

Content Conversion and Generation on the Web: A Pattern Language

Oliver Vogel

IT Solutions Architectures
PwC Consulting
Switzerland
mail@ovogel.de

Uwe Zdun

Institute for Computer Science
University of Essen
Germany
zdun@acm.org

Abstract. Content conversion and generation is required by many interactive, web-based applications. Simplistic implementations of content converters, builders, and templates often cannot satisfy typical requirements such as high performance, end-user customizability, personalization, dynamic system updates, integration with multiple channels. We will present a pattern language resolving the main forces in this context. A `GENERIC CONTENT FORMAT` can be used to represent content from any content source. `PUBLISHER AND GATHERER` are central instances to convert to and from the `GENERIC CONTENT FORMAT`, and to handle other central content management tasks such as cache lookup and storage. Conversions are performed by `CONTENT CONVERTERS`. There are three alternative patterns to generate content on request: `CONTENT FORMAT BUILDERS`, `FRAGMENTS`, and `CONTENT FORMAT TEMPLATES`. A `CONTENT CACHE` is used to store and retrieve the content in a central repository.

1 Introduction

Interactive, web-based applications usually have to generate formatted content on request. That is, the content is not or only partly available in pre-built files. The generated content often has to be formatted in different markup languages, such as HTML, WML, and XML. Often other formats, such as graphical user interfaces or textual representations, have to be supported as well. Moreover, the content usually has to be provided to different channels with different protocols, such as HTTP, COM, CORBA, MMS, and UMTS.

Interactive, web-based applications have to be capable to represent their services using HTML pages. An HTTP server transfers HTML pages with the HTTP protocol. A web user agent, such as a browser, communicates with a web server, and the web server “understands” that certain requests have to be handled interactively. Thus, it forwards the request and all its information to another module, thread, or process. This handler may handle the request solely and generate an HTML page in response. Or it may translate and forward the HTTP request to a legacy system’s API, and then the response has to be decorated with HTML markup.

On the first glance, content creation on the web seems to be a simple effort, especially when a given legacy system with a distinct API should be reengineered to the web. In our experience, this naive view is fundamentally wrong, and it leads to severe problems when the resulting system have to be further evolved later on (see [Zdun02b] for a detailed discussion). In many systems HTML pages are simply generated by string concatenation:

```
String htmlText = new String();
String name = legacyObject.getName();
htmlText += "<BR> <B> Name: </B>";
htmlText += name;
```

Such hard-coding of HTML markup in the program will inevitably lead to problems because central requirements of modern web engineering are violated. Such central requirements for interactive, web-based applications are:

- Content, representation style, and application behavior should be changeable ad hoc.
- Web-based applications typically have to represent the business logic on the web in a coherent way, say, in a common representation style.
- In many cases, the same content has to be presented to other channels, possibly with different representation formats than HTML, as well.

- Often new functionality has to be rapidly integrated, perhaps within a few hours, and it should be possible to evolve the system incrementally.
- In many cases, the running system cannot be stopped during changes.
- Many (large-scale) web applications have very high performance and memory demands.
- Many applications require highly personalized presentations of content.
- Customization of content and behavior by non-programmers, such as content editors, domain experts, and end-users, is often required.

These requirements are met by many different web architectures. In this paper we discuss a pattern language that documents “successful” solutions in the realm of converting and generating content on the web. These patterns lead, in a mostly technology neutral form, to flexible and generic software architectures for web applications. The pattern’s consequences and variants lead to the decision which technological choices are appropriate. During the stepwise and sequential application of the patterns different consequences and forces have to be compared with the technological options and the concrete application’s requirements.

1.1 Intended Audience

This paper is intended for software and information architects faced with the development of highly dynamic, personalized content centric web applications. The patterns within this paper can be used as a roadmap for building architectures capable of serving clients with dynamic web pages in a consistent and efficient manner. This is a living document and therefore your input and participation is very much appreciated. Thus, if you harvest new patterns, variants or can supply known users feel free to contact one of the authors.

1.2 A Note on the Form

For convenience and clarity each of our patterns has the same format. In this paper we use a modification of a form called Alexandrian form that is inspired by the writings of Christopher Alexander, especially “A Pattern Language” [AIS+77]. Each of our patterns begins with a name. This is followed by an introductory paragraph, which sets the context of the pattern and its basic relations to other patterns in the pattern language. Then, there are three diamonds to mark the beginning of the problem, and, in bold type, the problem is summarized in one or two sentences. The following body of the problem explains the problem in more detail, and especially discusses the set of forces in focus of the pattern. Then, again in bold type, the solution is given in form of an instruction. In the following paragraphs, the solution is discussed in more detail, examples are given, diagrams visualize the solution, dependencies to related patterns are introduced, and consequences of applying the pattern are discussed. Another three diamonds show that the main body of the pattern is finished. And finally, there is a discussion of variants and known uses of the pattern.

2 Pattern Language Overview

The pattern language consists of the patterns summarized in Table 1 as thumbnails. GENERIC CONTENT FORMAT is the most general pattern in the language. It is used to represent content from any content source. Usually, the pattern language is applied incrementally. Typically, at first, an initial GENERIC CONTENT FORMAT is defined to start off, and it is refined as the application evolves.

PUBLISHER AND GATHERER are central instances to convert to and from the GENERIC CONTENT FORMAT, and to handle other central content management tasks. Therefore, it is quite usual to design and build PUBLISHER AND GATHERER very early in a project. Usually, PUBLISHER AND GATHERER have to be integrated with the mapping of URLs (or other document/service IDs) to service implementations. This task is often handled by the MESSAGE REDIRECTOR pattern [GNZ01]. If multiple channels have to be served, often the PUBLISHER AND GATHERER has to be integrated with a SERVICE ABSTRACTION LAYER [Vogel01] as well.

Conversions are performed by CONTENT CONVERTERS. Converters are triggered by the PUBLISHER AND GATHERER. For each supported content format, one converter has to be written for conversion to and from the GENERIC CONTENT FORMAT. These may be hand-built or use one of the patterns for content generation.

There are three alternative patterns for content generation: CONTENT FORMAT BUILDERS, FRAGMENTS, and CONTENT FORMAT TEMPLATES. Here, we want to introduce a major distinction into template-based approaches that generate HTML pages by substituting certain elements in template files, and constructive approaches that construct a web page on the fly. CONTENT FORMAT BUILDERS are implementing a constructive approach. They are highly flexible and programmable, but not the fastest alternative and not well-suited for end-user customization. FRAGMENTS and CONTENT FORMAT TEMPLATES are template-based approaches. Potentially, FRAGMENTS offer a very high performance but can only assemble pre-built parts. A compromise are CONTENT FORMAT TEMPLATES that integrate program elements in the content source. Thus they are customizable with behavior and offer a sufficient performance, but they are less flexible and less well-integrated with the programming model than CONTENT FORMAT BUILDERS. Of course, there are several systems supporting more than one of the approaches in different combinations.

A CONTENT CACHE is used to store and retrieve the content in a central repository. Content caching is a central document management task, therefore, the CONTENT CACHE is usually triggered by the PUBLISHER AND GATHERER.

<i>Pattern Name</i>	<i>Problem</i>	<i>Solution</i>
GENERIC CONTENT FORMAT	How can we use content from different sources like legacy systems, DBMS or web services without having to know its concrete representation?	Provide a GENERIC CONTENT FORMAT which is used to represent content from any content source. Convert the content by using CONTENT CONVERTERS from its concrete representation into the generic representation before you process it within your web application.
PUBLISHER AND GATHERER	How can we convert to and from a GENERIC CONTENT FORMAT (semi-) automatically, provide access to all content required on the target platforms centrally, and integrate other content management tasks such as caching?	Provide PUBLISHER AND GATHERER as central instance(s) to retrieve and store all content either in the GENERIC CONTENT FORMAT(s), or in other formats delivered to target platforms. The PUBLISHER AND GATHERER trigger conversions, lookup in the cache, and other central content management tasks.
CONTENT CONVERTER	How can we automatically convert content in one format to a different format, and/or update the content according to a set of change rules?	For each required conversion type, provide a CONTENT CONVERTER that has callback methods to be called when a conversion should take place. In general, content conversion includes input processing of the input format, data conversion/manipulation, and output processing to the target format.
CONTENT FORMAT BUILDER	How can we build up content in different content formats dynamically and reuse the same code for different content formats? How do we avoid hard-coding content format specifics in the business logic code?	Provide an abstract CONTENT FORMAT BUILDER class that determines the common denominator of the used interfaces. Build special classes that implement that interface for each supported content format, as well as special methods (e.g. as callbacks) for required specialties.
FRAGMENTS	How can web pages be designed in order to allow the generation of web pages dynamically by assuring the consistency of its content? Moreover, how do you provide these dynamic web pages in a highly efficient way?	Provide an information architecture which represents web pages from smaller building blocks called FRAGMENTS.
CONTENT FORMAT TEMPLATE	How can we build up content in target content format and allow the content editor to add highly dynamic content parts in a simple way that yields a high performance?	Provide a template written in the content format that contains special code in a template language to be substituted by a template engine.
CONTENT CACHE	How can you increase the performance of web page delivery and thereby increase efficiency of the underlying web architecture?	Provide a central CONTENT CACHE for caching already created dynamic content. Consider the life time of the content and cache them as long as it is still valid in the application's context.

Table 1. Pattern Thumbnails

3 Patterns for Converting and Generating Content on the Web

Pattern: GENERIC CONTENT FORMAT

In a web application many different forms of content have to be provided. Usually content to be presented on the web can be given in different formats, through different legacy APIs, and over different channels. The code for conversion to/from different formats should be reusable, and the number of conversion should be minimal. Often different programming language and programs should be able to access the same information base.



How can we easily use content from different sources like legacy systems, DBMS or web services without having to know its concrete representation?

Suppose you are developing a web application, which retrieves content from a RDBMS and displays it using HTML. Usually the logic necessary to generate the HTML page operates directly on the content. Therefore it has to know its concrete format (database schema). This approach works well, if the number of input formats (N) and the number of output formats (M) are very small as there is a *N*M dependency between the different formats*.

Otherwise for every new dependency between an input and an output format a new CONTENT CONVERTER has to be created. This leads to a very *complex software architecture* and to an increased *maintenance effort*. Changes in any content format influences the *channel specific presentation logic* directly and prohibits the easy *integration of new content sources*. Thus it would be advantageous if the presentation logic would not depend on any concrete content format.

Therefore:

Provide a GENERIC CONTENT FORMAT which is used to represent content from any content source. Convert the content by using CONTENT CONVERTERS from its concrete representation into the generic representation before you process it within your web application.

The generic representation is usually readable and changeable easily, so that, for instance, end users can manipulate it without programming experience. Nowadays XML is often used to represent the GENERIC CONTENT FORMAT. The GENERIC CONTENT FORMAT should enable modeling arbitrary content models including primitive types like String, Integer and Double and compound types like Address, Customer or Account. Furthermore binary data like images and multimedia should be supported. By using a GENERIC CONTENT FORMAT new content sources can be integrated without having to modify presentation logic responsible for generating output formats like HTML and WML. The dependency between the input and output formats is thereby reduced to N+M.

The COMPOSITE [GHJV94] pattern can be applied to model the information architecture required to support GENERIC CONTENT FORMAT in the software architecture of a web application system.

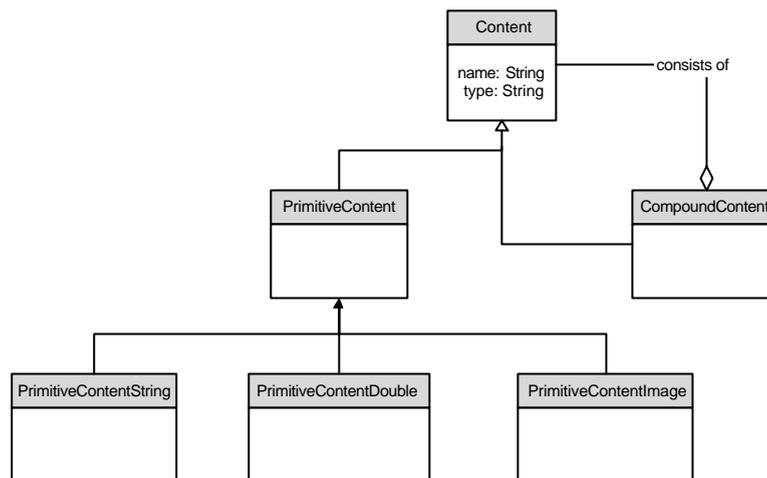


Fig. 1. Generic Content Format Representation Using the Composite Pattern

Figure 1 illustrates the generic structure of an information architecture following the GENERIC CONTENT FORMAT pattern concept. *PrimitiveContent* abstractions are used to represent primitive data types like Integer, String and Double as well as Images or arbitrary binary content. *CompoundContent* can contain other content like *PrimitiveContent* or other *CompoundContent*. An Address may i.e. exist of a *PrimitiveContent* Street of type String and a *PrimitiveContent* Number of type String. Often content is represented using a XML vocabulary expressing the abstractions necessary to model a GENERIC CONTENT FORMAT, e.g.:

```

<GenericContent>
  <PrimitiveContent name="Firstname" type="String">John</PrimitiveContent>
  <PrimitiveContent name="Lastname" type="String">Doe</PrimitiveContent>
  <PrimitiveContent name="Income" type="Income">100000</PrimitiveContent>
  <CompoundContent name="Address">
    <PrimitiveContent name="Street" type="String">Edgware Road</PrimitiveContent>
    <PrimitiveContent name="Number" type="String">2A</PrimitiveContent>
    <PrimitiveContent name="City" type="String">London</PrimitiveContent>
    <PrimitiveContent name="ZipCode" type="String">4NW</PrimitiveContent>
    <PrimitiveContent name="Country" type="String">United Kingdom</PrimitiveContent>
  </CompoundContent>
</GenericContent>
  
```

The GENERIC CONTENT FORMAT pattern offers a set of benefits: GENERIC CONTENT FORMAT serves as a “*data glue*” for integrating content from heterogeneous sources. It reduces the necessary *number of converters* to N input format converters plus M target format converters. If only well-defined formats have to be supported, *conversions can be automated* using the GENERIC CONTENT FORMAT. A GENERIC CONTENT FORMAT is a basics to implement an *efficient content conversion and generation* architecture, which is a main intent of the pattern language.

The GENERIC CONTENT FORMAT pattern can also incur the following liabilities: A GENERIC CONTENT FORMAT has to be defined centrally, thus, as applications evolve, it may be *hard to evolve the GENERIC CONTENT FORMAT non-centrally* (in a distributed and collaborative working environment). Therefore, initial formats have to be well designed for the particular domain, and extension processes have to be defined. Conversion can mean to *lose information* if the expression power of other supported formats and the GENERIC CONTENT FORMAT vary. It may be hard to guess automatically in unknown documents which parts of the GENERIC CONTENT FORMAT conform to which part of the unknown document.

◆ ◆ ◆

The pattern has different variants. We have primarily discussed a hierarchical variant that was supplemented by a COMPOSITE structure. The pattern also occurs in non-hierarchical structures. E.g. RDF is a graph-based GENERIC CONTENT FORMAT that can be linearized to hierarchical XML structures.

We have discussed typed data for the GENERIC CONTENT FORMAT. In some variants types are omitted, and a central data conversion type such as a String is used for all data. Then, each supported type must be convertible to and from Strings.

Known Uses:

- BEA WebLogic Integration uses a GENERIC CONTENT FORMAT to receive and send data from and to clients connected to its integration platform.
- ORACLE PortalToGo uses a SimpleResult data structure to represent content in a device independent manner. It generates device specific pages based on the content represented in the GENERIC CONTENT FORMAT.
- Credit Control Platform stores credit control information coming from different credit control systems in GENERIC CONTENT FORMAT and uses it to render HTML pages. The Credit Control Platform has been developed for a leading Swiss bank.
- SOAP [BEK+00] is an XML-based remote procedure call (RPC) protocol. SOAP envelopes are a typed GENERIC CONTENT FORMAT.
- RDF is a graph-based GENERIC CONTENT FORMAT for providing meta-data on the web.
- The document archiving system in [GZ01] provides a data capsule format for document archiving, containing the document plus metadata on the document. In future system versions, the capsule format should be XML.

Pattern: PUBLISHER AND GATHERER

In the context of content conversion, usually multiple input and target formats have to be supported, say, n input formats and m target formats. In the worst case, $n*m$ content converters have to be provided to perform all necessary conversions. If a GENERIC CONTENT FORMAT is given, such as XML, we can limit the necessary conversions from to $n+m$ converters. However, other issues with regard to central content management are not resolved yet: the correct content generation and conversion processes still have to be (automatically) triggered, and some formats (such as images) do not have to be converted at all, but are delivered to the target platform as is. Content changes have to be propagated to the representation in the GENERIC CONTENT FORMAT. If a cache is used, the cache has to be searched before conversion or generation is triggered, and possibly the result has to be entered into the cache.



How can we convert to and from a GENERIC CONTENT FORMAT (semi-) automatically, provide access to all content required on the target platforms centrally, and integrate other content management tasks such as caching?

Multiple different clients have to access the content in the generic format. Access to content in the generic format should be provided by a *central instance* that can be accessed from all client platforms.

Some *parts of the content do not have to be converted* to the GENERIC CONTENT FORMAT, say, because all target platforms are able to deal with a given input format. For instance, if we build web applications often image formats, such as JPEG or GIF, do not have to be converted because they can be natively accessed by the environments on all target platforms such as web browsers, mobile devices, TV applications, or widget sets.

Sometimes, *multiple GENERIC CONTENT FORMATS* have to be created. E.g. in the web context, often web content is converted to XML, unsupported image formats are converted to GIF or JPEG, and proprietary text formats are converted to PDF.

Often, it has to be decided for input and output processing whether conversions have to take place. That is, some content is delivered statically, some content is dynamically processed on the fly. Content change detection and content change propagation can also induce dynamic changes in already processed static content. Therefore, the balance of *static and dynamic content* has to be handled centrally.

Access to *different devices* on which the content is stored, such as disk drives, network devices, databases, optical devices, etc., should be accessible to clients via a unique interface so that clients can abstract from the storage devices used.

Central access points to web portals and services often have very high hit rates, therefore, they have to provide a highly *scalable* architecture.

Therefore:

Provide PUBLISHER AND GATHERER as central instance(s) to retrieve and store all content either in the GENERIC CONTENT FORMAT(S), or in other formats delivered to target platforms. The PUBLISHER AND GATHERER trigger conversions, lookup in the cache, and other central content management tasks.

PUBLISHER AND GATHERER are usually explicit entities, such as objects or processes. Sometimes, say in smaller projects, they are represented by the same entity. Usually, there are distinct access points on these entities for each specific type of content, say, PUBLISHER AND GATHERER are two objects and there is a method for each type of content. Another example is an implementation of PUBLISHER AND GATHERER as two daemons that fork handlers for each individual request. The content may be stored in a cache and/or on different devices, say, on the disk, in the memory, in a database, on optical devices, or on a network device. A CONTENT CACHE is used to abstract from these storage device specifics.

For each specific content type supported, the PUBLISHER AND GATHERER can access CONTENT CONVERTERS to and from the content's original format. The CONTENT CONVERTERS may have to operate on the fly. Once the content is converted to the GENERIC CONTENT FORMAT, and it is stored in the PUBLISHER AND GATHERER'S CONTENT CACHE. Out of various reasons, it may have to be converted again. For instance, content changes may be induced by content change detection and content change propagation. Often different variations of the content in the GENERIC CONTENT FORMAT may be required, say, because different styles have to be supported, or the content has to be transformed to correspond to different DTDs and schemas.

Let us consider a PUBLISHER AND GATHERER on two separate Java classes with methods for each type of content. In a simple publisher class methods for retrieving each individual content type are provided. A document in the GENERIC CONTENT FORMAT (here: XML) can directly be delivered with `getXml`, if it is found in the cache. Each document has a unique document ID, for instance denoted by an URL. We would have to trigger building a page from FRAGMENTS here as well, if this functionality is supported. Other formats, such as HTML, are either already converted and stored in the cache, or they have to be converted from XML. If a conversion took place, we can enter the generated HTML document in the cache.

```
class Publisher {
    CacheHandler xmlCache;
    CacheHandler htmlCache;
    ContentConverter htmlConverter;
    ...
    public XmlDocument getXml (DocumentID docID) {
        return xmlCache.get(docID);
    }
    public HtmlDocument getHtml (DocumentID docID) {
        HtmlDocument htmlDoc = htmlCache.get(docID)
        if (htmlDoc == null) {
            XmlDocument xmlDoc = getXml(docID);
            htmlDoc = htmlConverter.convertFromXml(xmlDoc);
            if (htmlDoc)
                htmlCache.enter(docID, htmlDoc);
        }
        return htmlDoc;
    }
    ...
}
```

Similarly, a gatherer can directly store XML input into the document cache (or on any other storage device), and entries for the document in depending caches, such as the HTML cache, are invalidated. If HTML input is received, the XML and HTML cache entries are invalidated, and the new document is converted to XML.

```
class Gatherer {
    CacheHandler xmlCache;
    CacheHandler htmlCache;
    ContentConverter htmlConverter;
    ...
    public void storeXml (DocumentID docID, XmlDocument xmlDoc) {
        xmlCache.store(xmlDoc);
        xmlCache.propagateChangeToDependingCaches(xmlDoc);
    }
    public void storeHtmlAsXml (HtmlDocument htmlDoc) {
        if (xmlCache.get(docID)) {
            xmlCache.invalidate(docID)
        }
        if (htmlCache.get(docID)) {
            htmlCache.invalidate(docID)
        }
        xmlCache.store(docID, htmlConverter.convertToXml(htmlDoc));
    }
    ...
}
```

}

Typically, the PUBLISHER AND GATHERER have to be integrated with other channels than the web. Depending on the URL different channels have to be served. Usually the publisher is triggered by a MESSAGE REDIRECTOR [GNZ01] used for indirecting URL calls to implementations. Each of these implementations is a service that should be published to the web (and other channels). The URL usually denotes which document or service is requested, which format is required, and which protocol is used. One or more publishers can be integrated as services into this architecture (see Figure 2), or the MESSAGE REDIRECTOR can be part of the publisher, if the publisher is the only service supported.

This service abstraction architecture is a SERVICE ABSTRACTION LAYER [Vogel01]. It is quite common for PUBLISHERS AND GATHERERS to be combined with a SERVICE ABSTRACTION LAYER if multiple services are offered to a number of channels.

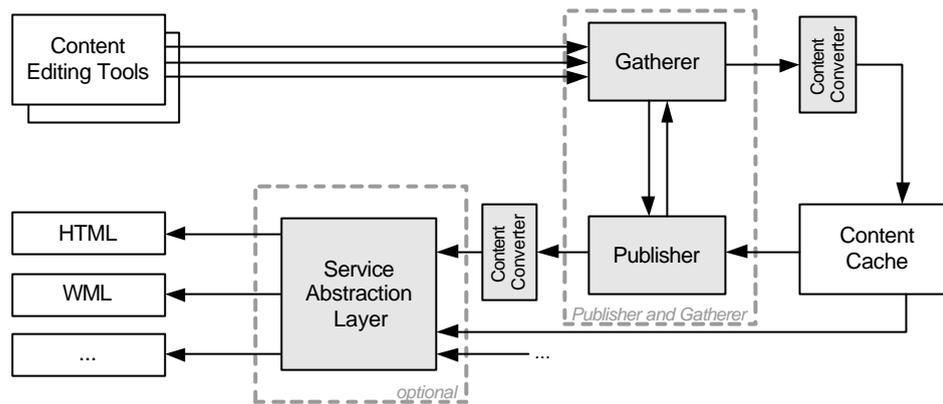


Fig. 2. Content Converters, Publisher and Gatherer, Content Cache, and Service Abstraction Layer

The PUBLISHER AND GATHERER pattern offers a set of benefits: PUBLISHER AND GATHERER are *central* instances that enable service access from *different platforms* and with *different protocols*. Correct *content conversion and generation* is triggered automatically, and caching is handled. PUBLISHER AND GATHERER can be easily integrated with sophisticated service abstraction architectures.

The PUBLISHER AND GATHERER pattern can also incur the following liabilities: Using a central instance means that we have to care about *scalability and performance* issues. To enable automatic conversion means that *all converters have to be written and maintained*, whereas hand-built architectures can only rely on the relevant converters.

◆ ◆ ◆

PUBLISHERS AND GATHERERS can be implemented in different variants. First, we can decide whether PUBLISHER AND GATHERER are implemented as two separate entities or as one entity of the programming language. In many more advanced server architectures PUBLISHER AND GATHERER are separated. Often they can be forked or redirect to other servers to provide a higher scalability of the architecture. Often there is a central instance to receive requests, and multiple workers to handle individual requests. This architecture is actually quite typical for PUBLISHERS AND GATHERERS in systems with high hit rates.

In SERVICE ABSTRACTION LAYERS [Vogel01] the publisher can either be used as a service or as a MESSAGE REDIRECTOR [GNZ01] for resolving URLs.

Known Uses:

- Simple servers that allow for putting and retrieving data (and programs) are simplistic implementations of the pattern variant with one entity: example are FTP servers and HTTP PUT/POST-enabled HTTP servers.
- xoComm [NZ00] has a worker object for each request that resolves different request methods (like GET, PUT, POST, etc.) using different methods. In xoComm there is also a central server and multiple workers to handle individual requests.
- Actiweb [NZ01] is a web object and mobile code system based on xoComm. It uses the “events” generated by the corresponding worker of the web server, and translates the URLs in an invoker component. Depending

on the URL, either normal web pages are delivered, an agent immigration or RPC invocation is handled, or a web object is triggered.

- The document archiving system in [GZ01] provides central gatherer entities for archiving of different content formats, and a document retrieval handler. All handlers are daemons that are provided for initial access only. Upon a request, a handler is forked from the central instance and handles the request.
- In the document management system DocMe a central `gatherd` and a `publishd` are provided. Internally, all gathered information is converted. Here, different constructive converters are provided, e.g. from MS Word format and similar formats used by end users as content editors. The system approximates how the documents should look like in different formats, such as HTML, TV broadcasted data, etc. Using the central PUBLISHER AND GATHERER the system caches the information, handles multiple document versions in the cache, change detection and propagation, user and rights management, and document classification issues.
- AOL Digital City [Davidson00] has an architecture with a central Pub server and multiple front end servers. A switch server multiplexes a client onto one of the front end servers.

Pattern: CONTENT CONVERTER

The context of representing information on “new media” target platforms, such as the web, mobile phones, mobile devices, TV settop-boxes, etc., implies that information has to be presented in different target formats. Usual target formats for the web include XML, WML, HTML. Sometimes formats, such as PDF, have to be supported as well. Often pictures in formats, such as GIF, JPEG, PNG, have to be generated.

For implementing a PUBLISHER AND GATHERER that triggers automatic conversions, converters from all input formats to the generic format, from the generic format to all target formats, and possibly converters from representations in the generic format to other representations in the generic format have to be provided.



How can we automatically convert content in one format to a different format, and/or update the content according to a set of change rules?

Content in different formats has to be generated for an interactive web application. In many cases, *static and dynamic content generation* have to be combined. That is, sometimes a representation has to be generated on-the-fly, sometimes statically in advance. Important consideration in this context are *performance* and *scalability* issues: for high-performance web applications (typically with high hit rates) generating all content on-the-fly is usually costly in terms of memory and performance, and this imposes severe requirements on the scalability of the application. Therefore, as far as possible conversion should take place in advance. Often static and dynamic content fragments have to be combined (see also the CONTENT FORMAT TEMPLATE and FRAGMENTS pattern).

For document to be updated, *change propagation* results in a simple set of update rules. These can either mean to delete or replace the document/fragment, or the given content has to be manipulated with one or more change rules. A conversion from a given format to the same format is required in which the change rules are triggered automatically.

In the context of *migrating legacy applications* to the web (or other new media platforms), usually the original format has to be supported as well. Thus, we cannot change the legacy application to directly support web-enabled output as its primary output format. Either the legacy format or the web format have to be converted.

Converting one content format to another often means to reduce the expressibility of the application to the *common denominator* of all target (and input) formats involved. Otherwise we have to live with *lossy conversions*.

Usually, conversions should take place either on request or upon certain events. Therefore, in such cases, content conversions have to be triggered by an *event-based API*.

Therefore:

For each required conversion type, provide a CONTENT CONVERTER that has callback methods to be called when a conversion should take place. In general, content conversion includes input processing of the input format, data conversion/manipulation, and output processing to the target format.

A CONTENT CONVERTER is constructed from three elements that are ordered in a CHAIN OF RESPONSIBILITY, each of them is optional:

1. *Input processing* creates a representation in memory from a given *input format*. As a result an intermediate representation is created. Usually, this is a representation in memory. In exceptional cases, such as operating on very large data sets (that do not fit into memory), we may use different intermediate representations. If the conversion is very simple, we can also directly operate on the input format.
2. *Data conversion and manipulation routines* on the intermediate representation (i.e. most often in memory) apply a set of change rules. The result is manipulated data in the intermediate format.
3. *Output processing* is used to create/convert the intermediate format to the target format.

The CHAIN OF RESPONSIBILITY [GHJV94] and the produced data formats are depicted in Figure 3. All parts of the CHAIN OF RESPONSIBILITY are optional, however, most often all parts are present. E.g. if steps 2 and 3 are performed on the input format, input processing is not required. If there is only a 1:1 conversion from one format to another one without any manipulations (e.g. to adapt the differences of the two formats) then step 2 is obsolete. If the intermediate format is equal to the target format then step 3 is not required.

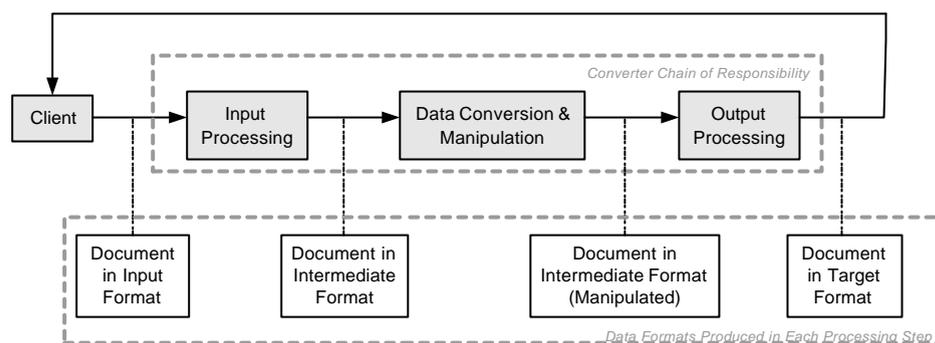


Fig. 3. CONTENT CONVERTERS: CHAIN OF RESPONSIBILITY and Produced Data Formats

In the example case of XML, we have to distinguish XML input processing and output processing. XML input processing means to parse and validate the XML text. After parsing, the relevant information has to be recognized and/or searched in the XML document. Once the information is located it can be extracted and connected to the business logic. After performing conversions using the business logic, or with the internal (e.g. DOM tree based) representation, or by applying XSLT style sheets, we can either recreate XML text or other target formats during XML output processing. Converters may be programmed by hand or use a processor. In the XML context, in general, there are three different models to handle XML processing:

- *Event-Based Processing*: SAX [Megginson99] is a simple API for event-based parsing of XML text. Expat is an XML parser that provides another event-based model. In general event-based parsers produce a flow of events from a given XML text (like start, end, data of a XML node). Usually, the APIs are relatively simple, and the memory usage is low. The basic idea is to catch the relevant events as they are processed. Therefore event-based processing is especially well-suited when the target information has to be accessed only once.
- *Tree-Based Processing*: DOM [W3C00] and xRDF [NZ02] are models that create a tree-based representation for XML (and RDF respectively) in memory. Therefore, the document can easily be searched and queried using a tree traversal API for random access in memory. DOM parsing usually suffers from high memory usage (about 2-3 and more times of the document size, depending on the DOM implementation).
- *Rule-Based Processing*: In XSLT [Clark99] rules are given that are to be applied when specified patterns are found in the source document (which is XML text). These patterns are specified using the Xpath language. Xpath is used to locate and extract information from the source document. While event- and tree-based processing requires the developer to write a program for information extraction, XSLT mostly requires writing style sheets which are themselves XML documents. Thus non-programmers are (to a certain degree) capable of writing these rules. However, the non-imperative model of XSLT is often not well-accepted by developers used to imperative programming models.

We will now discuss code examples for input processing in the three models with XML and Java. For input processing in the event-based model, we have to firstly instantiate a parser. Then in a next step we have to parse a file:

```

SAXParserFactory pf = SAXParserFactory.newInstance();
pf.setValidating(true);
parser = factory.newSAXParser();
...
parser.parse(file, ...);
...

```

The intermediate representation is then created in memory using event callbacks for each event in the XML text occurring during parsing, like start, end, PCDATA of an XML node:

```

public void startElement(String name, AttributeList attrs) {
    if (name.equals("A")) {
        convertAStartTagToIntermediateFormat(attrs);
    } else if (name.equals("B")) {
        convertBStartTagToIntermediateFormat(attrs);
    } else {
        ...
    }
    return;
}
public void endElement(...) {...}

```

For input processing in the tree-based model we have to instantiate a document tree builder object first. Then we have to parse the file as well:

```

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(false);
DocumentBuilder builder = factory.newDocumentBuilder();
...
Document document = builder.parse(file);

```

A tree structure is generated in memory. DOM provides a low-level API to traverse this tree as an intermediate format in memory, e.g.:

```

NodeList nodes_i = document.getDocumentElement().getChildNodes();
for (int i = 0; i < nodes_i.getLength(); i++) {
    Node node_i = nodes_i.item(i);
    if (node_i.getNodeType() == Node.ELEMENT_NODE &&
        ((Element) node_i).getTagName().equals("A")) {
        handleElementA();
    }
    ...
}

```

In the rule-based model, we have to instantiate the rule engine. Here a style sheet file is read in which we find the rules for transformations similar to regular expressions. These are used for transformation of the input format to the target format. That means, conversion and transformation are combined in one step.

```

TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer = factory.newTransformer(
    new SAXSource(new InputSource(styleSheetFile)));
transformer.transform(new SAXSource(new InputSource(file)), new StreamResult(out));

```

Often the content is accessed on demand, say, when an HTTP request is coming in. The conversion can also be triggered and configured in an event-based manner. Typical events are changes of content. But often conversions are triggered upon other events, say, content can be pre-processed when the system is idle or has a low work-load.

Often the converter is able to operate back and forth. The converter unifies all different conversions to and from the target format. Therefore, usually the converter has two TEMPLATE METHODS on an abstract converter class that call the three CHAIN OF RESPONSIBILITY methods for input processing, conversions, and output processing. One TEMPLATE METHOD converts to the target format, and one to the GENERIC CONTENT FORMAT such as XML. Special converter classes implement the hook methods for the target format that they represent (such as HTML). Figure 4 illustrates this design.

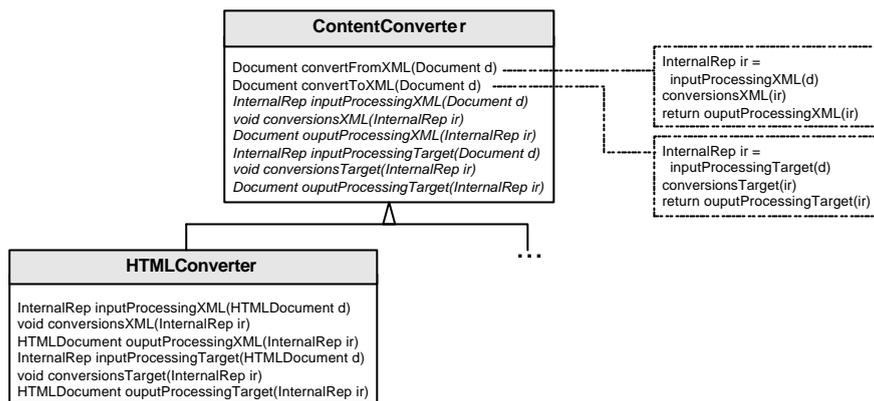


Fig. 4. Generic XML and Special HTML CONTENT CONVERTER Classes

CONTENT CONVERTERS can be used to create static content in advance as far as possible from dynamic inputs, so that the content that has to be created on the fly upon a request is minimal. Often static and dynamic content FRAGMENTS have to be combined to create one page. CONTENT FORMAT TEMPLATES and FRAGMENTS can be used for specifying in a static page where dynamic parts have to be inserted. CONTENT FORMAT BUILDER can have a special method to load static content e.g. from disk during content generation. Thus, all three content generation patterns are capable to use content pre-generated with CONTENT CONVERTERS.

In turn, CONTENT FORMAT BUILDER can be used to build up content dynamically in a specific format using a generic interface. Thus, of course, it can be used to build up the target format processed by the CONTENT CONVERTER.

The CONTENT CONVERTER pattern offers a set of benefits: It *unifies different APIs* for data transformation and manipulation to one abstract converter interface. Thus, in a content management environment different converters *can be applied in an automated fashion*. Automatic data conversion is required for automatically updating dynamic data in CONTENT CACHES and for dynamically applying conversion in PUBLISHER AND GATHERER. Moreover, the pattern allows for *combining different content conversion approaches* such as the event-based, tree-based, and rule-based approach. Since the approaches have substantially different characteristics, in many settings, it is not the best solution to support only content conversion but no dynamic content generation. Content conversion is an *efficient* alternative to (re-)constructing the content page using fragments, templates, or builders when a new or changed input arrives.

The CONTENT CONVERTER pattern can also incur the following liabilities: content conversion offers only a *limited expressibility* compared to fragments, templates, or builders. Therefore, higher-level manipulations of content should be implemented using these patterns. However, they can be triggered by a CONTENT CONVERTER. In many problem settings there are certain exceptional conversions that should be handled differently. Here, the CONTENT CONVERTER offers only *limited diversity of conversions* because it does not make much sense to produce an own converter for each exception. Better solutions are to provide an INTERCEPTOR [GZ01] or other callback mechanisms on the converter object for these cases.



There are different CONTENT CONVERTER variants. Since all three parts of a CONTENT CONVERTER are optional all parts can be omitted. The internal creation of content can be hand-built, or it can use CONTENT FORMAT BUILDER, TEMPLATES, or FRAGMENTS.

In some variants, the CONTENT CONVERTER object is also used to store the internal (generic) and the target format (instead of using an external CONTENT CACHE). This makes especially sense in some automatic type conversion systems, such as Tcl_Objs or some SOAP implementations. Here, the CONTENT CONVERTER object potentially “knows” the two representations in the two supported formats. However, at any time one of them may be undefined, if it is possible to create the content without losing information in both directions. The conversion is then performed when the typed or untyped object is requested the next time. When the information changes in one of the representations, the other representation is automatically invalidated.

Known uses:

- SAX [Megginson99] parsers and Expat are the basics for numerous event-based parsing architectures.
- DOM [W3C00] is the basics for numerous tree-based parsing architectures.
- xoRDF [NZ02] is a tree-based conversion architecture for RDF data that is extensible with multiple other interpretations using a VISITOR framework.
- XSLT [Clark99] is the basics for numerous rule-based parsing architectures.
- Sysmedia's Magenta is a content management system for interactive digital television that also allows for content conversion.
- Docme is a content management system that automatically converts input formats, such as Word, to an internal format. Conversion is mainly performed by "guessing" how the Word text should be mapped to the target format.
- The document archiving system in [GZ01] converts all inputs into an archive capsule format.
- Tcl_Obj's are a content converter that stores either the target format or a string representation or both.
- Credit Control Platform uses efficient, format specific, code generated CONTENT CONVERTERS to convert credit reports from different credit control systems into a GENERIC CONTENT FORMAT [Vogel00]. A modeling tool can be used to describe the schema of the input format. Based on the schema specific CONTENT CONVERTERS are created.

Pattern: CONTENT FORMAT BUILDER

In interactive web applications, content in HTML format and most often in multiple others formats has to be dynamically generated. Sometimes the same application has to support the same format in different variants. E.g. HTML may be delivered pretty-printed in a debugging version and compressed for optimizing file size in the released version. CONTENT CONVERTERS require a facility to build up a representation in a target format dynamically.



How can we build up content in different content formats dynamically and reuse the same code for different content formats? How do we avoid hard-coding content format specifics in the business logic code?

Different content formats have *different characteristics and specialties*. As an example of this diversity, consider for instance classical widget sets and markup formats, such as HTML and XML. Moreover, format types are heterogeneous in different incarnations. For instance, some widget sets have highly static and monolithic programming interfaces (such as Swing, AWT, or MFC), whereas other interfaces are highly dynamic (such as TK). Some markup formats such as XML are well-formed and can be validated with a DTD or schema, whereas HTML, for instance, is only loosely defined.

Converting one content format to another often means to reduce the expressibility of the application to the *common denominator* of all target (and input) formats involved. Otherwise we have to live with *lossy conversions*.

Often, we have to create the same content in the *same format in different ways*. Consider, for instance, generation of HTML text. Ideally, we would like to have pretty printed and indented HTML output that is easily readable. However, for larger pages this may become problematic: pretty printing HTML text means to insert a lot of white spaces and carriage returns. Therefore, in such cases, we require a more compressed output. When different platforms have to be supported, often we want to leave away marked parts of the content, such as leaving away larger pictures in HTML text for supporting a mobile devices.

Therefore:

Provide an abstract CONTENT FORMAT BUILDER class that determines the common denominator of the used interfaces. Build special classes that implement that interface for each supported content format, as well as special methods (e.g. as callbacks) for required specialties.

The classes' instances enable the application to incrementally build up pages in the user interface and to retrieve the result. Usually for each user interface element we have methods for starting and ending the element, so that elements may be placed in between.

Sometimes, the CONTENT FORMAT BUILDER builds up a string, say, for generating XML or HTML directly. Often, the CONTENT FORMAT BUILDER'S internal data representation is a COMPOSITE that is built up incrementally from the content format elements (which are then represented as objects). This variant has the advantage that the

content format in the memory can also be changed. That is, if the internal format of a CONTENT FORMAT BUILDER and a CONTENT CONVERTER are identical (e.g. a DOM tree), we do not have to perform input processing in the CONTENT CONVERTER after generating content on the CONTENT FORMAT BUILDER, but we can directly use the internal format generated.

CONTENT FORMAT BUILDERS let us abstract over specialties and characteristics of different user interfaces. However, we have to “simulate” the more advanced format is the less advanced ones, or reduce the output to the common denominator. Another variant is to live with lossy conversions.

Sometimes, living with lossy conversions is intended, say, if we want to provide a rich web interface, and reduced content for smaller mobile devices or settop boxes. In such cases, we can either leave certain parts of the content away during the building process, or use two variants of the CONTENT FORMAT BUILDER as STRATEGIES. [GHJV94]. Note that it is often easier and less memory/performance consuming to use CONTENT FORMAT TEMPLATES to create multiple different variants of the same content in the same format. Here, the content to be provided only on some platforms can be marked in the template definition.

In Figure 5 a typical design of a CONTENT BUILDER is shown. An abstract CONTENT BUILDER class determines the common interface for all derived builders. Here, four special Builder classes are derived: the GENERIC CONTENT FORMAT XML, HTML pages on the web, MMS pages for mobiles, and DVB-J Java classes that represent pages on interactive digital television platforms such as the Multimedia Home Platform.

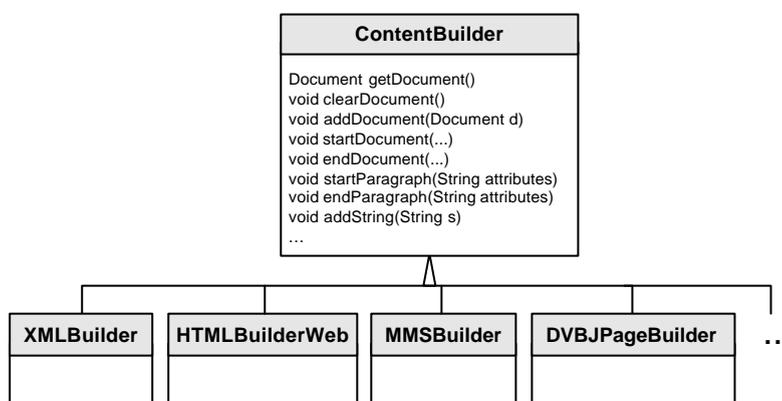


Fig. 5. Abstract Content Builder CONTENT BUILDER and Special BUILDERS

Usually, the content bricks are ordered hierarchically e.g. in a COMPOSITE or in a simple hierarchical list of strings. Each content brick that may have children has methods for starting the brick and ending it. E.g. a paragraph in an HTML builder may have children, thus, it has to be started and ended:

```

void startParagraph(String attributes) {
    addStringIncr("<P ");
    addString(parseArguments(attributes));
    addStringIncr(">\n");
}
void endParagraph() {
    addStringDecr("</P>");
}
  
```

Leafs, such as strings, have only a method for adding the leaf. In startParagraph and endParagraph we have used the methods addStringIncr, addString, and addStringDecr for adding the leafs that markup the paragraph. Only addString is a method supported by the abstract CONTENT BUILDER. addStringIncr and addStringDecr are methods for possibly increasing/decreasing the indent level of HTML text before adding a string. Thus, they represent a specialty of the HTML format.

CONTENT FORMAT BUILDERS let us generically program how to build up the content format, thus, they are a generic constructive approach. In contrast, CONTENT FORMAT TEMPLATES and FRAGMENTS are a template-based approach for the same problem (but both have a different set of forces in focus).

The CONTENT FORMAT BUILDERS pattern offers a set of benefits: The CONTENT FORMAT BUILDER allows for abstracting from *multiple target formats*. Compared to implementing each target format by hand, the CONTENT FORMAT BUILDER result in *shorter code* that is *easier to maintain*, say in cases of changing web standards, new

features, etc. CONTENT FORMAT BUILDER *avoid scattering format specifics throughout the business logic code*. The constructive approach of the CONTENT FORMAT BUILDER is in comparison to template or fragment-based approaches more *flexible*. *Syntax errors in the target format can be detected a priori*, say, the builder can raise an error, if a content element is opened but not closed.

The CONTENT FORMAT BUILDERS pattern can also incur the following liabilities: The constructive approach of the CONTENT FORMAT BUILDER is in comparison to template or fragment-based approaches rather *slow*. Problems of *lossy conversions* and reducing all inputs to the *common denominator* of the target formats can only be avoided by programming specialties of target formats for all other formats by hand. CONTENT FORMAT BUILDERS require programming efforts to create and customize content, thus, they are hardly applicable at the end-user level without tool support.



Known Uses:

- Antti Salonen's Htmllib is a CONTENT FORMAT BUILDER written in XOTcl for the HTML target format. It builds up a Tcl list dynamically on the builder object. It supports the most important parts of HTML's functionality.
- WebShell [Vckovski01] uses Tcl procedure to implement each part of the construction of a web page. These are combined in a special method that assembles and delivers the web page. The code of this procedure already resembles the document to be created. Tcl lists are used for combing the children of a node, e.g. a list with three children may look like:

```
ul {
  li "Element 1"
  li "Element 2"
}
```
- In the EU project TPMHP we are building a Java-based content-format builder for the Multimedia Home Platform which should support DVB-J, HTML, and MMS pages.
- The conference management system, described in [Zdun02a], uses HTML builder objects.

Pattern: FRAGMENTS

Instead of providing static web pages only, today's web sites offer dynamically generated web pages, enriched with real time information like stock quotes in a sometimes highly personalized manner. Examples of such web sites are financial, news and sports sites. These sites form a deep web which is by several orders of magnitude larger than the traditional surface web. Architects faced with the development of such sites must cope with new requirements making the design of highly dynamic systems a real challenge as content must be served efficiently and consistently.



How can web pages be designed in order to allow the generation of web pages dynamically by assuring the consistency of its content? Moreover, how do you provide these dynamic web pages in a highly efficient way?

Generating web pages from dynamic content is an expensive task as *content has to be fetched from data stores* like RDBMS, XMLDBMS or even from other web systems by accessing web services. This leads to *increased I/O* operations and often *network overhead* as backend system are incorporated over the intra- or even the internet.

Furthermore, assembling of the retrieved content to web pages results in a *processing overhead* as content might have to be converted into a GENERIC CONTENT FORMAT and web pages are regenerated completely as no means are available to determine which parts of a web page have changed. Often web pages as a whole are the most fine grained building blocks of web systems. Therefore, web pages cannot be served in an *efficient* manner as the whole web page has to be regenerated.

The *consistency of the content* displayed on the web page is another key challenge. It has to be avoided that different parts of a web page are inconsistent. Consider a web page showing stock quotes belonging to the user's portfolio. To get more detailed information on a specific stock the user can click on a hyperlink bringing up a details page. The information on that page may not be older or inconsistent with the one displayed on the former page. In order to assure that web pages are generated consistently intelligent means must be available to identify that underlying content has changed. This enforces a *flexible and intelligent information architecture*.

Therefore:

Provide an information architecture which represents web pages from smaller building blocks called FRAGMENTS. Connect these FRAGMENTS so that updates and changes can be propagated along a FRAGMENTS chain.

FRAGMENTS are pieces of information which have an independent meaning and identity. A single stock quote, news or user profile information are examples of FRAGMENTS. These independent parts can be assembled to compound parts like whole web pages. Thus FRAGMENTS can contain other fragments and reference others. Fragments can thereby build a dependency chain or object dependency graph. If FRAGMENTS lower in the chain change, the higher FRAGMENTS have to be revalidated and regenerated. Thus, only the parts of a web page which have actually changed have to be regenerated leading to a decreased processing overhead.

As FRAGMENTS have an independent meaning in the user's conceptual model they can build the basic entities for caching strategies. It is important to understand that FRAGMENTS are a concept of the used information architecture and are completely independent of base technologies like J2EE or .NET. Therefore the same information architecture can be used on different technology platforms. Furthermore, FRAGMENTS have names and identities and can be associated with a system identifier enabling FRAGMENTS to be stored and read using through different services in different places [Kriha01]. Thereby a FRAGMENTS based information architecture fits nicely into the overall software architecture of a web application system as they can be represented by conventional means like classes.

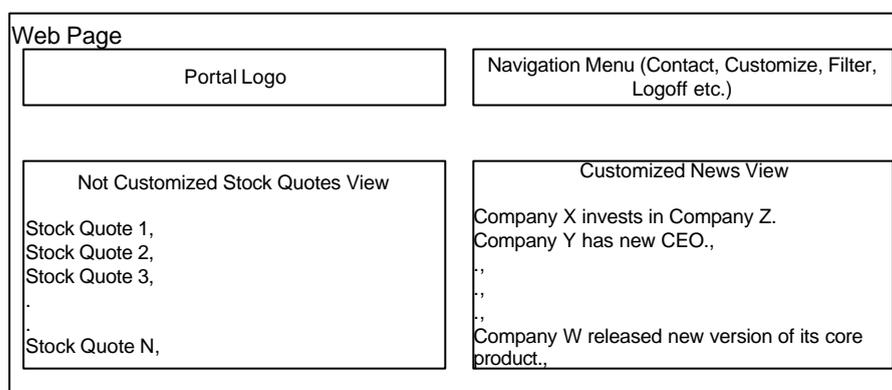


Fig. 6. An example web pages containing personalized and non personalized parts

The illustration above shows a web page of a financial portal site constructed from smaller building blocks. The portal logo and the navigation menu are user independent and thus appear on every portal page. The not customized stock quotes view is build upon dynamic content but not personalized. Therefore, it can be reused across different portal pages. Instead the customized news view is personalized by the user and is specifically generated for that particular user. However, several users could have the very same configuration or the different news items could appear on different web pages as well. Thus, there is a reuse potential for the news view or news items. Furthermore the stock quotes view and the news view are themselves build from smaller building blocks namely stock quotes or news items respectively.

Using the fragments concept the web page is a compound fragment containing the portal logo fragment, the navigation menu fragment, the stock quotes fragment and the news fragment. The stock quotes and news fragment are compound fragments as well build from stock quote and news item fragments. Like the GENERIC CONTENT FORMAT a FRAGMENTS architecture can be designed using the COMPOSITE pattern.

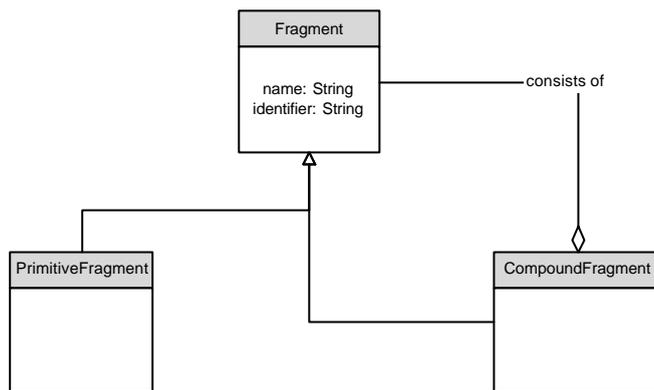


Fig. 7. Generic Fragments Structure using the Composite Pattern

Using the COMPOSITE pattern arbitrary FRAGMENT trees can be assembled. In order to tell which FRAGMENTS make up which other FRAGMENTS Fragment Definition Sets (FDS) are used. Fragment Definition Sets are fragments itself and build an object dependency graph necessary to invalidate FRAGMENTS and to detect which parts of a fragment have to be regenerated. The Fragment Definition Sets can themselves be modeled using the COMPOSITE pattern.

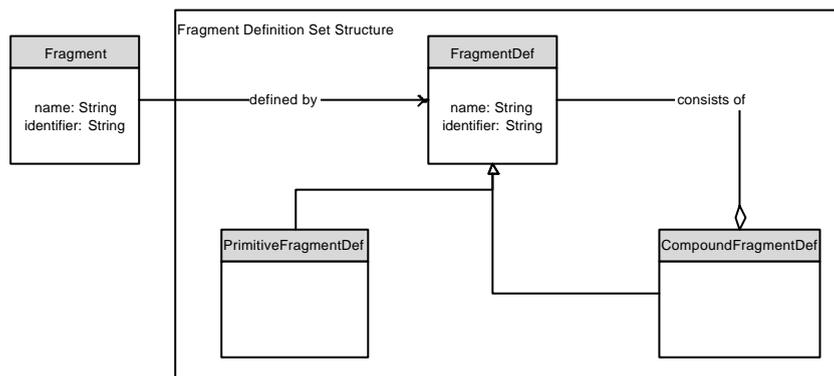


Fig. 8. Generic Structure of a Fragment Definition Set

FRAGMENTS are defined by FRAGMENT definitions. Combining the definition and the instance level of the information architecture leads to a dynamic object model system as described in [RTJ00].

Besides using FRAGMENTS to structure web pages, FRAGMENTS are also ideal candidates to model dependencies between different formats of the same content.

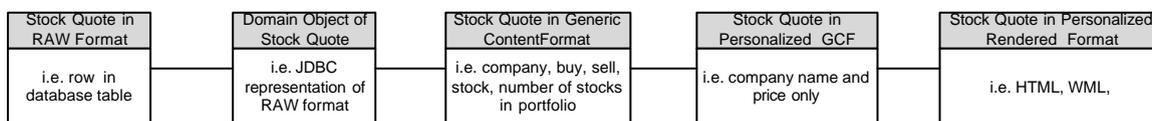


Fig. 9. Fragments Chain of the same Content

If any part of the fragment chain changes, its successor has to be revalidated and regenerated. The upper part in the chain, the rendered fragment, is usually part of a web page chain triggering the revalidation of the affected parts of the web page after its regeneration. To detect and to propagate fragment changes special algorithms can be used. For example, a Data Update Propagation (DUP) algorithm can be used to propagate changes along the FRAGMENT chain by assuring consistent updates as described in [CIW00]. Another approach is to include special validator objects containing the logic necessary to determine if fragments have become invalid and therefore have to be updated. The validators can either be configured using a rule based approach or be created programmatically

[Kriha01]. Moreover, caching can be integrated within the FRAGMENTS architecture as explained in CONTENT CACHE.

The FRAGMENTS pattern offers a set of benefits: Compared to the other content generation patterns, FRAGMENTS potentially offer the highest *performance*. Fragment elements are highly *personalizable*, and offer a *good integration with a layered CONTENT CACHE*. The other content generation patterns can be combined with the FRAGMENT approach.

The FRAGMENTS pattern can also incur the following liabilities: FRAGMENTS only *assemble pre-built parts*. They are *not highly programmable* and do not offer *behavioral abstractions*. However, these problems can be eliminated by combining them with the other content generation patterns. In pre-built FRAGMENTS content changes have to be detected and propagated to ensure *content consistency*.



Known Uses:

- Olympic Games 2000 Web Site is built by IBM using a system for the dynamic creation of web content [CIW00].
- Edge Side Includes are a new evolving technology used to describe cacheable and non cacheable Web page components. These components can be aggregated, assembled and delivered at runtime [ESI02].
- AOL Server's SOB (small objects) is an interface for dynamic publishing of editorial content [Davidson00]. SOBs can be placed as FRAGMENTS in templates. They are used in AOL Digital City and Movie Guide.

Pattern: CONTENT FORMAT TEMPLATE

In interactive web applications, content in HTML format and most often in multiple others formats has to be dynamically generated. Sometimes the same application has to support the same format in different variants. E.g. HTML may be delivered pretty-printed in a debugging version and compressed for optimizing file size in the released version. CONTENT CONVERTERS need a facility to build up a representation in a target format dynamically.

CONTENT FORMAT BUILDERS operate in the same context. But they build up the content in a programmatic and constructive approach. In some domains, this can lead to significant liabilities regarding end-user customizability and performance compared to static HTML content.

A FRAGMENT is a template-based approach. It codes only the fragment ID into the document, but it does not include the dynamic content itself. Thus, dynamic behavioral aspects of content that can be coded into the documents themselves is limited.



How can we build up content in target content format and allow the content editor to add highly dynamic content parts in a simple way that yields a high performance?

An important limitation of CONTENT FORMAT BUILDER is that it requires programming to create and customize the content created. *End-user-level customizability*, however, is important for many web applications since web developers are easier to hire (and less costly) than qualified programmers.

Compared to static HTML content, CONTENT FORMAT BUILDERS are rather slow. For high-performance systems a *performance* closer to using static content is required. Most often only small parts of a page are dynamic, and others are given statically. In suitable cases, we should not build up the whole page dynamically, but use the static content where possible.

Often, we have to create the same content in the *same format in different ways*. When different platforms have to be supported, often we want to leave away marked parts of the content, such as leaving away larger pictures in HTML text for supporting a mobile devices.

WEB FRAGMENTS solve both of these issues to a certain extent. However, for highly dynamic content elements we still have to create these fragments e.g. using CONTENT FORMAT BUILDERS. Therefore, in such cases the problems appear again during construction of the fragments.

Therefore:

Provide a template written in the content format that contains special code in a template language to be substituted by a template engine.

A CONTENT FORMAT TEMPLATE enriches the content with meta-information. A (little) language has to be defined for specifying the substitutions to be performed by the template engine. In some variants this is a whole scripting language.

A simple example of CONTENT FORMAT TEMPLATE are JSPs that contain Java code to be substituted. The substitution rules can also be applied e.g. with XML. The principle is to let the template engine find special tags containing the Java code and execute it before delivering the pages, e.g. here the data is computed dynamically:

```
<%@page import="java.util.*" %>
<HTML>
...
<BODY>
<H2>Date and Time</H2>
  Today's date is: <%= new Date() %>
</BODY>
</HTML>
```

Scripting approaches often fit better to the task of dynamically enriching web content because scripting languages are actually designed for dynamic string substitution and other prominent tasks of web templates. A typical example are AOLServer's ADP templates that are using Tcl. E.g. in the following example a web page is created dynamically in which the user's browser type and the time is displayed:

```
<%
  set header [ns_conn headers]
  set browser [ns_set iget $headers User-Agent]
  set time [clock seconds]
%>
<html>
  <body>
    Time: <%= $time %>
    Browser: <%= $browser %>
  </body>
</html>
```

The CONTENT FORMAT TEMPLATE can internally be realized using CONTENT FORMAT BUILDERS. Other combinations of the patterns are also possible. E.g. templates may be embedded in CONTENT FORMAT BUILDER'S client code.

The CONTENT FORMAT TEMPLATE pattern offers a set of benefits: At first glance, template production looks *very simple and straightforward*. That is, web page design can be separated from program development, and it is possible for web designers to create dynamic pages. In general, the approach is more *efficient* than purely constructive approaches on top of CONTENT FORMAT BUILDERS. In contrast to FRAGMENTS more high-level *dynamic interactions* can be supported in the content format. Simple *behavioral customizations can be performed by the end-user*.

The CONTENT FORMAT TEMPLATE pattern can also incur the following liabilities: In many approaches such as JSP and ASP the *promise to be simple and straightforward turns out to be unrealistic in practice* because complex programming language elements have to be understood by the web designers. Real applications have complex interdependencies. Since templates only act at the *local level of a single document* they can hardly cope with these issues. A second liability results from this problem: recurring elements often have to be recoded for every use in a template, that is, there is only *limited reuse* of template code. The *page design and business logic of the application are usually not separated*.



There are many CONTENT FORMAT TEMPLATE variants based on popular programming languages that are embedded in HTML code. We can generally distinguish between approaches aiming at the end-user/web designer level, and more complex approaches. Another aspect to distinguish the approaches is caching and interpretation. Some approaches always compile pages, some approaches cache pages once they are created, and other approaches always interpret the pages.

Known Uses:

- JSP and ASP are CONTENT FORMAT TEMPLATE platforms that deliver similar functionality. ASP pages are written in Visual Basic and JSP pages are written in the Java. They both use tags to allow embedded code in an HTML page, session tracking, and database connection. As an important difference, JSP pages are compiled into Java servlets, whereas ASP pages are always interpreted (but they may be cached). Thus, in most case JSP pages should be more efficient. JSP additionally allows for defining custom tags. As a disadvantage, both approaches require very “low-level” programming and are therefore hardly applicable at the end-user level. Scripting approaches for building templates on the web are often easy to customize. In Java based applications, however, we have to care for integration with Java entities such as Java beans.
- PHP introduces a new language for web page templates. It is small, light-weight, efficient, and easy for non-programmers to use. However, as a disadvantage the language is only created for one use: on the web.
- The Apache modules mod_perl, mod_tcl, and WebShell [Vckovski01] allow for combining templates, written in Tcl and Perl, with the Apache web server.
- Zope is a rather complex system for integrated web development that resides on the Python language, and also allows for templates.
- Some approaches provide combinations with CONTENT FORMAT BUILDERS: WebShell [Vckovski01], ActiWeb [NZ01], and Brent Welch’s TclHttpd can construct pages dynamically, and embedded template elements in the HTML code used to construct an HTML page.
- AOLServer’s ADP templates integrate HTML, Tcl, and the AOL Server interfaces. They are used on numerous high-performace web sites, including AOL Digital City and Movie Guide [Davidson00].
- Credit Control Platform supports different CONTENT FORMAT TEMPLATES. Data Visualizers can be specified on a meta level using a special modeling tool [BIV00]. Concrete CONTENT FORMAT TEMPLATES can be generated for different technologies like JSP, ASP and XSLT.

PATTERN: Content Cache

You are developing a web application system targeted to many users which has to support dynamic content in an efficient way. The processing time required to render web pages should be reduced.



How can you increase the performance of web page delivery and thereby increase efficiency of the web architecture.

Dynamic web application systems often lack in providing web pages in an *efficient way*. A FRAGMENTS architecture can be used in order to reduce the amount of parts of a web page having to be *regenerated* every time a new request enters the system. However, the performance of the overall web architecture might still be insufficient.

Content changes that affect already created content have to be detected and propagated to *avoid content inconsistencies*.

Therefore:

Provide a central CONTENT CACHE for caching already created dynamic content. Consider the life time of pre-created dynamic content and cache it as long as it is still valid in the application’s context.

The main reason for caching is to increase throughput and thereby performance. However, enabling caching in a consistent way is challenging as accurate cache invalidation algorithms have to be applied. Moreover, client and server side caching has to be considered. Whereas server side caching enables cache invalidation by introducing validator objects containing the knowledge when a cached piece of content becomes invalid explicitly, client side caching is often quite cumbersome.

First of all, clients, in most cases web browsers, must adhere to a protocol supporting the control of client side caching from the server side. Although, the most common protocol HTTP allows setting certain caching parameters most web browsers on the market still do not implement the HTTP specification accurately. This makes caching of dynamic content on the client side unreliable as you cannot be aware how the client’s browser implements the specification. You could limit access to your web site to certain, tested browsers only. But the next version or the same version on another platform might still behave differently. Thus, often the only choice is to turn off client side caching completely leading to a decrease of performance.

Server side caching is an effective means to speed up overall request satisfaction. To support efficient server side caching an information architecture must be in place which decomposes the information space along the

dimensions time and personalization and which distinguishes clearly between global pieces, individual selections of global pieces and really individual pieces [Kriha01]. An information architecture based on FRAGMENTS can be used to classify content. Moreover validator objects can be applied to determine, if a piece of information is still valid according to time and personalization constraints. The validator objects can either be configured using a rule-based approach or implemented programmatically. Different validator algorithms can be supplied using the STRATEGY pattern.

An important question has been neglected so far. Can or should dynamic content actually be cached? The answer is yes for a large scale web site. According to a report by Yahoo (Communications of the ACM, Personalization) 80% of all users do not customize their homepage. This means that besides the welcome message, everything appearing on the individual's portal page stays the same. Caching these parts truly increases the performance of the overall web site tremendously.

Assuming that hundreds of requests for the same stock quote entering the system at the same time would lead to hundreds of requests to the backend system for retrieving the very same piece of information affects system performance heavily. Only the first request should trigger the retrieval of the information all subsequent request should receive the information from the server side cache as long as it is valid. For most types of information an accuracy of a few seconds is acceptable. Therefore, every request should go through a CONTENT CACHE. The CONTENT CACHE checks if the requested piece of information is in the cache and if it is valid. If not, the content is loaded from the backend system and stored in the cache. Afterwards it is returned to the client. This applies for whole web pages as well for parts of web pages. The process of retrieving, converting and storing content in the CONTENT CACHE is shown in the following illustration.

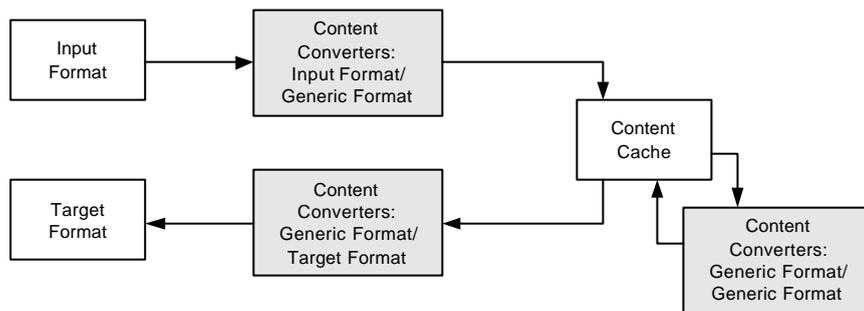


Fig. 10. CONTENT CONVERTERS (triggered by the PUBLISHER AND GATHERER) and the CONTENT CACHE

Content can be gathered and published by using the PUBLISHER AND GATHERER pattern. Typically CONTENT CONVERTERS have to be triggered before and/or after the content is placed in the CONTENT CACHE. The PUBLISHER AND GATHERER has to check whether the CONTENT CACHE contains a valid entry before it re-creates content dynamically.

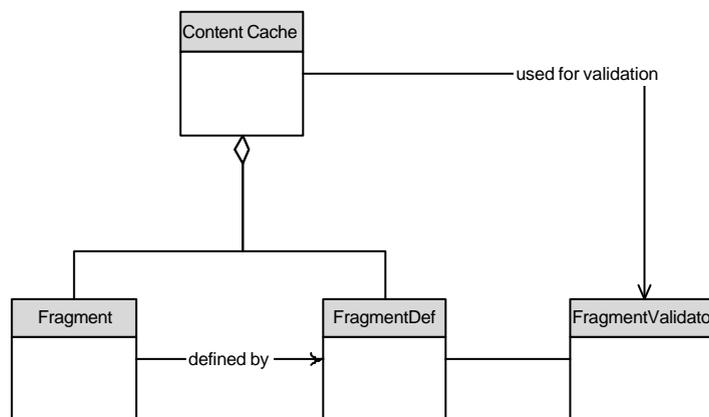


Fig. 11. Internal Structure of Content Cache

The *ContentCache* itself contains *Fragments* as well as *FragmentDefs* and uses associated *FragmentValidators* to validate *Fragments* of certain types.

Chains of FRAGMENTS representing the same content in different formats can be cached in the *ContentCache* too. Because of the behavior of *FragmentChains* the *ContentCache* is not the only active component within the caching process. FRAGMENTS within a chain automatically notify its successors upon content change triggering their revalidation and probably leading to the invalidation of the *ContentCache*. Thus, FRAGMENTS play an active role in the caching process as well.

The CONTENT CACHE pattern offers a set of benefits: In combination with FRAGMENTS the patterns allows for a highly *efficient* information architectures. Together with a PUBLISHER AND GATHERER it *integrates well* with CONTENT CONVERTERS.

The CONTENT CACHE pattern can also incur the following liabilities: Possible *inconsistencies* in the CONTENT CACHE have to be resolved. In exceptional cases change detection and propagation can be more costly than the performance gain of caching. In multi-threaded environments a CONTENT CACHE requires mutex locks which can result in *lock contention*. Therefore, it is important to *monitor hit rates and contention* closely.



There are different variants of CONTENT CACHES. A cache can be supplied as one central instances. As a variant, there can also be multiple caching instances, one for each content element. E.g. Tcl_Obj's use this style of caching: each Tcl_Obj is one cached element plus a CONTENT CONVERTER to/from a generic, string-based representation.

A CONTENT CACHE can support automatic invalidation of all dependent objects, or invalidation has to be handled by hand. Moreover, CONTENT CACHES can also support more advanced forms of content change detection and propagation such as object dependency graphs [CIW00].

If personalized FRAGMENTS are supported, an important variant is a layered CONTENT CACHE. Each caching layer than reflects one personalization layer in the FRAGMENTS.

Known Uses:

- Olympic Games 2000 Web Site uses a server side content cache to cache dynamic content [CIW00].
- ASP offer a cache for all created pages.
- AOL Server [Davidson00] implements resources for caching in a multi-threaded environment. Here, the cached data has to be mutex-protected during writing. AOL Digital City [Davidson00] uses this functionality for central content caching servers.
- AOL Server's SOBs (small objects) are FRAGMENTS that are placed in templates, and that are aggressively cached, e.g. in AOL Movie Guide [Davidson00].
- xoComm provides a similar caching structure on the client side as well [NZ00].
- Tcl offers Tcl_Obj's as central data conversion types. Tcl_Obj's cache a generic, string-based representation and/or the internal representation. The interpreter's command table provides a pool of all Tcl_Obj's in the interpreter.

4 Conclusion

In this paper we have presented patterns for dynamic generation of content on the web. The patterns are used in many different web architectures, and, to a certain extent, different available technological instances can be exchanged. E.g. different models of CONTENT CONVERTERS or different CONTENT GENERATION techniques can easily be exchanged. However, the base-line architecture stays the same, despite such important technological decisions. Since most basic technologies are based on XML, and since components, such as parsers and processors, are widely available for many different programming languages, we can assert that the patterns can be used for architectural decisions apart from concrete technological realizations. Therefore, they provide a good communication means with different stakeholders of the system in focus.

In our experience, the patterns yield architectures with a set of benefits and liabilities that vary slightly for different used implementation technologies, for different combinations of the patterns, for different sequences

through the language, and for different variants of the patterns. In this section we want to summarize the benefits and liabilities briefly.

The patterns strongly encourage architectures that provide a separation of concerns between content, styles, formats, and channels. That is the reason, why different technological choices can relatively easily be exchanged against each other. MESSAGE REDIRECTORS [GNZ01] can be used to implement the indirection to the channels, and add-ons for the channels can be transparently provided, such as logging or authentication.

Using SERVICE ABSTRACTION LAYER [Vogel01] multiple representation channels may be supported. CONTENT FORMAT BUILDER and CONTENT FORMAT TEMPLATE can be used to abstract from different content formats. Thus, a common denominator can be implemented with minimal programming effort. Both patterns provide a programmable alternative to using FRAGMENTS alone, and both can be integrated with FRAGMENT approaches.

All generational aspects are handled at runtime. Therefore, introducing changes into a running program is natively supported by many architectures based on the pattern language. However, since generation is always more performance intensive than delivering static HTML pages (e.g. stored in files or in a database), performance may be influenced negatively. Therefore, the balance between CONTENT FORMAT BUILDERS, CONTENT FORMAT TEMPLATES, and static content often has to be considered very carefully. In different applications, performance impacts may significantly vary. Thus often combinations of the patterns and caching are necessary to reach acceptable results. These forces are primarily resolved by the FRAGMENT and CONTENT CACHE patterns.

If CONTENT FORMAT BUILDERS are used exclusively, the user interfaces are reduced to the common denominator defined in the abstract user interface builder. Of course, certain CONTENT FORMAT BUILDERS may also ignore certain formatting instructions, say, like a WML CONTENT FORMAT BUILDER that does not fully support the HTML subset.

On first sight, the complexity architectures based on the patterns is higher than simple architectures, such as template-based approaches or CGI scripts. However, for larger tasks, the complexity of the simpler models usually grows exponentially, e.g. because of cut-and-paste code and missing integration models. Therefore, in real-world, large-scale web applications the complexity, and thus the maintainability and understandability, is rather influenced positively by applying the pattern language.

Acknowledgements

References

- [AIS+ 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. A Pattern Language – Towns, Buildings, Construction. Oxford Univ. Press, 1977.
- [BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. <http://www.w3.org/TR/SOAP/>, 2000.
- [BIV00] A. Bredenfeld, E. Ihler, O. Vogel, GENVIS, Model Based Generation Of Data Visualizers, In Proceedings of Technology of Object Oriented Languages and Systems Conference, France, 2000, <http://www.ovogel.de/publications/GENVIS.pdf>
- [CIW00] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content, In Proceedings of IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.
- [Clark99] J. Clark. XSL transformations (XSLT). <http://www.w3.org/TR/xslt>, 1999.
- [Davidson00] J. Davidson. Tcl in AOL digital city the architecture of a multithreaded high-performance web site. In Keynote at Tcl2k: The 7th USENIX Tcl/Tk Conference, Austin, Texas, USA, February 2000. <http://www.aolserver.com/docs/intro/tcl2k/>.
- [ESI02] Edge Side Includes (ESI) Overview, <http://www.esi.org>, 2002.
- [GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [GZ01] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. Accepted for publication in Journal of Software Maintenance:

- Research and Practice, 2001.
- [GNZ01] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001.
- [Kriha01] W. Kriha. Advanced Enterprise Portals, <http://www.kriha.org>, 2001.
- [Megginson99] D. Megginson. SAX 2.0: The simple API for XML. <http://www.megginson.com/SAX/index.html>, 1999.
- [NZ00] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. World Wide Web Journal, 3(1), 2000.
- [NZ01] G. Neumann and U. Zdun. Distributed web application development with active web objects. In Proceedings of The 2nd International Conference on Internet Computing (IC'2001), Las Vegas, Nevada, USA, June 2001.
- [NZ02] G. Neumann and U. Zdun. Pattern based design and implementation of a XML and RDF parser and interpreter: A case study. Accepted for ECOOP 2002, 2002.
- [RTJ00] D. Riehle, M. Tilman, and R. Johnson. Dynamic object model. In Proceedings of 7th. Pattern Languages of Programs Conference (Plop 2000), Monticello, Illinois, USA, August 2000.
- [Vckovski01] Andrej Vckovski. TclWeb. In Proceedings of 2nd European Tcl User Meeting, Hamburg, Germany, June 2001.
- [Vogel00a] O. Vogel, Usability of XML, Generation of efficient XML Content Converters, Speech at XML for Business, Switzerland, Zuerich / Regensdorf, 2000, <http://www.ovogel.de/publications/XML4Business.pdf>
- [Vogel01] O. Vogel. Service abstraction layer. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001. <http://www.ovogel.de/publications/ServiceAbstractionLayer.pdf>
- [W3C00] W3C. Document object model. <http://www.w3.org/DOM/>, 2000.
- [Zdun02a] U. Zdun. Dynamically generating web application fragments from page templates. In Proceedings of Symposium of Applied Computing (SAC 2002), Madrid, Spain, 2002.
- [Zdun02b] U. Zdun. Reengineering to the Web: Towards a Reference Architecture. In Proceedings of 6th European Conference on Software Maintenance and Reengineering, Budapest, Hungary, 2002