

# Software Architectures for Web Content Management

## Best Practices for Enterprises and E-Government

**Andreas Rüping**

Sodenkamp 21 A, 22337 Hamburg, Germany

[andreas.rueping@rueping.info](mailto:andreas.rueping@rueping.info)

[www.rueping.info](http://www.rueping.info)

## Introduction

Content management, you say? Doesn't that mean putting together a few HTML pages? And yes, probably a little PHP programming. Or some JSPs perhaps?

As a matter of fact, content management can be much more than this. I mean, not just a little more, but *much, much* more. Imagine you have to design a large web site with hundreds of pages, rich with content and navigation elements. You need forms for user interaction, search capabilities, community support and password-protected areas for registered users. The site must support several languages. And yes, it is supposed to be fast, reliable and secure.

Obviously, you are going to use a content management system — one that is open source or one that is commercially available. Such a tool will provide some help — actually it will indeed form the basis for your system. But there are still numerous questions that remain unanswered.

How can you make sure that there is a consistent overall layout for the entire site? How can you create useful navigation elements? How can you integrate an efficient search function? How can you implement the state model that lurks behind the desired user interaction? How can you support personalisation? What about caching? How do you make sure that authors and editors enjoy a reasonably straightforward workflow process? What techniques do you employ to make sure that the system scales well to a large number of users?

So here we are, finding ourselves right in the middle of a discussion of software architecture issues. True, for a small site with only about ten mostly static pages many of these questions are quite unimportant. But when it comes to larger sites, like company sites used for customer relationship management or e-government sites used for public administration, you need to be concerned with the software architecture as much as in any other development project.

## Getting Started

This paper is an excerpt from a larger collection of patterns on web content management. Web content management can be defined as the techniques and processes around the creation, maintenance, distribution and delivery of web content that is organised as a set of documents (Hackos 2002). These documents can contain text, pictures, multimedia objects — all kinds of objects that can become part of a web page.

The focus of this paper is on organising the content and defining templates for rendering the content. It consists essentially of the first two chapters from the overall pattern collection. Therefore patterns on other topics, such as user interaction, session management, search capabilities, personalisation, caching and deployment aren't included in this paper. Their thumbnails, however, are given in the appendix.

This paper assumes a J2EE environment. Although many of the patterns also apply to other technologies, the paper in its current state builds on the J2EE technology and draws its examples from this context.

## Terminology

Many terms in the context of content management are heavily overloaded, which is a bit of an obstacle to obtaining a common understanding. I'd therefore like to define the terms I'm using up front. The following table summarises the most important terms and explains their usage in this paper.

content	information that takes the form of documents, consisting of text, pictures, multimedia objects and links between them
content management	all techniques and processes around the creation, maintenance, distribution and delivery of content
content management system (CMS)	a software product that supports content management, consisting of the CMS repository, the CMS server and the CMS client
CMS repository	the database in which the CMS stores its content

CMS server	a tool that provides access to the documents stored in a CMS repository
CMS client	a tool that allows authors and editors to create and maintain content
author	a person who creates and maintains content for a web site
editor	a person who assumes responsibility for the content of a web site and performs quality assurance
web site	the set of web pages that can be reached at one internet domain, including the functionality they may integrate
portal	similar to a web site, the set of web pages at one internet domain, but with an emphasis on providing an integration platform for content and various applications
page	an HTML page in its entirety that a user receives as a response to an http request
page element	an individual part of a page, such as the main content area, a navigation bar, a logo, a header, a footer
document	the information unit in a CMS, consisting of the attributes and meta attributes of which the document is composed
document type	definition of the attributes and meta attributes of all documents of one kind, such as all articles, all news items, all order forms
attribute	a chunk of information assigned to a document, such as string, a date, an integer, a picture, an HTML or SGML fragment
meta attribute	an attribute assigned to a document automatically by the CMS, such as the creation date, the publication date, the author
template	a blueprint for the layout of all documents of one type, whose invocation on a document causes that document to be rendered
rendering	preparing a document for delivery, for instance for delivery to a web browser by generating HTML

## Running Example

Throughout the paper I'll use a running example to explain how the patterns work. This example is an amalgam of various content-centric web sites in whose development I was involved as a software engineer, designer, architect or consultant. I have blended requirements from several real-world web sites into this example which, while it is fictional, is nonetheless pretty realistic.

The example is a web portal for software engineering technology transfer. The portal is supposed to provide materials that will help users improve their software development practise, and will also include announcements to workshops and conferences. The portal also includes a protected area in which papers are offered to registered users. The detailed requirements will become clear as the we go into the actual patterns.

I use the example to explain the problem and the forces on the one hand, and the solution on the other (in the *Example Resolved* section). Please bear in mind that the example is only just that — an example which is meant to clarify the concepts behind the patterns. I cannot present the example at the level of detail that would be necessary to design a real portal, but have to omit certain details for brevity's sake.

# I. Content Organisation

---

## 1.1 Document Type Hierarchy

**Problem** How can you separate the content from its layout, while maximising the potential for reuse of any layout definitions you will have to provide?

**Forces** A web site typically contains information of different kinds, ranging from web articles over interactive forms to files offered for download. The different types of information are usually represented by different document types. Document types are a feature that virtually all content management systems offer. The fact that documents are typed allows you to decouple the contents from the layout: individual documents represent the information that is stored in their attributes, but are assigned the layout that is defined for their type.

Documents cannot be seen in isolation, however. What matters is the set of relationships that hold between different documents. Such relationships are typically expressed by the attributes of a document that hold references, or lists of references, to other documents. For instance, an article may include a list of references to other articles. A registration form may include the reference to a document that describes the actual event for which users can register.

When rendering a document for its presentation on the web, there are different ways how you could deal with references to other documents. You could include a hyperlink that leads users to the presentation of the other document. Or you could include a presentation of the referenced document inside the presentation of the original document, a teaser perhaps. Which option you choose is something you have to specify in the templates that define the possible layouts. In order to be able to define such layouts, you must be aware of the relationships that hold between the documents and you must be able to store them somehow.

This leads us to the concept of a document type model. The document type model is one concrete aspect of something larger: the information model behind the web site you plan to develop. In her book on *Content Management for Dynamic Web Delivery*, JoAnn Hackos explains that “an information model is an organizational framework that you use to categorize your information resources” (Hackos 2002). She recommends that each content management project begin with the creation of an appropriate information model that reflects the relationships between the various real-world entities that the documents represent. And indeed, virtually every content management system requires a document type model before any templates can be set up and content can be processed.

So far, so good. There are, however, things that can turn the setting up a document type model into a tricky business. We must keep in mind that the document types provide the platform for the definition of templates that specify the layout for the various kinds of content. There will have to be one or more templates for each document type to define what documents of that type will look like when they are presented on the web. The more document types there are, the more templates you will have to create and to maintain. In addition, if several document types have certain attributes in common, it is highly probable that their templates will have certain layout aspects in common. Those layout aspects will then have to be maintained in several places, which can easily lead to redundant code all over your templates. This isn't what you want.

Moreover, document types can be used for more than assigning a common set of attributes and a common layout to all documents of a kind. Document types can also be used to constrain the way that content is added to the site or is displayed to the users. For instance, a web site's download area might be constrained to contain only downloadable documents such as PDFs, but no articles or other HTML based content. Or users might want to constrain the results of a search function to only include references to articles, but not to any other materials.

**Example** The technology transfer portal is going to present the following types of information to the users:

report	an article that e.g. reports on certain technical or methodological advances
press release	an article that also appears as an official press release
vita	the vita of a community member, with a list of publications by that person
publication	a paper written by one or more community members in PDF format
workshop announcement	announcement to a workshop including an online registration form
conference announcement	announcement to a conference with a description of the programme given separately, but without the option for online registration

So, how can these different resources be cast into a document type model?

**Solution** Define an object-oriented document type hierarchy. Employ association to describe that a document holds references to other documents, and inheritance to model abstraction.

Each document type contains a set of attributes that define the individual documents. The document type definitions must be motivated by the application domain. Make sure to name the attributes in a way that is meaningful to authors and editors. After all, it's the authors and editors who will have to create documents and assign them specific types.

Attributes fall into the following categories:

- Attributes for the various elements a document may contain, such as formatted and unformatted text, hyperlinks, pictures, blobs, etc.
- References or lists of references to other documents within the content repository.
- Possibly an attribute that specifies a view variant, in case you choose to offer several VIEW VARIANTS (2.3) for documents of that type. However, no document type must contain any concrete layout assignments such as font sizes, type faces, etc.
- Meta attributes such as the author's name and the publication date. Most content management systems define such meta-attributes automatically (and assign appropriate values whenever a document is created, updated or published).

In addition to the basic document types — the ones that correspond one-to-one to the kinds of content you want to present on the web — you can define abstract types that summarise what several concrete document types have in common.

Defining an abstract document type makes sense in the following cases:

- if several document types have attributes in common, which can be extracted into a supertype,
- if the abstract type is useful for any constraints in the document type model or in any user functions.

The document type hierarchy forms the basis for the content you plan to create and to present on the web — a solid basis if you design it with care. It is primarily a mental model, but of course, the actual document type definitions will be part of the configuration of the content management system you're using. It depends on the actual system how the document type model is stored. Many systems expect an XML file.

**Benefits** + You create a level of indirection between the content and its layout and thereby decouple the layout from the individual documents. You can now assign one or more layouts to each document type by defining a TEMPLATE PER VIEW (2.1) and refine those views by introducing several VIEW VARIANTS (2.3). This makes it easy to define and maintain a consistent layout all over the web site.

- + The concept of abstract document types allows you to inherit attributes, which makes your model elegant and easier to maintain. If it is possible to inherit rendering functionality across the type hierarchy, you can reuse code for rendering web pages and avoid redundant code, with all the downsides that it necessarily implies.
- + Most content management systems allow you to constrain the references between documents. Given an appropriate document type hierarchy, you can specify, for instance, that an article should hold only references to other articles, which makes it impossible for authors to include references to any other kinds of document into an article. The concept of an abstract document type adds a lot of expressiveness to such constraints, as it allows you to refer to a set of document types instead of just one.
- + If you choose to provide your web site with a CONFIGURABLE SEARCH FUNCTION (4.1), you can now allow users to constrain their search results to match certain document types. Again, the concept of abstract document types adds much power to such constraints.

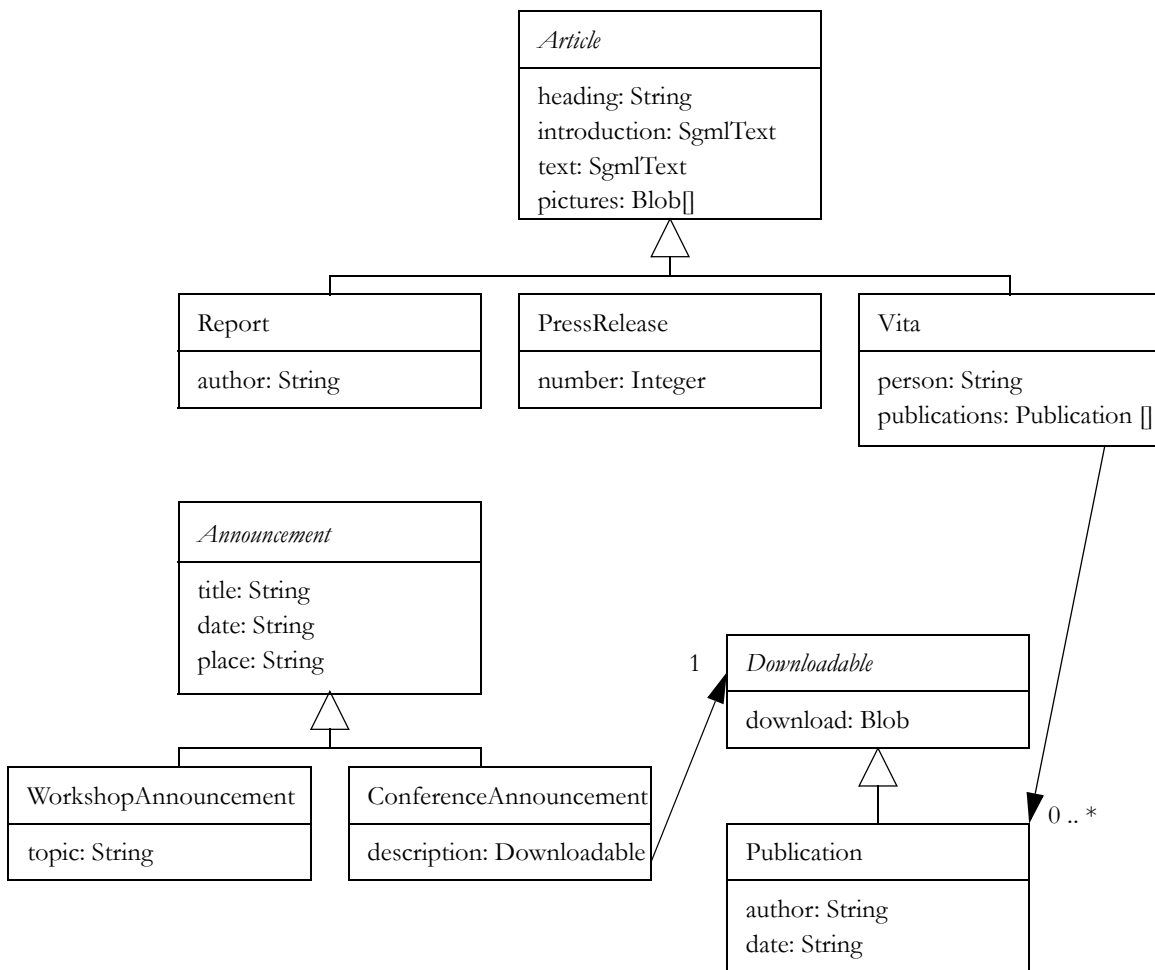
## Liabilities

- The document type hierarchy gives you a tentative set of document type definitions, but it's important to understand that these definitions might not be final. First, there are going to be additional document types which are motivated by the way users deal with the content, rather than by the content itself. For instance, if your site includes lists that represent IMPLICITLY LINKED DOCUMENTS (1.4), then these lists will require a document type of their own. Besides, your site will probably offer a CONFIGURABLE SEARCH FUNCTION (4.1), which results in an additional document type for the interactive form that is necessary for entering search queries. Second, some of the following patterns will have requirements on the tailoring of document types, and so also have an influence on the document type hierarchy. For instance, if the DECOUPLING OF CONTENT AND NAVIGATION (1.3) is used as a strategy to support several languages, document types must be tailored in such a way that no language-specific document has significant language-unspecific attributes. It's unimportant now to study all these aspects in detail. What's important is that the document type hierarchy that is implied by this pattern is a good starting point. But you must expect the hierarchy to be extended, adapted and refined by other patterns as we go on.
- A major drawback of this pattern is that not all content management systems support the concept of abstract document types. If your content management system does, fine. If not, you can come up with an implementation of your own, at least to some extent. For instance, when defining a TEMPLATE PER VIEW (2.1), you can implement rendering method by forwarding the request to the corresponding method of the supertype — home-made inheritance, if you will. All this, however, represents an increased effort on your part.
- The number of document types can become rather large, which makes content maintenance a bit complicated. You'll probably want to introduce no more document types than necessary. There are different ways to prevent this. First, if several document types are only marginally different you can

consider defining only one document type and introduce several VIEW VARIANTS (2.3) instead. Second, you should introduce abstract types only if there is a clear benefit, that is, if you can say what rendering methods can be reused or when the abstract type can serve as a constraint. In most cases, a moderate use of inheritance is a good idea, as it reduces what is known in object-oriented design as the fragile base class problem.

**Example Resolved**

Moderate use of abstraction leads to the following document type model for the technology transfer portal, represented here as a UML sketch. There are three abstract document types (*Article*, *Announcement*, *Downloadable*) that summarise the different domain-motivated types. Two document types hold references to other types: a conference announcement can include a downloadable, a programme leaflet perhaps, and a person's vita can include a list of publications.



## 1.2 Content Hierarchy

**Problem** How can the site's navigation structure be maintained in a straightforward and flexible way?

**Forces** The pages of a web site cannot be seen in isolation — they are connected through navigational links. It's fairly common to organise a web site in a more or less tree-like structure since such a hierarchy can easily be recognized by the users and makes travelling through the site straightforward. The hierarchical structure is usually presented to the users by one or more navigation elements, such as a navigation bar or a sitemap, which lead users through the entire site.

It's pretty natural that authors and editors will not want to manually maintain the navigation elements with all the links they may contain. Obviously, the navigation elements should be generated automatically. This, however, requires that the navigation hierarchy must be specified somehow. So, what ways are there for authors to specify what their site's navigation hierarchy should look like?

A first idea might be to take advantage of the fact that the content is probably stored in the CMS repository in a hierarchical way — in directories and subdirectories. Could it be possible to simply rely on the repository structure and to define the navigation hierarchy implicitly?

As a matter of fact, it isn't possible, in almost all cases. You could glue the site's navigation hierarchy to the repository structures, but that would be quite inflexible. Maybe the repository has to be organised in a way that matches the authors' and editors' responsibilities. What if all content that a certain author has the permission to maintain needs to go into one particular directory? What if the repository structures are supposed to match the customer's organisational departments? Such conflicts are impossible to avoid, so you have to decouple the navigation hierarchy from the repository structures.

Therefore you have to specify the navigation hierarchy explicitly. How can this be done?

One option is to add to each document type an attribute that holds a list of references to other documents — its children within the navigation hierarchy. But this would still be problematic for any document that might occur in more than one place within the site. This might be the case, and if it happens, it is completely unclear to which potential parent the document should be assigned. The truth is, the navigation hierarchy applies to fully-fledged pages with a unique place within the site, but not for domain-driven documents that may or may not be included in more than one page.

**Example** The technology transfer portal is going to have a mostly tree-like navigation hierarchy, with *News*, *Technology*, *Community*, *Publications* and *Conferences* as its main sections. These sections will be placed directly under the homepage, and will contain further subsections as the editors see fit. There is going to be one exception to the tree-like structure, though: pages that describe publications can

be included in multiple places, in the *Publications* Section obviously, but also in the *Community* section with people's personal homepages (which may include their papers) as well as in the *News* section.

**Solution** Establish a content hierarchy by introducing a special document type for pages. This document type's sole purpose is to embody the web site's navigation structures.

Defining a dedicated document type for pages means that you will have to extend the DOCUMENT TYPE HIERARCHY (1.1).

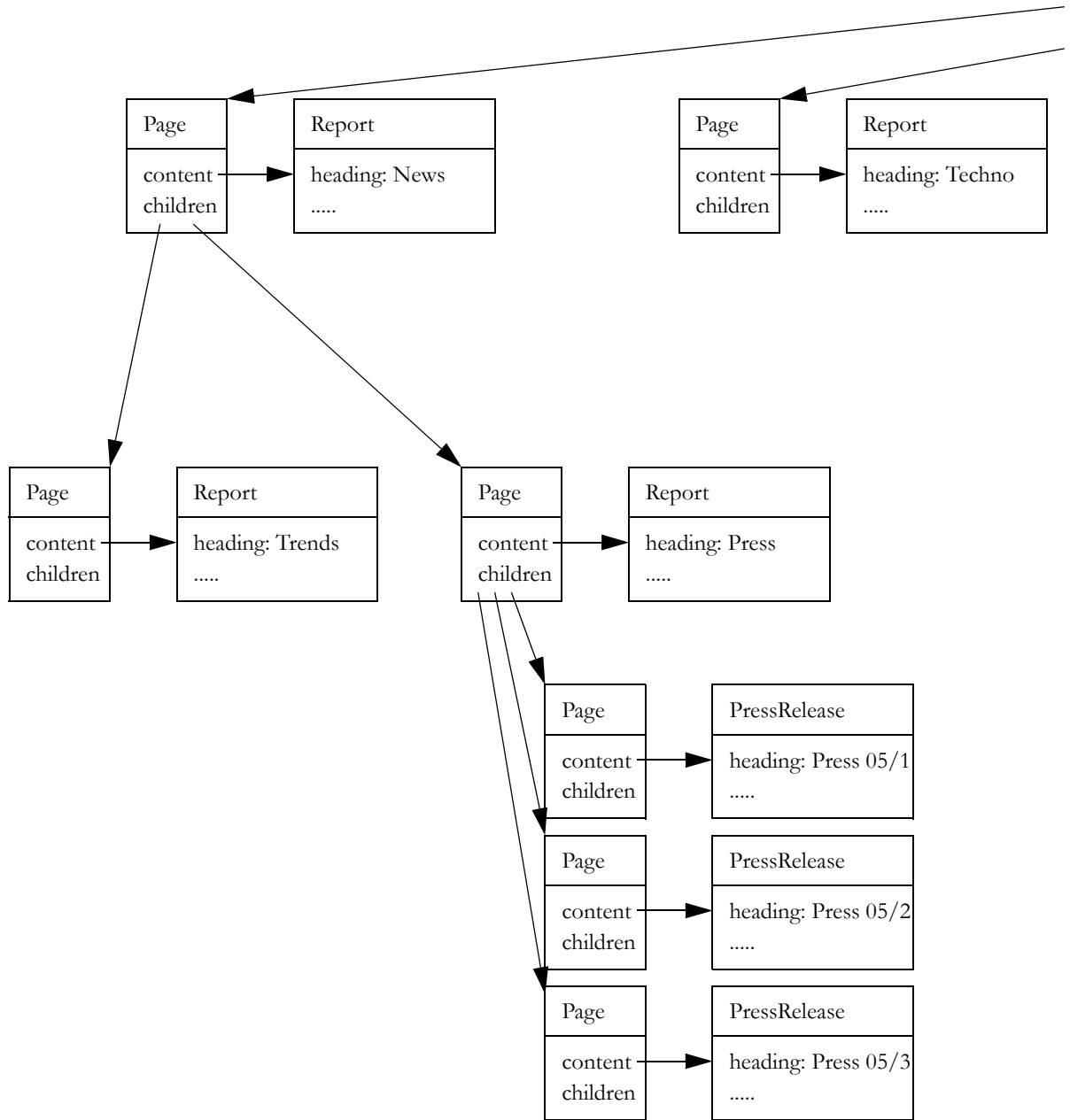
The new document type (which we will call *Page*) obtains the following two attributes:

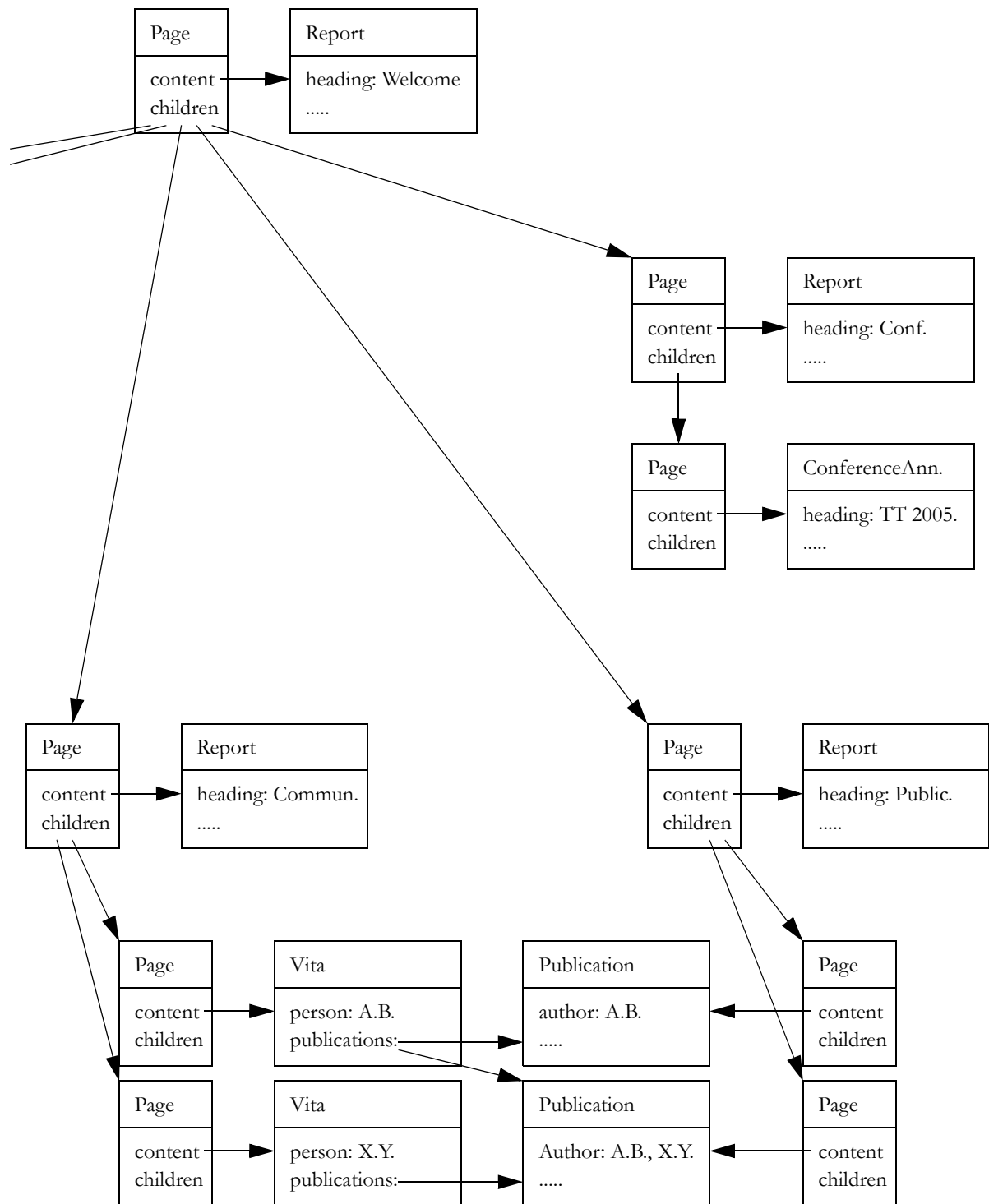
- content: a reference to the actual content document, an article for instance,
- children: a list of references to other pages which act as its children in the navigation hierarchy.

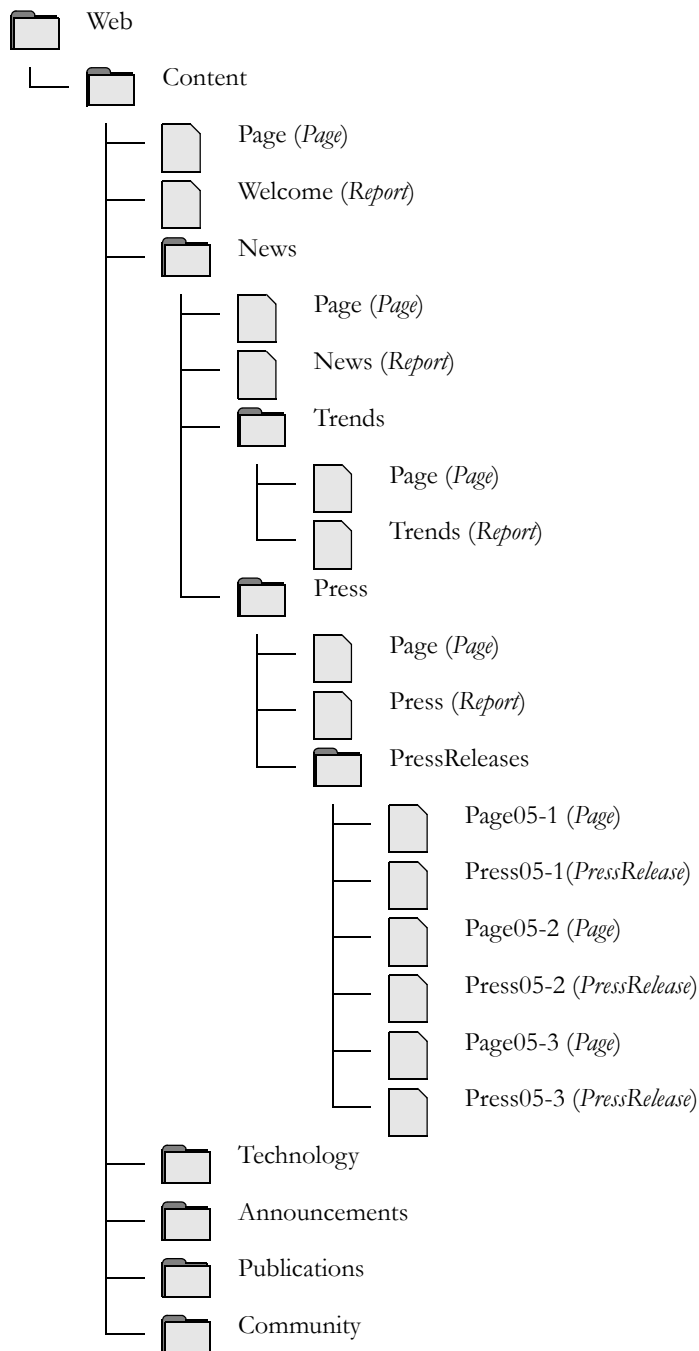
Given this definition, a page can be considered the outermost structure of what is requested by the users when they travel the site, while content documents, navigation elements or other possible page elements are all contained in a page. An http request will therefore probably use a URL that includes the page's path and name (without the need to apply any web server rewrite rules).

It's a good rule of thumb that the content repository should mirror the navigation hierarchy as far as possible. And while authors and editors are indeed free to organise the repository as they see fit, it is wise to set up the repository with directories that match the main navigational sections and to advise the authors and editors to evolve both structures similarly, unless strong organisational constraints are in their way.

- Benefits**
- + Now that a content hierarchy has been established and is explicitly specified in the *Page* documents, the foundations are laid for the automatic generation of navigation elements that you may want to include into the site. Various navigation elements are possible, ranging from a navigation bar to a sitemap, and what kinds of element you need is a matter of requirements analysis. Anyway, the AUTOMATIC GENERATION OF NAVIGATION ELEMENTS (2.2) now becomes possible, based on the navigation hierarchy described in the *Page* documents.
  - + The decoupled navigation and repository structures offer good flexibility for organising the CMS repository.
  - + If authors and editors indeed choose to organise the content repository along structures similar to the navigation hierarchy, finding resources in the CMS repository becomes pretty straightforward, given the fact that there is now a fairly easy mapping from URLs (which specify web pages) onto CMS paths (which specify resources in the repository).







- Liabilities**
- The navigation hierarchy serves as a pathway for the users to access the site, but it doesn't make search capabilities unnecessary. Most sites will require a CONFIGURABLE SEARCH FUNCTION (4.1) that takes keywords from the users and yields references to pages that match the user's request.
  - Setting a up a content hierarchy is fine, but it is only just that — one hierarchy. What are you going to do if, for instance, you have to support several languages or even several web sites, like an intranet and an extranet? In this case the content hierarchy has to unfold in the sense of a DECOUPLING OF CONTENT AND NAVIGATION (1.3).

- When a new page is added to the site, authors have to create at least two CMS documents: one for the page and one for the *real* content. This may seem like double work, but it is simply the price you have to pay for the flexibility you need. Sometimes, however, defining and linking the pages can become a tedious job, especially if much content needs to be added. An alternative can be to use lists of **IMPLICITLY LINKED DOCUMENTS** (1.4) for material that is updated frequently.

### **Example Resolved**

The large diagram on the previous two pages outlines an excerpt of the content hierarchy for the technology transfer portal, with *Page* documents referring both to their children and to their content. Keep in mind that the diagram does not show document types, but a hierarchy of document instances. You can see that the navigation hierarchy isn't exactly a tree since publications can be reached via several paths. The *Page* documents, however, are able to deal with that kind of structure.

The diagram to the left shows a bit of the directory structure in the content repository (with the *News* directory opened) which is similar, though not identical, to the content hierarchy.<sup>1</sup>

## **1.3 Decoupling of Content and Navigation**

**Problem** How can you organise the content in a way that supports several navigation structures simultaneously?

**Forces** The introduction of a specific document type for pages makes it possible to define and maintain a **CONTENT HIERARCHY** (1.2). This, however, might not be enough. There are cases in which you need to support more than one such hierarchy, for instance if you have to support several languages or if your web content is targeted at several sites, like perhaps an intranet and an extranet. Let's look at these cases in more detail.

If your site is to support several languages, you could simply introduce a language-specific attribute for each text. For instance, an article would require language-specific headings, introductions, main texts, perhaps even pictures (if pictures might contain text). This, however, is a rather inflexible solution. Imagine what would happen if you needed to support another language? You would have to change the **DOCUMENT TYPE HIERARCHY** (1.1), add new attributes, update the existing documents to match the new document types, and re-publish them. This would be a tedious job, and it would require re-publication of existing content although that content hasn't really changed. Overall, it's not a good idea.

---

1. For clarification, the document types are added to the documents in brackets and italics.

On the other hand, you can set up entirely separate hierarchies, each of which contains documents for one of the languages you need to support. It's not a good idea either, though. The different hierarchies will have things in common, especially much of their navigation structure. Don't expect authors and editors to be happy if they have to maintain the navigation structure redundantly for all languages. Rather, what is needed is a way to reuse the navigation structure across several languages while preserving the flexibility to add further languages without having to touch existing content.

There is similar, kind of inverse, situation when you have to deliver content to several sites. In this case you have to expect the navigation hierarchies to be quite different. For instance, an extranet might be a subset of intranet, with certain paths not being intended for public access. In other words, you need the flexibility to support different navigation structures but you want to reuse those parts of the content that appear in both sites.

Actually, the navigation hierarchies might not only have to be different, it might be necessary to keep them entirely separated. This lies in the different access permissions that different sites may require. For instance, an intranet typically requires a log-in, while an extranet does not. A web server or a servlet engine can take care of the log-in mechanism, but you have to specify the URLs which should be password-protected. In order to tell the URLs for intranet and extranet apart, their pages (which are addressed in the URL) must contain different paths. The likely consequence is that the pages for intranet and extranet will have to reside in different directories within the CMS repository. But even if this is the case, you still want to reuse the content that the two sites have in common.

**Example** While the technology transfer portal consists of only one site (which makes things easier), it is indeed faced with the requirement to support different languages. For the moment, English and German are sufficient. It's pretty likely though that a French version will be initiated in the near future, and in the long term more languages might follow, though nobody can say how many or which.

**Solution** **Establish several hierarchies that exist in parallel. These hierarchies are potentially separate from each other, but can share both parts of their navigation structure and parts of their content.**

What exactly these hierarchies look like depends on the requirements that are placed on the web site, or web sites, that you are building.

The following mechanisms allow you to implement the separation of content and navigation:

- You can extend the document type *Page* to feature not only a reference to a content document, but a list of references to several content documents. The template that renders the page can decide which document it uses based on current status information. For instance, a template can check the current locale (which it should be able to receive, for instance via the URL) and choose the reference to the language-specific document accordingly.

- You can extend the document type *Page* to feature conditional children links. Such links limit the parent-children relationship between pages. When the navigation structure of the site is determined, child pages will be considered only if their conditions are met. Possible conditions for a parent-child relationship include the current domain name. During the AUTOMATIC GENERATION OF NAVIGATION ELEMENTS (2.2), a template can compare the required domain with the current domain (which it can obtain from the URL) before including that child page in the navigation hierarchy. This technique allows you, for instance, to mark certain areas as intranet-only.
- You can set up different navigation hierarchies with individual homepages, and so lay the foundation to serve entirely different sites (distinguished, again, by the domain name contained in the URL by which the sites are requested). There are now separate sets of *Page* documents, but these *Page* documents can still reference the same content documents and so reuse content as far as possible.

While this is a fairly heterogeneous set of techniques, all techniques have in common that they weaken the mapping from pages to content, which so far was rather strict (a 1:1-relationship, with the exception that documents could appear in several paths). Obviously, the techniques can be combined, which gives you even more freedom to design variations of your site.

### **Benefits**

- + The less strict mapping from pages onto content gives you the freedom to design site variations with respect to languages, target audiences, and more.
- + It also paves the way towards a personalised site. Personalisation is typically based on a PROFILE CONTENT MAPPING (5.2) and it can mean, among other things, that certain areas of a site are visible to certain user only. The conditional children links can be used to specify such permissions. The AUTOMATIC GENERATION OF NAVIGATION ELEMENTS (2.2) includes checking the current user's permissions against the required permissions to determine whether or not a certain area of the site should be included in the navigation hierarchy presented to that particular user.
- + Despite the fact that you have multiple hierarchies, you can share content across the variations, and so avoid content that is stored redundantly in several documents.

### **Liabilities**

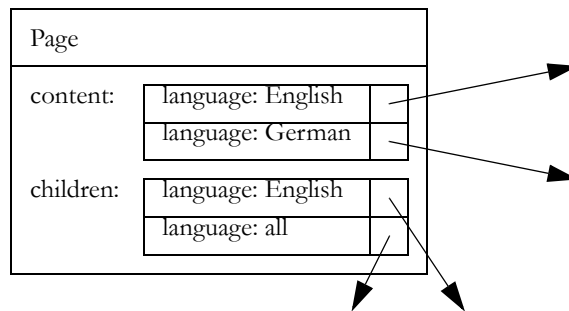
- There is an important consequence that this pattern has on the DOCUMENT TYPE HIERARCHY (1.1). If you reuse the navigation hierarchy but duplicate the content, for instance to accommodate several languages, you have to create multiple instances of all documents that are not entirely language-unspecific, even if they contain language-unspecific attributes. For example, if you duplicate all *Article* instances to obtain both English and German articles, you automatically duplicate the pictures an article may include. If there is a chance that pictures are language-unspecific, you duplicate content that doesn't really need to be duplicated. Plus, as in the case of pictures, that content can use up a lot of space — blobs can be large, after all. The consequence is that you should be careful to avoid the definition of document

types that include both language-specific and language-unspecific attributes. In such a case, you can improve your DOCUMENT TYPE HIERARCHY (1.1) by extracting language-unspecific attributes into document types of their own.

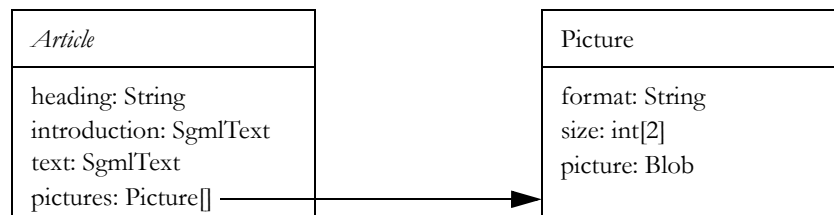
**Example Resolved**

Because the technology transfer portal needs to support several languages, we extend the document type *Page* to include named references to language-specific content. In addition, there are conditional references to the child pages in order to retain the possibility to make certain areas of the site available in specific languages only.

A typical *Page* instance now look as follows:



There is another change to the document model motivated by the liability mentioned above. The document type *Article* used to include pictures, but since there are going to be language-specific articles, it is now time to extract the pictures into a document type of their own.<sup>2</sup> This leads to the following update of the document model.




---

2. Actually, there are more reasons to introduce a separate document type *Picture*, like to be able to store the picture format and size — issues that we have ignored so far for simplicity's sake.

## 1.4 Implicitly Linked Documents

**Problem** How can content be maintained that changes frequently, without imposing on authors and editors the task to manually integrate the new content into the navigation hierarchy?

**Forces** The CONTENT HIERARCHY (1.2) defines a navigation structure for your site. To maintain that hierarchy, authors create pages, link them as necessary, and let them reference the actual content. This is generally fine. However, if new content is added frequently you might want to look for a better solution, one that spares authors the task of maintaining the *content* references within the *Page* documents.

To this end, we have to keep in mind that decoupling the navigation structure from the repository structure was motivated by the need for flexibility — only this way are you given the freedom to define a navigation hierarchy for the content that is driven by the user requirements and that isn't influenced by considerations for how to best organise the CMS repository.

But what if this flexibility isn't needed?

Let's think of material that is best organised as mere lists. There are materials that are typically presented as ordered or unordered lists — teaser lists, for instance, are common enough. Whatever such lists look like in detail, they have in common that they present items that are not structured any further. There is no navigation hierarchy among those items. But because no further structuring is applied to organise the items, you don't benefit at all from the flexibility to define an arbitrary navigation structure that includes these items.

**Example** There are two areas in the technology transfer portal that consist of materials that typically appear as mere lists. One is the set of press releases and the other is the set of publications, both of which are presented as a list in an overview page with the list entries containing links to the individual documents.

This is clearly different from other areas of the site. The technology area, for instance, typically contains pages that can be linked together in various ways. Here, authors need the freedom to define a navigation structure as they see fit. This freedom, however, is unnecessary when it comes to lists of press releases or publications.

**Solution** Define certain places within the CMS repository for content that is supposed to appear as a sorted or unsorted list. Introduce a dedicated document type for lists and implement a template that dynamically accesses the CMS directory and implicitly links all items stored there.

The document type for dynamic lists requires a number of attributes that specify the exact behaviour of a list. Those attributes fall into the following categories:

- Attributes that specify the elementary list properties. These include the specification of whether a list should be ordered or not, and if so, which ordering criteria apply.

- Attributes that specify possible constraints with regard to the kinds of document that are included in the list. For instance, you might want to restrict a list to include only articles or only downloadables. The fact that the DOCUMENT TYPE HIERARCHY (1.1) can contain abstract document types makes it easier to specify such constraints.
- An attribute that specifies how the items should be represented within the list. There might be more than one possible view for those items. A typical example is a specific teaser view, which makes a lot of sense for documents that appear in a list. If there are several views for a list item, a TEMPLATE PER VIEW (2.1) will be available, and the template that renders the list must know which template it has to call when invoking the rendering of the list items.
- An attribute that specifies where to find the list items. This attribute can be unnecessary, for instance if you can assume that list items are always stored in the same CMS directory as the list itself.

Authors can now create a *List* document and let it refer to a certain place within the repository. They can then create documents and simply store them in that specific place without linking them into a page — the documents appear automatically when users request the list in which they are contained.<sup>3</sup>

- Benefits**
- + Dynamic content represents an easy and efficient way to add content to a web site. Authors are freed from the tedious job of linking all documents manually into the CONTENT HIERARCHY (1.2). All they need to do is to maintain the list instead.
  - + Dynamic content also represents a departure from a rather static view of the content. We're on our way to templates that render content items that are being assembled dynamically, which ultimately leads us to a CONFIGURABLE SEARCH FUNCTION (4.1).

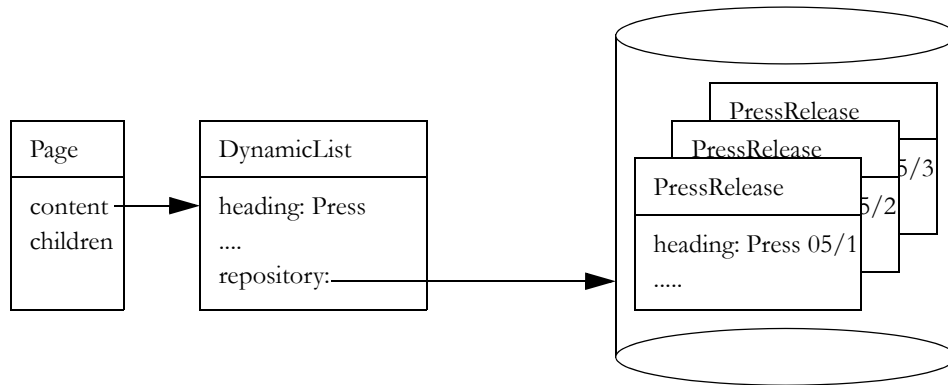
- Liabilities**
- There is no doubt that this pattern represents a performance penalty, however small it may be in practical cases. Instead of just iterating over a list of references, the template that renders the dynamic list needs to find out the place in the repository where it has to look for documents, and then look up all the documents stored in that place. Such a dynamic look-up costs some time, though the negative impact on performance can probably be ignored.
  - This pattern relies on looking up content items in a specific directory within the CMS repository. To implement the pattern, you need to access documents via their path in the CMS repository, as opposed to accessing documents via references from other documents. True enough, virtually

---

3. It should be mentioned that this pattern requires dynamic page generation. This isn't a real restriction, however, since we have generally assumed dynamic page generation throughout the entire paper.

every content management system can be expected to offer a function that allows you to access content via its path in the repository. Nonetheless, the implementation of this pattern is tool-specific in that it depends on the API that your CMS offers.

**Example Resolved** We choose two directories for dynamic content within the CMS repository, one for the press releases and the other for the publications. The following diagram describes the dynamic list that yields all press releases stored in the repository.



## II. Content Rendering

---

### 2.1 Template Per View

**Problem** How can you support the definition of different views for the same content?

**Forces** The separation of content and layout is the most important principle behind content management as far as software architecture is concerned. Documents represent content — their definition must not include any layout directives. Content management systems therefore provide for the definition of templates that allow you to define layout aspects separately from the individual documents. A template is essentially a blueprint for rendering documents of a certain type, which makes it possible to give all documents of one type a consistent layout.

However, a document will not always look the same but can have different appearances. There are various reasons for that. First of all, one and the same document can look different in different contexts. For example, when a document represents the main content of a page, it is likely to appear in full view so that all the information it holds is presented to the users. To the contrary, if the document appears as part of a list, such as an overview list or a list of search results, a teaser view is probably sufficient — one that only gives some essential information and in addition provides a link to a page that offers the full view.

Different appearances, however, can also be motivated by the users' preferences. For instance, many sites offer text-only versions of their content. If a text-only version of a document can be requested, yet another document view becomes necessary.

Finally, authors and editors may wish to retain the possibility to assign different layouts to certain documents as an editorial decision, regardless of context or user preferences. If you need to let authors and editors choose between, say, several full views, which all look a little different, the need to support different appearances increases further.

**Example** The technology transfer portal will obviously require templates for all document types, that is, for all kinds of article, for all kinds of announcement and for downloadables, too. Most of these templates present the document attributes in a suitable form. Downloadables include a link to the actual blob object.

All articles (report, press release, vita) have to support three different views. The full view includes heading, introduction, main text and pictures, the text-only view does without the pictures, and the teaser view consists of only the heading

and the introduction. Similarly, all announcements (for conferences and workshops) have to support a full view and a teaser view, with the teaser view consisting of title, date and place only. The teaser views will, for example, be used in the *News* page that offers a list of the site's most recent articles as well as a list of conferences and workshops that take place in the near future.

No print view is required for any document type. True, screen and print will use different page layouts, but this has an influence only on how the full page is composed from individual page elements, not on how the individual elements themselves look like.

Last not least, it should be mentioned that a template is needed that puts all the pieces together and renders the full page.

**Solution** For each document type, define a template for each view that is distinctly different from any other views.

In a Java setting, templates are typically JSPs or servlets. A template's job is to render a document, that is, to yield an HTML fragment that represents a specific view of that document. The fragment becomes part of the full page that is delivered to the client, the web browser that is.

In the most simplistic case, rendering consists of merely substituting document attribute values for the placeholders within some HTML fragment. In a more complex scenario, rendering also relies on functionality offered by other components. This begins with string operations, includes obtaining the current date and locale, and extends to calling the SERVER-SIDE SESSION MANAGEMENT (3.2) if any state-dependant output is required. Whatever task a template relies on, there is probably a BEAN PER TASK (6.2) that the template can consult.

Different views usually require different templates. As we have seen, different views can be caused by the following:

- differences in appearance motivated by the context in which a document is presented,
- differences in appearance motivated by the user preferences,
- differences in appearance motivated by the authors' or editors' request.

If two views are too similar to justify separate templates, you may opt to use a parametrised template instead. Most content management systems allow you too pass parameters to the templates, so you can use one template to generate different views as long as they are only slightly different.

The document type *Page* plays a special role when it comes to the definition of templates. *Page* represents the full page with all its elements and its position in the navigation hierarchy. As such, *Page* obviously needs a template that renders the full page — in other words, a template that calls the templates for all page elements and puts the results together. In the presence of CHANNEL-SPECIFIC PAGE LAYOUT (2.4), more templates for the full page become necessary.

Also, the templates that implement the AUTOMATIC GENERATION OF NAVIGATION ELEMENTS (2.2) belong to the document type *Page*. In a way, they too represent a view of a page: its position within the navigation hierarchy displayed one way or the other.

- Benefits**
- + You have a mechanism at hand that allows you to define layout aspects for all documents of a type. This is a pretty powerful concept that leads to a good separation of model and view.
  - + Since templates implement only one view (or at best views that are quite similar), they can be kept relatively simple. They can focus on the actual rendering, and leave other functionality to Java beans.

- Liabilities**
- Templates rely on the document attributes: text, images, links, etc. If they access the attributes directly, they depend directly on the content management system you use. This can be a disadvantage, as it introduces a close coupling between the custom-built code and the commercial product that is used. You can decouple those by introducing an CMS WRAPPER (6.1).
  - Templates not only rely on the document attributes, they also have to make sure that these attributes contain useful values, so that rendering the document does not result in any runtime error. Attribute validation, however, is not part of the actual rendering but rather a precondition for proper rendering. As such it should not be dealt with in any templates but in the CMS WRAPPER (6.1), too.
  - You can provide templates for different views made necessary by possible user requests, but this fact alone doesn't make sure that the right templates are used. For instance, you still have to ensure that a print view is delivered when the user hits the print button. The way to do this is to establish a CHANNEL-SPECIFIC PAGE LAYOUT (2.4).
  - Templates are rather static since they apply to document types rather than individual documents. If you need to give authors and editors the option to decide on specific layouts for individual documents, VIEW VARIANTS (2.3) can be the method of choice.
  - Introducing a template for each view can lead to an increasing number of templates and to templates that have part of their rendering functionality in common. For instance, a template for rendering a full view and one for rendering a text-only view share much of their code. As another example, templates of different document types might be similar or equal, especially with document types that have a common supertype in the DOCUMENT TYPE HIERARCHY (1.1). Either way, an inclusion or forwarding mechanism can be useful to avoid duplicate code. It can also be an option to assign a template to the supertype and let other document types inherit it, provided your CMS supports this approach.

**Example Resolved** The requirements state clearly that there have to be three different views for all articles and announcements, which results in templates named *renderFullView()*, *renderTextOnly()* and *renderTeaser()* being defined for all document types summarised by these two supertypes.<sup>4</sup>

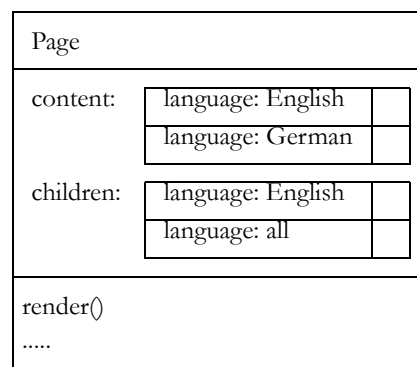
Because *renderTeaser()* is the same for all kinds of article, we assign it to *Article* rather than any subtype. This saves us the need to maintain duplicate templates redundantly. Let's assume for the moment that our CMS is able to use *Article*'s *renderTeaser()* when *renderTeaser()* for *Report*, *PressRelease* or *Vita* is requested. If this weren't the case, we could still avoid redundant templates though. We would have to define three templates named *renderTeaser()* for *Report*, *PressRelease* and *Vita*, but instead of doing the actual rendering, let them simply include *Article*'s *renderTeaser()*.

A similar argument holds for conferences and workshops. They also share their teaser view, so *renderTeaser()* is assigned to *Announcement* while *renderFullView()* and *renderTextOnly()* are assigned to *Conference* and *Workshop*.

The templates for the remaining document types look slightly different. This results from the fact that pictures and downloadables aren't used on their own, but rather in the context of another document: pictures are included in articles while downloadables are included in vitas and conference announcements. All that is needed is a set of templates that render the necessary HTML fragments for inclusion by another template.

The diagram on the previous page summarises the document types and their templates. It's the UML sketch we have used before with templates now being attached to the document types as methods.

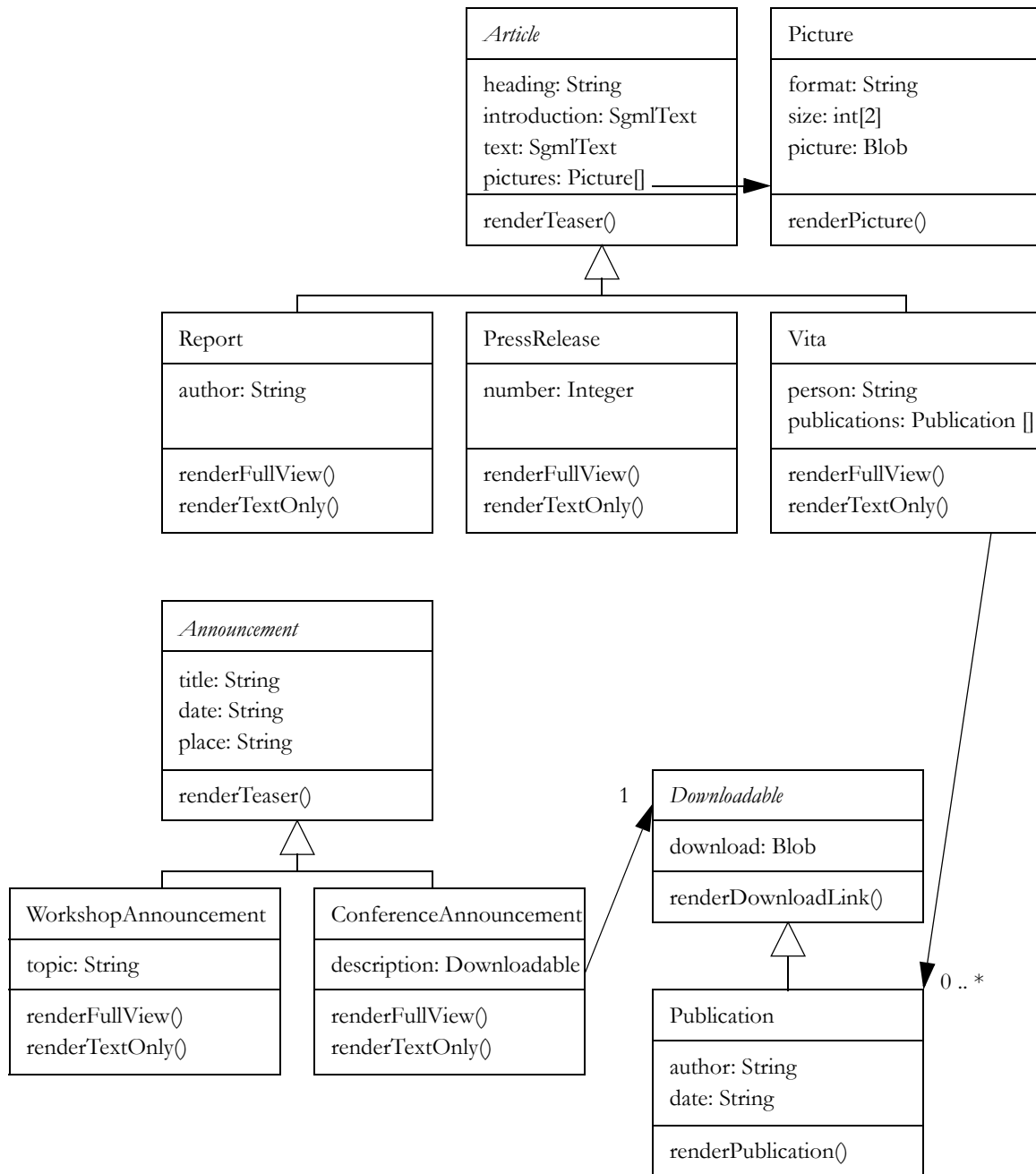
What about the document type *Page*? Obviously *Page* receives a template named *render()* which delivers the full page. Just like *Page* is the outermost document type, this template is the outermost template that has to assemble all elements a page contains.



To do so, *render()* interprets the content reference of a *Page* document, and call an appropriate template for the referenced document. Depending on the actual context, it invokes *renderFullView()*, *renderTextOnly()* or *renderTeaser()* on that document, and includes the HTML fragment it receives into the HTML for the full page.

---

4. In our example, templates are JSPs whose names in the implementation are mere strings. In our object-oriented document type model, however, we use names ending with brackets to emphasise the fact that templates here represents methods that can be invoked on document instances.



Actually, *render()* does a little more than this. Because the full page consists of more than the main content document, *render()* has to generate all other elements as well. It therefore generates a header and a footer and is also responsible for including the navigation elements. To generate the navigational elements more templates will be necessary, as we will see in the next pattern.

## 2.2 Automatic Generation of Navigation Elements

**Problem** How can you provide the users with meaningful navigation elements?

**Forces** The content is organised in a CONTENT HIERARCHY (1.2), so it is obvious that users rely on that hierarchy when travelling the site. Beginning with a home page, they travel along the paths and descend into the sections, subsections and so on. To help users navigate the site, it is absolutely common to offer navigation elements such as a hierarchical navigation, a breadcrumb navigation, a sitemap and perhaps more.

Some of the navigation elements are context-sensitive. They have two purposes: they inform users of their current place within the site and they offer links to related pages, whatever *related* means in a concrete instance. Moreover, the hierarchical hierarchy usually isn't displayed in its entirety. Typically only a part of it is offered to the users — the part that is most relevant for them given their current position within the site.

There are, however, more things to consider as far as the navigation elements are concerned. Your site might consist of more than one hierarchy. Perhaps a DECOUPLING OF CONTENT AND NAVIGATION (1.3) has been implemented to make intranet and extranet delivery easier. Navigation elements must take these add-ons to the primary CONTENT HIERARCHY (1.2) into account.

This leads us to the effect that personalisation has on any navigation elements. Personalisation is expressed through any user-specific appearance a site may have. This can include things such as the overall layout, which have nothing to do with the site navigation, but it can also include things like user-specific access permissions to certain parts of the site. Of course, navigation elements must match the user's permissions in the sense that they must only include links to pages that the current user is allowed to see.

Since all the information that is necessary to generate the navigation elements can be obtained from the CONTENT HIERARCHY (1.2), it is obvious that the navigation elements should be generated automatically with no manual maintenance involved.

However, there is also a risk involved here. Generating the navigation elements relies heavily on the references to child pages stored in the *Page* documents. If authors unintentionally create cyclical references in the CONTENT HIERARCHY (1.2), the templates that render the navigation elements could easily be trapped in an endless recursion, which will inevitably lead to a run time error when pages are requested. This, of course, must not happen. You must ensure that no run time errors occur, whatever mistakes authors and editors may make.

**Example** The technology transfer portal is going to use three different navigation elements. The hierarchical navigation offers an explorer-like tree view of the site, with the main sections included permanently, and subsections included if they lie

on the user's current path. The current path is given in the breadcrumb navigation. The site map offers a tree view of the entire site in a separate window. All navigation elements must refer to pages in the current user's language.

**Solution** Establish a component that calculates all necessary navigation-related items. Implement templates that take these items and render the navigation elements the site requires.

The component will typically be a Java class. Examples for useful methods include:

- a method that yields the path to the current page,
- a method that yields the children of the current page.

These methods take the current page as a parameter, and analyse the references stored in the *Page* documents of the site to calculate the desired results. If necessary, they also take additional input parameters, such as the current language, the current domain, a user profile — all this information can be essential in the presence of conditional links that are used for DECOUPLING OF CONTENT AND NAVIGATION (1.3) or of a PROFILE CONTENT MAPPING (5.2) for personalisation purposes.

The actual rendering of the navigation elements is done by a set of templates. Different navigation elements represent different views of the CONTENT HIERARCHY (1.2), so following the TEMPLATE PER VIEW (2.1) principle, you have to implement a template for each navigation element that is required.

What the templates look like in detail depends on your CMS and the underlying technology. JSPs are common enough. This is how the templates should work in principle:

- The templates belong to the document type *Page*. They have therefore access to the *Page* instance on which they are invoked.
- The templates are invoked on the page that the user currently visits. When rendering the navigation elements, the templates can therefore access the current page and its attributes.
- The templates use the methods offered by the navigation component. They can obtain path and hierarchy informations, and so build up trees that represent the navigation hierarchy or parts of it.
- The templates yield the navigation elements as HTML fragments which are then included into the full page.

The templates themselves need not be concerned with issues such as the current domain or the user profile. They simply take whatever items they receive from the navigation component.

It is wise, however, to implement those templates that travel the hierarchy recursively in such a way that they detect possible cycles. One option is to limit the recursion depth, so as to avoid any endless loops.

Some content management systems offer navigational elements as in-built functions. This may or may not turn out useful, depending on the requirements on the navigation in your specific project. Ideally, in-built navigation is a huge bonus, as it saves you the effort of implementing the Java component and the templates yourself. On the other hand, it can have a drawback as the CMS must make certain assumptions on the page structure and on the way how the references are stored that constitute the navigation hierarchy. You have to decide individually whether these assumptions are fine or not.

- Benefits**
- + Since the navigation elements are generated automatically, they come at no cost for manual maintenance.
  - + Since the calculation of any hierarchical relationships is done by the navigation component, the templates are mostly free of any model-related aspects and can focus on the actual rendering of the navigation elements. The solution therefore supports the separation of content and layout.

- Liabilities**
- The templates can only detect cyclical references that have entered the CONTENT HIERARCHY (1.2). Actually, cyclical references should never make it that far. You can apply WORKFLOW-BASED VALIDATION (8.1) to ensure that pages cannot be published if they contain cyclical references.
  - If the navigation elements depend on the current session state, as they do when personalisation enters the scene, caching becomes difficult. CACHED ELEMENTS (7.1) are possible only when no session information is needed for their rendering. PERSONALISED CONTENT ELEMENTS (7.2) can be an option to reduce the problem.

**Example Resolved** The technology transfer portal uses a Java class *Navigation* that implements the two methods suggested above:

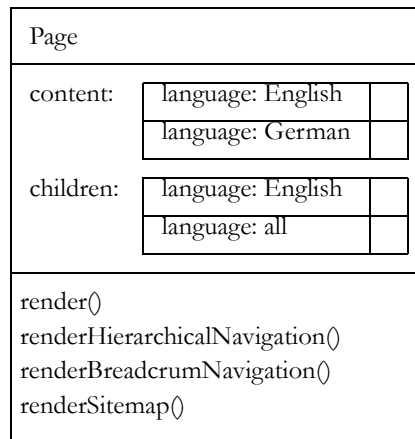
```
public class Navigation {
    .....
    public Page[] getPath (Page currentPage) {.....}
    public Page[] getChildren (Page currentPage){.....}
    .....
}
```

In addition, there are three templates for *Page* that render the three required navigation elements, all of them JSPs:

- A first JSP renders the navigation hierarchy. When invoked on the current page, it obtains the current page's path from the *Navigation* class. Beginning with the homepage and descending to the current page along the path, it obtains the children of all path elements from the *Navigation* class and so builds up an explorer-like structure that is opened where it includes the page the user currently visits.
- A second JSP renders the breadcrumb navigation. All this JSP needs to do is to obtain the current page's path from the *Navigation* class and to render the path elements.

- A third JSP renders the site map. This JSP is the only one that is unspecific of the current page. It begins at the home page, travels through the CONTENT HIERARCHY (1.2) recursively by obtaining child pages from the *Navigation* class, and so builds up a representation of the entire site.

Adding these three templates leads us to an update of the UML sketch for the document type *Page*.



## 2.3 View Variants

**Problem** How can you allow authors to fine-tune the layout of individual documents?

**Forces** Implementing a TEMPLATE PER VIEW (2.1), you can make it possible for documents to feature different appearances. The document appearance can so be adapted to specific contexts, like a teaser view when a document is part of a list, or to different user preferences, like perhaps a text-only view. Implementing different templates is fine when views must be rendered that are significantly different from each other.

Sometimes, however, views are required that are slightly different, but differ only a little. In such a case, separate templates would have many things in common and would increase the amount of redundant code, with all problems associated with it. Small layout variations simply don't justify separate templates. You are better off with some kind of parametrisation.

There is something else we need to consider. Template assignment is done for document types, not for individual documents. Which template is chosen depends for rendering a specific document may depend on the context or on user preferences, but it does not depend on the actual document. All documents of a kind are treated uniformly.

Authors, however, may have the wish to assign a special layout to individual documents. They probably won't be interested in various layouts that are completely different, as this would destroy the site's consistent look and feel, but they may wish to choose between small layout variations for each document individually. Examples include layout variations for embedding pictures into a text as well as for single-column vs. two-column presentations.

**Example** Articles in the technology transfer portal can contain a number of pictures that are part the full view of an article. There are, however, different ways how the pictures can be integrated into a text, for instance, they can be mixed into the text block or they can appear at the bottom of the text in a gallery style. It is up to the authors of an article to decide what the picture placement should be.

**Solution** **Add an attribute that specifies a view variant to those document types that have to support different appearances for individual documents.**

Authors can decide on a view variant for each document individually by assigning an appropriate value. Templates can interpret that value in essentially two different ways:

- A template can adapt its rendering the document to match the view variant, and so becomes a parametrised template.
- A template can call another template depending on the view variant. In this case, the parametrised template acts as a dispatcher and doesn't have to do any rendering itself. The view variant is used to determine dynamically what template should be invoked on a document. It extends the TEMPLATE PER VIEW (2.1) principle by taking it to dynamic dispatch.

Because view variants apply to individual documents, you must be careful not to include any layout directives that have to be consistent for the entire site. This is true for parametrised templates, but even more important if the variant is used to determine templates dynamically, since these templates can produce layouts that are pretty different. Either way, you need to balance the authors' desire to define layout variations for individual documents against the need for a consistent look and feel across the entire site.

Not all templates depend on the view variant — templates can, of course, simply ignore it. For instance, a template that renders the text-only view doesn't have to interpret a view variant that concerns the placement of any pictures.

**Benefits**

- + View variants represent a powerful mechanism that allows authors to choose from different appearances for individual documents.
- + View variants introduce a degree of parametrisation into the templates, and so can reduce their number. They also reduce the amount of redundant code that would otherwise be found in separate yet overlapping templates.

- Liabilities**
- View variants can have a negative impact on the complexity of templates. This is true whether you deal with parametrised templates or introduce a template dispatcher. You have to take care not to over-use the concept.
  - Using view variants, the assignment of a layout to a document is no longer coded into the system entirely but is determined at runtime, at least partially. On the one hand this can represent a slight performance penalty, though one which can probably be ignored. On the other hand, it involves the risk that illegal values for view variants can lead to runtime errors when a template is executed, unless the template checks that value before it goes to interpret it. This is indeed what a template should do, and in addition it is useful to apply a **WORKFLOW-BASED VALIDATION** (8.1) so that only legal values for a view variant can be entered into the content repository.

**Example Resolved** The two different ways to embed pictures in an article clearly don't justify two different templates. It's possible to stick with just one template *renderFullView()* for each kind of article. An additional attribute to the document type *Article* defines the actual view variant and is interpreted by the *renderFullView()* template of *Report*, *PressRelease* and *Vita*.

<i>Article</i>
heading: String introduction: SgmlText text: SgmlText pictures: Picture[] viewVariant: String
renderTeaser()

## 2.4 Channel-Specific Page Layout

**Problem** How do you organise the layout of the individual page?

**Forces** A typical page consists of a number of elements. First of all, there is the main document that a page represent, an article for instance. But usually there are more elements. It's perfectly common to have a header and a footer, with things such as a headline, a logo, and a copyright note. Next, there are probably some navigational elements. And perhaps there is even more, like a special teaser column that advertises the most recent bits of information, a form for entering keywords into a search function, or a form that allows users to log in.

When a page is requested, all its elements have to be rendered. Clearly this involves invoking a template for each document contained in a page. This, however, invites a first question. If there are different views to a document and there is a `TEMPLATE PER VIEW` (2.1), which template should be used?

There are more open issues. Not all page elements represent documents — logos, small forms, and small notes might all be mere page elements, with no document type representing them, so there is no 'natural' template that can be used to render them. You might simply include their rendering into the rendering of the full page or you can introduce specific templates on *Page*. Whatever you do, you have to take into account that if rendering an element depends on the session state, the element cannot be cached. Still, you need to maximize the potential for `CACHED ELEMENTS` (7.1), which will influence the tailoring of the rendering functionality.

The next thing to keep in mind is that there are layout requirements that address the entire site, rather than individual documents. Templates represent layout specifications for documents but provide no layout specification for the site in its entirety. This leaves you with the need to define things such as fonts, font sizes, colours, margins and so on, which have to apply to all page elements consistently. Moreover, you must be able to change these definitions relatively easily.

Things are, however, even more complicated. It's likely that the pages have to come in different styles for screen and print. Screen and print might look quite different — for instance, the print layout can often do without any navigational elements. Almost all web browser can interpret such styles and let a page look different, depending on whether it is to be viewed or to be printed.

Finally, you may have to support several output channels. Each output channel requires a specific rendering of the full page. In addition to the standard output on the web browser, a possible output channels is what has come to be known as barrier-free access — a version of the entire site that is text-only and that is formatted according to certain requirements which, for instance, allow pages to be fed into a speech-output device. This makes the site usable for people who are blind or partially sighted. Barrier-free access is mandatory for many e-government sites in many countries.

**Example** The pages of the technology transfer portal consist of the following elements:

- a header that includes a logo, the title bar, the breadcrumb navigation, a link to the sitemap (which appears in a separate window), a small search form (for a search function that is to be added later) and a login form (for access to a protected area, which, too, is to be added later),
- the hierarchical navigation,
- the main content,
- a footer which includes a copyright note.

The following diagram shows how the page looks like in principle:

Logo	Title Bar	Login Form
		Search Form
	Breadcrumb Navigation	Sitemap Link
Hierarchical Navigation	Main Content	
Copyright Note		

This, however, is only the screen view. The print view does not include the hierarchical navigation, the link to the sitemap as well as the search and login forms. In addition, the portal requires a text-only variant for barrier-free access, which looks in principle like the standard screen and print views, only that all elements included in the page must do without any pictures.

**Solution** Implement a template for each output channel that renders the full page. This template assembles all page elements in those styles that the channel needs to support. Define the styles to match the layout requirements placed on the full page.

We need to look at this in more detail. Implementing a template for the full page consists of the following steps.

First, you need to meet the preconditions:

- You have to define templates for all those page elements for which templates don't yet exist (headers and footers for instance). Only if the page element in question represents real content is it justified to introduce a new document type for it. In all other cases such a template can belong to *Page* and rely on CONFIGURATION ELEMENTS (1.5) if it uses any strings or pictures.
- If any of these page elements can take on different appearances, you have to define a TEMPLATE PER VIEW (2.1).
- When tailoring the page elements and defining their templates, make sure that each element either depends on the session state (like a form will) or is completely independent of the session state. Keep in mind that personalised elements always depend on the session state, therefore design specific templates for elements that might undergo a PROFILE CONTENT MAPPING (5.2).

Next you can define the templates for the full page.

- Devise a template for the full page for each output channel, for instance one standard template and one that delivers a barrier-free version of a page.
- Let this template invoke appropriate templates for all page elements in all styles that the output channel has to support. For instance, if the output channel should support both screen and print, and screen and print look different for a certain page element, both templates must be invoked (assuming there is a TEMPLATE PER VIEW (2.1)). The page elements to be included range from the main content over the navigation elements (which the AUTOMATIC GENERATION OF NAVIGATION ELEMENTS (2.2) will render) to any other elements that may be required (and that you may have defined in the first step).
- Define the styles that the output channel needs to support, for instance screen and print. For each style, define which page elements it includes, what their position are, as well as any overall layout directives such as fonts, font sizes, colours, margins, and so on.
- Let the template for the full page render an HTML page that is complete with header and body, that includes the style definitions and includes all fragments yielded by the other templates invoked.

As far as the implementation is concerned, 'styles' is likely to mean cascading style sheets (CSS). Cascades represent some kind of style inheritance that allows you to avoid any redundant style definitions. Web browsers interpret these styles and deliver the correct output for both screen and print.

Whenever a user requests a page, a template for the full page is invoked on that document. Because you can provide separate templates for different output channels, users can obtain different layouts of the full page by making different requests. In a way, this represents the TEMPLATE PER VIEW (2.1) principle applied to full pages: each output channel defines its own view of the full page, and for each such view a template is provided.

How the user request is mapped onto the invocation of a specific template on a specific page is a matter of configuration. You have to specify this mapping either in your content management system or in your web server.

- Benefits**
- + It's easy to provide the entire site with a consistent layout, as the layout specifications are made within the style definitions that apply to all pages. Moreover, modifications to those layout definitions are also fairly easy, as they are concentrated in one place.
  - + It is possible to define different layout for different output channels. Above all, barrier-free access can be offered at relatively little cost, and without adding significant complexity to the software architecture.
  - + Because there is quite some flexibility in how the full page is composed from its elements, we're on our way to a personalised site that applies a PROFILE CONTENT MAPPING (5.2) to specify who the full page is composed for an individual user.
  - + Careful tailoring of the page elements and their templates makes efficient caching strategies possible.

- Liabilities**
- The solution assumes that the users' web browsers are able to deal with CSS style sheets. There are different versions of CSS around, and while most browsers these days understand CSS at least to some degree, there is no question that the solution depends on the technology on the client's side. You therefore have to make REALISTIC CLIENT-SIDE ASSUMPTIONS (6.3) when defining the styles for your pages.
  - Different output channels, different styles, different views of the individual page elements — the flexibility offered by this pattern might become a disadvantage if you take it too far. Be sure to include views and styles for the full page and its elements only when they are indeed necessary, so that the number of templates doesn't get unnecessarily large.

**Example Resolved** Let us look at the individual page elements of the technology transfer portal first. The templates for rendering the main content, the hierarchical navigation and the breadcrumb navigation have already been defined.

The templates for rendering the title bar, the logo and the copyright note are all fairly simple. The necessary text strings and pictures are stored as CONFIGURATION ELEMENTS (1.5) somewhere within the CMS, and the templates use them to render the required HTML fragments. The templates are assigned to the document type *Page* though they can be regarded as 'static' methods — they don't rely on any of the attributes of *Page*.

The templates for rendering the sitemap link and the two forms are assigned to *Page* as well, and they don't rely on any *Page* attributes either. The template that renders the sitemap link is extremely simple, while the templates that render the forms are more complex. We'll take a more detailed look at those later when we discuss forms and the session states associated with them.

There's one thing, however, that we can note already: the two forms will depend on the session state and will not be cacheable. The fact that they receive templates on their own is a precondition for a good caching strategy. If we had chosen to define just one template for the entire header with all elements contained in it, the entire header could not be cached, as some part it would depend on the session state. This would have had a negative impact on the performance as far as rendering the title bar, logo, copyright note and sitemap link is concerned.

Let's now look at the rendering of the full page. We need two templates — *render()* for the standard view and *renderBarrierFree()* for the barrier-free version of the site. As a result, the UML sketch for the document type *Page* now looks as follows.

Page		
content:	language: English	
	language: German	
children:	language: English	
	language: all	
render() renderBarrierFree() renderTitleBar() renderLogo() renderCopyrightNote() renderHierarchicalNavigation() renderBreadcrumNavigation() renderSitemap() renderSitemapLink() renderSearchForm() rnderLoginForm()		

Both templates for rendering the full page, *render()* and *renderBarrierFree()*, have to support both a screen and a print style. These styles differ only in what elements are included in a page and what their positions are, but they do not differ in the way the individual elements look like. For instance, the main content appears at a different position on the printed page than it does on the screen, but the main content as such is same in both cases. The hierarchical navigation simply isn't included in the printed page, but there is no need to define a special view of the navigation either. Therefore, no special *renderPrint()* template is necessary for any document type.

So what does *render()* do? The following pseudo code summarises how the full page is rendered.

```
<html>
<head>
.....
<link rel="stylesheet" type="text/css" media="screen"
href="screen.css">
<link rel="stylesheet" type="text/css" media="print"
href="print.css">
.....
</head>
<body>
.....
<%
    document.renderLogo()
    document.renderTitleBar()
    document.renderBreadcrumNavigation()
    document.renderLoginForm()
    document.renderSearchForm()
    document.renderSitemapLink()
    document.renderHierarchicalNavigation()
    .....
    document.content.renderFullView()
    .....
    document.renderCopyrightNote()
%>
.....
</body>
</html>
```

The second output channel is delivered quite similarly. *renderBarrierFree()* does not include *document.renderLogo()* and uses a text-only view of the main content by invoking *document.render.renderTextOnly()*.

The two styles named *screen.css* and *print.css* define the layout of the page elements for screen and print as well as the overall layout. The following excerpt from the CSS style for print demonstrates how the style definitions are done in principle.<sup>5</sup>

```
/* print.css */
.logo{visibility: hidden;}
.titleBar{top: 0px; left: 0px; width: 640px; height: 80px;}
.breadcrum{top: 80px; left: 0px; width: 640px; height: 80px;}
.loginForm{visibility: hidden;}
.searchForm{visibility: hidden;}
.sitemapLink{visibility: hidden;}
.hierarchicalNavigation{visibility: hidden;}
.content{top: 160px; left: 0px; width: 640px; height: 320px;}
.copyright{top: 480px; left: 0px; width: 640px; height: 80px;}
.....
font-family:Verdana,Arial,Helvetica,sans-serif;
font-size: 12px;
```

---

5. This is just an excerpt — a full style sheet is much longer. The details are irrelevant here, since we are concerned with the principles rather than the details behind CSS definitions.

# Appendix

---

## Pattern Thumbnails

The following table summarises the patterns that are part of the overall collection of patterns on content management, but aren't included in this paper.

<b>1.5</b>	<b>Configuration Elements</b>
Problem	How can authors and editors configure the site's appearance beyond the definition of CMS documents?
Solution	Find a place in the CMS repository where you can deposit all kinds of configuration elements that influence the way the content is presented. The configuration elements can then be accessed by all templates.
<b>3.1</b>	<b>Form-Induced State Model</b>
Problem	How can you model the dynamics involved in your web site?
Solution	Devise a model that describes the states motivated by user interaction as well as the transitions between these states.
<b>3.2</b>	<b>Server-Side Session Management</b>
Problem	How can you implement a state model?
Solution	Establish server-side components that keep the session state and manage the state transitions.
<b>4.1</b>	<b>Configurable Search Function</b>
Problem	How can users look up content efficiently?
Solution	Implement two forms, one for entering search queries and one for navigating through the search results. Design the forms in such a way that users can configure both the search and the way how the results are displayed. Establish a component that manages search queries and their results in between user requests.
<b>4.2</b>	<b>Decoupled Keywords and Categories</b>
Problem	How can you enhance the search capabilities to become more precise?
Solution	Allow the authors and editors to add keywords to individual documents. Define categories that each consist of a set of keywords and use these to provide a topic-based search.

<b>5.1 URL-Based Authentication Realm</b>	
Problem	How can you establish a protected area to which only authenticated users obtain access?
Solution	Define one or more URL patterns that cover all protected pages and use an infrastructure component to enforce user authentication on all pages whose URLs match these patterns.
<b>5.2 Profile Content Mapping</b>	
Problem	How can define user-specific content?
Solution	Establish a component that manages a mapping from user profiles to permissions for individual content elements.
<b>6.1 CMS Wrapper</b>	
Problem	How can you strengthen the separation of model and view?
Solution	Provide a wrapper class for each document type. This wrapper class offers a method for each attribute that yields a version of that attribute that is ready to use by the document's templates with no further processing. Together, the wrapper classes form a kind of access layer on top of the CMS interface.
<b>6.2 Bean per Task</b>	
Problem	How can you make sure that the separation of model and layout isn't violated?
Solution	Define a bean for all distinct tasks on which the templates rely.
<b>6.3 Realistic Client-Side Assumptions</b>	
Problem	How can you save your system from any browser-specific rendering?
Solution	Make realistic assumptions on the technology on the client side.
<b>7.1 Cached Elements</b>	
Problem	How can you improve your web site's response time?
Solution	Plan to cache all those page elements that don't rely on the session state.
<b>7.2 Personalised Content Elements</b>	
Problem	How can you increase the amount of caching in the presence of personalisation?
Solution	If the degree of personalisation only leads to a small number of different content or layout variations, define specific templates for each variation.
<b>8.1 Workflow-Based Validation</b>	
Problem	How can you make sure that no document with illegal attributes can be published?
Solution	Implement validators that check document attributes for plausibility and integrate these validators into the workflow so that they validate the content upon publication.

<b>9.1 One Web Application</b>	
Problem	How many web application should you define for the content and its associated functionality?
Solution	Define one web application for all components that participate in generating the site.
<b>9.2 Content Versioning</b>	
Problem	How can you make sure that old versions of your site can be retrieved?
Solution	Apply versioning to all elements that are necessary to run your site.
<b>9.3 Dedicated Development and Production Environments</b>	
Problem	How can you meet the different requirements placed on the development process and on content maintenance?
Solution	Define separate environments for software development and for content production. Provide both with identical system infrastructure, but allow each to be supplied with a distinct version of the content.
<b>9.4 Dedicated Production and Delivery Servers</b>	
Problem	What can you ensure that content maintenance and the operation of your site don't interfere?
Solution	Set up separate servers for content production and maintenance on the one hand and for content delivery on the other. Implement workflows to manage content maintenance and distribution.

## Concluding Remarks

At the time of writing (February / June 2005) this paper is clearly a work in progress. I expect to add more patterns, reshape the existing ones and generally work on the material quite a bit.

Anybody who is interested in sharing their views on software architectures on top of content management is welcome to get in touch with me. I'll be happy to hear from you, whether it is with agreement, good ideas, concrete suggestions for improvement or experience reports — whatever. Feel free to contact me at “andreas.rueping@rueping.info”.

## Acknowledgements

Thanks a lot to Uwe Zdun who shepherded this paper for EuroPLoP 2005. His comments have already turned out very useful and will certainly help a lot with the further development of this document.

# References

Broemmer 2003

Darren Broemmer. *J2EE Best Practices — Java Design Patterns, Automation, and Performance*. John Wiley & Sons, 2003.

Buschmann Meunier Rohnert Sommerlad Stal 1996

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, 1996.

Dyson Longshaw 2004

Paul Dyson, Andy Longshaw. *Architecting Internet Solutions*. John Wiley & Sons, 2004.

Farley Crawford Flanagan 2002

Jim Farnley, William Crawford, David Flanagan. *Java Enterprise in a Nutshell*. O'Reilly, 2002.

Flanagan 2002

David Flanagan. *Java in a Nutshell*. O'Reilly, 2002.

Gamma Helm Johnson Vlissides 1995

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Hackos 2002

JoAnn T. Hackos. *Content Management for Dynamic Web Delivery*. John Wiley & Sons, 2002.

Rüping 2003

Andreas Rüping. *Agile Documentation — A Pattern Guide to Producing Lightweight Documents for Software Projects*. John Wiley & Sons, 2003.

Turner Bedell 2003

James Turner, Kevin Bedell. *Struts Kick Start*. Pearson Education, 2003.

Völter Schmid Wolff 2002

Markus Völter, Alexander Schmid, Eberhard Wolff. *Server Component Patterns — Component Infrastructures Illustrated with EJB*. John Wiley & Sons, 2002.