Professional SharePoint 2007 Development

Chapter 11: Building Document Management Solutions

ISBN-10: 0-470-11756-7
ISBN-13: 978-0-470-11756-9

11

Building Document Management Solutions

By John Holliday

The term “document management” has become a catch-all phrase for anything having to do with documents in an enterprise setting. It is an overly broad term that covers many different aspects of managing documents, from access control to version control to the auditing, review, and approval of content. To understand what document management means in the SharePoint environment, it helps to consider the evolution of document management systems over the last decade or so. It also helps to appreciate the value that SharePoint provides as a development platform for document management solutions.

Early document management systems were focused primarily on keeping track of revisions to documents that involved multiple authors, and operated in a manner similar to source code control systems. Individual authors checked out documents, thereby locking them so that other authors could not overwrite their changes. System administrators could specify who had permission to view or edit documents, and could generate reports of document activity. Other functions included the ability to automatically number each major or minor revision and revert at any time to a specific version of the document, generating the final content from information stored within the database.

The notion of metadata became a key characteristic of legacy document management systems. *Metadata* is information about a document, as opposed to the document content itself. For example, the current version number is an example of metadata, since it is information about the document. Other examples are the title, subject, comments and keywords associated with the document.

Most document management systems store document metadata in a central database. In fact, many of the early document management systems were written as database applications. This worked well at a time when the only business process being modeled was the generic document revision cycle. It starts to break down, however, when you want to model other business processes.

This is where SharePoint emerges as a superior platform for developing document management solutions. SharePoint refines the notion of document metadata to distinguish between system, class, and instance metadata. System-level metadata is maintained internally by SharePoint for all documents. Class-level metadata is stored within the SharePoint database for a given document library or content type and can be customized easily to include domain-specific information. Instance-level metadata is stored within each document instance as a set of document properties, and moves along with the physical document. This is especially important for managing documents in disconnected environments.

Because the term document management is so imprecise, it follows that the idea of what comprises a document management solution might be different depending on the context. One solution might implement a custom document approval process. Another might apply a business rule to determine who has access to a given set of documents. Yet another might apply a set of business rules that capture information about milestones achieved as a document is being edited.

So, what is a document management solution and how do you go about building one in SharePoint? The best way to answer this question is to examine the tools provided by the SharePoint platform in the context of a real scenario.

This chapter develops a solution for managing project proposals. The solution distinguishes between fixed-bid and time-and-material projects and provides metadata for controlling the content of each proposal as it evolves through the revision cycle. The solution will include custom fields for specifying the fixed-bid amount and the estimated hours for time-and-materials projects.

While developing a document management solution, it quickly becomes evident that business rules control not only the type of metadata that can be associated with a document but the range of acceptable values for each field. When a document is first created and at any stage during the revision cycle, the business rules examine the metadata and perform the appropriate actions to ensure that the document state is valid. This collection of business rules is often called a *content management policy* because it summarizes your policies and procedures for a particular type of document.

As you work through the chapter, developing the proposal management solution, you will explore SharePoint content types as a way to capture the essential characteristics of a document in one place. Content types offer tremendous advantages when building document management solutions; they allow you to encapsulate class-level metadata and associate it with code you can run on the server to implement custom business rules. The proposal management solution will include two policies: one to limit the acceptable bid amounts to a certain pre-approved range of values, and another to require a minimum number of estimated hours for time-and-material projects.

Finally, the chapter develops an extensible framework for defining document management policies using XML in conjunction with SharePoint event receivers. You also create custom commands that enable system administrators to use the STSADM command-line tool to administer proposal management policies after the solution is deployed.

Understanding the Document Lifecycle

Effective document management requires an understanding of the document lifecycle and the transformations that can occur as a document moves from one phase to the next. The basic document lifecycle consists of three phases: creation, revision, and publication. Each phase can be broken down into sub-phases that describe the different stages through which a typical document progresses over time (see Figure 11-1).

When a document is created, some information is needed to determine its initial content. You can call this *initiating metadata* for lack of a better term. At the very least, the initiating metadata for a SharePoint document consists of the location of the document template that should be used. Typically, the document template itself takes care of collecting any additional information that might be required to render the document in its initial state. For example, when creating a new proposal in Word, you might popup a dialog to interrogate the user prior to rendering the default document content.

After the document is created, its content is edited through one or more revision cycles. Throughout these iterations, various contributors may check out the document for editing and then check it back in until certain milestones have been achieved. The determination of whether a given milestone has been reached is often subjective, as there can be many different types of milestones, depending on the type of document and the kind of solution being developed. Nevertheless, to the extent you can identify such milestones, you can develop supporting metadata for them.

A typical requirement of a document management solution is to keep track of when changes are made and by whom. SharePoint supports this directly for all lists and list items. Another typical requirement is to enforce a particular numbering scheme for tracking versions, enforce checkout/checkin policies, or customize the way notifications are sent when changes are made.

Publication usually involves moving the finished document into a separate repository or routing it to one or more people for approval or review. As part of the publication process, it might be necessary to clean up the document content or convert it to a special file format for publication. For example, after publishing a document it might be necessary to convert it to PDF to ensure that it can be viewed by users who don't have Microsoft Word on their desktops.

Chapter 12 examines the publication phase and SharePoint's built-in support for automatic document conversion.

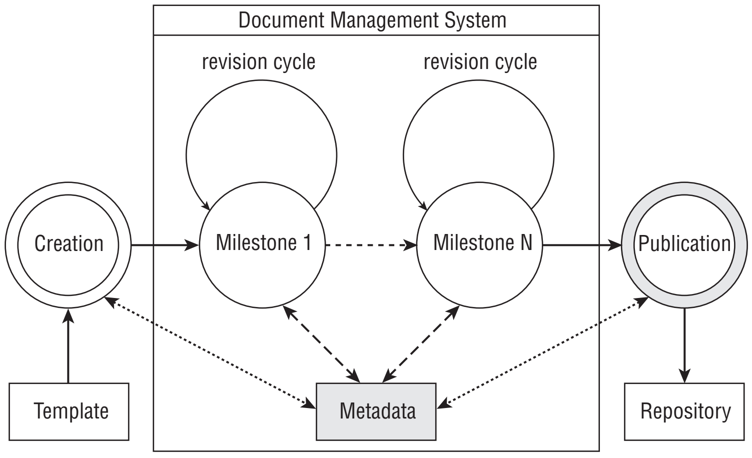


Figure 11-1

Defining Metadata Using Content Types

Metadata is the fuel that drives document management in SharePoint 2007, and the best way to work with document metadata is to define a content type. There are many benefits to using content types; the main one being that content types allow us to specify the custom fields needed to manage a document as it moves through the different stages of its lifecycle.

Solution developers are used to working with classes and objects, and properties and methods, where each class defines the properties and methods for instances of that class. They then create objects to represent instances of each class and invoke methods on those objects to apply business rules that retrieve or modify the state of the properties associated with each instance. Building document management solutions will be much easier if you can map the core elements (document, metadata, repository, etc.) onto familiar abstractions like *class* and *object* that you are used to working with.

SharePoint 2007 content types provide just such an abstraction. The content type acts as a sort of document class, defining the columns and event receivers that comprise each instance. The columns are like properties, and the event receivers are like methods. Take it one step further and say that the `ItemAdding` event receiver acts as a `constructor`, and the `ItemDeleting` event receiver acts as a *destructor* for each document instance.

The first step in defining a new content type is to determine from which of the built-in content types to derive the new content type. In object-oriented terms, you are choosing the base class for the new content type. SharePoint includes a number of default content types, all derived from the System content type, which serves as the root of the content type hierarchy.

Figure 11-2 shows some of the default content types and their identifiers.

SharePoint employs a special numbering scheme for identifying each content type, which it uses as a shortcut for creating new content type instances. Without such a numbering scheme, it might have been prohibitive to enable content type inheritance, since SharePoint would have needed to search through the database trying to resolve content type dependencies. This way, it only needs to examine the identifier, which it reads from right to left. For example, the Picture content type identifier is 0x010102, which SharePoint reads as id 02 (Picture) derived from Document (0x0101) derived from Item (0x01) derived from System (0x).

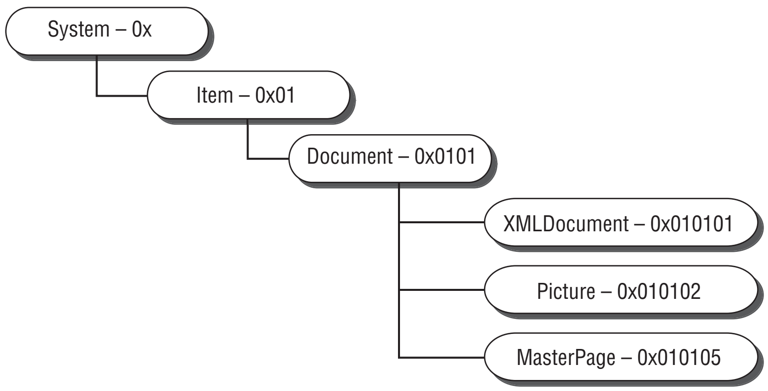


Figure 11-2

For custom content types that you define yourself, the identifier includes a suffix, which is the globally unique identifier (GUID) associated with our type, separated by 00 as a delimiter. For example, the project proposal content type defined later in this chapter has the ID

0x0101004A257CD7888D4E8BAEA35AFCDFDEA58C

Again, reading from right to left, you have 4A257CD7888D4E8BAEA35AFCDFDEA58C derived from Document (0x0101) derived from Item (0x01) derived from System (0x). The 00 serves as a delimiter between the GUID and the rest of the identifier, as shown in Figure 11-3.

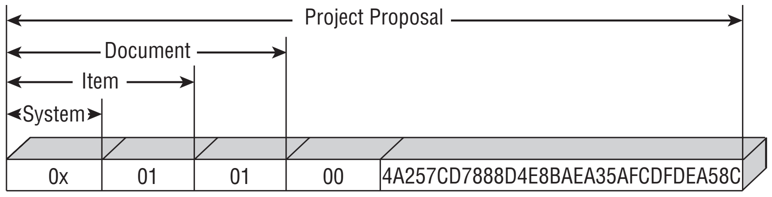


Figure 11-3

Each content type references a set of columns (also called fields), which compose the metadata associated with the type. It is important to note that content types do not declare columns directly. Instead, each content type includes column references that specify the identifiers of columns declared elsewhere within the SharePoint site. Column references are declared in XML using `FieldRef` elements.

Our project proposal content type is based on the built-in Document content type, which provides the following metadata fields:

- **Name** — The name of the file that contains the document content
- **Title** — The title of the document (inherited from the `Item` content type)

Next, you select from the built-in SharePoint fields to capture the common elements of a project proposal:

- **Author (Text)** — The author of the proposal
- **Start Date (DateTime)** — The date on which the project will start
- **End Date (DateTime)** — The date on which the project will end
- **Status (Choice)** — The current document status
- **Comments (Note)** — Additional comments
- **Keywords (Text)** — Keywords

In addition to the built-in columns, you need a few additional columns to complete the type definition.

- **ProposalType (Choice)** — The kind of proposal
- **EstimatedCost (Currency)** — The total cost of the proposed work
- **BidAmount (Currency)** — The proposed amount of the bid

- **EstimatedHours (Number)** — The total number of hours
- **HourlyRate (Currency)** — The proposed hourly rate

SharePoint provides two methods for declaring content types: using XML or using the Windows SharePoint Services object model. In actual practice, a hybrid approach is often useful. This is because while XML makes it easier to declare fields and other elements at a high level, it also makes it harder to work with the content type from elsewhere in your solution. Once the essential elements have been identified, the object model provides more control over how those elements are used and how they interact with one another. What you need is an easy way to declare the type while preserving your ability to add enhanced functionality through code.

The following sections explore both methods. You will use XML to declare the metadata for your custom project proposal content type and then the object model to control the behavior of each instance. As an alternative to using XML for the content type declaration, you will also explore ways to create content types entirely through code.

Declaring Content Types Using XML

To define a content type using XML, the following steps are required:

1. Create a content type definition file.
2. Create a field definition file that describes any custom fields.
3. Create a feature definition file that references the content type definition.
4. Install the feature into SharePoint.

The Content Type Definition File

Listing 11-1 shows the content type definition XML for the project proposal type. The content type ID is specified and indicates that the project proposal inherits the fields of the built-in `Document` type. The content type name "Project Proposal" is used to reference our type from code and from the SharePoint user interface.

Listing 11-1: Project proposal content type definition

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
        <!-- _filecategory="ContentType" _filetype="Schema"
_filename="contenttype.xml" _uniqueid="cff96a1e-6a52-4462-a3d0-d01471b8bfef" -->
        <ContentType ID="0x0101004a257cd7888d4e8baea35afcdfdea58c"
                Name="Project Proposal"
                Group="ProSharePoint2007"
                Description="A Content Type for Managing Project Proposals"
                Version="0">
                <FieldRefs>
                        <FieldRef ID="{246D0907-637C-46b7-9AA0-0BB914DAA832}"
Name="Author"/>
```

```
                        <FieldRef ID="{76A81629-44D4-4ce1-8D4D-6D7EBCD885FC}"
Name="Subject" />
                        <FieldRef ID="{9DC4BA7E-6C50-4e24-9797-355131089A2E}"
Name="ProposalType" Required="TRUE"/>
                        <FieldRef ID="{24A18FDA-927A-4232-88AE-F713FFD3FBB4}"
Name="EstimatedCost"/>
                        <FieldRef ID="{41495470-9EA3-46e0-9A34-0E0C3DE1A445}"
Name="BidAmount"/>
                        <FieldRef ID="{D46C0900-5617-414c-97E5-E5626DBC1495}"
Name="EstimatedHours"/>
                        <FieldRef ID="{79368859-EAF8-4361-8A9D-C3CBD9C88697}"
Name="HourlyRate"/>
                        <FieldRef ID="{64cd368d-2f95-4bfc-a1f9-8d4324ecb007}"
Name="StartDate" />
                        <FieldRef ID="{8A121252-85A9-443d-8217-A1B57020FADF}"
Name="EndDate" />
                        <FieldRef ID="{1DAB9B48-2D1A-47b3-878C-8E84F0D211BA}"
Name="Status" />
                        <FieldRef ID="{52578FC3-1F01-4f4d-B016-94CCBCF428CF}"
 Name="Comments" />
                        <FieldRef ID="{B66E9B50-A28E-469b-B1A0-AF0E45486874}"
Name="Keywords" />
                </FieldRefs>
        </ContentType>
</Elements>
```

Adding fields to a content type requires the use of `FieldRef` elements. This is because content types do not declare fields directly but instead refer to site columns that are defined globally within the current SharePoint execution context.

You will see shortly how site columns are declared, but for now note that each `FieldRef` element specifies the unique identifier of a separate Field element defined elsewhere within the solution. This includes both the built-in site columns that are shipped with Windows SharePoint Services, and any custom fields you define.

This presents a bit of a problem when building content type definition files. You already know the identifiers of the custom fields because you created them yourself. But for the built-in site columns, you first have to locate the appropriate identifiers that are recognized by SharePoint. When creating content types through the user interface, SharePoint looks up the field identifiers automatically. A bit of additional work is required when building a solution that installs custom content types at runtime.

The built-in site columns are declared in the `fieldswss.xml` file, which is located in the `12\TEMPLATE\FEATURES\fields` folder. To declare a content type based on built-in site columns, you must search through this file to find the field you want, and then copy the GUID from the file into your content type definition XML.

To simplify the process of looking up field identifiers, use a simple XSL style sheet to display the `fieldswss.xml` file as an HTML table. The style sheet shown in Listing 11-2 produces the table shown in Figure 11-4.

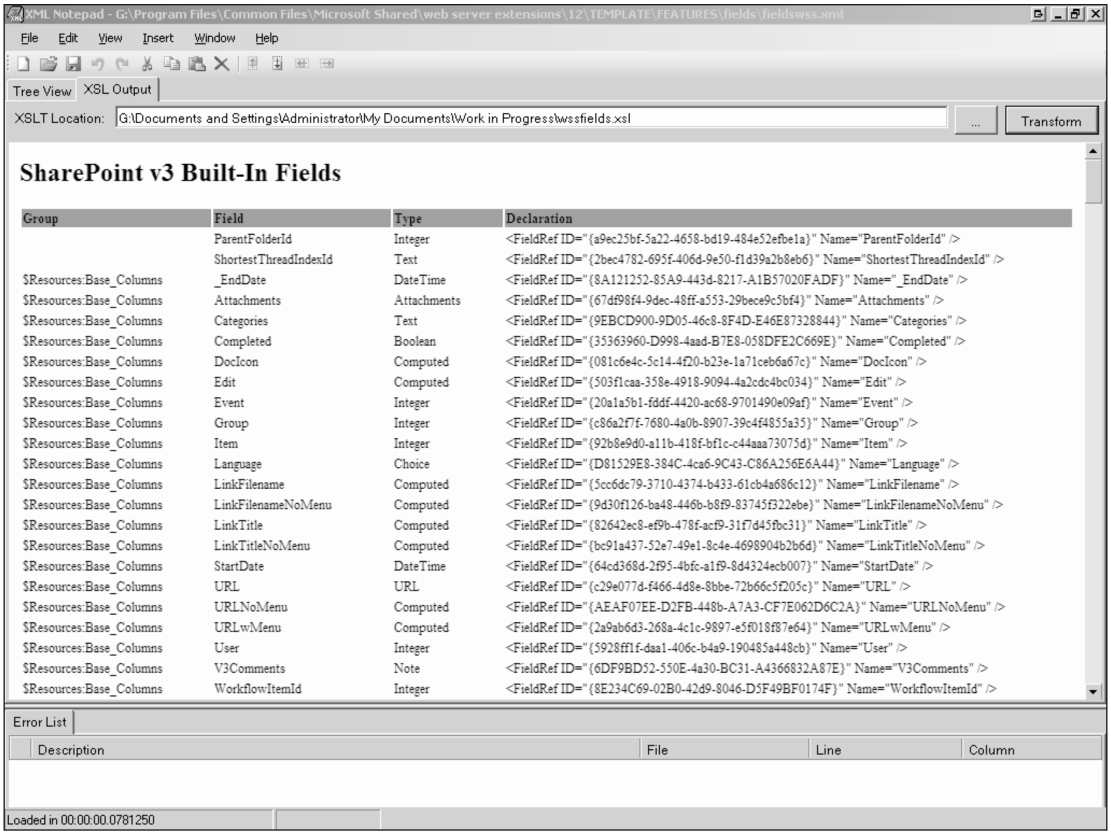
Listing 11-2: XSL style sheet to locate built-In field IDs

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:wss="http://schemas.microsoft.com/sharepoint/">
        <xsl:output method="html" version="1.0" encoding="utf-8" indent="yes"/>
        <xsl:template match="wss:Elements">
                <html>
                        <body>
                                <h2>SharePoint v3 Built-In Fields</h2>
                                <table border="0" width="100%"
style="font-size:9pt;">
                                    <tr bgcolor="#9acd32">
                        <th align="left">Group</th>
                        <th align="left" width="100">Field</th>
                        <th align="left">Type</th>
                        <th align="left">Declaration</th>
                    </tr>
                    <xsl:apply-templates>
                        <xsl:sort select="@Group"/>
                        <xsl:sort select="@Name"/>
                    </xsl:apply-templates>
                </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="wss:Field">
        <tr>
            <td width="100"><xsl:value-of select="@Group"/></td>
            <td width="100"><xsl:value-of select="@Name"/></td>
            <td width="100"><xsl:value-of select="@Type"/></td>
            <td>
                &lt;FieldRef ID=&quot;<xsl:value-of select="@ID"/>&quot;
                    Name=&quot;<xsl:value-of select="@Name"/>&quot; /&gt;
            </td>
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

This table comes in handy when writing content type definition XML files. In this example, you include not only the group, field name, and underlying data type but also a text string you can easily copy and paste into the content type definition file.

The Field Definition File

For the five custom fields of the project proposal, you have to create a set of site columns that will be deployed along with the solution. Site columns are defined in XML using CAML Field elements. Each Field element declares the unique identifier, field type, field name, description and other properties.



XML Notepad - G:\Program Files\Common Files\Microsoft Shared\web server extensions\12\TEMPLATE\FEATURES\fields\fieldswss.xml

XSLT Location: G:\Documents and Settings\Administrator\My Documents\Work in Progress\wssfields.xsl

SharePoint v3 Built-In Fields

Group	Field	Type	Declaration
	ParentFolderId	Integer	<FieldRef ID="{a9ec25bf-5a22-4658-bd19-484e52efbe1a}" Name="ParentFolderId" />
	ShortestThreadIndexId	Text	<FieldRef ID="{2bec4782-695f-406d-9e50-f1d39a2b8eb6}" Name="ShortestThreadIndexId" />
$Resources:Base_Columns	_EndDate	DateTime	<FieldRef ID="{8A121252-85A9-443d-8217-A1B57020FADF}" Name="_EndDate" />
$Resources:Base_Columns	Attachments	Attachments	<FieldRef ID="{67df98f4-9dec-48ff-a553-29bece9c5bf4}" Name="Attachments" />
$Resources:Base_Columns	Categories	Text	<FieldRef ID="{9EBCD900-9D05-46c8-8F4D-E46E87328844}" Name="Categories" />
$Resources:Base_Columns	Completed	Boolean	<FieldRef ID="{35363960-D998-4aad-B7E8-058DFE2C669E}" Name="Completed" />
$Resources:Base_Columns	DocIcon	Computed	<FieldRef ID="{081c6e4c-5c14-4f20-b23e-1a71ceb6a67c}" Name="DocIcon" />
$Resources:Base_Columns	Edit	Computed	<FieldRef ID="{503f1caa-358e-4918-9094-4a2cdc4bc034}" Name="Edit" />
$Resources:Base_Columns	Event	Integer	<FieldRef ID="{20a1a5b1-fddf-4420-ac68-9701490e09af}" Name="Event" />
$Resources:Base_Columns	Group	Integer	<FieldRef ID="{c86a2f7f-7680-4a0b-8907-39c4f4855a35}" Name="Group" />
$Resources:Base_Columns	Item	Integer	<FieldRef ID="{92b8e9d0-a11b-418f-bf1c-c44aaa73075d}" Name="Item" />
$Resources:Base_Columns	Language	Choice	<FieldRef ID="{D81529E8-384C-4ca6-9C43-C86A256E6A44}" Name="Language" />
$Resources:Base_Columns	LinkFilename	Computed	<FieldRef ID="{5cc6dc79-3710-4374-b433-61cb4a686c12}" Name="LinkFilename" />
$Resources:Base_Columns	LinkFilenameNoMenu	Computed	<FieldRef ID="{9d30f126-ba48-446b-b8f9-83745f322ebe}" Name="LinkFilenameNoMenu" />
$Resources:Base_Columns	LinkTitle	Computed	<FieldRef ID="{82642ec8-ef9b-478f-acf9-31f7d45fbc31}" Name="LinkTitle" />
$Resources:Base_Columns	LinkTitleNoMenu	Computed	<FieldRef ID="{bc91a437-52e7-49e1-8c4e-4698904b2b6d}" Name="LinkTitleNoMenu" />
$Resources:Base_Columns	StartDate	DateTime	<FieldRef ID="{64cd368d-2f95-4bfc-a1f9-8d4324ecb007}" Name="StartDate" />
$Resources:Base_Columns	URL	URL	<FieldRef ID="{c29e077d-f466-4d8e-8bbe-72b66c5f205c}" Name="URL" />
$Resources:Base_Columns	URLNoMenu	Computed	<FieldRef ID="{AEAF07EE-D2FB-448b-A7A3-CF7E062D6C2A}" Name="URLNoMenu" />
$Resources:Base_Columns	URLwMenu	Computed	<FieldRef ID="{2a9ab6d3-268a-4c1c-9897-e5f018f87e64}" Name="URLwMenu" />
$Resources:Base_Columns	User	Integer	<FieldRef ID="{5928ff1f-daa1-406c-b4a9-190485a448cb}" Name="User" />
$Resources:Base_Columns	V3Comments	Note	<FieldRef ID="{6DF9BD52-550E-4a30-BC31-A4366832A87E}" Name="V3Comments" />
$Resources:Base_Columns	WorkflowItemId	Integer	<FieldRef ID="{8E234C69-02B0-42d9-8046-D5F49BF0174F}" Name="WorkflowItemId" />

Figure 11-4

Listing 11-3 shows the custom field declarations for the project proposal content type.

Listing 11-3: Project proposal custom field declarations

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
    <Field
        ID="{9DC4BA7E-6C50-4e24-9797-355131089A2E}"
        Description="Select the proposal type from the available choices."
        Type="Choice"
        Name="ProposalType"
        DisplayName="Proposal Type"
        StaticName="_ProposalType"
        >
        <CHOICES>
          <CHOICE>$Resources:ProposalManager,_Choice_FixedBid</CHOICE>
          <CHOICE>$Resources:ProposalManager,_Choice_TimeAndMaterials</CHOICE>
```

```
            </CHOICES>
        </Field>
        <Field
            ID="{24A18FDA-927A-4232-88AE-F713FFD3FBB4}"
            Description="The estimated cost of performing the work."
            Type="Currency"
            Name="EstimatedCost"
            DisplayName="Estimated Cost"
            StaticName="_EstimatedCost"
            />
        <Field
            ID="{41495470-9EA3-46e0-9A34-0E0C3DE1A445}"
            Description="The total bid amount"
            Type="Currency"
            Name="BidAmount"
            DisplayName="Bid Amount"
            StaticName="_BidAmount"
            />
        <Field
            ID="{D46C0900-5617-414c-97E5-E5626DBC1495}"
            Description="The estimated person-hours for the project."
            Type="Number"
            Name="EstimatedHours"
            DisplayName="Estimated Hours"
            StaticName="_EstimatedHours"
            />
        <Field
            ID="{79368859-EAF8-4361-8A9D-C3CBD9C88697}"
            Description="The negotiated hourly rate."
            Type="Currency"
            Name="HourlyRate"
            DisplayName="Hourly Rate"
            StaticName="_HourlyRate"
            />
    </Elements>
```

A Note about Resources

SharePoint uses a special syntax to enable XML definition files to reference strings and other resources at runtime. This powerful new feature enables developers to create more robust solutions with built-in localization support. Resource tags can also be used to keep the code synchronized with the XML definition files, thereby saving time during development.

The `$Resources` part of the tag indicates that the string should be retrieved from a RESX file stored in the `12\Resources` folder. The next part of the tag specifies the name of the RESX file and the resource identifier to retrieve. In this case, the `12\Resources\ProposalManager.resx` file contains a string resource named `_Choice_FixedBid` that in turn holds the text "Fixed Bid".

The Feature Definition File

The final step is to create a feature definition that references the content type so you can activate the feature within the SharePoint environment. Listing 11-4 shows the feature definition file for the Proposal Management application.

Listing 11-4: Proposal management feature declaration

```
<Feature  Title="ProposalManagerFeature"
    Id="63d38c9c-3ada-4e07-873f-a278443e910c"
    Description=""
    Version="1.0.0.0"
    Scope="Web"
    Hidden="TRUE"
    DefaultResourceFile="core"
    ReceiverAssembly="ProposalManager, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=9f4da00116c38ec5"
    ReceiverClass="ProSharePoint2007.ProposalManagerFeature"
    xmlns="http://schemas.microsoft.com/sharepoint/">
</Feature>
```

Defining Content Types in Code

There are many advantages to using the Windows SharePoint Services 3.0 object model instead of XML to define content types. These advantages include:

- ❑ No need to refer to the GUIDs of built-in site columns
- ❑ The ability to create dynamic types that depend on runtime conditions
- ❑ The ability to build a library of reusable content type components

Automatic Resolution of Built-In Field Identifiers

Setting up field references for content types declared using XML requires that the unique field identifier be known ahead of time. When creating field references in code, you only need to supply the associated field name. SharePoint retrieves the identifier automatically.

For example, the following code segment creates a `Project Proposal` content type based on the built-in `Document` type, and then adds an Author column to the new content type. The Author column is provided by SharePoint as one of the built-in site columns available in the Document Columns group.

```
using (SPSite site = new SPSite("http://localhost")) {
    using (SPWeb web = site.OpenWeb()) {
        SPContentType baseType =  web.AvailableContentTypes["Document"];
        SPContentType proposal = new SPContentType(
            baseType, web.ContentTypes, "Project Proposal");
        web.ContentTypes.Add(proposal);
        proposal.FieldLinks.Add(new SPFieldLink(web.AvailableFields["Author"]));
    }
}
```

Dynamic Content Type Definitions

With XML content type definitions, the fields are declared statically at design time. Once the content type is deployed and provisioned, its fields cannot be changed without rewriting the solution. On the other hand, by using the object model, you can set up the content type differently depending on external conditions. This way you can build smarter solutions that adjust automatically to accommodate changes in the run-time environment.

Building a Library of Reusable Content Type Components

When working with the Windows SharePoint Services 3.0 object model, it is useful to create a set of helper components to simplify solution development. This can greatly reduce the steps needed to build a solution because the low-level details of working with the object model are tucked away inside higher-level abstractions that are easier to declare and use. This is especially important when building document management solutions based on content types because you ultimately want to encapsulate the business rules within the content type itself. Having a library of core components means that you don't have to start from scratch each time you need a new content type.

Listing 11-5 shows a generic `ContentType` class that is used as a wrapper for the underlying `SPContentType` object instance.

Listing 11-5: A generic content type wrapper class

```
using System;
using Microsoft.SharePoint;

namespace ProSharePoint2007
{
    /// <summary>
    /// A utility class for manipulating SharePoint content types.
    /// </summary>
    public class ContentType
    {
        SPContentType m_contentType = null;

        /// <summary>
        /// Default constructor.
        /// </summary>
        public ContentType()
        {
        }

        /// <summary>
        /// Creates a wrapper for an existing content type instance.
        /// </summary>
        /// <param name="contentType"></param>
        public ContentType(SPContentType contentType)
        {
            m_contentType = contentType;
        }

        /// <summary>
        /// Adds a content type to a SharePoint list.
        /// </summary>
        public static void AddToList(SPList list, SPContentType contentType)
```

```
{
    list.ContentTypesEnabled = true;
    list.ContentTypes.Add(contentType);
    list.Update();
}

/// <summary>
/// Removes a content type from a SharePoint list.
/// </summary>
public static void RemoveFromList(SPList list, string contentTypeName)
{
    foreach (SPContentType type in list.ContentTypes) {
        if (type.Name == contentTypeName) {
            list.ContentTypes.Delete(type.Id);
            list.Update();
            break;
        }
    }
}

/// <summary>
/// Loads a preexisting[content type.
/// </summary>
public virtual SPContentType Create(SPWeb web, string typeName)
{
    try {
        m_contentType = web.AvailableContentTypes[typeName];
    } catch {
    }
    return m_contentType;
}

/// <summary>
/// Creates a new content type.
/// </summary>
public virtual SPContentType Create(SPWeb web, string typeName,
                    string baseTypeName,
                    string description)
{
    try {
        SPContentType baseType = (baseTypeName == null
            || baseTypeName.Length == 0) ?
            web.AvailableContentTypes[SPContentTypeId.Empty] :
            web.AvailableContentTypes[baseTypeName];

        m_contentType = new SPContentType(
            baseType, web.ContentTypes, typeName);
        m_contentType.Description = description;
        web.ContentTypes.Add(m_contentType);
    } catch {
    }
    return m_contentType;
}

/// <summary>
/// Conversion operator to access the underlying SPContentType instance.
```

```
        /// </summary>
        public static implicit operator SPContentType(ContentType t){
            return t.m_contentType;
        }


        #region Field Methods

        /// <summary>
        /// Adds a new field having a specified name and type.
        /// </summary>
        public SPField AddField(string fieldDisplayName,
                        SPFieldType fieldType, bool bRequired)
        {
            SPField field = null;
            try {
                // get the parent web
                using (SPWeb web = m_contentType.ParentWeb) {
                    // create the field within the target web
                    string fieldName =
                        web.Fields.Add(fieldDisplayName,
                                fieldType, bRequired);
                    field = web.Fields[fieldName];
                    // add a field link to the content type
                    m_contentType.FieldLinks.Add(
                        new SPFieldLink(field));
                    m_contentType.Update(false);
                }
            } catch {
            }
            return field;
        }

        /// <summary>
        /// Adds a new field based on an existing field in the parent web.
        /// </summary>
        public SPField AddField(string fieldName)
        {
            using (SPWeb web = m_contentType.ParentWeb) {
                SPField field = web.AvailableFields[fieldName];
                try {
                    if (field != null) {
                        m_contentType.FieldLinks.Add(
                            new SPFieldLink(field));
                        m_contentType.Update(false);
                    }
                } catch {
                }
            }
            return field;
        }
        #endregion
    }
}
```

With this helper class in the component library, it's easy to declare a new project proposal content type. It can either be instantiated from an XML definition associated with a feature or be created entirely in code. Listing 11-6 shows the declaration for the project proposal type derived from the generic content type wrapper class.

Listing 11-6: Using the content type wrapper class

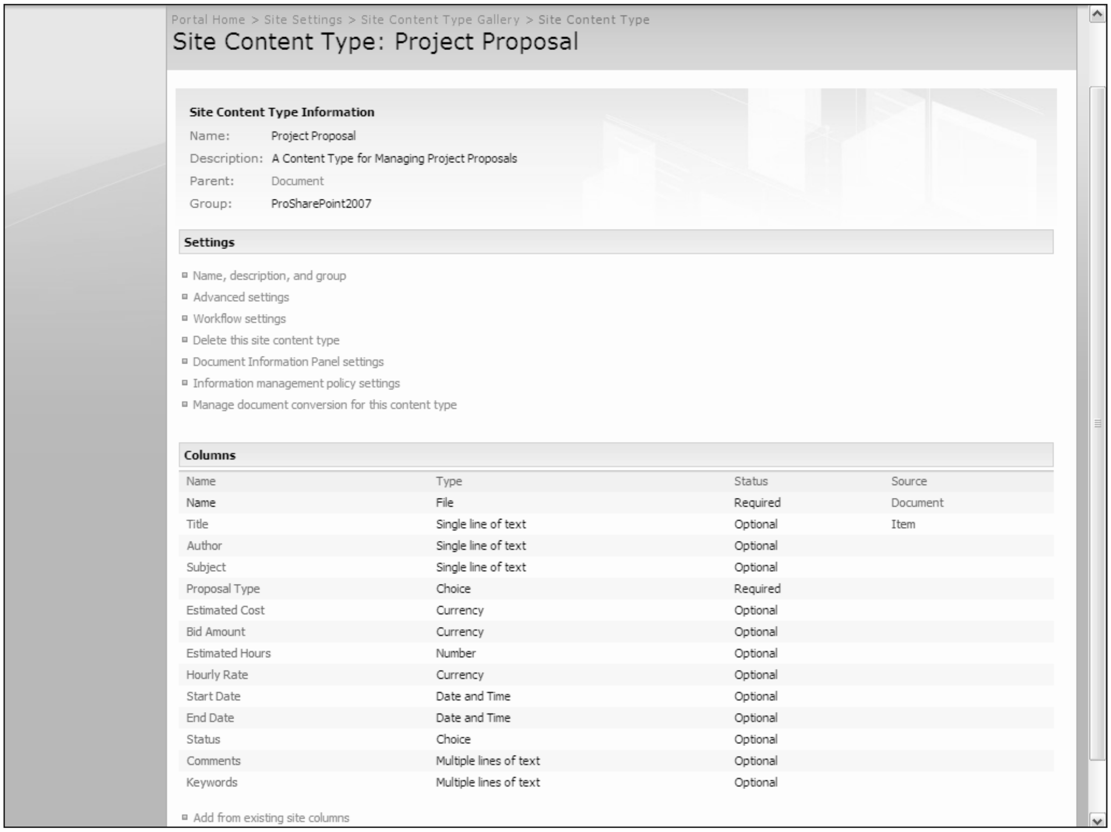
```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.SharePoint;

namespace ProSharePoint2007
{
    /// <summary>
    /// A helper class that encapsulates the ProjectProposal content type.
    /// </summary>
    class ProjectProposalType : ContentType
    {
        /// <summary>
        /// Creates the type using the XML content type definition.
        /// </summary>
        public SPContentType Create(SPWeb web)
        {
            return this.Create(web, "Project Proposal");
        }

        /// <summary>
        /// Creates the type using the SharePoint object model.
        /// </summary>
        public override SPContentType Create(SPWeb web, string typeName,
                            string baseTypeName,
                            string description)
        {
            // Call the base method to create the new type.
            SPContentType tProposal = base.Create(web, typeName,
                baseTypeName, description);

            // Create the fields programmatically.
            if (tProposal != null) {
                // built-in fields
                AddField("Author");
                AddField("Subject");
                AddField("StartDate");
                AddField("EndDate");
                AddField("Status");
                AddField("Comments");
                AddField("Keywords");
                // custom fields
                AddField(Strings._Field_ProposalType);
                AddField(Strings._Field_EstimatedCost);
                AddField(Strings._Field_BidAmount);
                AddField(Strings._Field_EstimatedHours);
```

```
                    AddField(Strings._Field_HourlyRate);
                }
                return tProposal;
            }
        }
    }
```

This code produces the content type definition shown in Figure 11-5.

In order to use the content type in a SharePoint site, you must deploy the type definition and then attach it to a list or document library for which content types have been enabled. Before you can achieve this, you need an additional piece of helper code to set up the document library to hold the proposal documents.

Listing 11-7 shows a `ProposalLibrary` class created for this purpose. When creating the document library, you remove the default `Document` content type so that users cannot create or upload standard documents. Finally, you create a new instance of the `ProjectProposal` content type and add it to the document library using the `AddToList` static method of the `ContentType` helper class.


Portal Home > Site Settings > Site Content Type Gallery > Site Content Type

Site Content Type: Project Proposal

Site Content Type Information

Name:	Project Proposal
Description:	A Content Type for Managing Project Proposals
Parent:	Document
Group:	ProSharePoint2007

Settings

- Name, description, and group
- Advanced settings
- Workflow settings
- Delete this site content type
- Document Information Panel settings
- Information management policy settings
- Manage document conversion for this content type

Columns

Name	Type	Status	Source
Name	File	Required	Document
Title	Single line of text	Optional	Item
Author	Single line of text	Optional	
Subject	Single line of text	Optional	
Proposal Type	Choice	Required	
Estimated Cost	Currency	Optional	
Bid Amount	Currency	Optional	
Estimated Hours	Number	Optional	
Hourly Rate	Currency	Optional	
Start Date	Date and Time	Optional	
End Date	Date and Time	Optional	
Status	Choice	Optional	
Comments	Multiple lines of text	Optional	
Keywords	Multiple lines of text	Optional	

- Add from existing site columns

Figure 11-5

Listing 11-7: A custom proposal document library class

```
/// <summary>
    /// A class that represents the proposals document library.
    /// </summary>
    class ProposalDocumentLibrary
    {
        SPDocumentLibrary m_docLib = null;
        public ProposalDocumentLibrary(SPWeb web)
        {
            try {
                SPListTemplate template =
                    web.ListTemplates["Document Library"];
                System.Guid guid =
                    web.Lists.Add(
                        Strings._ProposalLibrary_Title,
                        Strings._ProposalLibrary_Desc,
                        template);
                m_docLib = web.Lists[guid] as SPDocumentLibrary;
            } catch {
            }
            // Initialize the base library properties.
            m_docLib.OnQuickLaunch = true;
            m_docLib.EnableVersioning = true;
            m_docLib.EnableModeration = true;
            m_docLib.EnableMinorVersions = true;

            // Remove the default "Document" content type.
            ContentType.RemoveFromList(m_docLib, "Document");

            // Add the custom proposal content type.
            ContentType.AddToList(m_docLib, new ProjectProposalType().Create(web));
        }
    }
```

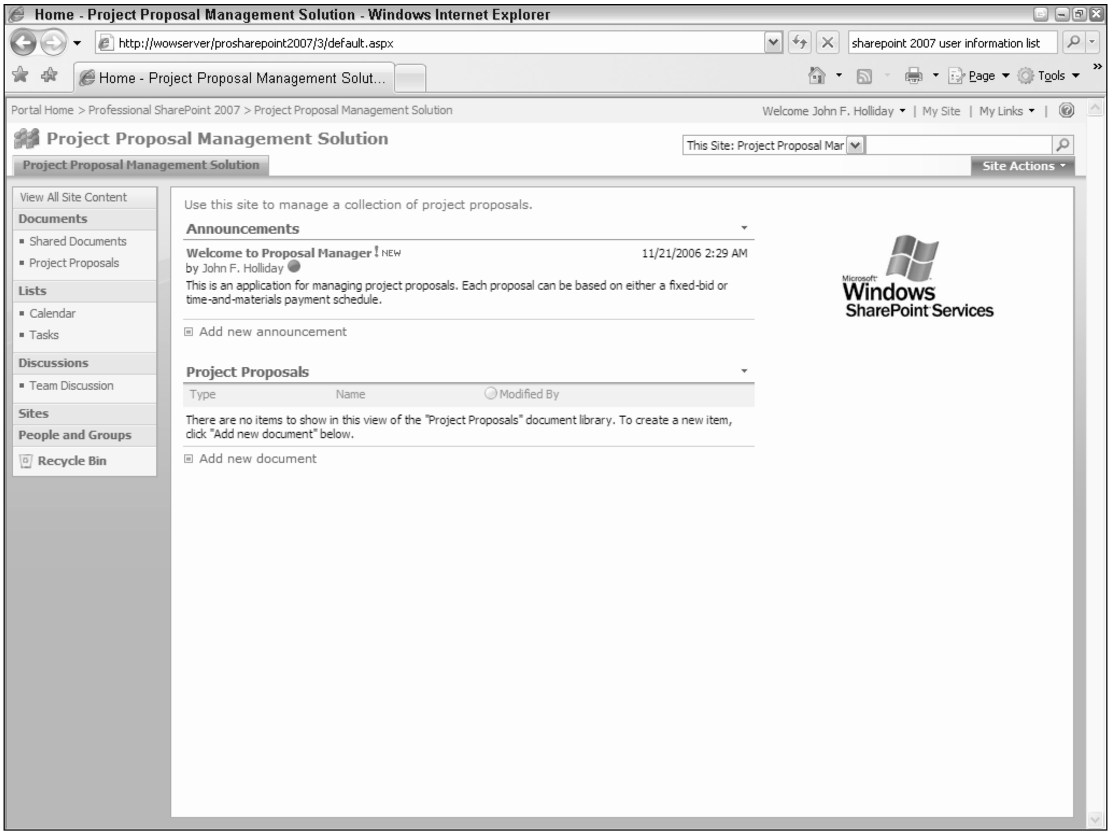
The easiest way to deploy a new content type is to include it as part of a custom feature. Here, you create a `ProposalManagement` feature to enable all of the proposal management tools on a site. As part of the feature implementation, you create an `SPFeatureReceiver` class for the `FeatureActivated` event that handles the deployment details for your custom content types. Listing 11-8 illustrates this process.

Listing 11-8: Provisioning a content type upon feature activation

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI.WebControls.WebParts;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebPartPages;
```

```
namespace ProSharePoint2007
{
    [Guid("63d38c9c-3ada-4e07-873f-a278443e910c")]
    partial class ProposalManagerFeature : SPFeatureReceiver
    {
        [SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
        public override void FeatureActivated(
                    SPFeatureReceiverProperties properties)
        {
            if (properties == null) {
                return;
            }

            SPWeb web = properties.Feature.Parent as SPWeb;

            // Create a library to hold the proposals and add a
            // default list view to the left Web Part zone.

            AddListViewWebPart(web,
                new ProposalDocumentLibrary(web),
                "Left", PartChromeType.Default);
        }

        /// <summary>
        /// Creates a ListViewWebPart on the main page.
        /// </summary>
        private void AddListViewWebPart(SPWeb web, SPList list,
                        string zoneId,
                        PartChromeType chromeType)
        {
            // Access the default page of the web.
            SPFile root = web.RootFolder.Files[0];

            // Get the Web Part collection for the page.
            SPLimitedWebPartManager wpm = root.GetLimitedWebPartManager(
            System.Web.UI.WebControls.WebParts.PersonalizationScope.Shared);

            // Add a list view to the bottom of the zone.
            ListViewWebPart part = new ListViewWebPart();
            part.ListName = list.ID.ToString("B").ToUpper();
            part.ChromeType = chromeType;
            wpm.AddWebPart(part, zoneId, 99);
        }
}
```

Now you have a site definition that includes the `ProposalManager` feature. When a site is created based on this site definition, the `FeatureActivated` event receiver creates a document library called Project Proposals that is automatically associated with your `Project Proposal` content type. Figure 11-6 shows the home page of a site created from the site definition.


Home - Project Proposal Management Solution - Windows Internet Explorer
http://wowserver/prosharepoint2007/3/default.aspx
Portal Home > Professional SharePoint 2007 > Project Proposal Management Solution
Welcome John F. Holliday | My Site | My Links
Project Proposal Management Solution
Site Actions
View All Site Content
Documents
Shared Documents
Project Proposals
Lists
Calendar
Tasks
Discussions
Team Discussion
Sites
People and Groups
Recycle Bin
Announcements
Add new announcement
Project Proposals
Type
Name
Modified By
Add new document
Windows SharePoint Services

Use this site to manage a collection of project proposals.

Welcome to Proposal Manager ! NEW 11/21/2006 2:29 AM
by John F. Holliday
This is an application for managing project proposals. Each proposal can be based on either a fixed-bid or time-and-materials payment schedule.

There are no items to show in this view of the "Project Proposals" document library. To create a new item, click "Add new document" below.

Figure 11-6

Managing Document Creation

In any document management solution, it's important that each new document adhere to organizational standards. This is especially true when metadata is being used to drive business processes. Even with a well-defined set of metadata fields and established guidelines for filling them out, busy knowledge workers often forget to provide this information because they are too focused on the document content. If the metadata is incomplete or inconsistent, then automated business processes that depend on it will fail. This leads to inefficiency and additional costs associated with finding and correcting the missing data.

Windows SharePoint Services 3.0 provides an enhanced callback mechanism that enables solution developers to control the document creation process and take prescriptive action based on the current state of the document. By implementing *event receivers*, you can write custom code that is called during document creation.

Document creation occurs in three stages:

1. The document template is opened in the appropriate editor program. The user edits the content and metadata and then saves the document into the library,
2. SharePoint retrieves the metadata and places it in a property bag, which it passes to the `ItemAdding` event receivers that have been registered for the library.
3. Unless one of the `ItemAdding` receivers cancels the document as noted below, SharePoint uses the properties to construct a new document list item, which is then added to the library. Next it calls the `ItemAdded` event receivers that have been registered for the library.

The `ItemAdding` event is called synchronously, suspending the document creation process until the event receiver returns control to SharePoint. By contrast, the `ItemAdded` event is called asynchronously after the document has been added to the library. Using this architecture, you can implement an `ItemAdding` event receiver to cancel the `Add` operation if the document metadata does not meet your requirements.

The key issues to address are:

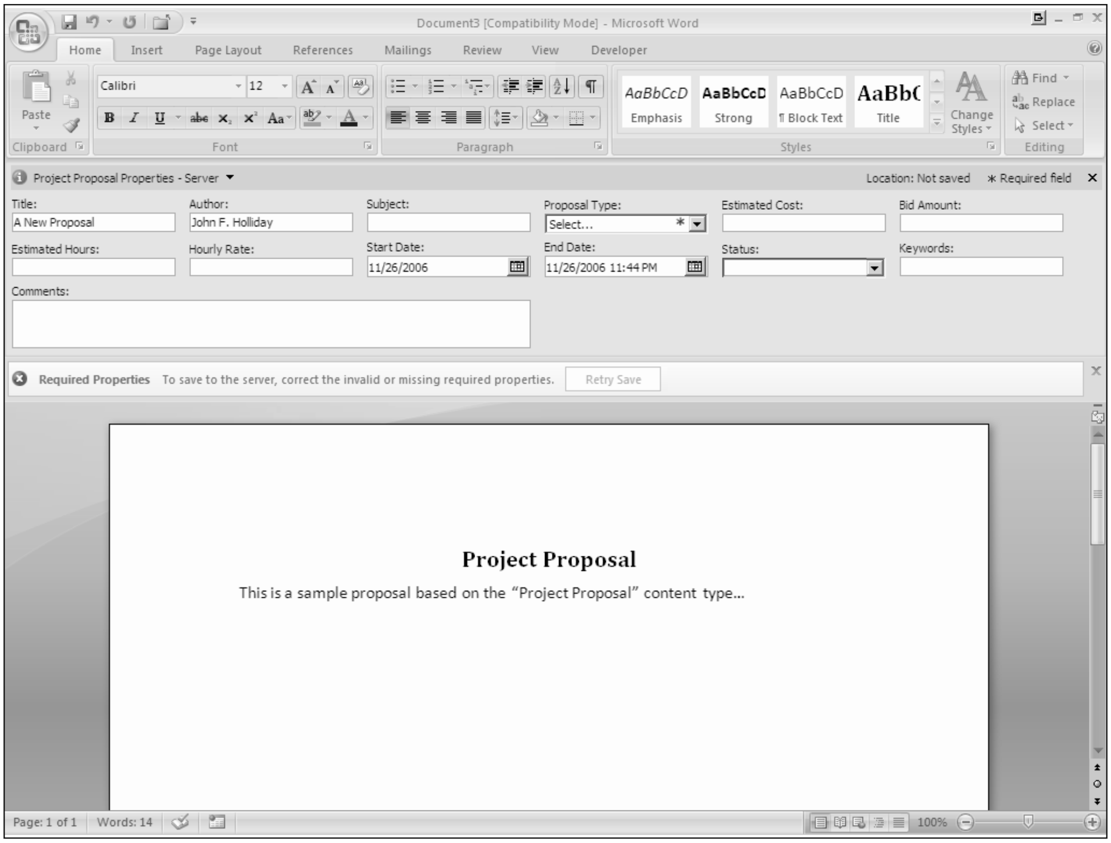
1. Have all required metadata fields been supplied with the document?
2. Are the values of the required metadata fields consistent with each other?
3. Is the metadata consistent with our document management policy?

Checking for Required Metadata

First, ensure that all of the required metadata fields have been supplied. For the project proposal, you want to ensure that the user has selected a project type, because the other business rules depend on this value. If it is not supplied, then you reject the document and notify the user.

The usual way to ensure that required metadata fields have been filled out is to set the `Required` attribute to "TRUE" when defining the content type. This works well for simple document types but not in more complex scenarios, where the set of required fields may change depending on an externally defined policy. It may still be necessary to use the `Required` attribute for documents that are created on the client. For instance, Microsoft Word will throw a generic exception instead of displaying a user-friendly error message when the `ItemAdding` event is canceled. Using the `Required` attribute forces the user to enter the item and also enables Word to display a visual cue indicating required fields, as shown in Figure 11-7.

When the `ItemAdding` event receiver is called, SharePoint passes a `SPItemEventProperties` object as a parameter. This object holds the property values for all of the metadata supplied by the user when creating the document. Using this object is a bit tricky because the same type of object is also passed to the other event receiver methods. However, different fields are supplied at different stages of the document lifecycle. The following table shows the relationship between selected fields of the `SPItemEventProperties` object and the Add/Update pairs of event receiver methods.



Project Proposal Properties - Server

Location: Not saved * Required field

Title: A New Proposal	Author: John F. Holliday	Subject:	Proposal Type: Select... *	Estimated Cost:	Bid Amount:
Estimated Hours:	Hourly Rate:	Start Date: 11/26/2006	End Date: 11/26/2006 11:44 PM	Status:	Keywords:
Comments:					

Required Properties To save to the server, correct the invalid or missing required properties.

Project Proposal

This is a sample proposal based on the "Project Proposal" content type...

Figure 11-7

Method	Description	Field	Available?	Comments
`ItemAdding`	Called before an item is added to the list.			
		`ListId`	Yes	Identifies the list that will contain the new item.
		`ListItem`	No	The list item has not yet been created.
		`Before-Properties`	No	Item properties are only available after the item is created.
		`After-Properties`	Yes	Item properties that will be used to populate the new item.

Continued

Method	Description	Field	Available?	Comments
`ItemAdded`	Called after an item is added to the list.			
		`ListId`	Yes	Identifies the containing list.
		`ListItem`	Yes	Identifies the new list item.
		`Before-Properties`	No	No item properties existed prior to item creation.
		`After-Properties`	Yes	Item properties that were used to populate the new item.
`ItemUpdating`	Called before an item is updated.			
		`ListId`	Yes	Identifies the containing list.
		`ListItem`	Yes	Identifies the list item.
		`Before-Properties`	Yes	Holds a hashtable of item properties before the update.
		`After-Properties`	Yes	Holds a hashtable of item properties that will be applied when the update is processed.
`ItemUpdated`	Called after an item is updated.			
		`ListId`	Yes	Identifies the containing list.
		`ListItem`	Yes	Identifies the list item.
		`Before-Properties`	Yes	Holds a hashtable of item properties before the update.
		`After-Properties`	Yes	Holds a hashtable of item properties after the update.

To make things easier and to simplify the code, you can declare a static class that takes these dependencies into account. Listing 11-9 shows how to implement the `ItemAdding` event receiver to check for the required metadata using a static wrapper class to process the raw `SPItemEventProperties` object.

Listing 11-9: ItemAdding event receiver

```
/// <summary>
/// A helper class for testing various conditions on SharePoint list items.
/// </summary>
public class ItemCondition
{
    public static bool HasProposalType(SPItemEventProperties properties)
    {
        object value = properties.AfterProperties["Proposal Type"];
        return value != null && value.ToString() != string.Empty;
    }
}

/// <summary>
/// Synchronous before event that occurs when a new item is added
/// to its containing object.
/// </summary>
/// <param name="properties">
/// A Microsoft.SharePoint.SPItemEventProperties object
/// that represents properties of the event handler.
/// </param>
[SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
public override void ItemAdding(SPItemEventProperties properties)
{
    try {
        ValidateItemProperties(properties);
    } catch (Exception x) {
        properties.ErrorMessage = x.Message;
        properties.Cancel = true;
    }
}

/// <summary>
/// Helper method to determine whether an item meets validation requirements.
/// </summary>
/// <param name="properties"></param>
private void ValidateItemProperties(SPItemEventProperties properties)
{
    if (!ItemCondition.HasProposalType(properties))
        throw new SPItemValidationException(properties, "You must select a proposal
type.");
}
```

Checking Metadata Consistency

In the case of the project proposal, you check for different field values depending on the type of project that was selected. For instance, time-and-material projects require the user to provide an initial estimate of the hours needed to complete the project, while fixed-bid projects require an estimate of the total bid amount. You will use these values later when implementing other business rules, so it is vital that the appropriate values are entered correctly.

When an item is added to a list, SharePoint calls the `ItemAdding` *event receiver, but it does not call the* `ItemUpdating` *event receiver. Conversely, when an item is edited, SharePoint calls only the* `ItemUpdating` *event receiver. Therefore, any consistency checking code should be called from both the* `ItemAdding` *and the* `ItemUpdating` *event receivers.*

Postprocessing of Metadata

After a document has been added to the library, it is often useful to perform postcreation tasks such as making entries into a tracking log or notifying users that a new document has been created. In the proposal-tracking system, you want to keep a running record of the progress each proposal makes throughout its lifecycle. You can do this easily by setting up a SharePoint list to record the date and time a given event occurs.

You begin by defining a second content type called a `ProposalTrackingRecord` and enable it for a custom list called "Proposal History." The `ProposalTrackingRecord` is derived from the built-in `Item` content type, and consists of only the title field. Use the title to display the text of the tracking event.

```
<ContentType ID="0x0100a0cada319e714c1fab64c519c065d421"
        Name="Proposal Tracking Record"
        Group="ProSharePoint2007"
        Description="A list item for tracking proposal-specific events."
        Version="0">
        <FieldRefs/>
</ContentType>
```

You can use a `ProposalTrackingEvent` enumeration to model the different kinds of events you wish to track. In addition to the standard document lifecycle events, you can also capture important proposal-specific milestones such as when a proposal is approved for submission to the client, or when a proposal is accepted by the client.

```
    enum ProposalTrackingEvent
    {
        Created,
        Modified,
        MajorRevision,
        MinorRevision,
        CheckedOut,
        CheckedIn,
        ApprovedForSubmission,
        AcceptedByClient,
        Published,
        Deleted,
        Archived,
    }
```

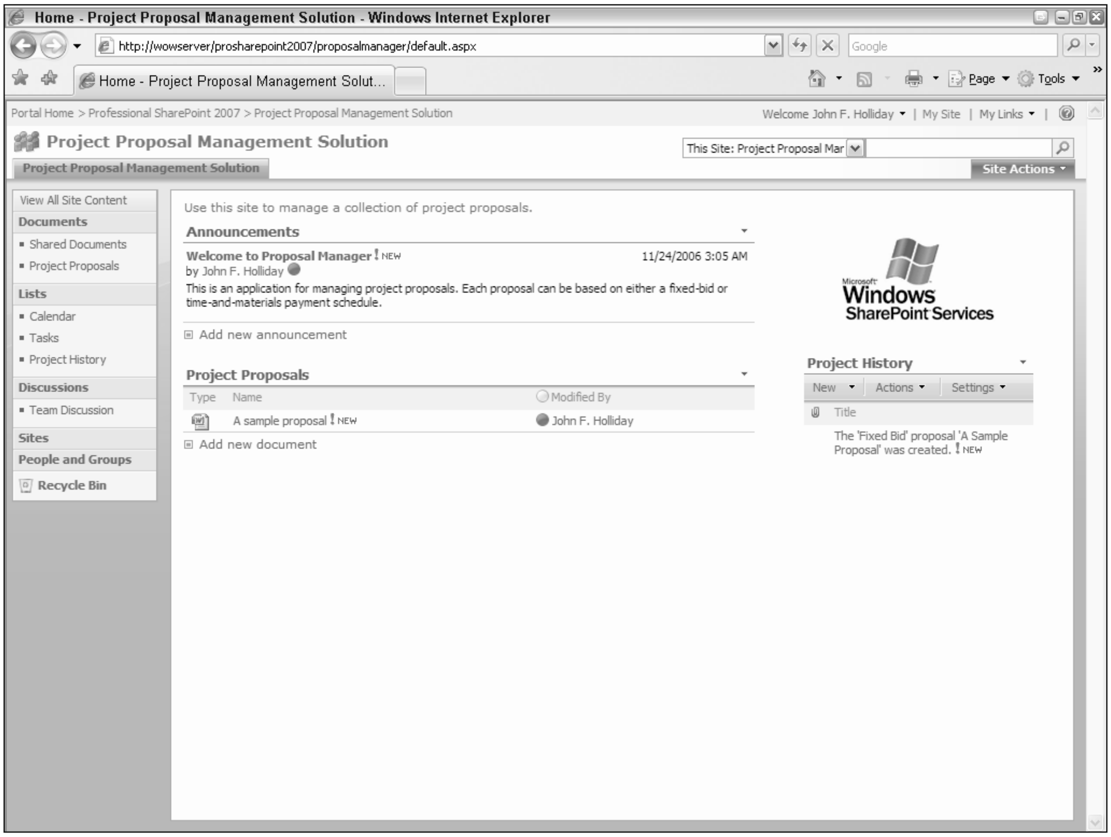
Finally, in the same way that you created the Proposals document library, you now create the Proposal History custom list and an associated `ListView` in the right Web Part zone.

```
// Create a custom list to hold the proposal history and
// add a default list view to the right Web Part zone.

AddListViewWebPart(web, new ProposalHistoryList(web),
            "Right", PartChromeType.TitleOnly);
```

To capture postcreation events, implement an `ItemAdded` event receiver for the `ProjectProposal` content type. This event is fired after the metadata has been validated and the new document has been added to the library. When the event receiver is called, you obtain a reference to the Proposal History list and make the appropriate entries using the `AddRecord` helper method:

```
[SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
public override void ItemAdded(SPItemEventProperties properties)
{
    // Add an entry to the proposal history list.
    SPWeb web = properties.ListItem.ParentList.ParentWeb;
    ProposalHistoryList history = ProposalHistoryList.FromWeb(web);
    history.AddRecord(ProposalTrackingEvent.Created,
        properties.AfterProperties);
}
```

Now, whenever a project proposal is added to the proposals library, a new tracking record is created based on the type of event (in this case, `ProposalTrackingEvent.Created`) and the properties stored in the new item. Depending on the event type, you can use these properties to capture a more detailed picture of the context surrounding the event. This analysis could be recorded separately or could cause a custom workflow to be initiated. Figure 11-8 shows the project history list with a tracking record entry.


Home - Project Proposal Management Solution - Windows Internet Explorer
http://wowserver/prosharepoint2007/proposalmanager/default.aspx
Google
Home - Project Proposal Management Solut...
Page
Tools
Portal Home > Professional SharePoint 2007 > Project Proposal Management Solution
Welcome John F. Holliday | My Site | My Links
Project Proposal Management Solution
This Site: Project Proposal Mar
Project Proposal Management Solution
Site Actions
View All Site Content
Documents
Shared Documents
Project Proposals
Lists
Calendar
Tasks
Project History
Discussions
Team Discussion
Sites
People and Groups
Recycle Bin
Microsoft Windows SharePoint Services
New
Actions
Settings

Use this site to manage a collection of project proposals.

Announcements

Welcome to Proposal Manager ! NEW — 11/24/2006 3:05 AM
by John F. Holliday
This is an application for managing project proposals. Each proposal can be based on either a fixed-bid or time-and-materials payment schedule.

Add new announcement

Project Proposals

Type	Name	Modified By
	A sample proposal ! NEW	John F. Holliday

Add new document

Project History

Title
The 'Fixed Bid' proposal 'A Sample Proposal' was created. ! NEW

Figure 11-8

Managing the Document Revision Cycle

Referring back to the generic model of the document lifecycle, note that all documents follow the general pattern of creation/revision/publication. Unlike the creation phase, which happens only once, the revision phase occurs repeatedly until the document is ready for publication. During the revision phase, many different types of events can occur, depending on the state of the document metadata and the status of its content.

The SharePoint object model defines eight pairs of events that can occur during the document revision cycle. These are captured by the following elements of the `SPItemEventReceiverType` enumeration. Using these events, you can control all aspects of document revision.

- `ItemUpdated/Updating`
- `ItemCheckedIn/CheckingIn`
- `ItemCheckedOut/CheckingOut`
- `ItemUncheckedOut/UncheckingOut`
- `ItemAttachmentAdded/Adding`
- `ItemAttachmentDeleted/Deleting`
- `ItemFileMoved/Moving`
- `ItemDeleted/Deleting`

We shall ignore the `ItemFileConverted` *event and the* `ItemFileMoved/Moving` *event pair because they happen outside the document revision cycle.*

During each stage of the revision cycle, you can use document metadata to analyze the current state of the document in terms of the problem domain and then update the metadata in the appropriate way. This revised metadata can be used to further constrain the behavior of the document or to control the actions of the people involved in editing it.

The following sections explore the construction of both generic and domain-specific tools for analyzing metadata to assist in answering the question "what happens next?" in the context of these events.

Building Custom Tools for Metadata Analysis

Sometimes it is useful to distinguish between metadata maintained by the system and custom metadata you defined within the problem domain. For the former, it's easy to create a library of reusable tools that can be used to quickly analyze system-defined properties. For instance, SharePoint can automatically track major and minor versions of each document in a library and can enforce moderation (approval) and check-out policies. Using the object model components associated with these properties, you can create components that perform useful functions, such as:

- Determine if the major or minor version number exceeds a certain limit
- Determine the current approval status of the document

❑ Compute the average length of time a document remains checked out

❑ Count the number of times a given user has checked in or reviewed a document

You can take the same approach for domain-specific metadata such as, in this example, where you might want to define a set of high-level methods for working with project proposals. These general and specific methods can make it much easier to build custom business rules. For example, you might need the following two rules:

❑ Compare the estimated cost of a fixed-bid proposal to a predefined limit.

❑ Compare the estimated man-hours of a time-and-materials proposal to a predefined minimum.

To support these rules, you can create a wrapper class for retrieving the bid amount and estimated hours from a set of project proposal properties. The properties are passed to the `SPItemEventReceiver` as described earlier. Listing 11-10 shows the `ProjectProposal` wrapper class that is derived from a generic wrapper for SharePoint list items.

Listing 11-10: A project proposal wrapper class

```
/// <summary>
/// A wrapper class for a project proposal instance.
/// </summary>
class ProjectProposal : SharePointListItem
{
    /// <summary>
    /// Constructs a wrapper for the underlying list item.
    /// </summary>
    public ProjectProposal(SPListItem item):base(item)
    {
    }

    #region Property Value Accessors

    /// <summary>
    /// Retrieves the proposal type as reflected by the item properties.
    /// </summary>
    public static ProposalType GetProposalType(SPItemEventDataCollection
properties)
    {
        ProposalType type = ProposalType.TimeAndMaterials;
        try {
            object value = properties[Strings._Field_ProposalType];
            string choice = value.ToString();
            if (choice.Equals(Strings._Choice_FixedBid))
                type = ProposalType.FixedBid;
        } catch {
        }
        return type;
    }
```

```
        /// <summary>
        /// Retrieves the proposal bid amount.
        /// </summary
        public static decimal GetBidAmount(SPItemEventDataCollection properties)
        {
            decimal amount = 0M;
            try {
                object value = properties[Strings._Field_BidAmount];
                amount = Decimal.Parse(value.ToString());
            } catch {
            }
            return amount;
        }

        /// <summary>
        /// Retrieves the estimated person hours for a proposal.
        /// </summary>
        public static decimal GetEstimatedHours(SPItemEventDataCollection properties)
        {
            decimal amount = 0M;
            try {
                object value = properties[Strings._Field_EstimatedHours];
                amount = Decimal.Parse(value.ToString());
            } catch {
            }
            return amount;
        }
        #endregion
    }
```

Ensuring Metadata Consistency between Revisions

Using your library of metadata analysis components, you can easily test for a range of conditions whenever a proposal document is updated. First, you perform the same metadata consistency check as when adding an item, rejecting the update from the `ItemUpdating` event receiver if a problem exists. Then, you use the project proposal wrapper to test for the two conditions you defined in the business rules.

```
[SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
public override void ItemUpdating(SPItemEventProperties properties)
{
const decimal MinimumBid =5000M;
const decimal MinimumHours = 300M;
try {
    ValidateItemProperties(properties);
    switch (ProjectProposal.GetProposalType(
            properties.AfterProperties)
    {
    case ProposalType.FixedBid: {
        if (ProjectProposal.GetBidAmount(
            properties.AfterProperties) < MinimumBid) {
            throw new ApplicationException(
                    "Bids must be higher than " +
                        MinimumBid.ToString());
            }
            break;
        case ProposalType.TimeAndMaterials: {
            if (ProjectProposal.GetEstimatedHours(
                properties.AfterProperties) < MinimumHours) {
                throw new ApplicationException(
                    "Hours must be greater than " +
                        MinimumHours.ToString());
            }
            break;
        }
    } catch (Exception x) {
            properties.ErrorMessage = x.Message;
            properties.Cancel = true;
        }
    }
```

At this point, you can perform other tests suitable for proposal documents. As more conditions are added, you can easily extend the framework to accommodate them by implementing the appropriate static methods on the `ProjectProposal` class and making the corresponding call from the event receiver.

Managing Checkin and Checkout

SharePoint supports enforced checkout for document libraries and lists. Setting the `ForceCheckout` property to `true` causes SharePoint to require that users check out a document before editing it. However, you may need to place additional constraints on document items. For example, you may need to control which users are allowed to check out a document, or keep track of which users checked out which documents, or calculate the average length of time a given user keeps documents checked out, and so on.

A good example of when this might be necessary is in a document library that is set up to enforce approval via the built-in `_ModerationStatus` field. After the document has been approved, you might wish to prevent users other than the approver from making further changes. You can do this by implementing a `CheckingOut` event receiver, checking the moderation status and then comparing the moderator to the current user. If they do not match and the document has been approved, then you abort the checkout with an appropriate message to the user.

Developing XML-Driven Document Management Solutions

Writing code whenever you want to change the policies associated with a content type can become a tedious operation. It would be better if you could define the more volatile aspects of the policy in an external file and then process that file during execution in order to determine if metadata has been provided consistently and completely. The problem is where should you put the file?

You could certainly put the file in a well-known location, but that would require too much knowledge of the operating environment. What you are really trying to do is separate the policy from your content type implementation so that you can change it more easily.

This section shows how to control the behavior of a class of documents using rules defined in an XML schema associated with a content type. After developing a simple project proposal management schema you create a default policy based on the schema and then attach it to the content type definition. As events are generated by SharePoint during the document revision cycle, you retrieve the policy from the content type and apply the policy.

Developing a Proposal Management Policy Schema

Referring back to the document lifecycle diagram, you can see how policies will affect the overall document revision cycle. Figure 11-9 shows the revision cycle with policies attached. You want to ensure that the total bid amount for fixed-bid proposals is never less than $5000. On the other hand for time-and-materials proposals, you want to reject any revision where the estimated hours are less than 300. And you want to be able to change these values without recoding the solution.

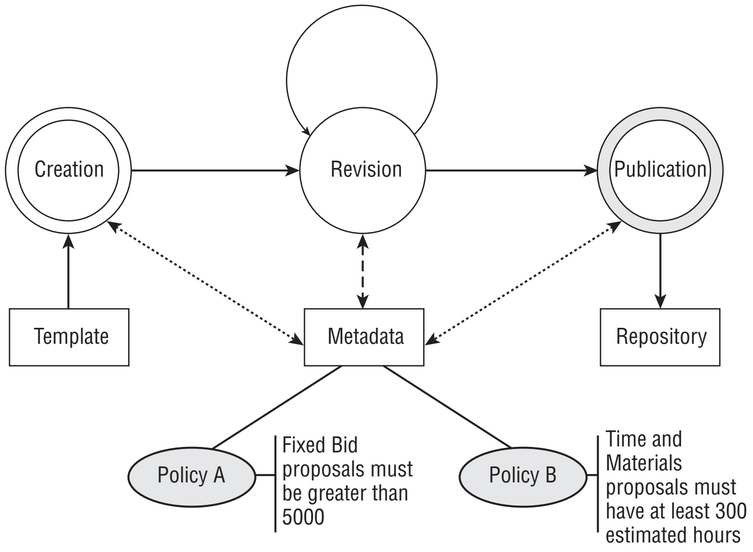

Creation
Revision
Publication
Template
Metadata
Repository
Policy A
Fixed Bid proposals must be greater than 5000
Policy B
Time and Materials proposals must have at least 300 estimated hours

Figure 11-9

You can meet these needs by defining a simple schema to manage project proposals. The schema will distinguish between the two different types of proposals and provide elements for specifying acceptable bid amounts and estimated hours. In addition, you need a way to control what happens when the policy has been violated. This example shows how to display an error message to the user stating the condition and the expected values. Figure 11-10 depicts the proposal management policy schema shown in Listing 11-11.

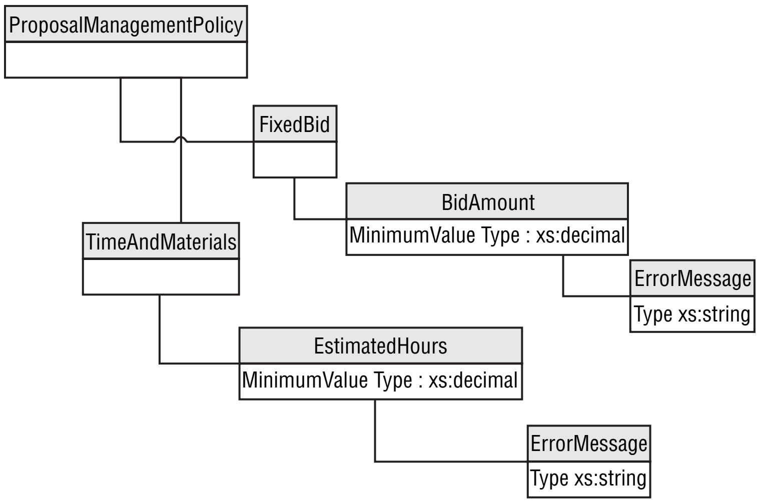


Figure 11-10

Listing 11-11: Proposal management policy schema

```
<xs:element name="ProposalManagementPolicy"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType>
    <xs:sequence>
        <xs:element name="FixedBid">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="BidAmount">
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element
                                    name="ErrorMessage"
                                    type="xs:string" />
                            </xs:sequence>
                            <xs:attribute name="MinimumValue"
                                type="xs:decimal"
                                use="required" />
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="TimeAndMaterials">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="EstimatedHours">
```

```
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element
                                        name="ErrorMessage"
                                        type="xs:string" />
                                </xs:sequence>
                                <xs:attribute name="MinimumValue"
                                    type="xs:decimal"
                                    use="required" />
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    </xs:element>
```

Using this schema definition, you can create a default policy for new project proposals. To make it easier for system administrators who will be editing these files, the example uses a simple value substitution scheme that looks for tokens in the error message text and replaces them with attribute values.

```
<ProposalManagementPolicy
        xmlns="http://schemas.johnholliday.net/proposalmanagementpolicy.xsd">
    <FixedBid>
        <BidAmount MinimumValue="5000">
            <ErrorMessage>The bid amount must be at least $$MinimumValue$$
.</ErrorMessage>
        </BidAmount>
    </FixedBid>
    <TimeAndMaterials>
        <EstimatedHours MinimumValue="300">
            <ErrorMessage>The estimated hours must be at least $$MinimumValue$$
.</ErrorMessage>
        </EstimatedHours>
    </TimeAndMaterials>
</ProposalManagementPolicy>
```

Attaching the Policy to a Content Type

In order for this to work, you need to associate the default policy with the project proposal content type. As it turns out, SharePoint has a built-in facility for doing this.

SharePoint maintains a collection of XML documents for each content type. These documents are accessible through the `XmlDocuments` Element in the content type definition. There are two ways to associate an `XmlDocument` with a content type: in the content type definition XML, used to provision the content type, and through the Windows SharePoint Services 3.0 object model. Which method to use depends on when the information is required and whether it will change.

If the information is static or is needed during the provisioning process, then the best option is to include it in the content type definition XML. On the other hand, if the information is dynamic, then the object model is probably the best choice. That way, you can drive other behaviors or control user interactions based on the contents of the associated XML document.

In this example, both conditions are true. You need the information from the default policy during the provisioning process so you can accept or reject new proposals, and you also need the policy information to be dynamic because you want to enable administrative users to modify the policy at any given time.

Adding the default policy to the content type definition XML requires that you insert an `XmlDocuments` element that contains a child `XmlDocument` element. You then place the contents of the custom XML document inside the `XmlDocument` element.

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
<!-- _filecategory="ContentType" _filetype="Schema" _filename="contenttype.xml"
_uniqueid="cff96a1e-6a52-4462-a3d0-d01471b8bfef" -->
<ContentType ID="0x0101004a257cd7888d4e8baea35afcdfdea58c"
    Name="Project Proposal"
    Group="ProSharePoint2007"
    Description="A Content Type for Managing Project Proposals"
    Version="0">
    <FieldRefs>
        <FieldRef ID="{246D0907-637C-46b7-9AA0-0BB914DAA832}" Name="Author"/>
        <FieldRef ID="{76A81629-44D4-4ce1-8D4D-6D7EBCD885FC}" Name="Subject" />
        <FieldRef ID="{9DC4BA7E-6C50-4e24-9797-355131089A2E}" Name="ProposalType"
                                 Required="TRUE"/>
        <FieldRef ID="{24A18FDA-927A-4232-88AE-F713FFD3FBB4}" Name="EstimatedCost"/>
        <FieldRef ID="{41495470-9EA3-46e0-9A34-0E0C3DE1A445}" Name="BidAmount"/>
        <FieldRef ID="{D46C0900-5617-414c-97E5-E5626DBC1495}" Name="EstimatedHours"/>
        <FieldRef ID="{79368859-EAF8-4361-8A9D-C3CBD9C88697}" Name="HourlyRate"/>
        <FieldRef ID="{64cd368d-2f95-4bfc-a1f9-8d4324ecb007}" Name="StartDate" />
        <FieldRef ID="{8A121252-85A9-443d-8217-A1B57020FADF}" Name="EndDate" />
        <FieldRef ID="{1DAB9B48-2D1A-47b3-878C-8E84F0D211BA}" Name="Status" />
        <FieldRef ID="{52578FC3-1F01-4f4d-B016-94CCBCF428CF}" Name="Comments" />
        <FieldRef ID="{B66E9B50-A28E-469b-B1A0-AF0E45486874}" Name="Keywords" />
    </FieldRefs>
    <XmlDocuments>
        <XmlDocument NamespaceURI="$Resources:ProposalManager,
ProposalManagementPolicyNamespace">
            <ProposalManagementPolicy>
                <FixedBid>
                    <BidAmount MinimumValue="5000">
                       <ErrorMessage>The bid amount must be at least $$MinimumValue$$
.</ErrorMessage>
                    </BidAmount>
                </FixedBid>
                <TimeAndMaterials>
                    <EstimatedHours MinimumValue="50">
                        <ErrorMessage>The estimated hours must be at least
$$MinimumValue$$.</ErrorMessage>
                    </EstimatedHours>
                </TimeAndMaterials>
            </ProposalManagementPolicy>
        </XmlDocument>
    </XmlDocuments>
</ContentType>
</Elements>
```

SharePoint uses the `NamespaceURI` attribute to index the collection of XML documents associated with a content type. This means that you must specify a namespace attribute for the `XmlDocument` element or you will not be able to locate the document when calling the SharePoint object model from your code.

The SharePoint SDK states that you can add any number of XML documents to the `XmlDocuments` *collection, and that any XML document can be used as long as it is valid XML. In practice, I've found that not only must the XML be valid, but it must not cause an exception to be thrown while SharePoint is deserializing the XML fragment. For instance, if namespaces are used, they must resolve properly. Also, it must be an XML fragment and not an XML document. In other words, it must not include an* `xml` *directive. If an exception occurs, then SharePoint will silently consume the exception but will not completely load the content type. One side effect of this is that the content type will be created, but without the expected metadata fields. Similarly, if the* `NamespaceURI` *attribute is missing, SharePoint will abort the load.*

Processing the Policy in Response to Events

The logical place to process policy files is in the synchronous event receiver methods `ItemAdding` and `ItemUpdating`, because you can cancel the operation and display an error message to the user. In order for this to work, you need to add a static method to the `ProjectProposalType` wrapper class.

The `ApplyPolicy` method extracts the XML document containing our default policy and then uses the `XmlSerializer` to deserialize it into a C# class. The `ProposalManagementPolicy` class is shown in Listing 11-12. By passing the `SPItemEventProperties` parameter along, you can then invoke the appropriate methods on the deserialized class to analyze the properties of the new or modified item in the context of the policy values. You use an extra parameter to specify the context in which the policy is to be applied (add, update, delete, etc.).

```
public static void ApplyPolicy(SPItemEventProperties properties,
                    ProposalManagementPolicy.PolicyContext context)
{
    ProposalManagementPolicy policy = null;
    ProjectProposalType proposalType = new ProjectProposalType();
    using (SPWeb web = properties.OpenWeb())
        {
            if (proposalType.Create(web) != null) {
                string xml = proposalType.GetXmlString(
                    Strings.ProposalManagementPolicyNamespace);
                policy = ProposalManagementPolicy.FromXml(xml);
            } else {
                policy = ProposalManagementPolicy.ReadFrom(
                    Strings.DefaultPolicy);
            }
        }
        if (policy != null) {
            policy.ApplyPolicy(properties.AfterProperties, context);
        }
}
```

Listing 11-12: Proposal management policy class

```
using System;
using System.IO;
using System.Data;
using System.Text;
using System.Diagnostics;
using System.ComponentModel;
using System.Xml;
using System.Xml.Serialization;
using Microsoft.SharePoint;

namespace ProSharePoint2007
{
    [Serializable]
    [XmlType(AnonymousType=true)]
    [XmlRoot(Namespace="", IsNullable=false)]
    public class ProposalManagementPolicy {

        /// <summary>
        /// Specifies the context in which a given policy is being applied.
        /// </summary>
        public enum PolicyContext
        {
            Add,
            Update,
            CheckIn,
            CheckOut,
            Delete
        }

        /// <summary>
        /// Loads a ProposalManagementPolicy object from a file.
        /// </summary>
        public static ProposalManagementPolicy ReadFrom(string fileName)
        {
            XmlSerializer ser = new XmlSerializer(
            typeof(ProposalManagementPolicy), "");
            return (ProposalManagementPolicy)ser.Deserialize(
            new StreamReader(fileName));
        }

        /// <summary>
        /// Loads a ProposalManagementPolicy object from an xml string.
        /// </summary>
        public static ProposalManagementPolicy FromXml(string xml)
        {
            xml = xml.Replace(Strings.XmlDirective_Utf16,
                    Strings.XmlDirective_Utf8).Trim();
            if (!xml.StartsWith(Strings.XmlDirective_Utf8))
                xml = Strings.XmlDirective_Utf8 + xml;

            XmlSerializer ser = new XmlSerializer(
                        typeof(ProposalManagementPolicy), "");
            return (ProposalManagementPolicy)ser.Deserialize(
                        new StringReader(xml));
        }

        /// <summary>
        /// Loads a ProposalManagementPolicy object from a content type.
        /// </summary>
        /// <param name="contentType">the containing content type instance</param>
        public static ProposalManagementPolicy FromContentType(
                            ContentType contentType)
        {
            string xml = contentType.GetXmlString(
                Strings.ProposalManagementPolicyNamespace);
            XmlSerializer ser = new XmlSerializer(
                typeof(ProposalManagementPolicy), "");
            return (ProposalManagementPolicy)ser.Deserialize(new
StringReader(xml));
        }

        /// <summary>
        /// Applies the policy to a given set of proposal properties.
        /// </summary>
        public void ApplyPolicy(SPItemEventDataCollection properties,
                    PolicyContext context)
        {
            // Check the properties to determine if the policy was satisfied.
            switch (ProjectProposal.GetProposalType(properties)) {
                case ProposalType.FixedBid: {
                        ApplyFixedBidPolicy(properties, context);
                        break;
                    }
                case ProposalType.TimeAndMaterials: {
                        ApplyTimeAndMaterialsPolicy(properties, context);
                        break;
                    }
            }
        }

        /// <summary>
        /// Converts this instance to a text string.
        /// </summary>
        /// <remarks>
        /// Uses the XmlSerializer to produce an XML string representing
        /// the policy.
        /// </remarks>
        public override string ToString()
        {
            // Generate the XML string.
            StringBuilder policy = new StringBuilder();
        XmlSerializer ser = new XmlSerializer(GetType(), "");
        ser.Serialize(new StringWriter(policy), this);

        // Check for and remove the xml directive.
        string xml = policy.ToString();
        xml = xml.Replace(Strings.XmlDirective_Utf8, "");
        xml = xml.Replace(Strings.XmlDirective_Utf16, "");

        return xml.Trim();
    }

    /// <summary>
    /// Updates the policy associated with a given content type.
    /// </summary>
    /// <param name="contentType"></param>
    public void UpdatePolicy(ContentType contentType)
    {
        // Attach the policy to the content type.
        contentType.AddXmlString(Strings.ProposalManagementPolicyNamespace,
                    this.ToString());
    }

    /// <summary>
    /// Applies policies that pertain to fixed bid proposals.
    /// </summary>
    /// <param name="properties"></param>
    void ApplyFixedBidPolicy(SPItemEventDataCollection properties,
                PolicyContext context)
    {
        if (ProjectProposal.GetBidAmount(properties)
        < FixedBid.BidAmount.MinimumValue) {
            switch (context) {
                case PolicyContext.Update:
                    throw new ProposalManagementPolicyException(
            FixedBid.BidAmount.FormattedErrorMessage);
            }
        }
    }

    /// <summary>
    /// Applies policies that pertain to time and materials proposals.
    /// </summary>
    /// <param name="properties"></param>
    void ApplyTimeAndMaterialsPolicy(SPItemEventDataCollection properties,
                    PolicyContext context)
    {
        if (ProjectProposal.GetEstimatedHours(properties)
        < TimeAndMaterials.EstimatedHours.MinimumValue) {
            switch (context) {
                case PolicyContext.Update:
                    throw new ProposalManagementPolicyException(
            TimeAndMaterials.EstimatedHours.FormattedErrorMessage);
            }
        }
    }
```

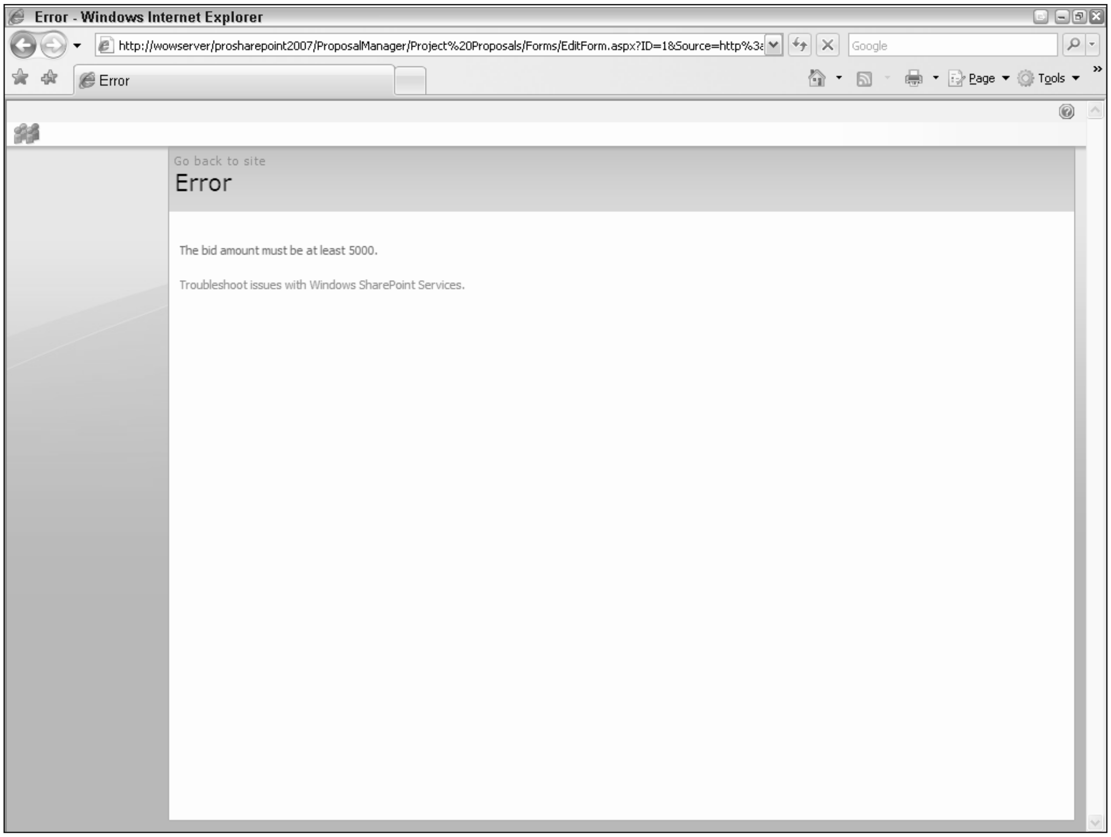
```
        private ProposalManagementPolicyFixedBid fixedBidField;
        private ProposalManagementPolicyTimeAndMaterials timeAndMaterialsField;

        /// <remarks/>
        public ProposalManagementPolicyFixedBid FixedBid {
            get {
                return this.fixedBidField;
            }
            set {
                this.fixedBidField = value;
            }
        }

        /// <remarks/>
        public ProposalManagementPolicyTimeAndMaterials TimeAndMaterials {
            get {
                return this.timeAndMaterialsField;
            }
            set {
                this.timeAndMaterialsField = value;
            }
        }
    }

    #region XML Serialization Support

    /// <remarks/>
    [Serializable]
    [XmlType(AnonymousType=true)]
    public class ProposalManagementPolicyFixedBid {

        private ProposalManagementPolicyFixedBidBidAmount bidAmountField;

        /// <remarks/>
        public ProposalManagementPolicyFixedBidBidAmount BidAmount {
            get {
                return this.bidAmountField;
            }
            set {
                this.bidAmountField = value;
            }
        }
    }

    /// <remarks/>
    [Serializable]
    [XmlType(AnonymousType=true)]
    public class ProposalManagementPolicyFixedBidBidAmount {

        private string errorMessageField;
        private decimal minimumValueField;

        /// <remarks/>
        public string ErrorMessage {
            get {
                return this.errorMessageField;
            }
            set {
                this.errorMessageField = value;
            }
        }

        /// <summary>
        /// Builds the actual error message by performing field substitutions.
        /// </summary>
        public string FormattedErrorMessage
        {
            get
            {
                return ErrorMessage.Replace(
                "$$MinimumValue$$", MinimumValue.ToString());
            }
        }

        /// <remarks/>
        [XmlAttribute]
        public decimal MinimumValue
        {
            get {
                return this.minimumValueField;
            }
            set {
                this.minimumValueField = value;
            }
        }
    }

    /// <remarks/>
    [Serializable]
    [XmlType(AnonymousType=true)]
    public class ProposalManagementPolicyTimeAndMaterials {

        private ProposalManagementPolicyTimeAndMaterialsEstimatedHours
            estimatedHoursField;

        /// <remarks/>
        public ProposalManagementPolicyTimeAndMaterialsEstimatedHours
            EstimatedHours {
            get {
                return this.estimatedHoursField;
            }
            set {
                this.estimatedHoursField = value;
            }
        }
    }
```

```
        /// <remarks/>
        [Serializable]
        [XmlType(AnonymousType=true)]
        public class ProposalManagementPolicyTimeAndMaterialsEstimatedHours {

            private string errorMessageField;
            private decimal minimumValueField;

            /// <remarks/>
            public string ErrorMessage {
                get {
                    return this.errorMessageField;
                }
                set {
                    this.errorMessageField = value;
                }
            }

            /// <remarks/>
            [XmlAttribute]
            public decimal MinimumValue
            {
                get {
                    return this.minimumValueField;
                }
                set {
                    this.minimumValueField = value;
                }
            }

            /// <summary>
            /// Builds the actual error message by performing field substitutions.
            /// </summary>
            public string FormattedErrorMessage
            {
                get
                {
                    return ErrorMessage.Replace(
                    "$$MinimumValue$$", MinimumValue.ToString());
                }
            }
        }

        public class StringCollection : System.Collections.Generic.List<string> {
        }
        #endregion
    }
```

Now you can modify the `ItemUpdating` event receiver to process the attached policy:

```
[SharePointPermission(SecurityAction.LinkDemand, ObjectModel = true)]
public override void ItemUpdating(SPItemEventProperties properties)
{
    try {
        // Apply the current proposal management policy.
        ProjectProposalType.ApplyPolicy(properties,
            ProposalManagementPolicy.PolicyContext.Update);
    } catch (Exception x) {
        properties.ErrorMessage = x.Message;
        properties.Cancel = true;
    }
}
```

If an exception is thrown during the application of the policy, you set the `ErrorMessage` field and cancel the operation. SharePoint will then display an Error page whenever an attempt is made to update a proposal that does not conform to the established policy, as shown in Figure 11-11.


Error - Windows Internet Explorer
http://wowserver/prosharepoint2007/ProposalManager/Project%20Proposals/Forms/EditForm.aspx?ID=1&Source=http%3a
Google
Error
Page
Tools
Go back to site

Error

The bid amount must be at least 5000.

Troubleshoot issues with Windows SharePoint Services.

Figure 11-11

Creating Policy Administration Tools

Associating a custom policy file with a content type is powerful, but administrative users will not be able to change the policy unless they can associate a different file with the content type at runtime. Two approaches come to mind. You could provide a custom user interface for entering the acceptable bid amount and estimated hours and then generate an XML file, or you could create a command-line tool.

Working with XML files via command-line utilities is a lot easier than developing a UI, especially if the schema is changing frequently during development. In addition, you might like to enable a machine-driven process or a script to modify the policy rather than require human interaction. Fortunately, the wise and benevolent SharePoint gods made it quite easy to extend the `stsadm` command-line tool with your own custom commands. Listing 11-13 shows a custom `stsadm` extension for setting the proposal management policy that will be applied to new or existing proposals based on your project proposal content type.

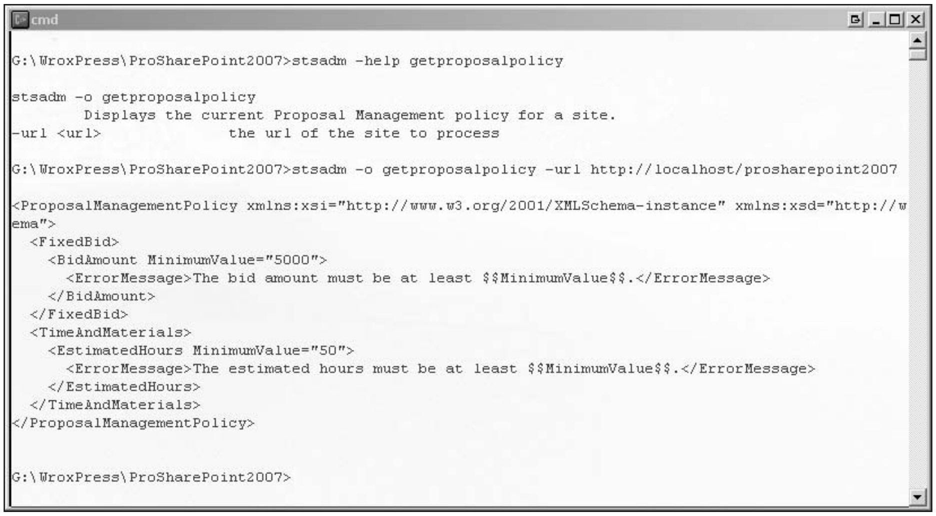
Listing 11-13: A custom STSADM command to apply proposal management policy

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Text;
using Microsoft.SharePoint;
using Microsoft.SharePoint.StsAdmin;
using ProSharePoint2007;

namespace ProposalManager.Admin
{
    /// <summary>
    /// Implements a custom STSADM command to display the proposal management
    /// policy specification for a given site.
    /// </summary>
    public class GetProposalPolicy : ISPStsadmCommand
    {
        #region ISPStsadmCommand Members

        string ISPStsadmCommand.GetHelpMessage(string command)
        {
            string msg = "Displays the current Proposal Management policy.";
            return msg + "\n" + "-url <url>\t\tthe url of the site to process";
        }

        int ISPStsadmCommand.Run(string command,
            System.Collections.Specialized.StringDictionary keyValues,
            out string output)
        {
            int result = 1;
            const string ProposalContentTypeName = "Project Proposal";

            try {
            // validate the arguments
            if (keyValues["url"] == null)
                throw new ApplicationException("No url supplied.");
                // open the website
                using (SPSite site = new SPSite(keyValues["url"])){
                    using (SPWeb web = site.OpenWeb()){
                        // load the Project Proposal content type
                        ContentType ctProjectProposal
                            = new ContentType();
                        if (ctProjectProposal.Create(web,
                            ProposalContentTypeName) == null) {
                            throw new ApplicationException(
                            string.Format(
                            "Failed to locate Content Type '{0}'",
                                ProposalContentTypeName));
                        }

                        // load the policy
                        ProposalManagementPolicy policy
                            = ProposalManagementPolicy.FromContentType(
                                ctProjectProposal);

                        // convert to text and return result
                        output = policy.ToString();
                        result = 0;
                    }
                }} catch (Exception x) {
                    output = x.ToString();
                }

                return result;
            }

            #endregion
        }
    }
```

To deploy the command, you simply install the assembly into the Global Assembly Cache and create a command definition file named `stsadmcommands.proposalmanager.xml` in the `12\CONFIG` folder.

```
<?xml version="1.0" encoding="utf-8" ?>
<commands>
    <command name="setproposalpolicy"
          class="ProposalManager.Admin.SetProposalPolicy,
            ProposalManager.Admin, Version=1.0.0.0,
            Culture=neutral, PublicKeyToken=3c6b2ee283bb579c"/>
    <command name="getproposalpolicy"
          class="ProposalManager.Admin.GetProposalPolicy,
            ProposalManager.Admin, Version=1.0.0.0,
            Culture=neutral, PublicKeyToken=3c6b2ee283bb579c"/>
</commands>
```

Figure 11-12 shows the new command being executed.



```
G:\WroxPress\ProSharePoint2007>stsadm -help getproposalpolicy

stsadm -o getproposalpolicy
        Displays the current Proposal Management policy for a site.
-url <url>              the url of the site to process

G:\WroxPress\ProSharePoint2007>stsadm -o getproposalpolicy -url http://localhost/prosharepoint2007

<ProposalManagementPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://w
ema">
  <FixedBid>
    <BidAmount MinimumValue="5000">
      <ErrorMessage>The bid amount must be at least $$MinimumValue$$.</ErrorMessage>
    </BidAmount>
  </FixedBid>
  <TimeAndMaterials>
    <EstimatedHours MinimumValue="50">
      <ErrorMessage>The estimated hours must be at least $$MinimumValue$$.</ErrorMessage>
    </EstimatedHours>
  </TimeAndMaterials>
</ProposalManagementPolicy>


G:\WroxPress\ProSharePoint2007>
```

Figure 11-12

Summary

Managing the document lifecycle depends on metadata. Content types are the way to manage metadata within SharePoint. You can encapsulate document metadata by combining it with custom behaviors implemented in content type event receivers. You then use those custom behaviors to control all aspects of the document creation and revision phases. Constructing reusable libraries of document-specific components can greatly simplify the task of building document management solutions.

You can extend the notion of encapsulation to include high-level document management policies declared using XML. First you create a custom document management policy schema and then use it to create a default set of policies for your content type. Using the Windows SharePoint Services object model, you can attach a default policy to the content type at runtime and then retrieve it, using XML deserialization, to process the policy against individual document items. Finally, you can extend the `stsadm` command line tool with custom commands specific to your document management solution.