Timeline intervals must have an integer length; an interval is a type of event.

`AttributeEventArgs.Node` is the `DomNode` whose attribute has changed or will change. It is adapted to `BaseEvent`, the DOM adapter base class for events. If the adaptation is unsuccessful, the changed `DomNode` does not correspond to an event and does not need to be checked.

`AttributeEventArgs.AttributeInfo` is the `AttributeInfo` for the changed attribute of the `DomNode`. The method checks that this corresponds to the attribute for the start time of the event. If so, the new start time value is obtained from `AttributeEventArgs.NewValue` and checked that it is greater than or equal to zero. If not, it raises an exception.

If the changed attribute is not a start time, the validator attempts to adapt the `DomNode` to `Interval`. If successful, it checks whether the changed attribute is for the interval length. In that case, it checks that the new interval length is an integer and raises an exception if it isn't.

Topics in this section

Links on this page to other pages

[ATF Circuit Editor Sample](#), [ATF Diagram Editor Sample](#), [ATF DOM Tree Editor Sample](#), [ATF FSM Editor Sample](#), [ATF Simple DOM Editor Sample](#), [ATF State Chart Editor Sample](#), [ATF Timeline Editor Sample](#), [Authoring Tools Framework](#), [Context Classes](#), [Creating Command Clients](#), [Creating Control Clients](#), [DOM Tree Editor Programming Discussion](#), [General Timeline Classes](#), [IInstancingContext](#) and [IHierarchicalInsertionContext](#) Interfaces, [Implementing a Document and Its Client](#), [Instancing In ATF](#), [Property Descriptors](#), [SettingsService Component](#), [Simple DOM Editor Programming Discussion](#), [Timeline Interfaces](#), [Timelines in ATF](#), [Value Editors and Value Editing Controls](#), [WinForms Timeline Renderers, Controls, and Manipulators](#)

Tree List Editor Programming Discussion

The [ATF Tree List Editor Sample](#) demonstrates how to use the ATF tree controls `TreeListView` and its enhancement, `TreeListViewEditor`. `TreeListView` uses `TreeListViewAdapter`, which adapts `TreeListView` to display data. Tree List Editor shows all styles of a `TreeListView` as well as an unadorned `TreeListView`, implementing them all as components.

Tree List Editor also demonstrates how to create data and display its properties in a `PropertyEditor` component.

Programming Overview

The `CommonEditor` component is the base class for most of the tree editors in Tree List Editor. It shows how to use the `TreeListViewEditor` class, which uses a `TreeListView` with a `TreeListViewAdapter`. `CommonEditor` creates a `UserControl` to hold the `TreeListViewEditor` and related tool buttons and provides the control host client for the `UserControl`. The tool buttons generate tree data and perform other functions that are handled by `CommonEditor`'s methods.

The `RawTreeListView` component displays a folder's file hierarchy and operates somewhat like the `CommonEditor`, but uses a `TreeListView`.

Tree List Editor creates a data item class `DataItem` derived from `CustomTypeDescriptor` to display data in the trees. This class uses a structure very similar to the classes for generated data in the [ATF Property Editing Sample](#). This also allows displaying data in a `PropertyEditor` component. The `DataContainer` class can contain a collection of `DataItem` and implements `ITreeListView` and other interfaces to allow viewing data in a tree. `DataContainer`'s methods also generate tree data.

The `TreeListViewAdapter` wraps a `TreeListView` and allows supplying data to the `TreeListView` through its `View` property, which implements the `ITreeListView` interface. This `View` property contains a `DataContainer` object, which supplies what the `TreeListViewAdapter` needs to display data.

Contents

- [Programming Overview](#)
- [Tree List Editor Components](#)
- [CommonEditor Component](#)
 - [CommonEditor Constructor](#)
 - [BtnClick\(\) Method](#)
 - [CommonEditor Control Host Client](#)
 - [Editors Derived from CommonEditor](#)
- [RawTreeListView Component](#)
 - [RawTreeListView Constructor](#)
 - [IInitializable.Initialize\(\) Method](#)
 - [MySorter Class](#)
 - [Other Methods](#)
- [Tree Data Generation](#)
 - [DataItem Class](#)
 - [DataContainer Class](#)
 - [DataComparer Class](#)
- [TreeListViewAdapter Class](#)
 - [TreeListViewEditor View Property](#)
 - [TreeListViewAdapter View Property](#)

Tree List Editor Components

Tree List Editor follows the standard WinForms initialization pattern discussed in [WinForms Application](#). In addition to standard ATF components, it adds the following components of its own to the MEF TypeCatalog:

- [List](#): Add the standard list view editor, based on the `CommonEditor` component. For details on this and the other editor components (except `RawTreeListView`), see [CommonEditor Component](#).
- [CheckedList](#): Add the checked list view editor, based on the `CommonEditor` component.
- [VirtualList](#): Add the virtual list view editor, based on the `CommonEditor` component.
- [TreeList](#): Add the tree list view editor, based on the `CommonEditor` component.
- [CheckedTreeList](#): Add the checked tree list view editor, based on the `CommonEditor` component.
- [RawTreeListView](#): Add a tree list editor for displaying a hierarchical file system. For details, see [RawTreeListView Component](#).

CommonEditor Component

The `List`, `CheckedList`, `VirtualList`, `TreeList`, and `CheckedTreeList` components all derive from the `CommonEditor` component, and these components all reside in the file `Editor.cs`.

`CommonEditor` is the common editor used for all the tree view editors. The other components' implementation is trivial, because all needed capability is provided in `CommonEditor`.

`CommonEditor` derives from `TreeListViewEditor`:

```
class CommonEditor : TreeListViewEditor, IInitializable, IControlHostClient
```

`TreeListViewEditor` adds extra functions to a `TreeListView`, including a `TreeListViewAdapter` and a right-click context edit menu. The `TreeListView` class provides a tree with a list view. `TreeListViewAdapter` wraps a `TreeListView` and allows a user to supply data to it through the `ITreeListView` interface. For more information on the operation of `TreeListViewAdapter` with this sample, see [TreeListViewAdapter Class](#).

CommonEditor Constructor

`CommonEditor`'s constructor specifies a display name, tree list style, and flags indicating which buttons to display:

```
public CommonEditor(
    string name,
    TreeListView.Style style,
    Buttons flags,
    IContextRegistry contextRegistry,
    ISettingsService settingsService,
    IControlHostService controlHostService)
```

The constructor creates a `UserControl` control to hold the `TreeListViewEditor` plus any buttons desired.

The `name` parameter specifies text to display in the user interface for the tree, which is on the window for the tree.

The `TreeListView.Style` parameter is an enumeration specifying the tree style:

- `List`: A list.
- `CheckedList`: A list with check boxes next to items.
- `VirtualList`: A list supporting thousands of items, instantiated as needed.
- `TreeList`: A tree with columns (default).
- `CheckedTreeList`: A tree list with check-boxes next to the items.

`Tree List Editor` demonstrates all these styles in its five editor windows.

The `flags` parameter tells `CommonEditor` which buttons to add to the tree control. The parameter is a flag bit mask, so it can specify any combination of buttons. Each button is created with the `CreateButton()` method, which creates a new `Button` with the desired text, sizing it to fit the text, and positioning it on the `UserControl`. All the buttons have the same `Click` event handler `BtnClick()`, but have their `Tag` property set to the `Buttons` enumeration value for that button. The constructor invokes `RegisterControl()` with the Control Host Registry for the `UserControl`.

BtnClick() Method

The `BtnClick()` event handler uses `Button.Tag` to determine what function to perform. Some of these functions generate data and place it in the tree. For a discussion of data generation, see [Tree Data Generation](#).

CommonEditor Control Host Client

`CommonEditor` implements `IControlHostClient` very simply for the `UserControl`. Its `Activate()` method just sets the active context to the `TreeListViewEditor.View` property, which holds the `ITreeListView` associated with data displayed in the control. For more information on data handling in the tree, see [Editors Derived from CommonEditor](#) and [Tree Data Generation](#).

Editors Derived from CommonEditor

All the other tree editors derive from `CommonEditor`. These other components do very little. Here's the entire `CheckedList` definition, for example:

```
[ImportingConstructor]
public CheckedList(
    IContextRegistry contextRegistry,
    ISettingsService settingsService,
    IControlHostService controlHostService)
    : base(
        "Checked List",
        TreeListView.Style.CheckedList,
        Buttons.AddFlat,
        contextRegistry,
        settingsService,
        controlHostService)
{
    TreeListView.NodeSorter =
        new DataComparer(TreeListView);

    View = new DataContainer();

    TreeListView.NodeChecked += TreeListViewNodeChecked;
}
```

The constructor calls the base constructor to specify the editor name, to indicate it's a `CheckedList`, and to specify that only the "Add Flat" button appears with this editor.

The `TreeListView.NodeSorter` property specifies the way nodes are sorted in the `TreeListView`. The `TreeListView` property is in the `TreeListViewEditor` and contains its `TreeListView`. For information on the sorting class, see [DataComparer Class](#).

The `View` property is also in the `TreeListViewEditor` and is a `ITreeListView` indicating the data view for the `TreeListView`. The `DataContainer` class implements `ITreeListView` and represents a set of data generated and placed in the tree. For details, see [Tree Data Generation](#).

This constructor also specifies the handler for the `TreeListView.NodeChecked` event:

```
private void TreeListViewNodeChecked(object sender, TreeListView.NodeEventArgs e)
{
    Outputs.WriteLine(
        OutputMessageType.Info,
        "{0} {1}",
        e.Node.Label,
        e.Node.CheckState == CheckState.Checked
            ? "checked"
            : "unchecked");
}
```

This method writes a message to the Output window in Tree List Editor. This window is created and handled by the `Outputs` component, which provides static methods to access its Output window. Tree List Editor imports `Outputs`.

RawTreeListView Component

`RawTreeListView` creates the Raw TreeListView Usage window in Tree List Editor to display a folder's contents in a tree. In this respect, it has similarities to the [ATF File Explorer Sample](#). That sample's `FolderViewer` component displays the entire file hierarchy of the "C:" drive's contents as a tree. For details, see [FolderViewer Component](#).

`RawTreeListView` also operates somewhat like the `CommonEditor` component. It creates a `UserControl` to host a tree control and buttons. Its `CreateButton()` method is identical to `CommonEditor`'s. It also implements `IControlHostClient` to serve as the `UserControl` control host client, which functions similarly to `CommonEditor`'s client. However, `RawTreeListView` uses a `TreeListView` for the tree control rather than a `TreeListViewEditor`.

RawTreeListView Constructor

The constructor creates all the buttons needed, adds them to the `UserControl`, and subscribes to events. The event handlers are all in `RawTreeListView`. The `TreeListView` is also created and added to the `UserControl`. Finally, the constructor invokes `RegisterControl()` with the Control Host Registry to register `UserControl`.

IInitializable.Initialize() Method

`RawTreeListView` also implements `IInitializable.Initialize()`, which does nothing. However, implementing `IInitializable`

forces this component to be instantiated, which would not happen otherwise, because nothing imports it. For more details on MEF initialization, see [Initializing Components](#).

MySorter Class

MySorter is the sorter that RawTreeListView provides for the tree, as seen in this code from RawTreeListView's constructor.

```
m_control = new TreeListView { Name = NameText, AllowDrop = true };
m_control.NodeSorter = new MySorter(m_control);
```

MySorter is somewhat similar to the DataComparer class used by CommonEditor, but sorts folder and file names rather than generated data. For information on how DataComparer works, see [DataComparer Class](#).

Other Methods

Most of the methods and code in RawTreeListView either handle events or enumerate the files in a folder to display them. The folder handling uses standard .NET classes, such as DirectoryInfo and Directory.

Tree Data Generation

The file DataGenerator.cs contains several classes used in generating display data for the trees, in particular DataItem and DataContainer. These classes use a structure very similar to the classes for generated data in the [ATF Property Editing Sample](#). For information on how that sample generates data, see [Creating Application Data](#).

DataContainer in particular works with the TreeListViewAdapter class to display generated data. To learn how this works, see [TreeListViewAdapter Class](#).

DataItem Class

DataItem represents a set of generated data displayed in a tree; the PropertyEditor component can also display certain DataItem properties. DataItem derives from System.ComponentModel.CustomTypeDescriptor, which is a simple default implementation of the System.ComponentModel.ICustomTypeDescriptor interface:

```
class DataItem : CustomTypeDescriptor
```

ICustomTypeDescriptor supplies dynamic custom type information for an object and is useful for describing data items in an application. The ItemBase class in the Property Editing sample is the basic data unit and also derives from CustomTypeDescriptor.

PropertyEditing Attribute

DataItem has a set of properties, such as Parent and Children, that allow it to represent hierarchical data. Consider these two properties:

```
/// <summary>
/// Gets whether item has children</summary>
[PropertyEditingAttribute]
public bool HasChildren
{
    get { return m_children.Count > 0; }
}

/// <summary>
/// Gets item's children</summary>
public ICollection<DataItem> Children
{
    get { return m_children; }
}
```

The Children property gets an ICollection<DataItem> of all the child data.

Note that the property HasChildren is marked with the attribute [PropertyEditingAttribute], but Children isn't.

[PropertyEditingAttribute] is a custom attribute that Tree List Editor uses to mark the properties to be displayed in the Property Editor. It's defined this way:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class PropertyEditingAttribute : Attribute
{}
```

The `PropertyEditingAttribute` class itself is marked with the attribute `[AttributeUsageAttribute()]`, which tells how the defined attribute can be used. In this case, the `AttributeTargets.Property` enumeration value means the attribute applies to C# properties. `AllowMultiple = true` means that the attribute can be used more than once in a class. The Property Editing sample also defines a custom attribute that it uses to mark fields, rather than properties. Unlike that sample, the attribute `[PropertyEditingAttribute]` has no parameters, so the class is very simple, requiring nothing but its declaration. The properties here are marked with the full attribute name `[PropertyEditingAttribute]`, although `[PropertyEditing]` also works, by convention.

PropertyPropertyDescriptor Class

`DataItem` defines a property descriptor class `PropertyPropertyDescriptor` for C# properties. Note that the word "property" is used in two senses here: for a C# property, as well as for a property represented by a property descriptor.

```
public class PropertyPropertyDescriptor : PropertyDescriptor
```

`PropertyPropertyDescriptor` derives from `System.ComponentModel.PropertyDescriptor`, and its implementation is somewhat similar to the Property Editing sample's analogous class `FieldPropertyDescriptor`, a descriptor for fields, rather than C# properties. For a description of `FieldPropertyDescriptor`, see [Creating Application Data](#).

GetProperties() Method

When a selection of tree items occurs, through a chain of events, a collection of property descriptors is requested by calling the `GetProperties()` method for each selected item that implements `ICustomTypeDescriptor`, which is `DataItem` in Tree List Editor. This eventually results in the properties for the last selected item being displayed in the Property Editor window by the `PropertyEditor` component.

Like the Property Editing sample, the class deriving from `CustomTypeDescriptor`, `DataItem`, implements the `GetProperties()` method:

```
public override PropertyDescriptorCollection GetProperties()
{
    var props = new PropertyDescriptorCollection(null);

    foreach (var property in GetType().GetProperties())
    {
        var propertyDesc =
            new PropertyPropertyDescriptor(property, GetType());

        propertyDesc.ItemChanged += PropertyDescItemChanged;

        foreach (Attribute attr in propertyDesc.Attributes)
        {
            if (attr.GetType().Equals(typeof(PropertyEditingAttribute)))
                props.Add(propertyDesc);
        }
    }

    return props;
}
```

This method iterates through all the C# properties defined in the `DataItem` class, obtained from `GetProperties()`, and creates a `PropertyDescriptor` for each property. It then checks all the `Attributes` for the data in the property descriptor. If any of the attributes is `PropertyEditingAttribute`, it adds the property descriptor to the `PropertyDescriptorCollection` object `props`. It does not add the property descriptor more than once, because the only attribute marking properties in `DataItem` is `[PropertyEditingAttribute]`.

This `GetProperties()` method resembles the one in the Property Editing sample. However, that one excluded properties that had the checked for attribute; this `GetProperties()` method includes such properties. This guarantees that only `[PropertyEditingAttribute]` marked properties are listed in the Property Editor.

DataContainer Class

This class provides the ability to display data in tree controls, and it also generates the data as `DataItem` objects. It contains data in its

`m_data` field, which is of type `List<DataItem>`.

DataContainer Interfaces

`DataContainer` implements these interfaces, each of which are discussed below:

```
class DataContainer : ITreeListView, IItemView, IObservableContext, ISelectionContext, IValidationContext
```

ITreeListView Interface

`ITreeListView` defines a view on hierarchical data and consists of the following:

- `IEnumerable<object> Roots`: Gets the root level objects of the tree view.
- `IEnumerable<object> GetChildren(object parent)`: Gets enumeration of the children of the given parent.
- `string[] ColumnNames`: Gets list column's names.

This is very similar to the `ITreeView` interface, except that `ITreeListView` allows more than one root object and also adds `ColumnNames`. The implementation here uses the `m_data` field along with `DataItem`'s properties `HasChildren` and `Children`.

IItemView Interface

This interface's `GetInfo` property gets a given object's `ItemInfo`. The `ItemInfo` class contains numerous properties to access information about an item displayed in a tree node (or list). The implementation of `GetInfo()` in `DataContainer` just fills out a few properties:

```
public void GetInfo(object obj, ItemInfo info)
{
    var data = obj.As<DataItem>();
    if (data == null)
        return;

    info.Label = data.Name;
    info.Properties =
        new object[]
    {
        data.Type.ToString(),
        data.Value
    };

    info.IsLeaf = !data.HasChildren;
    info.ImageIndex =
        data.HasChildren
        ? s_folderImageIndex
        : s_dataImageIndex;
}
```

First, this method adapts the item to a `DataItem`. After copying the `Name`, it creates an array of `objects` for the `Properties` property, containing a string of the item's type and its value. It sets `IsLeaf`, and then sets up `ImageIndex` to display a folder image for an item with children.

IObservableContext Interface

`IObservableContext` contains events for items being changed, added, or removed. This is required so the tree display can be refreshed after items change. These events are raised when data is modified.

When the `TreeListViewAdapter.View` property is set, `TreeListViewAdapter` adapts the value of `View` to `IObservableContext` and subscribes the adapted value to these events. These events are thus handled by methods in `TreeListViewAdapter`. Also recall that `View` contains a `DataContainer` object. For details, see [TreeListViewAdapter View Property](#) and [IObservableContext Events](#) in particular. `TreeListViewAdapter` does the same thing for the `IValidationContext` interface.

ISelectionContext Interface

The selectable objects are `DataItem` objects, as seen in this section of the `ModifySelected()` method, which is called when the "Modify Selected" button is clicked:

```

private void ModifySelected(IEnumerable<object> selection)
{
    Beginning.Raise(this, EventArgs.Empty);

    foreach (var obj in selection)
    {
        var data = obj.As<DataItem>();
        ...
    }
}

```

The `DataContainer` class creates a `Selection` object, which is a collection representing a selection. It's easy to implement `ISelectionContext` using the methods available on a `Selection` object. The `ISelectionContext` implementation is almost exactly the same as the implementation in the Property Editing sample's `ListEditingContext` class. For its description, see [ISelectionContext](#).

In its constructor, `DataContainer` subscribes to selection change events, where `m_selection` contains the `Selection` object:

```

m_selection.Changing += TheSelectionChanging;
m_selection.Changed += TheSelectionChanged;

```

`TheSelectionChanging()` simply raises an event:

```

private void TheSelectionChanging(object sender, EventArgs e)
{
    SelectionChanging.Raise(this, EventArgs.Empty);
}

```

Raising such selection events hooks into the event handling system that ultimately generates lists of property descriptors that property editors can use to display the common properties of selected items.

IValidationContext Interface

This interface provides events that validators can subscribe to and perform their validations at appropriate times:

- `Beginning`: Raised before validation begins.
- `Cancelled`: Raised after validation is cancelled.
- `Ending`: Raised before validation ends.
- `Ended`: Raised after validation ends.

Some of the data generation and modification methods raise these events.

When the `TreeListViewAdapter.View` property is set, `TreeListViewAdapter` adapts the value of `View` to `IValidationContext` and subscribes the adapted value to these events. These events are thus handled by methods in `TreeListViewAdapter`. For details, see [TreeListViewAdapter View Property](#) and [IValidationContext Events](#) in particular. `TreeListViewAdapter` does the same thing for the `IObservableContext` interface.

DataContainer Data Generation Methods

The event handler method `BtnClick()` handles clicking buttons on the tree windows, as described in [BtnClick\(\) Method](#). `BtnClick()` calls `DataContainer`'s data generation methods for some of the buttons, as seen in this section of the method:

```

private void BtnClick(object sender, EventArgs e)
{
    var btn = (Button)sender;
    var flags = (Buttons)btn.Tag;

    switch (flags)
    {
        case Buttons.AddFlat:
            DataContainer.GenerateFlat(
                View,
                (TreeListView.TheStyle != TreeListView.Style.TreeList) &&
                (TreeListView.TheStyle != TreeListView.Style.CheckedTreeList)
                ? null
                : TreeListViewAdapter.LastHit);
            break;

        case Buttons.AddHierarchical:
            DataContainer.GenerateHierarchical(
                View,
                TreeListViewAdapter.LastHit);
            break;
        ...
    }
}

```

The `GenerateFlat()` method, for instance, has a first parameter of `View`, which is a property of `TreeListViewEditor`, from which the `CommonEditor` class derives. However, the `View` property in `TreeListViewEditor` simply gets and sets uses the `View` property in `TreeListViewAdapter`. The `TreeListViewAdapter`'s `View` property holds an `ITreeListView`, that is, an object implementing `ITreeListView`. For Tree List Editor, this is a `DataContainer` object, because each of the tree editors makes an assignment like this:

```
View = new DataContainer();
```

For details, see [Editors Derived from CommonEditor](#). For details on the `View` property, see [TreeListViewAdapter View Property](#).

The second parameter of `AddHierarchical()` is either `null` (depending on the tree style) or the last data item selected.

Here's the `GenerateFlat()` method:

```

public static void GenerateFlat(ITreeListView view, object lastHit)
{
    var container = view.As<DataContainer>();
    if (container == null)
        return;

    container.GenerateFlat(lastHit.As<DataItem>());
}

```

The `view` parameter is adapted to `DataContainer`, which works because `view` is a `DataContainer` object. The private method `GenerateFlat()` is called, with a parameter of the last selected item, if any, which serves as the parent to which data objects are added:

```

private void GenerateFlat(DataItem parent)
{
    Beginning.Raise(this, EventArgs.Empty);

    var items = s_random.Next(5, 16);
    for (var i = 0; i < items; i++)
    {
        var data = CreateItem(parent);
        data.ItemChanged += DataItemChanged;

        if (parent != null)
            parent.Children.Add(data);
        else
            m_data.Add(data);

        ItemInserted.Raise(this, new ItemInsertedEventArgs<object>(-1, data, parent));
    }

    if (parent != null)
    {
        ItemChanged.Raise(this, new ItemChangedEventArgs<object>(parent));
    }

    Ending.Raise(this, EventArgs.Empty);
    Ended.Raise(this, EventArgs.Empty);
}

```

Notice that `GenerateFlat()` raises the `IValidationContext` events at the appropriate times: `Beginning` at the beginning, and then `Ending` and `Ended` at the end.

A `System.Random` object is used repeatedly in data generation to randomize the number of objects generated and the lengths of strings. For instance, the loop generates between 5 and 16 items, using the `CreateItem()` method to create `DataItem` objects.

The `ItemChanged` event is subscribed to for the new `DataItem` with the handler `DataItemChanged`. This handler raises the `ItemChanged` event of the `IObservableContext` interface. This is key to updating the tree with this new data. For details, see [ItemChanged and ItemRemoved Events](#).

`CreateItem()` creates all the `DataItem` objects:

```

private static DataItem CreateItem(DataItem parent)
{
    var enumLength = Enum.GetNames(typeof(DataType)).Length;
    var name = CreateString(s_random.Next(2, 11));
    var type = (DataType)s_random.Next(0, enumLength);

    object value;
    switch (type)
    {
        case DataType.Integer:
            value = s_random.Next(0, 51);
            break;

        case DataType.String:
            value = CreateString(s_random.Next(5, 16));
            break;

        default:
            value = type.ToString();
            break;
    }

    var data =
        new DataItem(
            parent,
            name,
            type,
            value);

    return data;
}

```

This method generates a random object name, randomly chooses the type of the generated object, and then generates a random integer or string for the object's value. It constructs a new `DataItem` from this data and returns it.

DataComparer Class

`DataComparer` provides the functions to sort items in the tree lists. It is defined for a `TreeListView.Node`:

```
class DataComparer : IComparer<TreeListView.Node>
```

Its main function, `Compare()`, shows that `TreeListView.Node.Tag` is adapted to `DataItem`:

```
public int Compare(TreeListView.Node x, TreeListView.Node y)
{
    if ((x == null) && (y == null))
        return 0;

    if (x == null)
        return 1;

    if (y == null)
        return -1;

    if (ReferenceEquals(x, y))
        return 0;

    var lhs = x.Tag.As<DataItem>();
    var rhs = y.Tag.As<DataItem>();
    ...
}
```

The `TreeListView.Node.Tag` property gets or sets the user data attached to the node, which is a `DataItem` in this case. For details on how `TreeListView.Node.Tag` is set to a `DataItem`, see [ItemChanged and ItemRemoved Events](#) in the section [TreeListViewAdapter Class](#).

TreeListViewAdapter Class

`TreeListViewAdapter` wraps a `TreeListView` and allows supplying data to the `TreeListView` through its `View` property, which implements the `ITreeListView` interface. The `TreeListView.Node.Tag` contains the data attached to the node, as already noted. This section shows how the `TreeListViewAdapter` makes that happen and how the `View` property is the key. `TreeListViewAdapter` also triggers updating the tree's display after data changes. Raising the appropriate events makes the updates occur.

This section also shows that by implementing `ITreeListView`, `DataContainer` allows displaying data in the tree with `TreeListViewAdapter`.

TreeListViewEditor View Property

As previously seen in [Editors Derived from CommonEditor](#), the tree editors deriving from `CommonEditor` set the `View` property of `TreeListViewEditor` in their constructors:

```
View = new DataContainer();
```

Here's the definition of `TreeListViewEditor.View`

```
public ITreeListView View
{
    get { return m_treeListViewAdapter.View; }
    set { m_treeListViewAdapter.View = value; }
}
```

This shows that the `TreeListViewEditor.View` property actually comes from `TreeListViewAdapter.View`, because `m_treeListViewAdapter` holds the `TreeListViewAdapter` associated with the `TreeListViewEditor`. Therefore, the `DataContainer` object is the value that `TreeListViewAdapter` sees in `View`. Note that `DataContainer` implements `ITreeListView`, as required.

TreeListViewAdapter View Property

`TreeListViewAdapter.View` contains the actual `DataContainer` object, as just observed. Here is part of the implementation of `TreeListViewAdapter.View`:

```
public ITreeListView View
{
    get { return m_view; }
    set
    {
        if (m_view != value)
        {
            if (m_view != null)
            {
                ...
                m_view = value;

                if (m_view != null)
                {
                    m_itemView = m_view.As<IItemView>();

                    m_selectionContext = m_view.As<ISelectionContext>();
                    if (m_selectionContext != null)
                    {
                        m_selectionContext.SelectionChanging += SelectionContextSelectionChanging;
                        m_selectionContext.SelectionChanged += SelectionContextSelectionChanged;
                    }
                    else
                    {
                        m_treeListView.NodeSelected += TreeListViewNodeSelected;
                    }

                    m_observableContext = m_view.As<IObservableContext>();
                    if (m_observableContext != null)
                    {
                        m_observableContext.ItemInserted += ObservableContextItemInserted;
                        m_observableContext.ItemChanged += ObservableContextItemChanged;
                        m_observableContext.ItemRemoved += ObservableContextItemRemoved;
                        m_observableContext.Reloaded += ObservableContextReloaded;
                    }

                    m_validationContext = m_view.As<IValidationContext>();
                    if (m_validationContext != null)
                    {
                        m_validationContext.Beginning += ValidationContextBeginning;
                        m_validationContext.Ending += ValidationContextEnding;
                        m_validationContext.Ended += ValidationContextEnded;
                        m_validationContext.Cancelled += ValidationContextCancelled;
                    }
                }
            }
        }

        Load();
    }
}
```

The property's setter attempts to adapt the new value of the `View` property to several context interfaces:

- `ISelectionContext`
- `IObservableContext`
- `IValidationContext`

If successful, it subscribes the adapted `DataContainer` value to a set of events in the interface. Because `DataContainer` implements all these interfaces, `DataContainer` objects subscribe to all these events. However, all these event handlers are in `TreeListViewAdapter`. These handlers allow displaying data in the tree.

IObservableContext Events

These events relate to items in the context changing. They are raised by the data generation methods in `DataContainer` to update the display, as shown in [DataContainer Data Generation Methods](#).

ItemInserted Event

Here's the handler for ItemInserted:

```
private void ObservableContextItemInserted(object sender, ItemInsertedEventArgs<object> e)
{
    if (m_treeListView.TheStyle != TreeListView.Style.VirtualList)
    {
        if (m_inTransaction)
            m_queueInserts.Add(e);
        else
            AddItem(e.Item, e.Parent);
    }
    else
    {
        if (e.Item is object[])
            VirtualListSize += ((object[])e.Item).Length;
        else
            VirtualListSize += 1;
    }
}
```

For non-virtual tree lists, the new item in `e.Item` is added to the queue `m_queueInserts` for later addition to the tree — or added immediately with the `AddItem()` method. The field `m_inTransaction` indicates whether a transaction is occurring or not, and is set in the `IValidationContext` events. For more details, see [IValidationContext Events](#).

`AddItem()` creates a `TreeListView.Node` for the item with the `CreateNodeForObject()` method, which also calls `UpdateNodeFromItemInfo()`, discussed further in [ItemChanged and ItemRemoved Events](#). `AddItem()` also adds nodes for children of the item, because this data can be hierarchical: a `DataItem` can have child `DataItems`.

ItemChanged and ItemRemoved Events

The event handlers for `ItemChanged` and `ItemRemoved` are similar to each other. Both handle virtual lists separately, and queue items that are to be changed or removed during a transaction — or perform the update immediately when there is no transaction. Here is `ObservableContextItemChanged()`, for instance:

```
private void ObservableContextItemChanged(object sender, ItemChangedEventArgs<object> e)
{
    if (m_treeListView.TheStyle != TreeListView.Style.VirtualList)
    {
        if (m_inTransaction)
            m_queueUpdates.Add(e);
        else
            UpdateItem(e.Item);
    }
    else
    {
        UpdateItem(e.Item);
    }
}
```

During a transaction, the item is added to `m_queueUpdate`; otherwise the tree is immediately updated with `UpdateItem`, which does the following:

```

private void UpdateItem(object item)
{
    TreeListView.Node node;
    if (!m_dictNodes.TryGetValue(item, out node))
        return;

    ItemInfo info =
        GetItemInfo(
            item,
            m_itemView,
            m_treeListView.ImageList,
            m_treeListView.StateImageList);

    UpdateNodeFromItemInfo(node, item, info);

    m_treeListView.Invalidate(node);
}

```

After getting item information from `GetItemInfo()`, `UpdateItem()` calls `UpdateNodeFromItemInfo()` with that information and then invalidates the tree view to force redrawing the tree.

Among other things, `UpdateNodeFromItemInfo()` makes the data stored in the tree easily accessible:

```

private static void UpdateNodeFromItemInfo(TreeListView.Node node, object item, ItemInfo info)
{
    node.Label = info.Label;
    node.Properties = info.Properties;
    node.ImageIndex = info.ImageIndex;
    node.StateImageIndex = info.StateImageIndex;
    node.CheckState = info.GetCheckState();
    node.Tag = item;
    node.IsLeaf = info.IsLeaf;
    node.Expanded = info.IsExpandedInView;
    node.FontStyle = info.FontStyle;
    node.HoverText = info.HoverText;
}

```

Note that `node.Tag = item` sets the `TreeListView.Node.Tag` property to the data item. For Tree List Editor, this is a `DataItem` object. The `Compare()` method takes advantage of this to access this data, as noted in [DataComparer Class](#). In addition, the `LastHit` property gets the data for the last selected node from `Tag`:

```

public object LastHit
{
    get
    {
        object lastHit = m_treeListView.LastHit;
        return lastHit.Is<TreeListView.Node>() ? lastHit.As<TreeListView.Node>().Tag : this;
    }
}

```

This finally explains how the `Tag` property gets set to the data object, which is a `DataItem` instance.

Reloaded Event and Load() Method

The `Reloaded` event occurs when the data for a tree has been reloaded, and its event handler merely calls the `Load()` method:

```

private void Load()
{
    Unload();

    if (m_view == null)
        return;

    if (m_treeListView.TheStyle == TreeListView.Style.VirtualList)
        m_treeListView.VirtualListSize = 0;

    // Add columns & data

    foreach (string columnName in m_view.ColumnNames)
        m_treeListView.Columns.Add(new TreeListView.Column(columnName));

    try
    {
        m_treeListView.BeginUpdate();

        if (m_treeListView.TheStyle != TreeListView.Style.VirtualList)
        {
            foreach (object root in m_view.Roots)
            {
                if (m_inTransaction)
                    m_queueInserts.Add(new ItemInsertedEventArgs<object>(-1, root, null));
                else
                    AddItem(root, null);
            }
        }
        finally
        {
            m_treeListView.EndUpdate();
        }
    }
}

```

`Unload()` clears the tree and all the work queues, such as `m_queueInserts`. The method `TreeListView.ColumnCollection.Add()` adds a column to the tree.

In the second loop, data items are obtained from the `Roots` property, which holds all the root data objects. This data is added to either to `m_queueInserts` or immediately added to the tree by `AddItem()`, as in the `ItemInserted` handler `ObservableContextItemInserted` shown in [ItemInserted Event](#). For the Tree List Editor, `Roots` is a property of `DataContainer` that enumerates the `DataItem` objects in this container.

In the end, `TreeListView.EndUpdate()` calls `EndUpdate()` on the underlying control, which resumes redrawing the tree. Drawing was prevented earlier by the `TreeListView.BeginUpdate()` method.

IValidationContext Events

`IValidationContext` provides events to encapsulate processes in transactions, so the operation can be cancelled if a problem occurs. These event handlers also modify tree data in a similar fashion to the `IObservableContext` events.

Beginning Event

The Beginning event handler simply sets flags:

```

private void ValidationContextBeginning(object sender, EventArgs e)
{
    m_inTransaction = true;
    m_treeListView.UseInsertQueue = true;
}

```

The `m_inTransaction` field is used throughout `TreeListViewAdapter` to determine whether a transaction is occurring, as seen in [ItemInserted Event](#).

Ending and Ended Events

`TreeListViewAdapter` really handles only the Ended event signalling the end of a transaction, using this method:

```

private void ValidationContextEnded(object sender, EventArgs e)
{
    if (!m_inTransaction)
        return;

    try
    {
        m_treeListView.BeginUpdate();

        try
        {
            m_treeListView.UseInsertQueue = true;

            foreach (var args in m_queueInserts)
                AddItem(args.Item, args.Parent);
        }
        finally
        {
            m_treeListView.UseInsertQueue = false;
            m_treeListView.FlushInsertQueue();
        }

        foreach (var args in m_queueUpdates)
            UpdateItem(args.Item);

        foreach (var args in m_queueRemoves)
            RemoveItem(args.Item);
    }
    finally
    {
        m_inTransaction = false;

        m_queueInserts.Clear();
        m_queueUpdates.Clear();
        m_queueRemoves.Clear();

        m_treeListView.EndUpdate();
    }
}

```

ValidationContextEnded() first stops display drawing by calling BeginUpdate().

If there are items to add in m_queueInserts, it calls AddItem() for each one. Update items in m_queueUpdates are processed with UpdateItem(), and items in m_queueRemoves are removed with RemoveItem(). All the work queues are emptied.

Finally, EndUpdate() redraws the tree.

Topics in this section

Links on this page to other pages

[ATF File Explorer Sample](#), [ATF Property Editing Sample](#), [ATF Tree List Editor Sample](#), [Authoring Tools Framework](#), [File Explorer Programming Discussion](#), [Initializing Components](#), [Property Editing Programming Discussion](#), [WinForms Application](#)

Using Dom Programming Discussion

The [ATF Using Dom Sample](#) is a simple and direct example of how to use the ATF DOM in an application. It contains an XML Schema, a schema metadata class file generated by DomGen, DOM adapters for types, and a schema loader. It also shows saving application data in an XML file.

This sample application has no UI. You can run it in a Windows Command Prompt to see its output text. It saves the application data it creates in an XML file `game.xml`.

Programming Overview

Using Dom has no UI and does not use MEF, and it mainly demonstrates how to create and handle application data with the ATF DOM.

It has a very simple data model with one base type that game objects are based on, and the model is defined in the XML Schema file `game.xsd`. Similarly to many of the other samples, Using Dom has a metadata class created by DomGen. It uses a simple schema loader based on `XmlSchemaTypeLoader` that mainly loads the schema and defines DOM adapters on data types. These DOM adapters primarily define properties for type attributes.

Contents

- [Programming Overview](#)
- [Using Dom Data Model](#)
- [Using Dom Schema Loader](#)
- [DOM Adapters in Using Dom](#)
- [Using Dom Main\(\) Function](#)
- [Schema Loader Invocation](#)
- [Creating Application Data](#)
- [Printing Application Data](#)
- [Saving Application Data](#)

The `Main()` function loads the XML Schema, which also results in defining the DOM adapters. It creates the application data by building a tree of `DomNodes` and setting their attributes. It shows two ways to do this, one by directly setting `DomNode` attributes and the other by setting DOM adapted object's properties. This first method is sufficient, so this sample does not even need to define DOM adapters.

The sample also prints out the application data to a Windows Command Prompt.

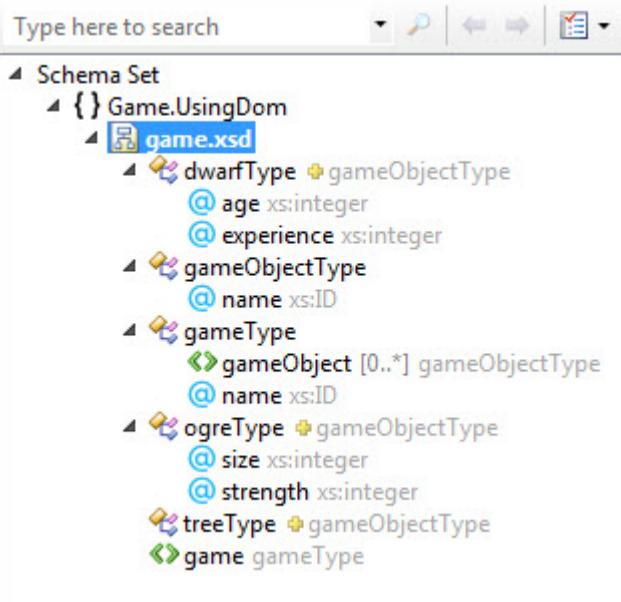
Using Dom persists its application data to an XML file with the `DomXmlWriter` class, which uses the loaded XML Schema.

Using Dom Data Model

Using Dom specifies its data model in the XML Schema file `game.xsd`. The "gameObjectType" is for game objects, and the other game objects are based on it: "dwarfType", "ogreType", and "treeType". The type "gameType" is the root type and contains "gameObjectType" type objects:

```
<xs:complexType name="gameType">
  <xs:sequence>
    <xs:element name="gameObject" type="gameObjectType" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:ID" />
</xs:complexType>
...
<xs:element name="game" type="gameType" />
```

The entire data schema is shown in this view of the Visual Studio XML Schema Explorer, with all types opened to show their attributes and contained objects:



Note that "treeType" has no attributes of its own, only the attributes of its base type "gameObjectType":

```

<xs:complexType name="treeType">
  <xs:complexContent>
    <xs:extension base="gameObjectType">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

```

This shows that a derived type does not have to define additional attributes. This definition does accomplish making "treeType" a "gameObjectType", and that's its only purpose. This allows "treeType" objects to be contained in the "gameType" object.

The command file GenSchemaDef.bat runs DomGen on the XML Schema file game.xsd:

```

...\\...\DevTools\DomGen\bin\DomGen.exe game.xsd GameSchema.cs Game.UsingDom UsingDom

```

DomGen creates the metadata class file GameSchema.cs defining the class GameSchema, which shows how the metadata classes are constructed. For instance, here's what is generated from the schema definition for the root type "gameType", shown previously:

```

public static class gameType
{
    public static DomNodeType Type;
    public static AttributeInfo nameAttribute;
    public static ChildInfo gameObjectChild;
}

```

The three class members are the following:

- Type: DomNodeType for the type of the DomNode associated with an object of "gameType".
- nameAttribute: AttributeInfo object for the attribute "name".
- gameObjectChild: ChildInfo object for the list of objects this type can contain, of type "gameObjectType".

The GameSchema class's Initialize() method initializes the gameType class this way:

```

gameType.Type = typeCollection.GetNodeType("gameType");
gameType.nameAttribute = gameType.Type.GetAttributeInfo("name");
gameType.gameObjectChild = gameType.Type.GetChildInfo("gameObject");

```

This accomplishes the following:

- Defines the node type Type using the XmlSchemaTypeCollection.GetNodeType() method to the type of "gameType".
- Sets nameAttribute with DomNodeType.GetAttributeInfo() to the attribute type for the "name" attribute.
- Initializes gameObjectChild using the DomNodeType.GetChildInfo() method to the child info for "gameObject" elements, which are the objects that the gameType object can contain.

Using Dom Schema Loader

The `GameSchemaLoader` class derives from `XmlSchemaTypeLoader`, the ATF base schema loader class. Its constructor loads the schema file:

```
public GameSchemaLoader()
{
    // set resolver to locate embedded .xsd file
    SchemaResolver = new ResourceStreamResolver(Assembly.GetExecutingAssembly(),
"UsingDom/Schemas");
    Load("game.xsd");
}
```

The `XmlSchemaTypeLoader.Load()` method calls `OnSchemaSetLoaded()` after the schema is loaded to perform any desired initialization:

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

        // register extensions
        GameSchema.gameType.Type.Define(new ExtensionInfo<Game>());
        GameSchema.gameType.Type.Define(new ExtensionInfo<ReferenceValidator>());
        GameSchema.gameType.Type.Define(new ExtensionInfo<UniqueIdValidator>());

        GameSchema.gameObjectType.Type.Define(new ExtensionInfo<GameObject>());
        GameSchema.dwarfType.Type.Define(new ExtensionInfo<Dwarf>());
        GameSchema.ogreType.Type.Define(new ExtensionInfo<Ogre>());
        break;
    }
}
```

The `foreach` executes only once, because there's only one collection of types here. After setting field values used by a couple of properties, `NameSpace` and `TypeCollection`, the loop defines DOM adapters.

The first three are for the root type "gameType", so they apply to the root `DomNode`:

- `Game` is for the type "gameType" itself.
- `UniqueIdValidator` makes sure that IDs for nodes are unique. Note that the schema for "gameType" specified an ID for its "name" attribute: `<xss:attribute name="name" type="xs:ID" />`.
- `ReferenceValidator` verifies that references in the DOM data are valid. This is usually used for references to IDs, and is not needed in this sample, because there are no data references in the data model.

The last three DOM adapters are defined for each of the game object types that have attributes. There is no DOM adapter for "treeType", because it has only the attributes from its base type, "gameObjectType", which does have a DOM adapter.

For details on what all the DOM adapters do, see [DOM Adapters in Using Dom](#).

DOM Adapters in Using Dom

A DOM adapter class provides the capabilities desired for a `DomNode` that can be adapted to the DOM adapter. In Using Dom, the DOM adapters provide properties for each attribute of the type the DOM adapter is defined for.

For example, here's the DOM adapter `GameObject` for the "gameObjectType" type, which is about as simple as it gets:

```

public class GameObject : DomNodeAdapter
{
    /// <summary>
    /// Gets or sets game object name</summary>
    public string Name
    {
        get { return GetAttribute<string>(GameSchema.gameObjectType.nameAttribute); }
        set { SetAttribute(GameSchema.gameObjectType.nameAttribute, value); }
    }
}

```

The "gameObjectType" type has just one attribute, "name". `GameObject`'s `Name` property uses the `DomNodeAdapter` methods `GetAttribute()` and `SetAttribute()` on the `nameAttribute` for the property's getter and setter. Recall that `nameAttribute` is defined as an `AttributeInfo` in `GameSchema`:

```
public static AttributeInfo nameAttribute;
```

And `DomNodeAdapter.GetAttribute()` takes an `AttributeInfo` parameter:

```
protected T GetAttribute<T>(AttributeInfo attributeInfo)
```

`Name`'s property definition is typical for attribute properties in the sample applications that use the ATF DOM.

The DOM adapter `Game` defined for "gameType" also defines a property for its "name" attribute, which is the name of the game. It also defines the `GameObjects` property for the game objects that an object of this type contains:

```

public class Game : DomNodeAdapter
{
    /// <summary>
    /// Performs initialization when the adapter is connected to the game's DomNode.
    /// Raises the DomNodeAdapter NodeSet event and performs custom processing.</summary>
    protected override void OnNodeSet()
    {
        base.OnNodeSet();
        m_gameObjects = GetChildList<GameObject>(GameSchema.gameType.gameObjectChild);
    }

    /// <summary>
    /// Gets or sets the game's name</summary>
    public string Name
    {
        get { return GetAttribute<string>(GameSchema.gameType.nameAttribute); }
        set { SetAttribute(GameSchema.gameType.nameAttribute, value); }
    }

    /// <summary>
    /// Gets a list of the game objects</summary>
    public IList<GameObject> GameObjects
    {
        get { return m_gameObjects; }
    }

    private IList<GameObject> m_gameObjects;
}

```

`GameObjects` simply returns the value in the `m_gameObjects` for getting its value. This DOM adapter also has an `OnNodeSet()` method, which sets the value of `m_gameObjects`. The `DomNodeAdapter.GetChildList()` method used here takes a `ChildInfo` parameter:

```
protected IList<T> GetChildList<T>(ChildInfo childInfo)
```

Thus getting the value of `GameObjects` returns an `IList<GameObject>`: a list of the DOM adapter objects for the game objects contained in the `DomNode`. DOM adapters adapt `DomNodes`, so this is essentially a list of the child `DomNodes` of the "gameType" node, the root `DomNode`.

The DOM adapters `Dwarf` and `Ogre` both derive from the `GameObject` DOM adapter, because their types are based on "gameObjectType":

```
public class Dwarf : GameObject
```

In general, if data type A is based on type B, the DOM adapter defined on A should derive from the DOM adapter defined on the type B.

Using Dom Main() Function

Main() puts all the pieces together.

Schema Loader Invocation

Main() creates a schema loader:

```
var gameSchemaLoader = new GameSchemaLoader();
```

As noted in [Using Dom Schema Loader](#), GameSchemaLoader's constructor loads the schema, and that also calls `OnSchemaSetLoaded()` that defines the DOM adapters, so the application can use them.

Creating Application Data

Using Dom offers two ways to create application data, one of which is commented out:

```
DomNode game = null;
// create game either using DomNode or DomNodeAdapter.
game = CreateGameUsingDomNode();
//game = CreateGameUsingDomNodeAdapter();
```

The `DomNode` `game` is the root `DomNode` for the tree of `DomNodes` that contain the application data. Using Dom creates application data by creating a tree of `DomNodes` and putting data into each `DomNode` by setting its attributes. The code here allows for two approaches to setting up the `DomNodes`:

- `CreateGameUsingDomNode()`: Create `DomNodes` and set their attributes directly.
- `CreateGameUsingDomNodeAdapter()`: Create `DomNodes` and set their attributes by using DOM adapters on the nodes.

These approaches offer an interesting contrast. First consider how these different methods set up the root `DomNode`.

`CreateGameUsingDomNode()` does it this way:

```
// create Dom node of the root type defined by the schema
DomNode game = new DomNode(GameSchema.gameType.Type, GameSchema.gameRootElement);
game.SetAttribute(GameSchema.gameType.nameAttribute, "Ogre Adventure II");
IList<DomNode> childList = game.GetChildList(GameSchema.gameType.gameObjectChild);
```

This method creates a new `DomNode` of the type `GameSchema.gameType.Type` with `ChildInfo` from `GameSchema.gameRootElement`, using this version of the `DomNode` constructor:

```
public DomNode(DomNodeType nodeType, ChildInfo childInfo)
```

Next, it sets the "name" attribute with the `DomNode.SetAttribute()` method. (Note that the DOM adapters described in [DOM Adapters](#) in [Using Dom](#) set attributes this way.) Finally, it gets a list of the root's child `DomNodes` with the `DomNode.GetChildList()` method (just as the DOM adapter did).

Contrast this to the approach of `CreateGameUsingDomNodeAdapter()`:

```
DomNode root = new DomNode(GameSchema.gameType.Type, GameSchema.gameRootElement);
Game game = root.As<Game>();
game.Name = "Ogre Adventure II";
```

It creates the root `DomNode` exactly the same way. However, it then initializes a `Game` object by adapting the root `DomNode` to the `Game` DOM adapter with the `Adapters.As()` method:

```
public static T As<T>(this IAdaptable adaptable)
```

This adaptation works, because `DomNode` implements `IAdaptable`:

```
public sealed class DomNode : IAdaptable, IDecoratable
```

The last line sets the `Name` property to "Ogre Adventure II". By the definition of the Game DOM adapter, setting this property sets the "name" attribute of its associated `DomNode` to the given value. So the result is exactly the same for both methods.

The "ogreType" is also handled differently in the two methods. Here's `CreateGameUsingDomNode()`:

```
// Add an ogre
DomNode ogre = new DomNode(GameSchema.ogreType.Type);
ogre.SetAttribute(GameSchema.ogreType.nameAttribute, "Bill");
ogre.SetAttribute(GameSchema.ogreType.sizeAttribute, 12);
ogre.SetAttribute(GameSchema.ogreType.strengthAttribute, 100);
childList.Add(ogre);
```

As before, this sets the ogre's attributes using the `DomNode.SetAttribute()` method. At the end, it adds the new `DomNode` as a child of the root `DomNode` using the `childList` obtained earlier from the root `DomNode`.

Here's `CreateGameUsingDomNodeAdapter()`'s approach to doing the same thing:

```
// Add an ogre
DomNode ogreNode = new DomNode(GameSchema.ogreType.Type);
Ogre orge = ogreNode.As<Ogre>();
orge.Name = "Bill";
orge.Size = 12;
orge.Strength = 100;
game.GameObjects.Add(orge);
```

Again, it creates the ogre `DomNode` exactly the same as in `CreateGameUsingDomNode()`. However, it then adapts this node to an `Ogre` DOM adapter object. It sets attribute values using `Ogre`'s `Name`, `Size`, and `Strength` properties. The `Name` property actually belongs to the `GameObject` DOM adapter that `Ogre` derives from:

```
public class Ogre : GameObject
```

The last difference is that the new `DomNode` is added as a child to the `GameObjects` property of the `Game` object `game`, because this property holds the list of child `DomNodes`, as described in [DOM Adapters in Using Dom for the Game DOM adapter](#).

At its end, `CreateGameUsingDomNode()` returns the `DomNode` it created in the beginning:

```
return game;
```

On the other hand, `CreateGameUsingDomNodeAdapter()` returns the `DomNodeAdapter.DomNode` property of the Game DOM adapter, which contains the adapted `DomNode`:

```
return game.DomNode;
```

The `CreateGameUsingDomNode()` method creates application data just fine, so the DOM adapters are not really required, except to illustrate this alternate approach to setting up data.

Printing Application Data

After application data is created, it is printed:

```
Print(game);
```

The `Print()` method prints a subtree of `DomNodes`:

```

private static void Print(DomNode game)
{
    Console.WriteLine("Game: {0}", game.GetAttribute(game.Type.GetAttributeInfo("name")));

    foreach (DomNode child in game.Children)
    {
        Console.WriteLine();
        Console.WriteLine("  {0}", child.Type.Name);
        foreach (AttributeInfo attr in child.Type.Attributes)
            Console.WriteLine("    {0}: {1}",
                attr.Name,
                child.GetAttribute(attr));
    }
    Console.WriteLine();
}

```

The first `WriteLine()` call prints data from the root `DomNode`'s attributes obtained with `DomNodeType.GetAttribute()`. The subsequent outer loop prints information on each child of the root, obtained from the child list in the `DomNode.Children` property. The inner loop iterates through all the attributes of each child, obtained from the list of attributes in the `DomNodeType.Attributes` property.

This is the output you get when running Using Dom in a Windows Command Prompt:

```

Game.UsingDom:ogreType
  name: Bill
  size: 12
  strength: 100

Game.UsingDom:dwarfType
  name: Sally
  age: 32
  experience: 55

Game.UsingDom:treeType
  name: Mr. Oak

```

Saving Application Data

It's easy to persist application data when you use the ATF DOM and an XML Schema for type definitions. The last section of `Main()` sets up the file path for the saved data file and then saves the data:

```

// create directory for data files
Directory.CreateDirectory(Path.Combine(ExecutablePath, @"data"));
string filePath = Path.Combine(ExecutablePath, "data\\game.xml");
var gameUri = new Uri(filePath);

// save game.
 FileMode fileMode = FileMode.Create;
using (FileStream stream = new FileStream(filePath, fileMode))
{
    DomXmlWriter writer = new DomXmlWriter(gameSchemaLoader.TypeCollection);
    writer.Write(game, stream, gameUri);
}

```

The most interesting part is the ATF specific line that creates a `DomXmlWriter` using this constructor:

```

public DomXmlWriter(XmlSchemaTypeCollection typeCollection)

```

The constructor needs an `XmlSchemaTypeCollection`, which contains the loaded schema — all the data model's definitions, plus names that can be used in the XML. The `DomXmlWriter.Write()` method writes the node tree, given its root, to the given stream and URI:

```

public virtual void Write(DomNode root, Stream stream, Uri uri)

```

The resulting file `game.xml` looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<game xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs=
"http://www.w3.org/2001/XMLSchema" name="Ogre Adventure II" xmlns="Game.UsingDom">
  <gameObject xsi:type="ogreType" name="Bill" size="12" strength="100" />
  <gameObject xsi:type="dwarfType" name="Sally" age="32" experience="55" />
  <gameObject xsi:type="treeType" name="Mr. Oak" />
</game>
```

First, notice that the XML tag names for the objects, `<game>` and `<gameObject>`, are the element names of the root type and the items contained in a "gameType" object.

Next, the XML attribute names are the attribute names defined in the types. For example, the `<game>` tag, of type "gameType", has the attribute "name", which is defined in the XML Schema. The first `<gameObject>` tag has an "xsi:type" of "ogreType", which is the type name of an ogre. Its other attributes, "name", "size", and "strength", come from the type definitions for "gameObjectType" and "ogreType".

In other words, the XML in the Schema nicely parallels the XML in the saved file.

Topics in this section

Links on this page to other pages

[ATF Using Dom Sample](#), [Authoring Tools Framework](#)

WinForms and WPF Apps Programming Discussion

These two applications are very simplified versions of the [ATF Simple DOM Editor Sample](#), whose coding is described in detail in [Simple DOM Editor Programming Discussion](#). These samples are adapted somewhat along the lines of the application developed starting with the Simple DOM Editor, described in [Creating an Application from an ATF Sample](#).

The main point of these samples is to show how to develop both WinForms and WPF samples using ATF. The same ATF techniques are used as in Simple DOM Editor, so these two samples don't illustrate anything additional about ATF, per se.

Topics

- [Programming Overview](#)
- [Application's Common Code](#)
- [WinForms Application](#)
- [WPF Application](#)

Programming Overview

This page discusses what the common code for the two samples has in common with and how it differs from the [ATF Simple DOM Editor Sample](#)'s code. The page also mentions the two samples' individual code, which is much smaller than their common code.

Application's Common Code

The bulk of the code in both [ATF Win Forms App Sample](#) and [ATF Wpf App Sample](#) is in common in the `WinGuiCommon` folder in the applications' source. Their common code provides nearly all the functionality of the samples, and demonstrates that ATF works well in both WinForms and WPF. This common code was taken from the [ATF Simple DOM Editor Sample](#).

What was kept from the Simple DOM Editor:

- Data model definition, in both the XML Schema, Schema class, and `SchemaLoader`.
- Main editor window with `ListView` control.
- Document class and client.
- DOM adapters for event, resource, and event sequence types.
- Contexts for events and event sequences.
- Editor for main `ListView`.

What was left out from the Simple DOM Editor:

- Palette window.
- Resources window.
- DOM node search, including `GenericSelectionContext`.
- Help.

In what remains, Simple DOM Editor was translated into these samples' common code mainly by changing from "EventSequence" to "WinGuiCommonData" in file names, type names, class names, and so forth. For example, the `WinGuiCommonData` class corresponds to the `EventSequence` DOM adapter.

There are other minor differences as well, such as changing the extension of documents to ".gad".

WinForms Application

The Win Forms App illustrates a typical WinForms application. It does the following:

- Initialization, such as calling appropriate `Application` class methods.
- Setting up MEF `TypeCatalog`, if using MEF.
- Creating a `MainForm` and running it.

For more information on developing a WinForms application with ATF, see [WinForms Application](#).

WPF Application

Similarly, Wpf App shows a typical WPF application. It has these things:

- A `WpfApp.App` class, defined in `App.xaml`, derived from the ATF class `AtfApp`.
- MEF initialization if using MEF, defined in `App.xaml.cs`.

For a fuller description of creating an ATF application for WPF, see [WPF Application](#).

Topics in this section

Links on this page to other topics

[ATF Simple DOM Editor Sample](#), [ATF Win Forms App Sample](#), [ATF Wpf App Sample](#), [Creating an Application from an ATF Sample](#), [Simple DOM Editor Programming Discussion](#), [WinForms Application](#), [WPF Application](#)

ATF Glossary

This glossary defines ATF technical terms and utilities.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

adaptable control

Control that can have adapters that add capabilities to the control. See control adapter.

adaptee

An object that is adapted to an adapter and behaves like an instance of the adapter's class. See adapter.

adapter

An object that dynamically casts an object to another type or class. An object can be its own adapter, if it can be cast to the desired type. An adapter for `A` converts something of a different type than `A` to the type `A`. An adapter may also adapt a control to displaying data, such as the `TreeListViewAdapter` class adapting data to be displayed in a `TreeListView` control.

adaptation

Technique used to allow one type of C# object to look like another, without using inheritance. Adaptation is the process of dynamically casting an object (adaptee) to a type of the class of the adapter. The adaptee can be treated as an instance of the adapter's class. In terms of software engineering design patterns, adaptation consists of the Adapter and Decorator pattern. See adapter, DOM extension, and DOM adapter.

adaptation framework

ATF framework that allows supporting different kinds of data models and managed data stores, such as a DOM or other CLR objects, by converting objects to other types. See adaptation.

annotation (DOM)

An entry in a type definition that contains additional information about a type that is not in the regular type definition. In an XML Schema, annotations are defined inside the standard XSD `<xs:annotation>` and `<xs:appinfo>` tags. In the ATF DOM, annotations can be used to convey information such as the icon used to represent a data type in a palette or the control used to edit a type's properties.

annotation (graph)

A text window added to a graph canvas. See graph.

application shell framework

ATF framework containing core application services and interfaces needed for applications with a GUI. It includes the components `ControlHostService`, `CommandService`, `SettingsService`, and `StatusService`, or components of similar functionality.

ATGI

Sony intermediate file format (`.atgi` extension) for exchanging digital assets among graphics software applications. See Collada.

attribute (DOM)

Information describing a type. For example, a type that defines graphical objects could have "x" and "y" attributes to specify a coordinate location on a canvas.

attribute (MEF)

A C# decorator for programming entities. In MEF, a class or item can be decorated with an attribute indicating information MEF uses. Attributes can designate that items are to be imported or exported, for instance.

C

catalog (MEF)

A container class holding components that MEF can compose for an application. There are several kinds of MEF catalogs, deriving from the base class `ComposablePartCatalog`. ATF mainly uses `TypeCatalog`, which creates a catalog from a collection of component types. See [compose](#).

circuit

A graph where nodes are circuit elements with pins, such as OR gates, and edges are wires connecting pins. See [graph](#), [pin](#).

Collada

Open standard file format (.dae extension) for exchanging digital assets among graphics software applications. The name comes from Collaborative design activity. See [ATGI](#).

command

An action invoked from a menu item or tool strip button.

command client

Client class that implements the `ICommandClient` interface for a command. In addition to performing a command, this interface provides methods to determine whether the command can be performed and to update the command state. See [command](#), [command state](#).

command group

Group of commands with related functions, both in menus and tool strips. For example, File > Save and File > SaveAs are closely related commands, grouped together. See [command](#).

CommandService

Component in the application shell framework that provides a service to handle commands in menus and tool strips. See [command](#) and [application shell framework](#).

command state

The `CommandState` class contains properties describing a command's state: the menu item name and whether its menu item has a check mark. See [command](#).

command visibility

Where command is visible, as on menus, tool strips, and so on, described by the `CommandVisibility` enum in the `CommandState`. See [command state](#).

component (MEF)

An object or a C# type that is added to a MEF container, so that the component can discover and use other components without having a hard-dependency between them. See [compose](#).

complex type (DOM)

A type describing elements with attributes and/or child elements. See [type definition \(DOM\)](#), [simple type](#).

compose, composition (MEF)

The process of putting together components into a cohesive whole in an application, where the components meet each others' import and export contracts, specified by their attributes. See [component](#).

connection (graph)

The entity or edge connecting nodes in a graph. See [edge](#), [graph](#).

context

An environment or interrelated conditions in which something exists or occurs. ATF has interfaces and classes to provide services for an application's data and operations in various contexts. The term context is often used to refer to an instance of a context interface or class. A context interface or class can be very general, working with different kinds of data.

context framework

ATF framework that tracks and works with application contexts. See context.

context registry

Component that tracks all context interface or class objects in an application, including the active context, maintaining a collection of context objects. A context registry implements `IContextRegistry`.

control adapter

Class derived from `ControlAdapter` that adds capabilities to an adaptable control, such as rendering, view changing, or selection. See adaptable control.

control client, control host client

Client class that implements the `IControlHostClient` interface for a control. The client specifies the control's behavior, if any, when the control gains or loses focus, or is closed. The control client is specified when registering a control with the `ControlHostService`. See `ControlHostService`.

ControlHostService

Component in the application shell framework that implements `IControlHostService` and is responsible for exposing client controls in the application's main form. See application shell framework.

D

data model

A model of user data in an application specifying the types of application data. See type definition (DOM).

data persistence

See persistence.

define extension (DOM)

Associating a data type with a DOM extension. A type can have any number of DOM extensions defined for it. See DOM extension, initialize extension.

Direct2D

Microsoft 2D vector graphics application programming interface (API).

document

A file containing application data. A class representing documents often implements the `IDocument` interface.

document client

Class that does the work of handling a document and implements `IDocumentClient` for operations on the document, mainly file-related, such as open, save, and close.

document framework

ATF framework that enables application components to track documents that a user is working on and which document is currently active.

Document Object Model (DOM) framework

A modeling, management, and persistence framework that provides data management for loading, storing, validating, and managing changes to application or game data. 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.

document registry

A component that tracks all documents an application has open. A document in this context is any object implementing `IDocument`.

DOM adapter (DOM)

An ATF DOM extension derived from the `DomNodeAdapter` class, which provides an infrastructure for connecting an extension to its underlying DOM node, as well as many useful methods and properties for DOM extensions. See DOM extension.

DomExplorer

A MEF component you can use to visualize the contents of a DOM node tree. See Dom node, component (MEF).

DOM extension (DOM)

A class that extends or adapts a `DomNode`'s data type to another type. If you adapt a `DomNode` to a DOM extension, the `DomNode` is dynamically cast to the DOM extension class's type. An extension can also listen to events on Dom nodes. See DOM adapter.

DomGen (DOM)

A utility that, given an XSD type definition file, generates a set of type metadata classes you can reference — instead of names inside the type definition file.

DOM metadata (DOM)

Classes that represent data associated with types. These classes can be used to provide information about the primitive data type associated with a type or the type's attributes, for instance. The `DomNodeType` metadata class represents the type of a `DomNode`.

DomNode, DOM node (DOM)

A node in a tree that represents a piece of application data. Each node has a `DomNodeType` that specifies the type of the node's data. See DOM node tree.

DomNodeAdapter (DOM)

See DOM adapter.

DOM node tree (DOM)

A tree of DOM nodes representing all the application's data. See Dom node.

DomNodeType (DOM)

Metadata class representing the type of a `DomNode`. See Dom node and DOM metadata.

E

edge (graph)

A connection between nodes in a graph. See connection, graph.

entity-relationship modeling (DOM)

A way of representing entities and the relationships between them. For example, DOM types have attributes associated with them.

export (MEF)

An attribute indicating that a component is making a class, property, or some item available to other components. See import.

G

ghost (timeline)

When a timeline object is moved, a ghosted copy of the timeline object moves to indicate the possible positions of the object. See [timeline](#).

graph

Collection of nodes with connecting edges. Graphs can be specialized to other forms, such as circuits and statecharts. ATF offers a rich facility for handling graphs. See [circuit](#), [statechart](#).

GridPropertyEditor

Property editor component that displays items with properties in a spreadsheet type control, allowing you to see the properties of all selected items. See [property editor](#), [PropertyEditor](#).

group (graphs)

Group of circuit items treated as a single element in the circuit. See [circuit](#).

group (timeline)

Container for zero or more tracks in a timeline. See [timeline](#) and [track](#).

H

hit

A result of a picking operation to click on some object. See [pick](#).

hit record

Encapsulated results of a hit from a picking operation to click on some object. See [hit](#) and [pick](#).

I

ID attribute (DOM)

An attribute to uniquely identify objects of a given type.

import (MEF)

An attribute indicating that a component needs a class, property, or some item from another component. See [export](#).

initialize extension (DOM)

Associating a `DomNode` with a DOM extension that is defined for the type of the `DomNode`. A node can have any number of DOM extensions associated with it. After initialization, a DOM node can be treated as an instance of the adapter's class. The extension can also listen for events on the `DomNode` and its children. See [DOM extension](#), [define extension](#).

instancing framework

ATF framework that works with object instances that can be edited, that is, copied, inserted, or deleted. It enables inserting objects, such as typing text onto a page, pasting from a clipboard, or dropping a circuit element onto a canvas.

interval (timeline)

An event of zero or greater time duration on a single track in a timeline. See [event](#), [timeline](#), and [track](#).

K

key (timeline)

An event of zero-length time duration on a single track in a timeline. See event, timeline, and track.

M

Managed Extensibility Framework (MEF)

A Microsoft library for creating extensions or components that can discover each other with no configuration required, avoiding hard dependencies.

manipulator (timeline)

Class to manipulate timeline objects, such as intervals, in a timeline control. See timeline.

marker (timeline)

An event of zero-length time duration on all tracks in a timeline. See event, timeline, and track.

master (graphs)

A collection of circuit elements that can be copied and reused. Changing one master changes them all. See circuit.

MEF component

See component (MEF).

metadata

Data describing other data. In the ATF DOM, metadata classes provide information about types in the data model. See DOM metadata.

N

node

A data point (DomNode) in a tree of application data. An item to be connected in a graph. See DomNode, graph.

P

palette

Container for items that can be dragged and dropped onto an application, typically onto a canvas control.

part creation policy (MEF)

An attribute indicating how a component is used by its importers. A component may be "Shared", one copy for all importers; "NonShared", one copy for each importer; or "Any", either "Shared" or "NonShared".

persistence

Storing application data, usually as a document, so that it can be later recalled and used or edited. Persistent data is most frequently stored in files. See document.

pick

An operation to click on some object that is then hit. See hit.

pin

A connection point on a circuit element in a circuit graph. See circuit, graph.

property

An attribute of a data object, which may be specified in a data model for the object's type. See attribute (DOM).

property descriptor

Metadata class describing a property/attribute of a data type. It contains information about the property, such as its name, description, and an appropriate value editor for the property, given its data type. Used by property editors. See value editor.

property editing framework

ATF framework that works with contexts that allow properties to be edited by controls. ATF offers a variety of property editor controls to edit object properties of various types. See attribute (DOM), property.

property editor

Control showing properties of selected items, containing value editing controls. ATF provides the `PropertyEditor` and `GridPropertyEditor` components for property editing.

PropertyEditor

Property editor component that displays items with properties in a two-column, multi-line control, showing properties of one selected item. See property editor, GridPropertyEditor.

prototype

A collection of items that can be copied as a named prototype and then be copied back into the application for reuse. See template.

R

reference

A relationship between arbitrary elements in a data model. In the ATF DOM, a reference can be an internal reference from one DOM node to another or an external reference to an external resource, such as a URI.

register command

Add a command to the user interface that can be executed from a menu item or tool strip control.

restriction (DOM)

A limitation on data type values that validators in the DOM can automatically check. Restrictions can be defined in the type definition. See validator.

S

schema (DOM)

A set of type definitions, usually in the XML Schema Definition Language. In the DOM, usually synonymous with type definition. See schema loader, type definition (DOM).

schema loader, schema type loader (DOM)

Class that reads a type definition file and converts types to metadata objects containing the types' information. Typically inherits from `XmlSchemaTypeLoader`. See schema, `XMLSchemaLoader`.

SettingsService

Component in the application shell framework that manages user-editable settings (preferences) and the persistence of these application settings. See application shell framework.

simple type (DOM)

A primitive type, such as integer and float, date and time, and string, as well as a special type like URI. See type definition (DOM), complex type.

statechart

A graph where nodes are states and edges are transitions, as in a state machine. Also known as a state transition diagram. See graph.

state transition diagram

See statechart.

StatusService

Component in the application shell framework that provides a global status text panel on the application's StatusBar and adds a display to show an operation's progress. See application shell framework.

sub selection

Selection inside another selection. For instance, a selected attribute of a selected item, as might be highlighted in a property editor.

T

template

An item that can be copied as a named template and then be copied back into the application for reuse. See prototype.

timeline

Graphical representation of a time-sequence. Timelines contain groups that can contain tracks that can contain events, which can be intervals, keys (zero-length intervals) and markers (zero-length events that are on all tracks). See event, group, interval, key, marker, and track.

track (timeline)

Container in a timeline's group containing zero or more events. See event, group, and timeline.

TypeConverter

Class in `System.ComponentModel` that value converters derive from. See value converter.

type definition (DOM)

A definition of a type in a data model. A type definition can be represented in a variety of languages, including XML Schema Definition Language, which is the default in ATF. See data model, type definition file (DOM).

type definition file (DOM)

A file containing type definitions for a data model. See data model, type definition (DOM).

type loader

See schema type loader.

U

UITypeEditor

Commonly used base class in `System.Drawing.Design` for a value editor with a value editing control that drops down a control or displays a modal dialog. See value editor, value editing control.

V

validator (DOM)

A DOM adapter that performs automatic validation of DOM data as it changes. See DOM adapter.

value converter

Class that converts a property value between its data representation in a value editor and a type displayable in a value editing control. Typically derived from `System.ComponentModel.TypeConverter`. See TypeConverter.

value editing control

Control class displaying a property value that interacts with a user to change the value. Typically associated with a value editor. See value editor.

value editor

Class associated with a property to edit its value, typically associated with one or more value editing controls. May derive from `UITypeEditor`. See value editing control.

W

Windows Forms (WinForms)

The graphical application programming interface using the Windows API and GDI, included as part of the Microsoft .NET Framework. You can create Windows Forms applications with ATF. See Windows Presentation Foundation (WPF).

WinForms

See Windows Forms.

Windows Presentation Foundation (WPF)

The graphical subsystem for rendering user interfaces in Microsoft Windows-based applications using DirectX instead of GDI and the Windows API. You can create WPF applications using ATF. See Windows Forms (WinForms).

wire

A connection between pins in a circuit graph. See connection, circuit.

X

XML Schema Definition Language (XSD), XML Schema

An XML-based language for specifying the structure of an XML document. In the ATF DOM, can be used to define the data model's types. For a reference, see <http://www.w3schools.com/schema>. See data model, type definition (DOM).

XmlSchemaTypeLoader

ATF DOM class that does much of the work of loading and managing an XML schema. See schema type loader.
