Interactive Television Applications using DSL Tools

Steve Cook
Microsoft Corporation
7 J J Thomson Avenue
Cambridge CB3 0FB, UK
+44 1223 479717
steve.cook@microsoft.com

ABSTRACT
This paper describes how to use the Microsoft DSL Tools to construct the Interactive Television Applications system as an example for the Model Driven Development Tools Implementers Forum at TOOLS'07.

Categories and Subject Descriptors
D.1.7 [**Visual Programming**]: Domain Specific Languages.

General Terms
Design, Languages.

Keywords
DSL Tools, Domain-Specific Languages, Visual Studio.

1. INTRODUCTION
The description of the Interactive Television Applications example was circulated in advance of the workshop [1] and will not be repeated here. The purpose of this paper is to describe the steps necessary to implement a simple version of the application using the Microsoft Domain-Specific Language Tools [2].

2. APPROACH
Interactive TV applications are constructed from three kinds of content: menus, votes and text pages. Menus provide the ability to navigate to nested content which can include menus. An application consists of a top content element, which would typically but not necessarily be a menu offering navigation to nested content.

An interactive TV application consists physically of an XML file, intended for inclusion in the digital stream sent to the TV. Obviously of the complete set of possible XML files, only a vanishingly small subset are valid TV applications, so native XML is not an appropriate medium for the producers and journalists who create these applications. The purpose of the exercise, then, is to design a diagrammatic language that enables producers and journalists to create TV applications without having to worry about the XML representation.

This is achieved by using the DSL Tools to create a modelling tool (a DSL) hosted in Visual Studio 2005. The producers and journalists can launch this tool, use its modelling facilities to create the application, and use the tool to generate the required XML.

For the purpose of the exercise, we'll pass over the question of whether Visual Studio is an appropriate host for an application intended for TV producers and journalists. Possibly it is not; but since the DSL Tools are (currently) solely intended for creating modelling tools that run within Visual Studio 2005, admitting this would immediately invalidate the exercise. Therefore we restrict the scope of our considerations to how to create the tool, and how to generate the XML files from its models: the purpose being for illustration of how the DSL Tools work, rather than the creation of a realistic application.

Having studied the use cases in the brief [1] and gained a general understanding of the problem, the first step in designing the DSL Tools version of the application – which we will henceforward call the TV Application Designer – is to define the *domain model* (many readers might call this the meta-model). This is a model of the concepts in the domain: menus, votes and text, and their relationships in a valid application.

We then define shapes that correspond to these concepts, so that they can be manipulated in the target tool. We define mappings between the concepts and their shapes. We define additional metadata that governs aspects of the user interface of the target tool, such as the toolbox from which new elements can be created. A couple of aspects of the target tool have to be created by dropping into C# code, and we do this. Finally, we create a template from which the required XML file can be generated.

The remainder of this paper describes each of these steps in more detail. For much more detail about the DSL Tools see the book by the author and colleagues [3].

3. THE USE CASES
Here we list the use cases together with a short analysis on how each one is to be handled in this exercise.

3.1 Case 1
"A producer would like to build an application for the soccer world cup finals, listing the teams and information about each, and allowing viewers to vote for the most likely winner."

The TV Application Designer provides the ability to create menus, votes and text pages, and configure them into applications. The top-level menu of this particular application will be connected to a submenu providing information about the teams, a voting page, and other information.

Conference'04, Month 1–2, 2004, City, State, Country.

3.2 Case 2

"As Case 1, but the teams should be listed by group (e.g. four teams in three groups). As the teams are known before their division into groups by lots, it should be possible to define the content for the teams first, and quickly add the structure of the groups, so that the application can be running for users as soon as possible during the program that broadcasts the division by lots."

Each group can be represented by a menu. Team information should be able to exist without having to be accessible via a menu. We take it from this that a content element should be potentially accessible from any number of menus including 0, and that if it does appear in multiple places it will be replicated in the output stream.

3.3 Case 3

"A producer would like to use a page of text to provide analysis of the recent events in a rugby match. A journalist with a laptop will need to change the text on the page throughout the match. The user interface used by the journalist should not allow him to change the structure of the whole application, only edit the text."

We address this requirement by storing the contents of text pages in separate text files, and reading those files to include their contents in the output stream. Keeping the journalist from changing the structure of the application is achieved simply by not deploying the target tool to the journalist, and instead offering him a text editor and asking him to deposit the resulting file in a well-known location. When the journalist informs the producer that he has done this, the producer can regenerate the application based on the new file.

3.4 Case 4

"A producer has decided that the wording of a particular text page was better before the last set of changes, and would like to revert it."

This is achieved simply by the producer keeping copies of the various file versions using some appropriate naming scheme and selecting the desired ones.

4. GETTING STARTED

To create a new DSL with the DSL Tools, the user must create a new Visual Studio solution populated with the components needed to build a DSL. The Visual Studio "New Project" dialog is launched selecting the Domain-Specific Language Designer template. This is a Visual Studio project template that creates a solution and pre-populates it with all of the parts needed to build a DSL. The template runs a wizard to collect data from the DSL author about the new language, including its name and file extension. At this point the DSL author can choose a pre-existing language as a starting point for the new language: the available options are Class Diagrams, Component Models, Task Flows and the Minimal Language. The first three of these give rich starting points for three popular flavours of modelling language: if the desired language is similar to one of these, considerable effort can be saved by choosing the right template as a starting point.

For the current simple example, the author chose the minimal language. This provides a very simple starting point, although it is a complete and valid language. At any point in the development of a DSL, if the language is valid then the code for the target tool can be generated, and tried out simply by pressing the F5 key which will build the language and launch it in a new copy of Visual Studio.

5. DOMAIN MODEL

The domain model for a modelling language is constructed using the DSL Designer, which is a modelling tool incorporated in the DSL Tools. This tool uses an automatically laid out tree-structured notation where domain classes are represented by rounded rectangles and domain relationships are represented by square-cornered rectangles connected by lines to domain classes. The core of the domain model is illustrated in Figure 1.

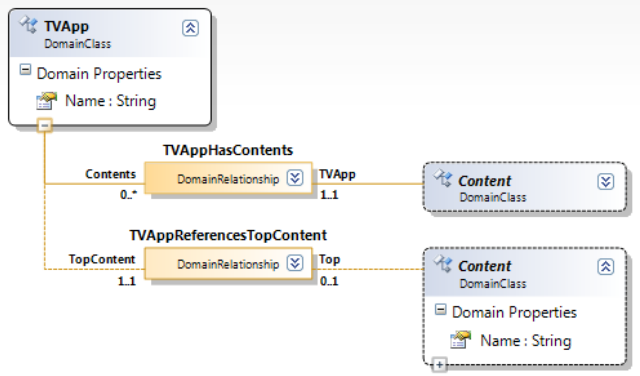

Figure 1. Core of Domain Model

The term *domain class* denotes an element within the domain model that has class-like semantics. Domain classes are implemented by C# classes generated according to a pattern defined by the DSL Tools. Instances of these classes, called *model elements*, live in the target tool and are managed by a component called the Store, which provides various services including undo-redo, reflection, rules and events.

Similarly, a *domain relationship* is an element within the domain model that has binary relationship-like semantics. In fact a domain relationship is also a domain class: every instance of a domain relationship (every *link*) can do everything that a *model element* can.

Armed with this knowledge we can see that the domain model in Figure 1 defines two domain classes called **TVApp** and **Content**. **TVApp** is the *root class*, and will have exactly one instance at model run time. **Content** is an *abstract* domain class (its name is in italics). Two relationships connect these classes: the **TVAppHasContents** relationship is a one-to-many *embedding* relationship, and the **TVAppReferencesTopContent** is a *reference* relationship that picks out a single content element which will be the top level content of the application. This is not the place to go into the fine details of the notation which are extensively documented in [2] and [3]; the typical reader will be able to deduce the essentials from the diagram and from the brief description above.

Every model in the target tool will be structured as a tree by instantiating the *embedding* relationships. This is an important property enforced by the DSL Tools, which ensures that the model can be saved in an XML file, and viewed through a model explorer. This, in fact, constitutes the semantics of embedding: it corresponds to element nesting in the XML file, and node nesting in the model explorer.

The domain class **Content** appears twice in the figure. This is because the domain model is a graph, but is laid out as a tree. The lower appearance has been selected as the full definition.

The remainder of the domain model is shown in Figure 2.

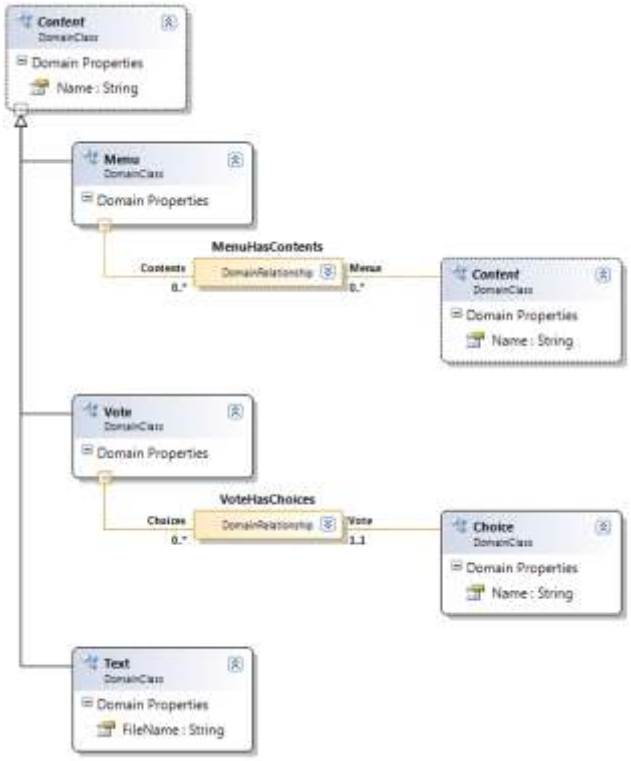

Figure 2: Subclasses of Content

This shows three subclasses of **Content**, called **Menu**, **Vote** and **Text**. **Menu** is related to **Content** through a many-to-many reference relationship called **MenuHasContents**. **Vote** is related to another domain class **Choice** through a one-to-many embedding relationship called **VoteHasChoices**.

Every domain class has a domain property called **Name**, either directly or via inheritance from **Content**. This follows straightforwardly from the requirements of the application. These properties carry additional metadata that mark them as an element name, which allow them to be automatically specially treated in the target tool. In general it is not necessary for a domain class to have a name property, but such properties are very useful for making the target tool have a simple user interface.

The **Text** class has a property **FileName**. This holds the pathname of the file that holds the textual content to be incorporated into the application.

6. SHAPES AND CONNECTORS

Shapes and connectors are defined in a separate area of the DSL Designer. The shapes defined for the example are shown in Figure 3. Every **Shape**, **Connector** and **Diagram** definition is also a domain class definition (for meta-thinkers, the domain model for the DSL designer has **Shape**, **Connector** and **Diagram** as subclasses of **DomainClass**).

Every DSL must have a definition for its **Diagram**. There will be one instance of the **Diagram** associated with a target model. For this example, the **Diagram** is created automatically and requires no additional work.

There are several different kinds of **Shape** available in the DSL Tools. For this example, two kinds are used: **GeometryShape** for representing menus and text pages, and **CompartmentShape** for representing votes. The other available shape kinds are **ImageShape**, **PortShape** and **Swimlane**.

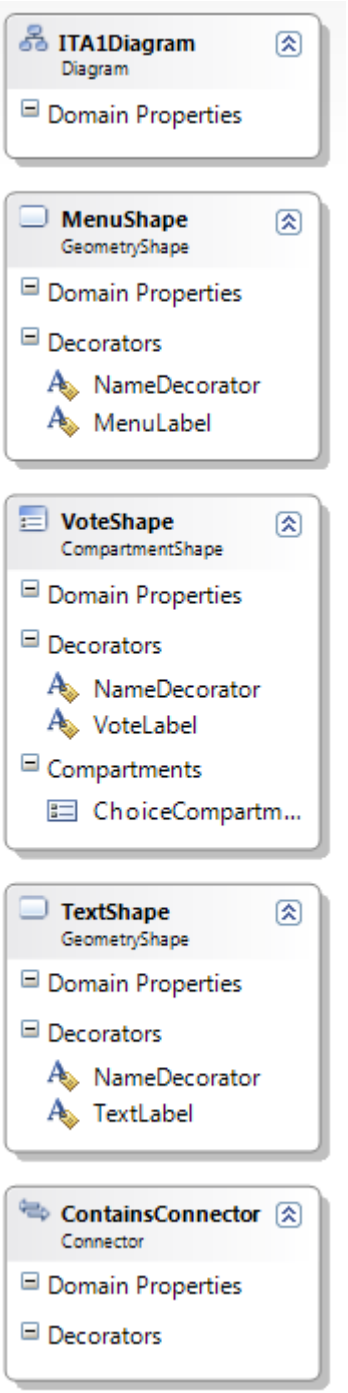

Figure 3. Diagram Element Definitions

A **GeometryShape** has a geometrical outline (rectangle, rounded rectangle, ellipse). The DSL author can set its display options such as size, line thickness, fill colour, outline colour and so on. A **CompartmentShape** contains a header section, and an expandable set of compartments, each of which can contain text derived from the model. Again there are numerous display options available, including whether the compartment section itself has a header.

Each shape definition incorporates a set of **Decorators**. A decorator is a small area positioned inside (or near) the shape which can contain additional content such as text, icons or an expand-collapse control. For the **MenuShape**, there is a text decorator called **MenuLabel** that labels the shape with the text "Menu", and one called **NameDecorator** that shows the name of the represented menu. The **VoteShape** and **TextShape** are similarly configured.

The **VoteShape** is a **CompartmentShape**, and has a single compartment, in which the set of available choices for the vote will be displayed.

A connector is defined to represent the line that associates a menu with its items of content. By analogy with UML containment this line is defined with an arrowhead at its target end, and a black diamond adornment at its source end. These adornments are set as properties of the **ContainsConnector**, as well as its thickness, colour and so on.

7. SHAPE MAPPINGS

Each shape is mapped to a domain class. A Diagram Element Map tool on the DSL Designer's toolbox enables the DSL author to create these maps by drawing a connector representing the map between the domain class and the corresponding shape. The maps are represented as lines on the DSL Designer's design surface (there is a display option to hide or show these lines).

Selecting a map line and showing the DSL Details window provides options for configuring the map. Configuring the **NameDecorator** decorators to display the **Name** properties of the associated classes is straightforward through the Decorator Maps section of the DSL Details window, which provides a control to select a property to associate directly with the decorator, as well as other more complex options.

The Compartment Maps section of the DSL Details window is used to configure the **ChoiceCompartment** to display the names of the choices associated with a vote. The compartment itself is associated with the set of elements reached via the relationship **VoteHasChoices**. Mapping the compartment to this set enables the target designer to include the ability to add a new choice to the set. This mapping is configured by setting the "Displayed Elements Collection Path" for the compartment to be `VoteHasChoices.Choices/!Choice`.

Paths such as this one are used in the diagram element maps to navigate between objects and relationships. The `VoteHasChoices.Choices` segment of the path denotes a hop from an element to a link: here the element is a **Vote**, and the hop occurs across the role whose property name is **Choices**. The `!Choice` segment is shorthand for `VoteHasChoices!Choice`, and is a hop from a link to the element that plays the **Choice** role. Read together, these segments describe the set of **Choice** elements that are associated via the **VoteHasChoices** relationship with the **Vote** at hand: this is the set represented by the compartment.

Having configured this path, the DSL Author then uses a pull-down control to select the **Name** property of the **Choice** class to be displayed in the compartment.

8. TOOLBOX

There's one more job to do to complete the first iteration of the TV Application Designer, which is to define its toolbox. The DSL Explorer in the DSL Designer has a node called Editor, and under this is a node called Toolbox Tabs. Here the author can create one or more toolbox sections, and in each section place a tool associated either with a domain class or a relationship. Each tool can be configured to have a name, a caption, a tooltip that will appear when the user hovers over the tool, a help keyword, and icons for the tool on the toolbox and the cursor to be displayed as the tool is used. For the TV Application Designer, tools are defined to create a **Menu**, a **Vote**, a **Text** and a **MenuHasContents** relationship.

At this point, the TV Application Designer can be launched for debugging. Pressing the "Transform All Templates" button in the Solution Explorer of the DSL solution causes all of the code for the TV Application Designer to be generated. Building this code and pressing the F5 key causes the TV Application Designer to be launched in a Debugging context that was automatically set up when the DSL solution was originally created. This designer enables the author to create models of TV Applications such as that in Figure 4.

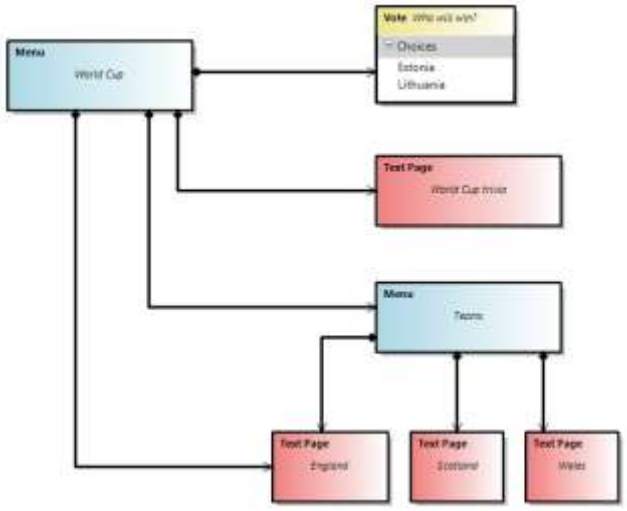

Figure 4. A Simple TV Application

9. VALIDATION

Referring back to the domain model, notice that the reference relationship **TVAppReferencesTopContent** has a minimum multiplicity of 1. This means that if no element of the model is selected to be the top element then the model is not valid. This kind of validation is called multiplicity underflow checking. It does not happen by default though: to enable it, the DSL author

must explicitly enable validation for the TV Application Designer. This is done by selecting the Validation node in the DSL Explorer, which appears adjacent to the Toolbox Tabs node. Setting the "Uses Load", "Uses Menu", "Uses Open" and "Uses Save" properties all to true means that models in the TV Application Designer will be validated whenever they are loaded, opened or saved, and also places a Validate entry on the context menu that appears when the user right-clicks over the diagram or any shape. If no top element is selected for the model, the error message "TVApp has no TopContent" will be displayed whenever validation occurs, and the user will be warned if they try to save the model in this state.

More complex validations can be written using code. There are two kinds: soft validations, that can be triggered from the the load, open, save and menu events, and hard validations that can be invoked to stop user gestures from completing. Multiplicity overflow is automatically wired into the DSL Tools as hard validations, while as we've seen, multiplicity underflow is soft and configurable.

10. CODE CUSTOMIZATIONS

So far we've created a tool that can create models of TV Applications. We still need to enhance it to add the capability to attach text files to the text pages and launch a text editor on those files. This requires the designer to be customized using a modest amount of C# code, using the APIs defined in the Visual Studio SDK and the .NET Framework. Having made these customizations, right-clicking over one of the Text shapes produces the context menu shown in Figure 5.

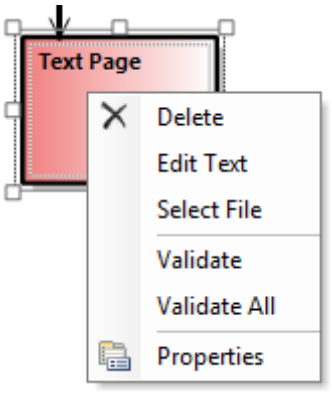

Figure 5. Customized context menu

Choosing the "Select File" entry launches an instance of the `System.Windows.Forms.OpenDialog` class, which is used to choose the text file to be associated with the page: the filename is stored in the **FileName** property. Choosing the "Edit Text" entry launches the Notepad program on the chosen file; if no file has been chosen then a new one can be created at this point. This provides a simple user interface for associating the text pages with existing files or for creating new ones.

11. GENERATING THE XML

When the user of the TV Application Designer is happy with their application, they need to generate the XML content that will be inserted in the digital stream to the TV. In the DSL Tools, generation of this kind can be done simply using the Text Templating engine. A template file is created that navigates the model and emits boilerplate text interspersed with text calculated from the contents of the model.

Here is the body of the template. The first line of it is boilerplate – it is the XML prolog that will be emitted as it stands into the output stream. The second line is boilerplate that contains, within <#= and #> brackets, an expression in C# that will be evaluated in the context of the template execution. The expression `this.TVApp` denotes the root instance of the model, which is an instance of the C# class `TVApp` generated from the domain class **TVApp**.

The following lines are bracked within <# and #> and consist of C# code executed at this point in the template expansion. This code calls the `GenContent()` method. Finally another line of boilerplate completes the template.

```
<?xml version="1.0" encoding="utf-8"?>
<TVApp name="<#= this.TVApp.Name #>">
<#
  if (this.TVApp.TopContent != null)
  {
    GenContent(this.TVApp.TopContent);
  }
#>
</TVApp>
```

Methods can be defined in the template after the body. Below, for example, is the `GenMenu()` method. The sections bracketed in <#+ #> are code that defines the method; the sections not so bracketed are boilerplate that will be emitted as the method is executed. Because this template is recursive (`GenContent()` calls `GenMenu()` and vice-versa) the methods `PushIndent()` and `PopIndent()` are used to control how the output is formatted.

```
<#+
public void GenMenu(Menu m)
{
  this.PushIndent("  ");
#>
<Menu name="<#= m.Name #>">
<#+
  foreach (Content element in m.Contents)
  {
    GenContent(element);
  }
#></Menu>
<#+
  this.PopIndent();
}
#>
```

The result of executing this template on the model in Figure 4 is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TVApp name="World Cup 2010">
  <Menu name="World Cup">
    <Vote name="Who will win?">
      <Choice name="Estonia">
      <Choice name="Lithuania">
    </Vote>
```

```
    <Text Name="World Cup trivia">Trivia
text
  2nd line of text
  3rd line of text
  etc
  </Text>
    <Menu name="Teams">
      <Text Name="England">England
    </Text>
      <Text Name="Scotland">Scotland
    </Text>
      <Text Name="Wales">Wales
    </Text>
    </Menu>
  </Menu>
</TVApp>
```

12. CONCLUSION

This paper has described how to use the Microsoft DSL Tools to construct the Interactive Television Applications system. The following topics were addressed: unfolding the language from a chosen starter template; designing the domain model; creating the shapes; mapping shapes to concepts; creating the toolbox; validation; code customizations; generating the output artifacts. Extensive further detail can be obtained via the book [3].

13. REFERENCES

[1] http://www.dsmforum.org/events/MDD-TIF07/InteractiveTVApps.pdf

[2] http://msdn.microsoft.com/vstudio/DSLTools/

[3] Cook, S., Jones, G., Kent, S. and Wills, A.C *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Microsoft .NET development series, Boston, MA, 2007.