5 Implementation of Extensions

The current version of Cuyahoga (1.5.0) is a mature and well designed web application. But as with any application, there is room for improvements.

This chapter discusses the issues the author wanted to resolve and the steps he took to extend the system.

5.1 General Issues:

Modules can use *UserControls* (.ascx) and *Page*s (.aspx) to provide an user interface for managing their content. Certain functionality is only available within special controls or pages (e.g. indexing for search is only implemented in *ModuleAdminBasePage*)

Some of the functionality (searching, content syndication, persistence, etc.) relies on very different components which do not share a common pattern for the different pieces of functionality

The modules have to provide all infrastructure services regarding the management of their content themselves, there are no basic services provided by the application

The content is isolated from other modules' content, no services are available for connecting, classifying, defining workflows for content etc. between modules

5.2 Concrete Examples

5.2.1 The Search Functionality

Modules (more precisely: the module controller) can implement the *ISearchable* interface which exposes events for notifying the search infrastructure about changes. Additionally, the interface defines a method that is called to retrieve all searchable content (when rebuilding the entire search index).

For edit/admin pages, the framework offers a base class, *ModuleAdminBasePage*, that module pages can extend. This base page offers some methods and events that simplify the access to core services (e.g. the *IndexBuilder*).

The *ModuleAdminBasePage* uses the *ModuleLoader* to load the module controller and inspects it if there are any special interfaces it is interested in (e.g. *ISearchable*).

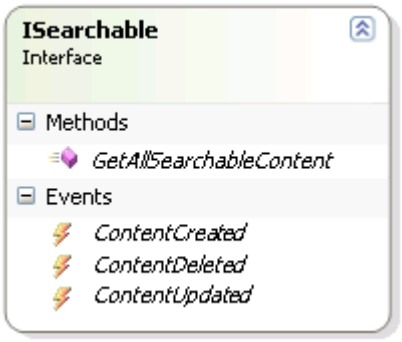

Figure 17: The ISearchable interface

The *ModuleAdminBasePage* registers event listeners for the events *ContentCreated*, *ContentDeleted* and *ContentUpdated* of the module implementing *ISearchable* (e.g. *ArticleModule*) and forwards content for (re)indexing to the *IndexBuilder* if one of these three events occurs.

Figure 18 illustrates this concept.

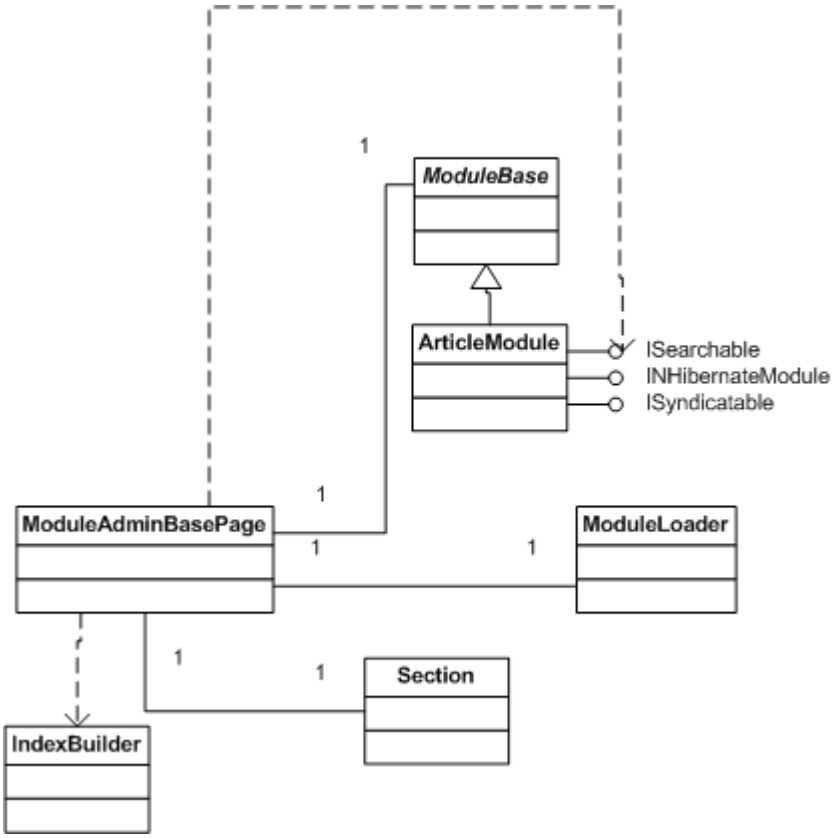

Figure 18: Making a module searchable

While exploring this concept two issues emerge:

First, only pages inheriting from *ModuleAdminBasePage* can directly submit content for indexing. The web controls that can be loaded by *PageEngine* can not inherit from *ModuleAdminBasePage* (since they're of type *System.Web.UI.UserControl*, not *System.Web.UI.Page*), so for these parts of a module there is only the possibility to manually add the functionality that is provided by *ModuleAdminBasePage*.

Second, the indexing is realized through a set of dependent components working quite differently than other module extensions (e.g. Syndication, Persistence, etc.). There is a base page needed that has dependencies on the *IndexBuilder* and the *ModuleLoader* and events have to be fired explicitly to let the base page know that there is content to be indexed.

A better solution would be a generally available indexing/search service that can be configured to automatically index the contents of a module.

5.2.2 Infrastructure Code

Recalling the introduction to Cuyahoga, modules are regarded as “little applications”. While the integration into the CMS works very well concerning the user interface and deployment, core infrastructure services (like persistence, versioning, indexing etc.) have to be implemented manually for each module.

Persistence stands out a from this list because it is supported very well by the usage of NHibernate and the availability of a data access object (*CommonDao*) that can handle basic CRUD operations for different entity types.

Nevertheless, the combination of different infrastructure services is not facilitated through a common pattern, so if multiple operations have to be carried out, e.g. persisting an entity and submitting it for full-text indexing, this has to be specified explicitly every time.

Ideally, there would be no need to explicitly call the respective components, so the module developer can avoid writing repetitive code and doesn't have the responsibility to check if he hasn't forgotten to fire the *OnContentCreated* event, for example.

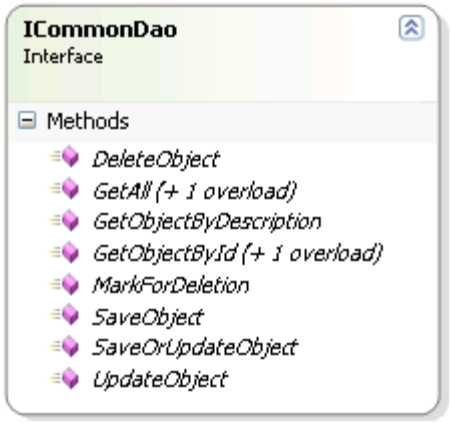

Figure 19: The ICommonDao interface

Code Listing 22 shows the calls needed for saving and indexing an *Article*.

```
public virtual void SaveArticle(Article article)
{
          article.Category = HandleCategory(article.Category);

          if (article.Id == -1)
          {
          this._commonDao.SaveOrUpdateObject(article);
          OnContentCreated(new IndexEventArgs(ArticleToSearchContent(article)));
          }
          else
          {
          this._commonDao.SaveOrUpdateObject(article);
          OnContentUpdated(new IndexEventArgs(ArticleToSearchContent(article)));
          }
}
```

Code Listing 22.: The ArticleModule's SaveArticle method

5.3 Defining Objectives

To be able to provide a solution for the issues mentioned, that is acceptable for an open source project being in productive use, following objectives have to be met alongside the implementation of the extensions:

- Cuyahoga has an active user base using the current version, so no breaking changes are possible
- New services must be able to exist side by side with the old infrastructure
- The existing components and the Microkernel should be leveraged whenever reasonable
- All new services have to provide a contract they adhere to
- The services should be capable of handling content from different types of modules
- The abstraction layer for content must be usable by different types of modules

5.4 Concrete Improvements

The following sections document the concrete changes and extensions that the author introduced to Cuyahoga. While working on the core objectives some by-products were created that are shortly discussed at the end of this chapter.

5.4.1 A New Foundation: The Content Abstraction Layer

To be able to leverage the infrastructure code and to provide common operations for all managed content, an abstraction layer is needed for all CMS content.

As usual in OOP, abstraction can be reached through inheritance, so the author decided to create a new base class for all managed content, the *ContentItem* class.

There are properties that all content shall provide, in order to provide a rich set of out-of-the-box functionality.

While the pros and cons of such a base class should be discussed later, the contract defined through the *IContentItem* interface is explained below:

1. *Category* instances that are assigned to this *ContentItem*
2. Read/Write permissions
3. The creation date
4. The original author
5. A globally unique identifier (GUID)
6. The entity id (also database primary key)
7. A locale definition (e.g. “en-US”) for specifying localized contents
8. The date of last modification
9. The last author who modified the content
10. The publication date (for date driven publishing)
11. The author specifying the publication
12. The publication end date
13. The section this content is (primarily) bound to
14. A summary text, which will be mainly used for auto-generated listings/search results
15. The title (mainly for auto-generated listings/search results)
16. A format definition for automatically creating links to this content (e.g.”/articles/view/{0}”), at the placeholder “{0}”, the actual Id will be inserted (available only after initial insert)
17. A version number, counting the revisions (and usable by the versioning infrastructure)

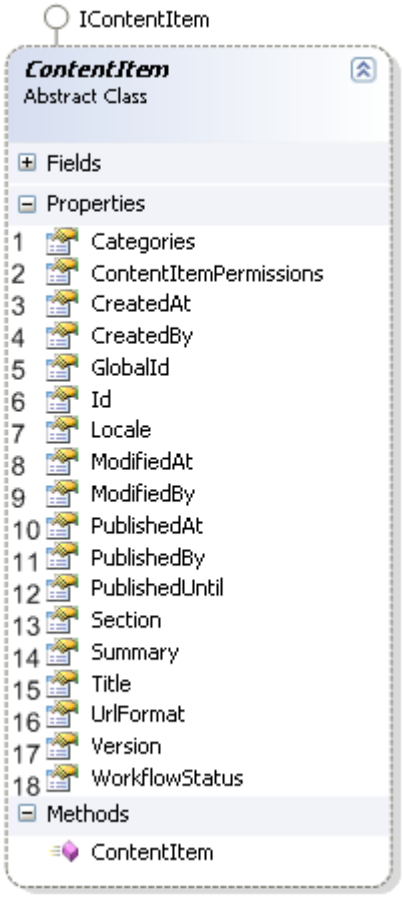

Figure 20: The ContentItem class

18. An enumeration defining certain workflow states (Draft, Review, Approved, Archived, Locked)

This *ContentItem* class is especially meant to provide a contract for structured content, e.g product data, news articles, forum posts, etc. However, there is other content, unstructured content, that should also benefit from this.

So the first non-abstract class inheriting from *ContentItem* is defined: *FileResource*

This class provides common information about files that should be managed through Cuyahoga.

1. Simple count of downloads of this *FileResource*
2. List of *Role*s that are allowed to download a file
3. The file's extension (e.g. ".doc", ".pdf" etc.)
4. The length in bytes
5. The file's mimetype (e.g. "image/jpeg")
6. The file's name (without extension)
7. The physical path to this file (e.g. "C:\Docs\mydoc.pdf")
8. A list of custom attributes (key value pairs) to store any additional needed information with the file (e.g. for pictures their resolution or movies their encoding etc.)

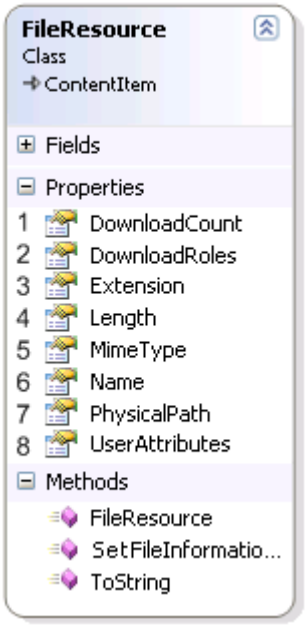

Figure 21: The FileResource class

After having defined contracts for *ContentItem*s and *FileResource*s, there need to be the infrastructure components/services that a developer can make use of without having to further bother with the details of their implementation.

The first aspect that comes to mind is the persistence of this content. Again, NHibernate provides a very feature-rich and solid base for these services.

The *ContentItem* class is mapped as any other persistent class using a NHibernate mapping file and is therefore not listed.

The *FileResource* class mapping file starts with the element "joined-subclass" which signals NHibernate that this class is part of a class hierarchy. Each class mapping for classes inheriting from *ContentItem* will have to start with this element.

The derived classes will have a database table of its own for the additional properties they introduce. The inherited properties, however will be inserted into the database table that is mapped to *ContentItem*.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">

<joined-subclass name="Cuyahoga.Core.Domain.FileResource, Cuyahoga.Core"
extends="Cuyahoga.Core.Domain.ContentItem, Cuyahoga.Core" table="cuyahoga_fileresource">

          <key column="fileresourceid" />

          <property name="Name" column="filename" type="String" length="255"
          not-null="false" />

          <property name="Extension" column="extension" type="String" length="10"
          not-null="false" />

          <property name="PhysicalPath" column="physicalpath" type="String" length="255"
          not-null="true" />

          <property name="Length" column="length" type="Int64" not-null="false" />

          <property name="MimeType" column="mimetype" type="String" length="255"
          not-null="false" />
          <property name="DownloadCount" column="downloadcount" type="Int32"
          not-null="false" />

          <bag name="DownloadRoles" cascade="none" lazy="true"
                    table="cuyahoga_fileresourcerole">
                    <cache usage="read-write" />
                    <key column="fileresourceid" />
                    <many-to-many class="Cuyahoga.Core.Domain.Role, Cuyahoga.Core"
                              column="roleid" />
          </bag>

          <map name="UserAttributes" cascade="all" lazy="true"
          table="cuyahoga_fileresourceuserattributes" order-by="attributekey asc">
                    <cache usage="read-write" />
                    <key column="fileresourceid"/>
                    <index column="attributekey" type="String" length="50"/>
                    <element column="attributevalue" type="String" length="255"/>
          </map>

</joined-subclass>

</hibernate-mapping>
```

Code Listing 23.: Mapping the FileResource class

Since NHibernate supports this Polymorphism in every aspect, it is possible to query for the subclass type only, or for all objects of type *ContentItem*. This greatly simplifies the code that has to handle the content in a generic way because it can dynamically be decided what specialization of *ContentItem* to handle.

As previously stated, the requirement to inherit from a base class to be able to take part in certain processes might look like a tough one.

However, since this base class only contains properties that are part of the business domain of a Content Management System, it is just a logical step into the direction of (re)unifying common domain aspects.

Besides that, the *ContentItem* class is optional and does not have to be used, if the module developer decides to rather implement a different solution.

Now, assuming that other modules let their content inherit from *ContentItem*, it allows for the

infrastructure services to operate on the modules content without having to have detailed knowledge about the individual types, as illustrated in Figure 22.

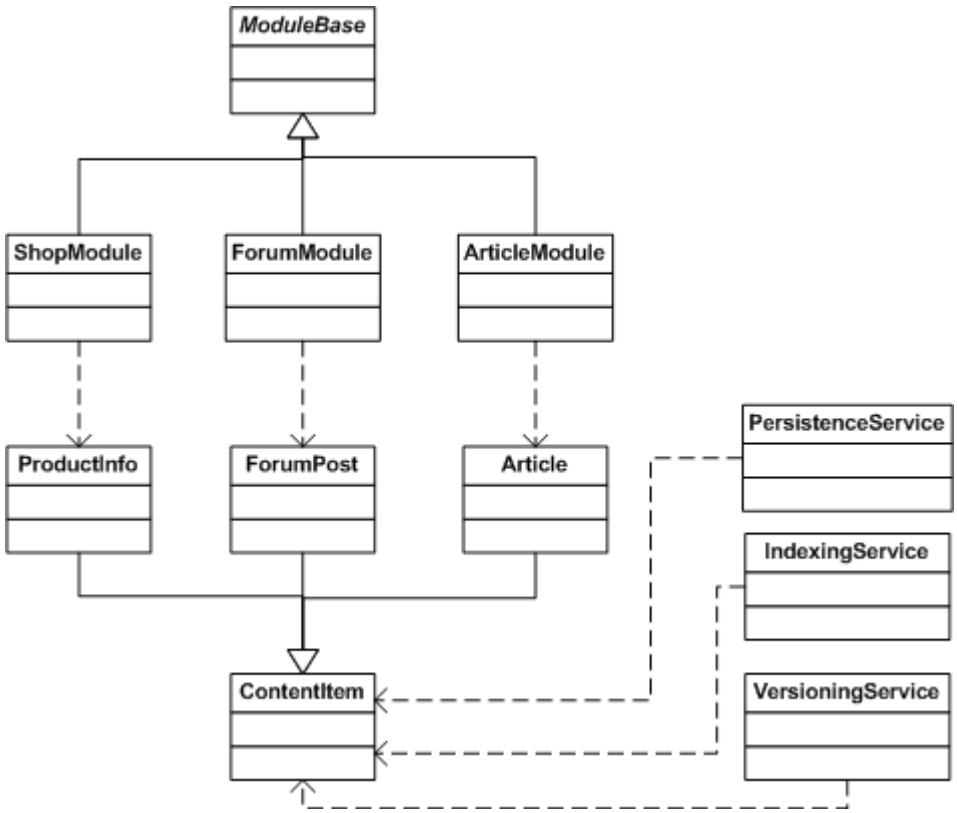

Figure 22: Different services making use of the content abstraction layer

Having the content abstraction layer in place does allow for providing common functionality but the module still needs to obtain an instance of the service and submit the content. The next section discusses how the services can be provided to the module.

5.4.2 Providing Infrastructure Services for Content

There are several basic services that come to mind when looking at content management. Most of the system's modules will support one or more of the following services for managing its content:

Security checks, caching, validation, classification, versioning, indexing for search, persistence, import/export, logging and many more.

The *ContentItem* class provides the most necessary properties for being processed by a CMS. The infrastructure services that have to deal with the *ContentItem* objects may require additional information.

5.4.2.1 Adding Specialized Data and Behaviour to Content

To keep the base class lean and avoid adding properties and methods that are too specific to a given service, the *ContentItem* class should use interfaces that offer this extra bit of data or behavior. So far, the author has defined the following interfaces:

```
namespace Cuyahoga.Core.Service.Search
{
    public interface ISearchableContent
    {
        string ToSearchContent();
        IList<CustomSearchField> GetCustomSearchFields();
    }
}
```

Code Listing 24: Definition of the ISearchableContent interface

The *ISearchableContent* interface demands to methods.

- *ToSearchContent*, which should return a string that will be added to the full-text index
- *GetCustomSearchFields*, which can be used to declare fields that should be indexed separately from the full-text index, so they can be queried by their field name. An object representing a book in a shop module for example could define a *CustomSearchField* "ISBN", so it can not only be retrieved using the full-text index but also by querying for its ISBN number.

```
namespace Cuyahoga.Core.Service.Versioning
{
    public interface IVersionableContent
    {
       VersioningInfo GetVersioningInfo();
    }
}
```

Code Listing 25: Definition of the IVersionableContent interface

The *IVersionableContent* interface only has one method, *GetVersioningInfo*. The returned object is used to describe the properties that should be included in the comparison of two versions of a *ContentItem*. The details of using this interface are demonstrated in the definition of the *VersioningService*.

```
namespace Cuyahoga.Core.Service.Validation
{
    public interface IValidateableContent
    {
        bool IsValid(ref string validationMessage);
    }
}
```

Code Listing 26: Definition of the IValidateableContent interface

The *IValidateableContent* interface can be used to determine if the *ContentItem* is in a valid state. This is mainly intended for validating content before insert or update operations. The validation Message argument can be altered to provide more specific and helpful information for what a user can do if the validation fails (e.g. "The credit card checksum is not correct")

```
namespace Cuyahoga.Core.Service.Logging
{
    interface ILoggableContent
    {
        string GetLogSignature();
    }
}
```

Code Listing 27: Definition of the ILoggableContent interface

Finally, the *ILoggableContent* interface would be used at any time when log information about a specific *ContentItem* is written to the log file. Using the shop module example again, the logger would not only write something generic like “Deleted ContentItem with Id 23453” but a more detailed information like “Deleted Book with ISBN: 3-86640-001-2, category: Science-Fiction”

5.4.2.2 Implementing the Search Service

As described, until version 1.5 of Cuyahoga, for indexing content the usage of several base classes and different components was necessary.

The rewritten search infrastructure will operate on *ContentItem* classes that implement the *ISearchableContent* interface and is made available through the Microkernel.

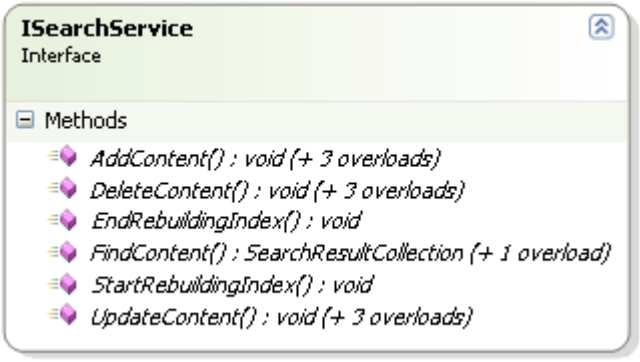

Figure 23: The ISearchService interface

The interface defines the methods the *SearchService* needs to offer. Internally, the *SearchService* still uses the (extended) *IndexBuilder* and *IndexQuery* classes that were present before but hides them behind this simple contract.

The overloads are needed to provide compatibility with the previous implementation. To show what exactly changed, the *StaticHtmlModule* will be taken as an example.

```
private SearchContent StaticHtmlContentToSearchContent(StaticHtmlContent shc)
          {
                    SearchContent sc = new SearchContent();
                    sc.Title = shc.Section.Title;
                    sc.Summary = Text.TruncateText(shc.Content, 200);
                    sc.Contents = shc.Content;
                    sc.Author = shc.ModifiedBy.FullName;
                    sc.ModuleType = shc.Section.ModuleType.Name;
                    sc.Path = this.SectionUrl;
                    sc.Category = String.Empty;
                    sc.Site = shc.Section.Node.Site.Name;
                    sc.DateCreated = shc.UpdateTimestamp;
                    sc.DateModified = shc.UpdateTimestamp;
                    sc.SectionId = shc.Section.Id;
                    return sc;
                    }
```

Code Listing 28.: Converting a StaticHtml object to an instance of SearchContent

For the *StaticHtmlModule* to be able to submit its content for the index, it first has to convert the *StaticHtml* object it manages to a helper object “SearchContent” because until now the *IndexBuilder* only accepted objects of type *SearchContent* for indexing.

After having converted the module specific content, the module's edit page needs to register the events that inform the *ModuleAdminBasePage*.

```
#region ISearchable Members

public SearchContent[] GetAllSearchableContent()
{
          StaticHtmlContent shc = GetContent();
          if (shc != null)
          {
                    SearchContent[] searchContents = new SearchContent[1];
                    searchContents[0] = StaticHtmlContentToSearchContent(shc);
                    return searchContents;
          }
          else
          {
                    return new SearchContent[0];
          }
}
public event Cuyahoga.Core.Search.IndexEventHandler ContentCreated;

protected void OnContentCreated(IndexEventArgs e)
{
          if (ContentCreated != null) ContentCreated(this, e);
}
public event Cuyahoga.Core.Search.IndexEventHandler ContentDeleted;

protected void OnContentDeleted(IndexEventArgs e)
{
          if (ContentDeleted != null) ContentDeleted(this, e);
}
public event Cuyahoga.Core.Search.IndexEventHandler ContentUpdated;

protected void OnContentUpdated(IndexEventArgs e)
{
          if (ContentUpdated != null) ContentUpdated(this, e)
}
#endregion}
```

Code Listing 29: The StaticHtmlModule implementing ISearchable

Using the new *SearchService* this is greatly simplified.

```
public StaticHtmlModule(ISearchService searchService)
{
          this.searchService =  searchService;
}

public void SaveContent(StaticHtmlContent content)
{
          [... saving to database ]
          this.searchService.AddContent(StaticHtmlContent content);
}
```

Code Listing 30: Indexing content with the new SearchService

Now, since the *SearchService* is registered with the Microkernel, the module only needs to define a constructor argument that matches *ISearchService* to obtain an instance and can call it at any time, anywhere in the module providing a *StaticHtmlContent* object that is inheriting from *ContentItem* and implements the *ISearchableContent* interface.

The rewritten *IndexBuilder* knows how to index the content, obsoleting the need of a helper object like *SearchContent*.

Code Listing 31 shows the shortened implementation of the new *BuildDocumentFromContentItem* method.

First, the method checks if the supplied *ContentItem* really implements *ISearchableContent* and otherwise throws an error.

Then, the simple properties (Title, CreatedBy etc.) are added to the Lucene *Document* as *Field*s.

Finally, the method loops through the collections of *Category*, *ContentItemPermission* and *CustomSearchField* objects and adds this information to the Lucene *Document*, as well.

```
private Document BuildDocumentFromContentItem(IContentItem contentItem)
{
  ISearchableContent searchInfo = contentItem as ISearchableContent;
  if (searchInfo == null) throw new
  ArgumentException("Argument must implement ISearchableContent");

  string plaintext = searchInfo.ToSearchContent();
  string path = string.Format(contentItem.UrlFormat, contentItem.Id);

  Document doc = new Document();
  doc.Add(new Field("title", contentItem.Title, Field.Store.YES,
  Field.Index.TOKENIZED));
  doc.Add(new Field("summary", contentItem.Summary, Field.Store.YES,
  Field.Index.TOKENIZED));
  doc.Add(new Field("contents", plaintext, Field.Store.NO, Field.Index.TOKENIZED));
  doc.Add(new Field("author", contentItem.CreatedBy.FullName, Field.Store.YES,

  [ ... ]

  foreach (Category cat in contentItem.Categories)
  {
    doc.Add(new Field("category", cat.Name, Field.Store.YES, Field.Index.UN_TOKENIZED));
  }
  foreach (ContentItemPermission permission in contentItem.ContentItemPermissions)
  {
    if (permission.ViewAllowed)
    {
    doc.Add(new Field("viewroleid", permission.Role.Id.ToString(), Field.Store.YES,
    Field.Index.UN_TOKENIZED));
    }
  }
   foreach (CustomSearchField field in searchInfo.GetCustomSearchFields())
  {
    Field.Store store = field.IsStored ? Field.Store.YES : Field.Store.NO;
    Field.Index index =
    field.IsTokenized ? Field.Index.TOKENIZED :Field.Index.UN_TOKENIZED;
    if (field.FieldKey != null && field.FieldValue != null)
    {
    doc.Add(new Field(field.FieldKey, field.FieldValue, store, index));
    }
  }
  return doc;
}
```

Code Listing 31.: The rewritten IndexBuilder converts ContentItems to Lucene Documents

5.4.2.3 Implementing the Versioning Service

A very important feature of any CMS is the ability to keep different versions of content. Until now, Cuyahoga does not support versioned content, nor do any of the modules available.

The versioning functionality should be implemented as simple (and robust) as possible but by doing so, it should not affect the domain model (e.g. by requiring special meta attributes etc.) and it should keep the usage of resources low (by not simply copying the whole entities for every version increment).

Furthermore, since the type of object will not be known in advance, the versioning service should be prepared to handle arbitrary, complex objects.

While the concrete algorithm is not of primary interest in the context of this thesis, at least the concept should be explained.

Upon initial saving of an entity, all versionable properties will be saved in the *VersionedProperty* collection of *VersionedItem*, resulting in a complete copy. The next time the entity is edited and updated, its properties will be matched against the previous version and a new entry for *VersionedItem* will be inserted in the database. If no change is detected, only the references from the old *VersionedProperty* entries to the new *VersionedItem* will be updated. If there happened a modification, the new entry for *VersionedProperty* will be created and associated with the new *VersionedItem*.

VersionedItem (before change):

Id	Type	Version
001	Article	1.0

VersionedProperty (before change):

FK VersionedItem	Type	PropertyKey	PropertyValue
001	string	Title	New products are out
001	string	Summary	We are pleased to anounce that...

VersionedItem (after change of property “Title”):

Id	Type	Version
001	Article	1.0
002	Article	2.0

VersionedProperty (after change of property “Title”):

FK VersionedItem	Type	PropertyKey	PropertyValue
001	string	Title	New products are out
001, 002	string	Summary	We are pleased to anounce that...
002	string	Title	**Very n**ew products are out

Using this approach, only the changed value for Title is added to the database, the Summary has not changed, so only a reference is added from the old Summary to the new *VersionedItem* entry.

Figure 24 shows all components and entities involved.

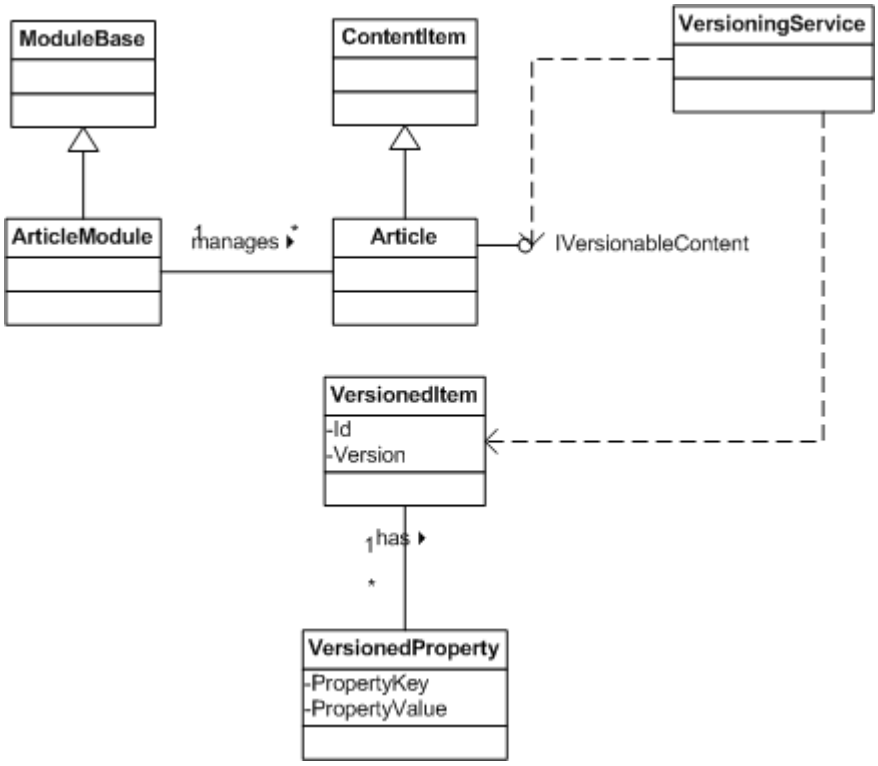

Figure 24: Interaction between the VersioningService and ContentItems

As previously explained, *ContentItem*s that should be versioned have to implement the *IVersionableContent* interface.

Figure 25: The IVersionableContent interface

GetVersioningInfo is supposed to return a *VersioningInfo* object that defines the properties (and child objects) that are included for the versioning process.

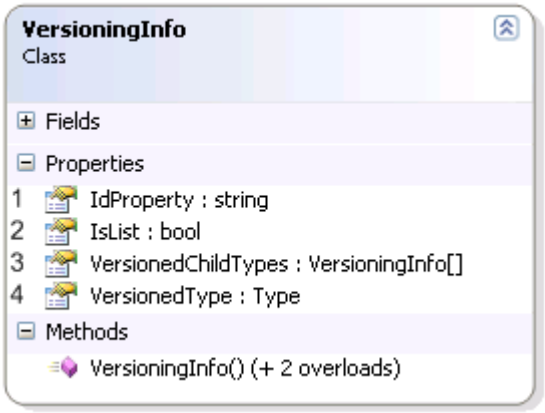

Figure 26: The VersioningInfo class

The *VersioningInfo* class is used to describe how and which complex child objects of the submitted ContentItem are to be handled by the *VersioningService*.

1. The property to be used as the identifier (only the *ContentItem* class itself is required to have an Id property, child objects are not.
2. Defines if the child object is contained within a list
3. The child object itself can have childs, too
4. The Type of the child object

By using this information, the *VersioningService* can recursively convert all versionable properties to their string representation and save the changes as shown above.

5.4.3 Chaining the Infrastructure Services

In a Content Management System most of the modules will support one or more of the following infrastructure services regarding managing its items: Security, Caching, Validation, Versioning, Index/Search, Persistence, Import/Export, Logging, Internationalization etc.

The problem is that every service object has to be instantiated and the content(s) that the operations are to executed upon need to be individually submitted to the respective service.

Looking at a concrete *ContentItem* that is being edited, multiple operations will be performed, if the content item has to be changed.

Figure 27 shows a series of such calls that are likely to happen, if one or more of the *ContentItem* properties are changed.

Figure 27: Chaining the infrastructure services

First, it has to be checked if the changes are valid.

Next, the content item will probably get re-indexed and the changes will be recorded to some version store.

Then, some info about the operations might be logged.

Last, the changed content item itself has to be saved/updated in the database.

Not all modules will always use the same set of services and some modules will probably want to implement a specialized version of a given service.

Using the new extensions to Cuyahoga, there are several ways this could be realized.

The simplest solution would be to put all needed service contracts to the module's constructor (Constructor Injection) and use them as needed but this again involves a lot of repetitive code.

Another solution would be to make the services “chainable”. A common way of chaining objects is to let all chainable objects implement the same interface and add a constructor parameter whose type matches this interface.

The service would then override the interface methods or properties it is interested in (e.g. the indexing component will probably not be interested in read-only operations) and finally forward the call to the “inner service” (if one has been supplied with the constructor argument)

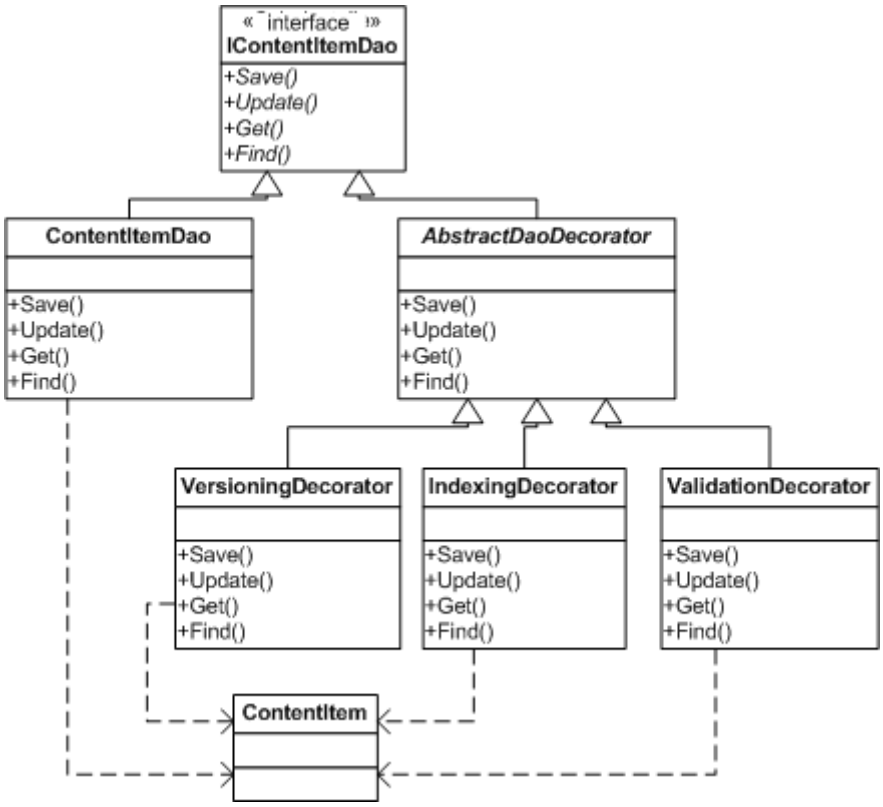

Figure 28: Implementing the Decorator Pattern

This approach is a well known pattern called Decorator Pattern [Buschmann et al. 2001]. A concrete implementation for the CMS scenario is shown in Figure 28.

There is an interface *IContentItemDao* that defines several very common operations (Get, Save, Find etc.) that can be overridden by the decorators. The decorators then can perform any operations needed and forward processing to the next decorator.

To make sure that all decorators behave the same way, an abstract class is used which the decorators have to inherit from. Code Listing 32 shows a (shortened) possible implementation.

One interesting detail is the usage for Generics. Generics are similar to C++ templates and allow for defining type placeholders that will be evaluated during runtime. Additionally the generic type (T) is constrained to only accept *IContentItem*.

```
public abstract class AbstractDaoDecorator<T> : IContentItemDao<T> where T :
IContentItem
{
    protected IContentItemDao<T> contentItemDao;

    //constructor injection
    public AbstractDaoDecorator(IContentItemDao<T> contentItemDao)
    {
        this.contentItemDao = contentItemDao;
    }

    #region IContentItemDao<T> Members

    public virtual T GetById(long id)
    {
        //forward call to inner dao
        return this.contentItemDao.GetById(id);
    }

    public virtual System.Collections.Generic.IList<T> GetAll()
    {
        //forward call to inner dao
        return this.contentItemDao.GetAll();
    }

    public virtual T Save(T entity)
    {
        //forward call to inner dao
        return this.contentItemDao.Save(entity);
    }

    public virtual void Delete(T entity)
    {
        //forward call to inner dao
        this.contentItemDao.Delete(entity);
    }

    #endregion
}
```

Code Listing 32: Implementation of the AbstractDaoDecorator class

So now, it is possible to chain several services, as Code Listing 33 shows.

```
public void SaveArticle(Article article)
{
          IContentItemDao<Article> contentItemDao = new VersioningDecorator<Article>(
                    new ValidationDecorator<Article>(
                              new IndexingDecorator<Article>(
                                        new ContentItemDao<Article>(
          ))));

          contentItemDao.Save(article);
}
```

Code Listing 33: Manually creating a Decorator Chain

The only remaining issue is that both solutions (using Constructor Injection versus the Decorator Pattern) are not perfect.

Using Constructor Injection, there is no need to directly obtain a reference to the desired services but the services have to be called individually for every operation.

Using the Decorator Pattern, the services are chained automatically. But the references can't be obtained so easily because the decorators are not available through the Microkernel and if they would, how would the Microkernel know which decorators should execute?

The final solution is a combination of these two approaches, exploiting the possibility to inspect the *ContentItem* for special interfaces (e.g. *ISearchableContent*) and configuring a “standard chain” in the Microkernel's component configuration.

Combining the Constructor Injection and the Decorator pattern is realized through registering the *IContentItemDao* interface as a service contract with the Microkernel.

Since all decorators have to implement this interface, they are obtainable via Dependency Injection. The Castle Microkernel has the ability to recognize service overrides, so mulitple variants of the *IContentItemDao* can be registered (one for the *VersioningDecorator*, one for the *ValidationDecorator*, etc.)

The order in which the decorators are called is only important for the actual *ContentItemDao*, which provides the database access and should therefore be called last.

Code Listing 34 shows an excerpt of the Microkernel's configuration file, where the “standard chain” is defined.

The parameter entries enclosed in “${...}” are named references to the specialized implementations of *IContentItemDao*. These are used for determining which actual service to supply as constructor argument.

```
    <!-- Decorator chain-->
    <component
      id="core.searchdecorator"
      service="Cuyahoga.Core.DataAccess.IContentItemDao`1, Cuyahoga.Core"
      type="Cuyahoga.Core.Decorators.SearchDecorator`1, Cuyahoga.Core">
      <parameters>
        <contentItemDao>${core.versioningdecorator}</contentItemDao>
      </parameters>
    </component>
    <component
      id="core.versioningdecorator"
      service="Cuyahoga.Core.DataAccess.IContentItemDao`1, Cuyahoga.Core"
      type="Cuyahoga.Core.Decorators.VersioningDecorator`1, Cuyahoga.Core">
      <parameters>
        <contentItemDao>${core.contentitemdao}</contentItemDao>
      </parameters>
    </component>
    <component
  id="core.contentitemdao"
  service="Cuyahoga.Core.DataAccess.IContentItemDao`1, Cuyahoga.Core"
  type="Cuyahoga.Core.DataAccess.ContentItemDao`1, Cuyahoga.Core">
    </component>
```

Code Listing 34.: Configuring the Decorator Chain with the Microkernel

Code Listing 35 shows a snippet from the *SearchDecorator* class. The constructor arguments are injected through the Microkernel. The Save method evaluates if indexing should happen at all and then inspects the supplied entity for the *ISearchableContent* interface. In case it is found, the content will be indexed and the execution flow is handed over to the next (inner) *IContentItemDao*.

```
class SearchDecorator<T> : AbstractDaoDecorator<T> where T : IContentItem
{
    private ISearchService searchService;

  public SearchDecorator(IContentItemDao<T> contentItemDao, ISearchService
searchService)
        : base(contentItemDao)
    {
        this.searchService = searchService;
    }

  private bool UseInstantIndexing
  {
        get
        {
            return (Boolean.Parse(Config.GetConfiguration()["InstantIndexing"]));
        }
  }

  public override T Save(T entity)
  {
        if (UseInstantIndexing)
        {
            if (entity is ISearchableContent)
            {
                this.searchService.AddContent(entity);
            }
        }
        //forward call to inner dao
        return this.contentItemDao.Save(entity);
  }
```

Code Listing 35: Implementation of the SearchDecorator class

5.5 Gains and By-Products

Using the presented content abstraction layer and the chained services, module developers are freed from providing a lot of infrastructure code and can utilize the services as needed.

The created services all share a common pattern and new services can easily be added in a predefined manner and without interfering with the existing core.

Since content now shares a common foundation, the CMS users can connect and categorize it, regardless of the type of module that manages it.

Some by-products that were implemented for making the new services available to the user have not been documented in this thesis and therefore should be shortly mentioned here.

The module loading strategy has been changed from on-demand to predefined, because performance issues arose during stress tests with dynamic loading. The CMS administrator can now “activate” the module in the back-end area

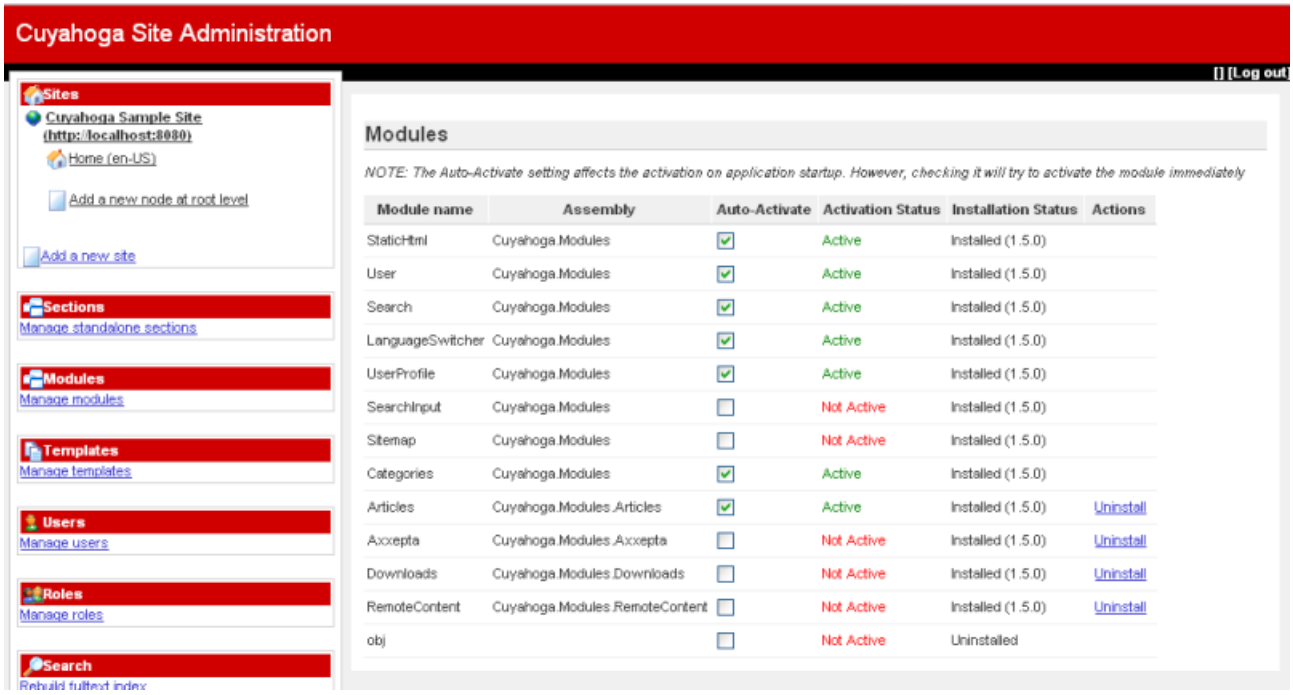

Figure 29: The extended module manager

A completely new administration area was created. The “category manager” allows for creating and editing content categories

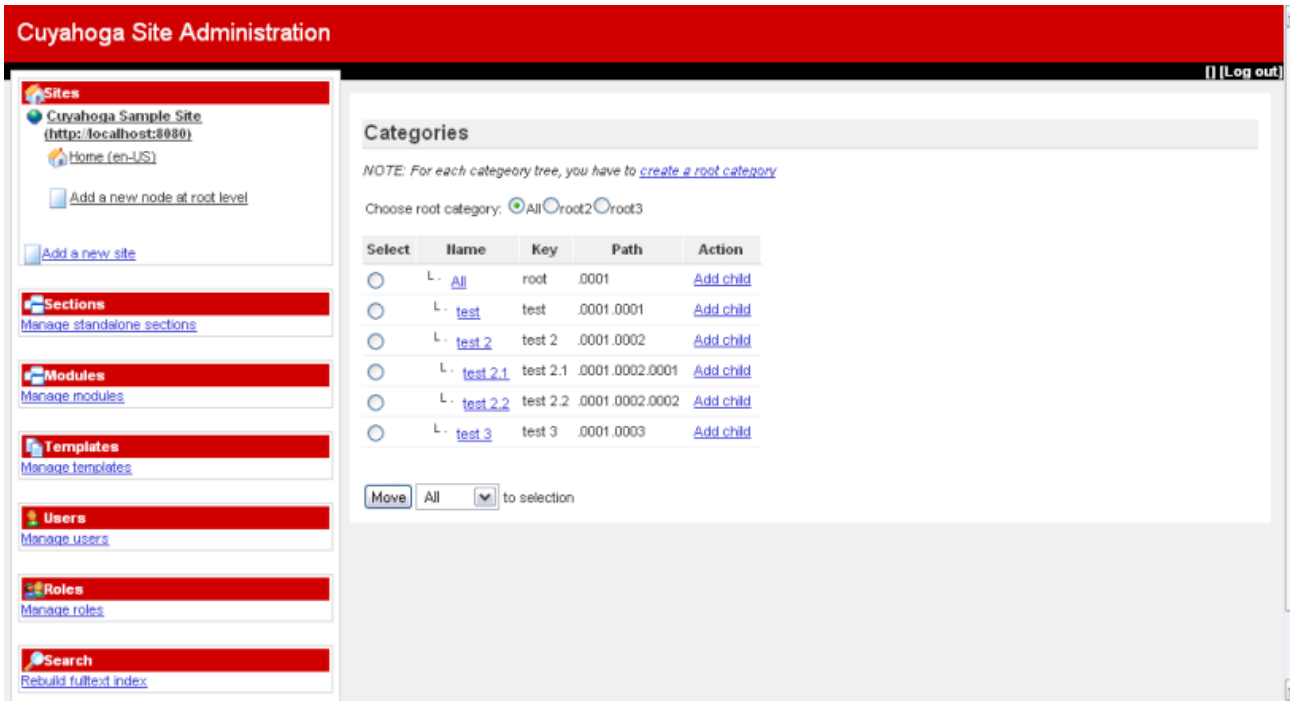

Figure 30: The new category manager

A category module was implemented to be able to display the categories on the CMS pages. The module implements the *IActionProvider* interface, so it can be connected with the search module for limiting the search to only selected categories

The search module was improved to reflect the new *SearchService* capabilities. A category filter was added and the search module now has to submit the *User* instance that issued the query, so the *SearchService* can automatically filter the result set based on access privileges.

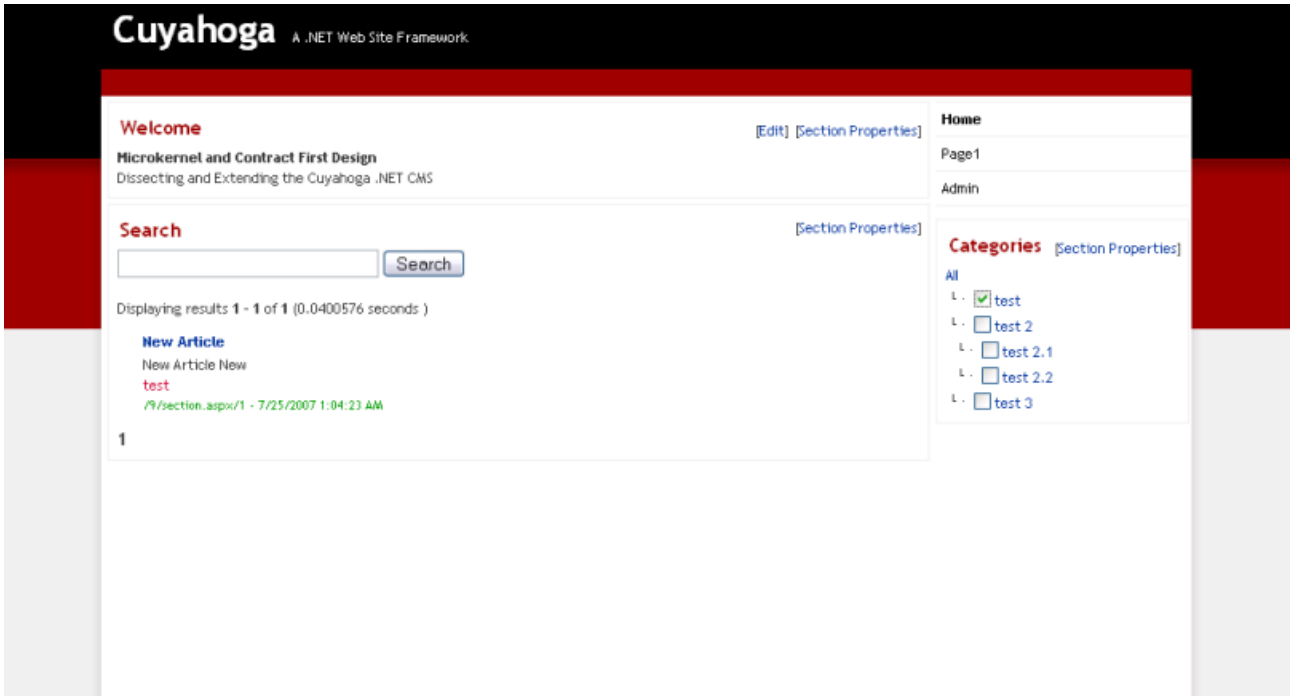

Figure 31 Combining search module and category module

The libraries used by Cuyahoga (Castle, NHibernate, Lucene.NET, etc.) where updated to the latest version and resulting conflicts were resolved.