# Manakin

## Developer's Guide

**The second version of the DSpace XML UI project**

Scott Phillips

Cody Green
John Leggett
Alexey Maslov
Adam Mikeal
Brian Surratt


Revision 1

October 2005

http://di.tamu.edu/

## Abstract

Manakin is the second release of the DSpace XML UI project that implements an XML-based interface enabling communities and collections to maintain their own distinct look-and-feel. The project builds upon the Cocoon framework together with the **D**igital **R**epository **I**nterface (DRI) schema and uses *Themes* to style the content and packages called *Aspects* for portability. These features combine to provide improvements in efficiency and increased modularity in content generation.

## Table of Contents



## Figures

# Introduction

This document is a guide for developing in the *DSpace XML UI: Manakin* project. DSpace is an open source digital repository system developed by MIT and HP used by many institutions worldwide. The DSpace information model is similar to the organizational structure of a university composed of colleges, departments, schools, labs, centers, etc. These organizational units may be mapped to communities and collections within DSpace. When DSpace is used as a centralized Institutional Repository, it is beneficial to provide the individual communities the ability to present these collections with their own distinct look-and-feel.

The DSpace XML UI project is currently under development by Texas A&M University in order to provide major improvements to the user interface of DSpace; the fundamental feature this project adds is the ability for each community and collection within DSpace to establish a unique look-and-feel, one that might extend outside of DSpace into an existing web presence. We believe providing communities with this ability will increase the adoption of DSpace.

Manakin is the second release of the XML UI project's customizable DSpace interface. This second release offers several improvements over previous versions in both efficiency and modularity. While previous versions implemented a **D**ocument **O**bject **M**odel (DOM)[1] based approach, Manakin builds upon the Cocoon[2] framework to use a **S**imple **A**PI for **X**ML (SAX)[3] based approach. To accomplish this customizable interface, Manakin uses *Themes* to style the content and packages called *Aspects* to modularize the generation of the content.

## Goals

The Cocoon based Manakin project has five design goals.

- **Allow each community & collection to maintain a distinct look and feel.**

  Allowing a unique "look-and-feel" for each community enables branding and a feeling of ownership by the community. This can increase the adoption of DSpace by these communities.

- **Separate business logic from stylistic design.**

  The separation of business logic from the stylistic design of the interface enables changes in style to be accomplished without modifying the underlying architecture, ensuring the maintainability of the DSpace code-base. This use of "single-source publishing" means interfaces in DSpace can be easily delivered in multiple formats.

- **Establish an interface-level component architecture.**

  The component architecture of Manakin consists of three tiers; at each tier, levels of componentization are present to enable parallel development. As the tiers descend, progressively less skill and experience are required of someone to be productive at that tier.

  1. Java / Cocoon development Tier

     Requires Java development and Cocoon expertise and is able to perform any functional customization to DSpace using the supplied component architecture.

  2. XML / XSL Theme Tier

     Requires XML and XSLT knowledge, but no Java experience. At this tier; information currently being displayed by DSpace can be further processed before being displayed to the user. Major site-wide presentation, structural, and limited computational changes may be made to an entire Theme, or an individual page.

  3. HTML / CSS style Tier

     Requires only basic XHTML knowledge and can change the style of how information is presented to the user across an entire Theme.

- **Enable internationalization & localization of content.**

  Although currently supported in the JSP implementation of DSpace, a major goal of this redesign is to further enhance internationalization & localization by incorporating the concepts into the designed modular architecture.

- **Provide an alternative interface that does not replace the existing JSP based interface.**

  There currently exists a functioning interface to DSpace, and Manakin will operate along-side this existing interface. This means that organizations using DSpace will have a choice of which interface to utilize. This is especially important to those who have invested in local customizations of the JSP interface which may not be compatible with Manakin.

## Why Cocoon?

Cocoon describes itself as a web development framework built around the concepts of **S**eparation **o**f **C**oncerns (SoC)[4] and a component-based architecture. The framework is based upon the **S**imple **A**PI for **X**ML (SAX). These two features provide the primary motivation for using Cocoon in this project.

The SoC design principle and component architecture allow for parallel development of the application, with each developer focusing on a single part without affecting others. DSpace development is represented by an open and very diverse community with very little centralized focus. Different members of the community may develop functionality that conflicts with other developers' changes. This has caused problems in the past, as new features incorporated into DSpace conflict with local customizations in the existing

JSP interface that many organizations maintain. Cocoon addresses this problem by allowing individual components to be replaced without affecting the entire system. The Manakin design builds on this foundation with the concepts of Themes & Aspects, (described later).

### Where Manakin fits in with DSpace

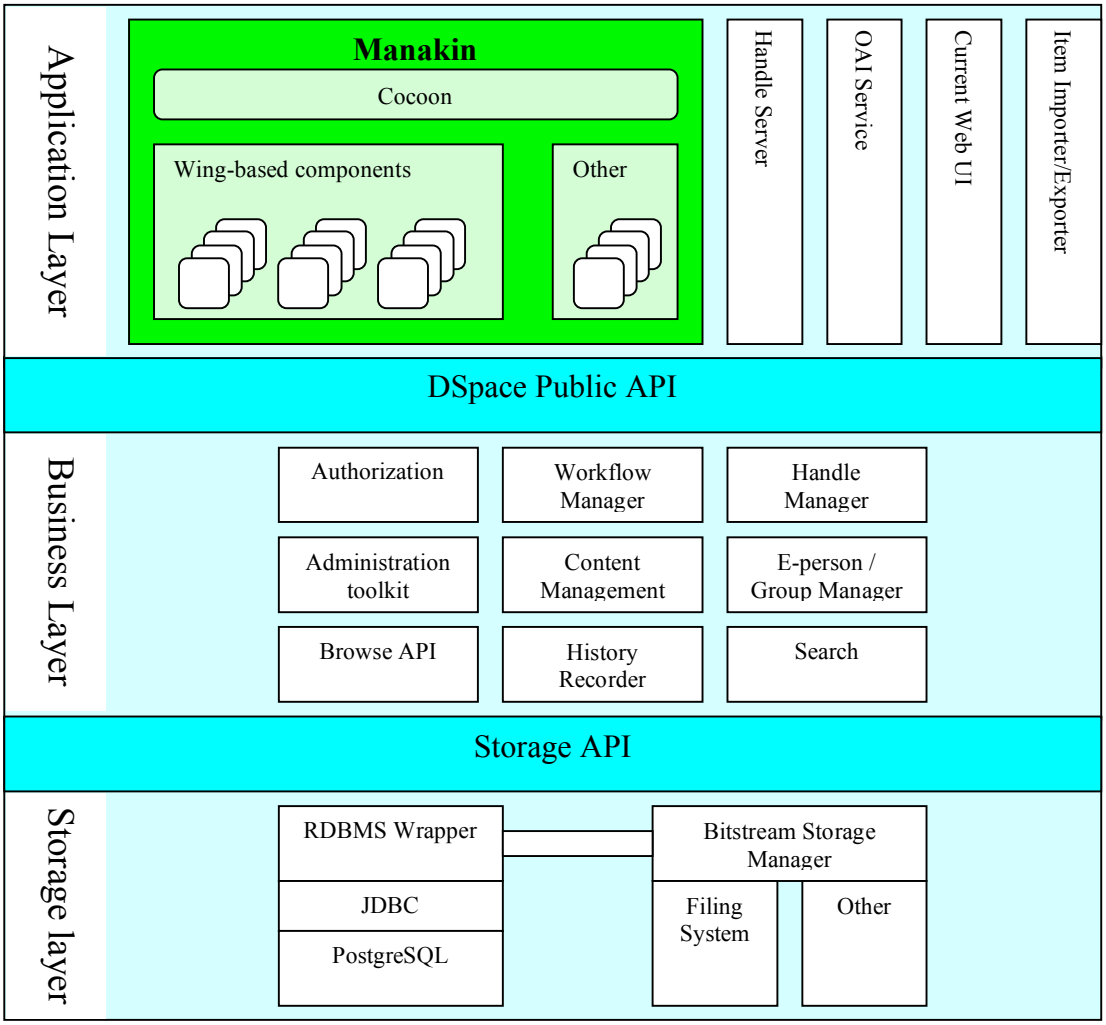


**Figure 1: Where Manakin fits within DSpace.**

Manakin resides in the application layer of DSpace and requires no modification to the DSpace core API. Figure 1 gives an architectural overview similar to the one found in the DSpace manual. Manakin is composed of a Cocoon element that interfaces with DSpace through a set of components. Some of these components are Wing-based, and others are normal Cocoon components. Wing-based components utilize the *Wing* framework for streamlined, DOM-like manipulation of the document at each stage of the pipeline. The Wing framework, developed in conjunction with Manakin, is discussed in detail later in this document.

## Cocoon Fundamentals

As described earlier, Apache Cocoon is a web development framework designed around the concept of **S**eparation **o**f **C**oncerns (SoC) using a component-based architecture. The framework focuses on publishing websites through using a **S**imple **A**PI for **X**ML (SAX) event-based pipeline.

The SoC design principle in Cocoon is realized by *components*. Components are operations that are joined together to form a *pipeline*. Several broad types of components exist: Matchers, Generators, Transformers, Serializers, Selectors, Views, Readers, and Actions. These components are not interconnected; there are no method calls from one component to the next. Instead, their interaction is guided by their constructor, which in most cases is the *sitemap* as described below.

### Pipelines

The *pipeline* is a fundamental concept of Cocoon. As a web request enters the pipeline, various components transform the content at each stage until it reaches the end of the pipeline, where the content is transmitted to the user. Websites are built through the arrangement of these pipeline components, which is sometimes referred to as a "Lego™-like" approach. Under this approach, a developer is able to build a dynamic website by hooking together components without resorting to any "real" programming.

### Sitemap

The *sitemap* is a set of XML documents that describe how all the Cocoon components are configured together. The sitemap contains two major parts: a component definition section which describes each type of component, and a pipeline section that defines how those components are arranged. The sitemap is the heart of any Cocoon-based website.

### Component Types

- Generators

  *Generators* create a stream of SAX events for processing by other components. Typically this is derived from a static XML file on the file system. However, this content may also be dynamically generated on the fly or may be the result of a new HTTP request, either internal to the website or to an external website. Typically, a pipeline's first component is a Generator.

- Transformers

  *Transformers* take a stream of SAX events and perform processing on them. The SAX stream may be processed to add or remove content from the pipeline. The most common Transformer is an XSLT transformer.

- Serializers

  *Serializers* are the end point of a pipeline. They transform the SAX events into binary or character streams for transmission to the client.

- Matchers

  *Matchers* are used to match requests against wildcard or regular expression patterns. Based on those matches, choices can be made on which components to include in the pipeline and which to ignore.

- Selectors

  *Selectors* are similar to matchers with additional flexibility. Where Matchers can only make binary decisions of "yes/no", a selector can make multiple choice decisions, similar to an "if – else – if – else" construct.

- Actions

  *Actions* do not add or remove content from the pipeline, nor do they make decisions about the flow of the pipeline. Instead they perform operations that may affect future decisions by matchers or selectors.

- Readers

  *Readers* are both the starting and ending point of a pipeline. Normal pipelines must start with a generator and end with a serializer. Readers combine these functions and are typically used for binary content such as images.

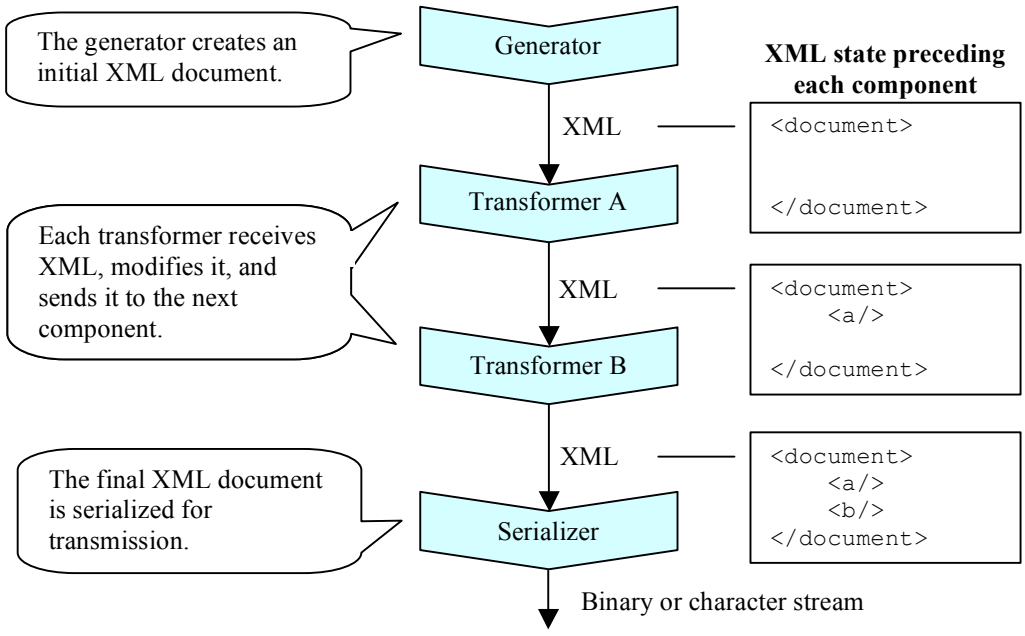


**Figure 2: Example Cocoon pipeline**

## The DRI Schema

The **D**igital **R**epository **I**nterface (DRI) schema is a centralized and generic schema allowing it to be applied to all DSpace pages. While a page is being constructed and before the Theme is applied, the document will always conform to the DRI schema. This common format allows Themes to be interchangeable across the whole site and not tied to any specific Aspect package or component. A full description of the DRI schema may be found in the DRI Schema Reference[5].

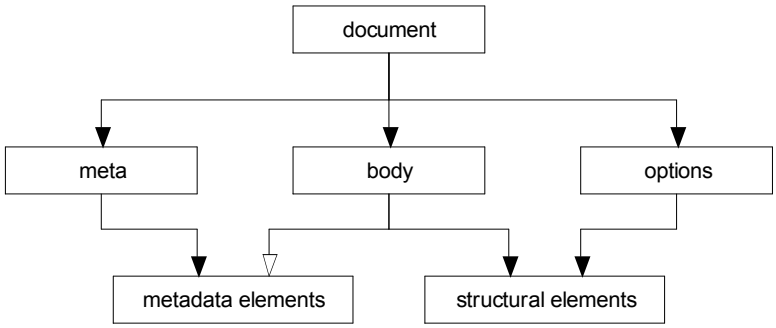


**Figure 3: Top level structure of a DRI document**

The schema consists of three top level elements: `<meta>`, `<body>`, and `<options>`, all contained inside a DRI `<document>`. These three top level elements contain two types of elements: metadata and structural. The `<meta>` element contains metadata information about the page being displayed, the user who is authenticated, the repository, and any artifacts that may be included. For artifacts, the metadata is encoded inside a **M**etadata **E**ncoding & **T**ransmission **S**tandard (METS) object. METS provides a standard to wrap the metadata together with bundles & bitstreams. The first implementation of Manakin will encode Dublin Core metadata inside the METS objects. However, it is reasonable to expect that in the future as DSpace expands to other metadata standards new components will be written to incorporate other metadata standards as well.

The `<body>` and `<options>` elements differ from the `<meta>` element because they contain structural elements. Structural elements are used to build a generic representation of a DSpace page. The `<body>` element can also make use of metadata content through the use of metadata references. These references appear within the structural elements and reference metadata contained within the `<meta>` element tree.

### Why DRI?

The DRI schema was developed and chosen over other schema standards such as XHTML or TEI for two reasons: the inclusion of a native metadata format, and ease of Theme maintainability. The inclusion of metadata in native METS format enables the Theme to choose the best method to render the artifact for display. Under other schemas that metadata would have to be translated to structural elements such as an HTML `<table>`. While encoded in structural elements, the versatility of the metadata is severely limited. The second reason for the selection of a new schema is ease of Theme maintainability. Popular schemas such as XHTML suffer from the problem of not relating elements together. For example, if a heading precedes a paragraph, the heading is

related to the paragraph not because it is encoded as such but because it happens to precede it. When these structures are attempted to be translated into formats where these types of relationships are explicit, the translation becomes tedious, and potentially problematic.

We have constructed this new schema by incorporating other standards when appropriate, such as Cocoon's i18n schema for internationalization, DCMI's Dublin Core, and the Library of Congress's METS schema. The design of structural elements was derived primarily from TEI as well as from many of the design patterns present in preexisting standards such as DocBook and XHTML. While the structural elements were designed to be easily translated into XHTML, they preserve the semantic relationships for use in more expressive languages.

## Design Overview

*Theme* and *Aspect* packages work together to deliver stylized content to the user. Themes are responsible for stylizing a **D**igital **R**epository **I**nterface (DRI) document, while Aspects are responsible for building a DRI document.

### Themes

Manakin *Themes* stylize the content generated by Manakin to a display format suitable for the user. The format typically is XHTML, however, nothing prevents the use of another format, such as PDF, or a graphical format like SVG. Themes are implemented as XSL stylesheets applied in multiple stages to a pipeline's contents. The stylesheets, along with any static resources that may be required by the theme such as images, multimedia, CSS stylesheets, help documents, and translations, are all packaged together for portability. Themes may be configured to apply to all pages in one particular community / collection, or they may be applied to many communities / collections. They can also be configured to apply to a singe arbitrary page.

### Aspects

In **A**spect-**O**riented **P**rogramming (AOP)[6] programs are broken down into distinct parts that overlap as little as possible. These parts are called *aspects,* and woven together to form the program. Manakin *Aspects* are the arrangement of Cocoon components (transformers, actions, matchers, etc) that implement a new set of coupled features for the system. These Aspects combine to form all the features of Manakin. Each of the system's Aspects are "chained together", so that for each page the system generates, every Aspect is given the chance to add its own content into the page. *Aspect chaining* allows new features to be overlaid onto an existing system while eliminating the need to patch or merge files, because all Aspects are kept structurally separate.

Aspects are implemented as Cocoon sub-sitemaps composed of components which may query the DSpace API, possibly changing the state of DSpace. They take a DRI document as input, incorporate their features into the document, and pass the modified DRI document along to the next Aspect; this use of DRI documents as both input and output is what makes Aspect chaining possible. In addition to modifying the in-process document for display, Aspects execute the code that determines how the page is displayed.

The order in which Aspects are executed is determined by the Aspect configuration file, `aspects.xml`. Aspects are packaged together with their source code, sitemaps, unique translations for the particular Aspect, and any other resources that may be required. This packaging enables portability of Aspects; they can be copied from one DSpace installation to another with minimal conflicts. The standard install of Manakin DSpace includes five aspects:

- Artifact Browser

  The *ArtifactBrowser* Aspect is responsible for browsing communities, collections, items, and bitstreams, viewing an individual item, and searching the repository.

- E-person

  The *E-person* Aspect is responsible for logging in, logging out, new user registration, forgotten passwords, editing profiles, and changing passwords.

- Submission

  The *Submission* Aspect is responsible for submitting new items to DSpace, the workflow process, and finally ingesting the new items into the DSpace repository.

- Administrative

  The *Administrative* Aspect is responsible for administrating DSpace, such as creating, modifying, and removing all communities, collections, e-persons, groups, registries, and authorizations.

- LocalExample

  The *LocalExample* Aspect is included as a guide to show how to extend Manakin for local customizations without modifying the standard Manakin Aspects.

To gain a fuller understand of Aspects' interaction with Manakin, consider the example of creating an Aspect to add a shopping cart to DSpace. This new shopping cart should provide several new features to the system, including the ability to add an item to the user's cart, the ability to manage the cart (adding or removing items), and the ability to purchase and grant access to the item(s).

This new Aspect and all associated files, including the source code, configuration, cocoon sitemap, and any static resources, would be contained in its own directory. When this new Aspect is added to the `aspect.xml` configuration file, any page served by Manakin will be passed through it, and depending upon the URL of the request, one of the following cases will occur: specific existing pages could be modified, content could be added to all pages, or entirely new pages could be created.

- The first case, modifying existing pages, is encountered when a standard item display page is being generated. The new Aspect would likely add a "Buy This Item" button to this page. This new button is added to the page content already generated by the standard ArtifactBrowser Aspect. In this case, the shopping cart is extending one of the standard Manakin Aspects.
- The second case, adding content to all pages, is encountered because once the user has added an item to their shopping cart and is continuing to browse the site, they will need the ability to navigate back to their cart. Since this possibility could occur anywhere within the site, the Aspect needs to add a "View Cart" link to all pages.
- The third case, creating new pages, will occur when the user wants to checkout or review their cart's contents. After the user clicks the "View Cart" link, they are taken to a new page and shown the contents of their shopping cart. This page did not exist in the system before the Aspect was added; it's entirely new content generated by the shopping cart Aspect. This case handles the actions that need to be performed on the server, such as adding or removing items from the cart, purchasing the cart, and finally enabling access to the content.

The shopping cart Aspect only deals with the features necessary to implement a shopping cart, so when the Aspect is turned off, there would be no indication of the shopping cart anywhere: no dead links, no dead pages, etc. However, when the Aspect is turned on, the links to the shopping cart, the "Buy This Item" button, and checkout pages would all be present without necessitating the merging or patching of any files.

## Processing a Request

Handling a DSpace page request consists of two major processes, content generation and style application. The content generation process involves building an XML representation of the requested page using the DRI schema; this process is handled by the Aspect chain. At the end of this process, the document will contain all the data required to build the requested page. The style process uses Themes to transform the generated DRI document into a display format (usually XHTML) for transmission to the user.

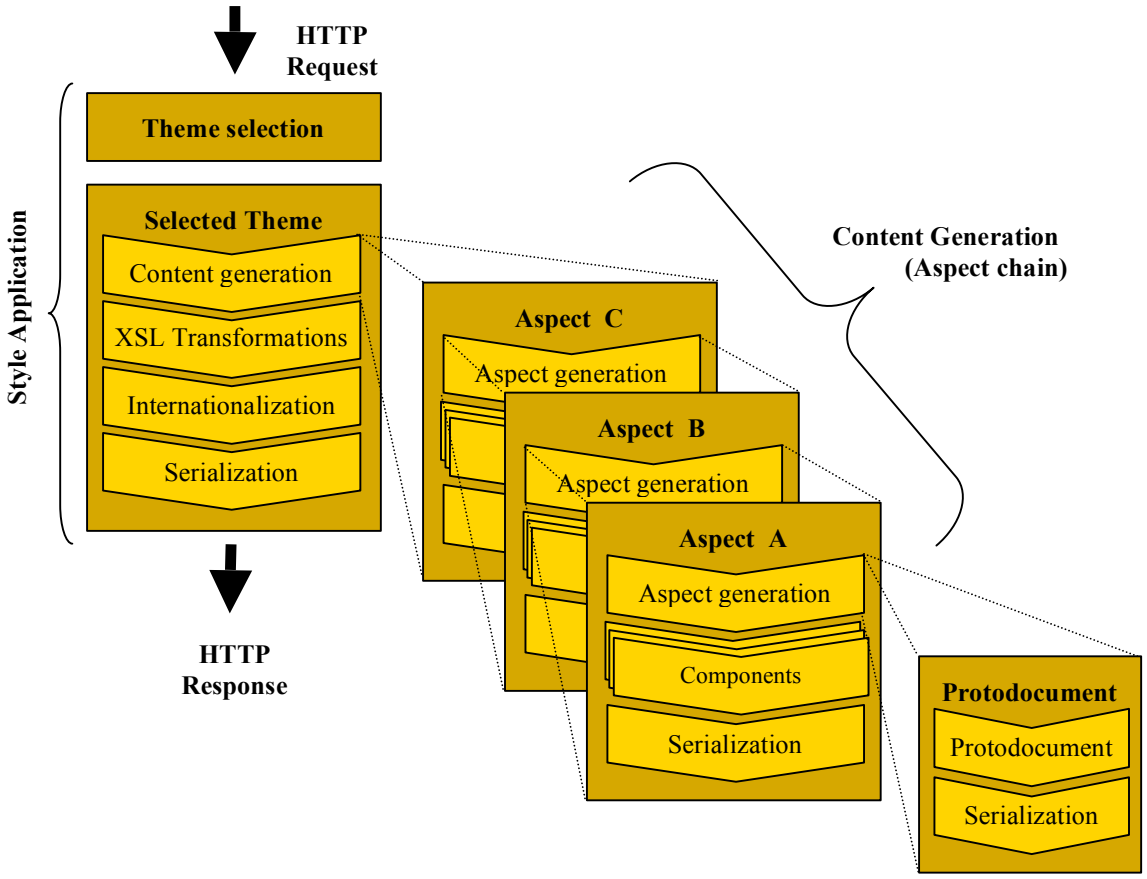

**Figure 4: Manakin Theme & Aspect flow diagram**

When Manakin receives a page request, it initiates the style application process first, which identifies and executes the appropriate Theme. The Theme's first step is to make an internal request for an unstylized DRI document, initiating the content generation process. The content generation process returns the DRI document to the Theme for transformation and serialization to the user.

When the content generation process is initiated, it begins by constructing an Aspect chain. The chain is built by linking all defined Aspects together in reverse order ending with the *protodocument*. The protodocument is the most minimal document that still conforms to the DRI Schema, consisting of nothing more than the root element and the major containers.

The linking occurs naturally, as each Aspect makes an internal request for the next Aspect in the chain. Although the Aspects are linked in reverse order, they are executed in natural order, as the chain is constructed using head recursion. This means that the final Aspect in the chain (Aspect A) performs its operations first, modifying the document, and then passing its output to the preceding Aspect, and so on until the initial Aspect is executed (Aspect C). The chain may be thought of as a stack, with each Aspect pushed on as it is linked, and when the stack is full, each Aspect is popped off and executed.

At the conclusion of the content generation process, a complete XML representation of the page exists in the form of a DRI document. This is an unstylized representation, containing only structural and metadata information. In order to be displayed to the user, the document needs to be transformed into a display format, such as XHTML. These transformations are the responsibility of the Theme.

Since the Theme initiated the internal request for a DRI document, starting the content generation process, the completed document is returned back to the Theme for further processing. Themes transform this document from the DRI schema into a suitable display format using XSLT. During the transformation the theme may insert original content into the document, such as help text or navigational links. This original content should be encoded using Cocoon's i18n schema to be translated in the next step.

After transformation, the i18n schema elements are translated into the natural language of the user by Cocoon's i18n transformer. Finally, the Theme will serialize the document for transmission to the user.

## Internationalization

Internationalization is the process of transforming a page into the language of the user. This is accomplished in Cocoon by using *keys* to lookup the translations of specific words and phrases in a *catalogue*; translated phrases are collected into catalogues which map to a specific key. When the page is generated by the Aspect chain these keys are embedded in the DRI document and encoded in Cocoon's i18n schema. The internationalization

transformer maps these keys into phrases in the user's language, and the phrases are inserted back into the DRI document, replacing the key.

While there may be many i18n catalogues in Manakin, the first and most important of these is the *core* catalogue. This catalogue contains all the translation definitions for the core DSpace components. This means that to translate a stock install of DSpace, a translator would only need to translate one catalogue. Of course, Aspects will exist that are not part of the stock DSpace install, such as a locally customized Aspect, or the preceding shopping cart example. The components that are specific to these Aspects will likely use phrases that also need to be translated. Since they are not part of a stock DSpace install, their translations need to reside outside the core catalogue. When this happens, the Aspect will use an alternative catalogue which resides inside the Aspect's package. This enhances the portability of non-core Aspects.

### Cocoon, MVC, & Flowscripts

The **M**odel **V**iew **C**ontroller (MVC)[7] design pattern is commonly used by many website frameworks. Under the Cocoon framework the model is represented by the contents of the pipeline. The view is the set of transformers that stylize the contents for the user. In Manakin the view is represented by the Theme transformer. The controller is composed of the pipeline components that generated the model and their arrangement together.

Flowscripts are a Cocoon concept representing the controller concept from the MVC design pattern. Flowscripts are a relatively new addition to the Cocoon framework and have not yet proven to be scalable, as they require a persistent javascript virtual machine. Since scalability is a primary concern of the Manakin implementation, Flowscripts will not be used; and instead more traditional sitemaps composed of actions, matchers, and selectors will be employed.

The orientation of our system around Aspects will enable future Aspects to use Flowscripts as their controller without affecting the rest of the system. If this technology proves itself to be useful & scalable, then an effort can begin to use it throughout the DSpace system incrementally.

## Sitemap Structure

As stated before, the heart of any Cocoon website are the sitemaps. Manakin's sitemaps are broken up into several files to separate the contents into smaller more manageable units. The concept being that each sitemap is simpler and easier to understand because it corresponds to a smaller part of the overall website. Figure 6 depicts the overall structure of the sitemap and sub-sitemaps in Manakin.

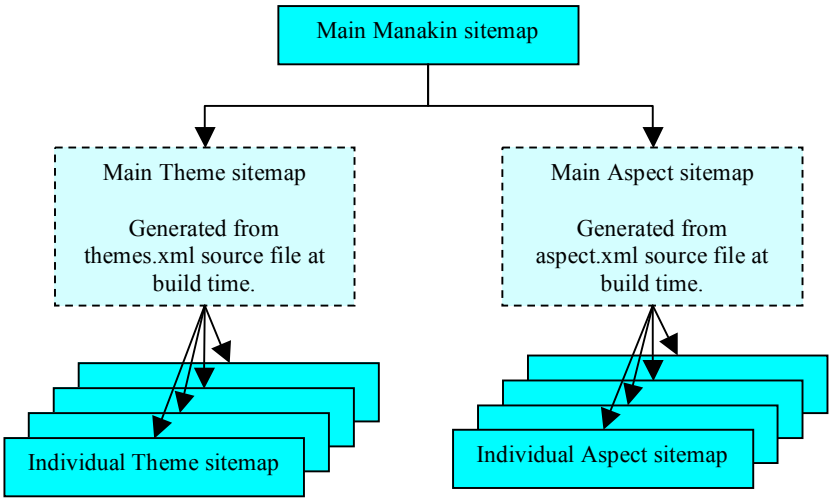


**Figure 5: Manakin sitemap structure**

The main Manakin sitemap defines components global to all functions of the system, such as the file generator and an XML serializer. The main sitemap does not do anything by itself; instead it routes requests to more specific sitemaps that can process the actual requests. There are four types of requests that are handled by the sitemap: normal DSpace page requests, **D**igital **R**epository **I**nterface (DRI) document requests, static Theme resource requests, and bitstream requests.

- **Normal DSpace page requests**

  Normal requests do not begin with any particular URL prefix as do the other types of requests. As such, all requests that do not match the other types of requests are routed to the main Theme sitemap by default. The Theme sitemap selects the appropriate Theme and stylizes content for user.

- **DRI Document requests**

  DRI document requests are created by the Aspect chain before they are sent to a Theme for stylization. To differentiate these requests from typical requests sent by the user's browser they always begin with the URL prefix `/DRI/`. These requests are generated internally by the Theme, meaning that for every page in DSpace there are two URLs; the published URL that the user sees, plus the same URL prepended with `/DRI/` that delivers the unstylized DRI document.

- **Static Theme resource requests**

  Requests for static Theme content always begins with `<context path>/themes/<theme name>/`. These requests are routed to the main Theme sitemap, where the appropriate Theme is selected and content delivered. This happens typically for thematic images, multimedia, or CSS stylesheets.

- **Bitstream requests**

  Requests for bitstreams are requests for content from the DSpace repository; this content is treated as binary data in most cases and is not styalized by the Themes. These requests always begin with the URL prefix `<context path>/bitsreams/` followed by the handle and bitsream identification.

### Main Theme sitemap

The main Theme sitemap is special because it is generated automatically at build-time. During the `ant` build process, the sitemap is generated dynamically from its XML source configuration (`themes.xml`); this enables customization of the system without requiring a highly experienced Cocoon developer. The generated sub-sitemap consists of a decision making tree, with each branch splitting off requests to individual Themes based upon their configuration.

### Main Aspect sitemap

The main Aspect sitemap, like the main Theme sitemap, is generated dynamically at build-time from its XML configuration source (`aspect.xml`). However, unlike the main Theme sitemap, the Aspect sitemap does not simply consist of a decision tree, instead it recursively chains Aspects together.

The `aspect.xml` configuration file is a series of simple one line commands to turn Aspects on or off. If an Aspect is commented out, then that Aspect is not executed and all the features associated with it will not be present in the system. In the example configuration below the Aspects are chained together with the Artifact Browser is executing first, followed by EPerson, and ending with the Administrative Aspect.

```
<aspect name="Artifact Browser" src="ArtifactBrowser/" />
<aspect name="EPerson" src="EPerson"                   />
<aspect name="Submission" src="Submission/"            />
<aspect name="Administrative" src="Administrative/"    />
```

## Wing-based Transformers

*Wing* is a framework that was developed to stream-line the process of writing components that transform DRI documents. Under Cocoon's SAX-based design, writing a transformer can be a tedious task of `startElement()` and `endElement()` events. The design goals of the Wing framework are two fold: first, hide and limit the annoyances of working with SAX by providing an API specific to the DRI schema that provides DOM-like interaction. Second, enable the XML Objects from the previous DOM-based version of the XML UI to be reused as Cocoon transformers with minimal refactoring.

The Wing framework is designed under an "additive" model, where each component may only add content to the active document, and none of the decisions that a Wing component makes are based upon content currently present in the document. A Wing-based component is like any other Cocoon component, except that it is a child class of `AbstractWingTransformer`. This abstract class handles the bookkeeping required by SAX, enabling component writers to ignore issues of SAX events and focus on the content being inserted into the document. Figure 6 depicts the class inheritance relationships of most Manakin Wing-based components.

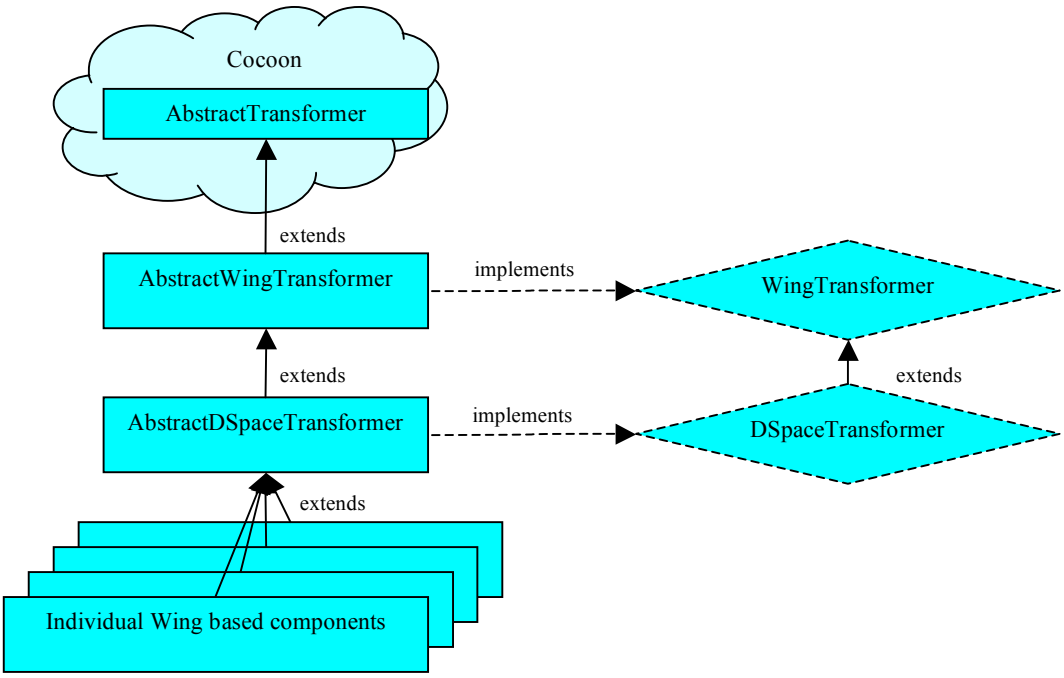


**Figure 6: Wing-based component inheritance hierarchy.**

The class hierarchy depicted above is divided into four layers. At the top layer there is Cocoon and its `AbstractTransformer`. From this Cocoon layer the definition of what it means to be a Transformer is derived. It may be useful to note that within the cloud there is a further hierarchy.

Below the Cocoon layer is the Wing layer, where SAX events are translated into DRI events. When these events are generated, a DOM-like interface is invoked to solicit the

content from the individual Wing-based components at the lowest level. The Wing layer does not have any knowledge of any DSpace concepts; instead, it is only aware of DRI concepts. This abstract layer is included to enable portability of the Wing framework.

Just below the Wing layer but above the implementing classes is the DSpace layer. Unlike the layer above, this layer is specifically here to handle DSpace concepts. This abstract component will provide such facilities as logging to the DSpace log, establishing a DSpace context, providing scoped communities and collections, and handling DSpace specific errors.

At the lowest level are the individual components inheriting directly from the `AbstractDSpaceTransformer`. These components are developed to transform the DRI document.

## Conclusion

The Manakin design incorporates the five design goals of the project through the use of the DRI schema, Themes, and Aspects. Each DSpace community and collection is able to maintain a distinct look-and-feel by using *Themes* to stylize each page for its particular purpose. Separating the business-logic from style-logic is achieved by the *DRI schema* abstraction layer between Themes & Aspects. An interface component level architecture is achieved by the use of Cocoon components and Manakin *Aspects*. Internationalization and localization is achieved by enabling each Aspect to maintain a separate *translation catalogue* or utilize the built-in core catalogue used by core DSpace components. Finally none of these features interfere with the existing JSP interface, because all interaction is accomplished through the existing DSpace API.

## Works Cited


[1] "W3C Document Object Model", http://www.w3c.org/DOM/

[2] "The Apache Cocoon project", http://cocoon.apache.org/

[3] "The Simple API for XML project", http://www.saxproject.org/

[4] "Separation of Concerns", http://wikipedia.org/wiki/Separation_of_concerns

[5] Maslov, Alexey; Green, Cody; Leggett, John; Mikeal, Adam; Phillips, Scott; Surratt, Brian."DSpace DRI Schema Reference", http://di.tamu.edu/projects/xmlui/manakin/

[6] "Aspect-Oriented Programming", http://wikipedia.org/wiki/Aspect-oriented_programming

[7] "Model View Controller", http://wikipedia.org/wiki/Model_View_Controller