Chapter 4

A Concern-Oriented Component Model for Adaptive Web Applications

"If we knew what it was we were doing, it would not be called research, would it?"[1]

The previous chapter reviewed and compared existing Web engineering solutions aimed at the development of adaptive Web-based systems. It investigated both model-driven approaches addressing the conceptual design and high-level specification of Web applications, as well as component-oriented and document-centric solutions primarily focusing on their presentation and implementation aspects. It was pointed out that component-based reuse is a crucial issue of Web engineering. Still, there is lacking support for the efficient creation of adaptive multimedia Web presentations from reusable and configurable implementation entities.

To fill this gap, this chapter presents a *concern-oriented component model*[2] for adaptive dynamic Web applications. The term *concert-oriented* denotes its explicit support for the clear separation of concerns involved in a Web application, i.e. it enables to compose adaptive Web presentations by the aggregation and linkage of reusable *document components* that encapsulate different application *concerns* such as content, structure, navigation, semantics, presentation (as well as their corresponding adaptation issues) on different abstraction levels. The resulting document component structures can be automatically translated to Web presentations that are adapted to a specific user, device, output format, or other context information.

The remainder of this chapter is structured as follows. First, in Section 4.1, the document-centric component concept is discussed, and a number of requirements towards a document model for adaptive Web presentations are mentioned. Based on these requirements, the following sections (4.2 to 4.4) present the component-based document model and its XML-based description language in detail. The different abstraction levels of document components, their support for adaptation, as well as the concept of document component templates are explained by examples. In Section 4.5 a pipeline-based document generator aimed at the on-the-fly publishing of component-based adaptive Web applications is presented. Finally, selected benefits of the proposed model are discussed in Section 4.6.

[1]Albert Einstein (1879 - 1955)

[2]The component model was developed within the scope of the AMACONT project and is also often referred to as the AMACONT component model.The thesis presents the author's contributions to the model, based on requirements towards the efficient authoring of adaptive Web applications from reusable components.

4.1 Declarative Document Components

Szypersky defines in [Szyperski 1998] the notion of a software component as follows:

Definition 4.1 (Software component) *A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [Szyperski 1998].*

This original definition considers a software component as a *binary* unit of composition that is typically based on an imperative implementation. Still, in recent years a number of declarative component approaches have emerged, aiming at describing not only the interfaces and properties, but even the functionality of reusable implementation entities in a declarative human-readable form [Dachselt 2004, Aßmann 2005]. This shift from the traditional program-centric to a document-centric application development paradigm has been especially characteristic for the World Wide Web. As prominent examples well-established standards such as SMIL (Synchronized Multimedia Integration Language [Bulterman et al. 2005]) or SVG (Scalable Vector Graphics [Ferraiolo and Jackson 2003]) can be mentioned, that facilitate to create complex Web and multimedia applications on top of XML-based declarative document descriptions. Furthermore, as discussed in Section 3.2, there already exists a number of approaches (WCML, CONTIGRA, CHAMELEON) that explicitly focus on reusing declarative Web implementation entities in a component-wise manner.

The component-based document model presented in this chapter was inspired by the previously discussed approaches WebComposition (WCML), CONTIGRA, and CHAMELEON. Adopting their document-centric component concept, it supports the development of personalized ubiquitous Web presentations from declarative reusable implementation entities called *document components* [Fiala et al. 2003a, Fiala et al. 2003b].

Document components are XML documents, instances of a specific XML-grammar describing adaptive Web content. They can be defined on different abstraction levels, each representing a separate application concern (e.g. content, structure, semantics, navigation, presentation) involved in a Web presentation. Document components are unequivocally identified by a unique identifier and further described by appropriate metadata. Acting as their interface definition, this metadata specifies their properties (such as their structure, layout) as well as their adaptive behavior. Web sites are constructed by configuring, aggregating, and interlinking components to complex component structures. During document generation, these abstract document structures are dynamically translated into Web pages in a concrete output format and are automatically adapted to the current usage context.

Even though document fragments are no (binary) software components according to the above mentioned software engineering definition of Szyperski, note that they show a lot of similarities to the classic component concept. They are system independent and reusable units, representing a certain functionality that can be combined to complex applications. Furthermore, they provide a clearly defined interface described by specific metadata, allowing for configuration and aggregation on higher component levels[3].

While the specifics of the component model and its XML-based description language are described in detail in the subsequent sections, the following list comprises the most important

[3]Though being different from the component definition of Szypersky, note that the concept of document components also corresponds to the notion of components of a hypermedia system as defined by the Dexter reference model [Halasz and Schwartz 1994].

requirements that were considered during their design. These requirements were derived from the previously discussed shortcoming of existing solutions (see Section 3.2.10) as well as the main goal of the thesis: the efficient component-based authoring of adaptive Web applications from reusable implementation artefacts[4].

1. **Rigorous separation of different concerns** involved in a Web presentation, such as content, structure, navigation, layout, and adaptation.
2. **Reusability** of (parts) of Web presentations on both different abstraction levels and of different granularity (i.e. from fine-granular atomic resources to coarse-grained complex Web document structures).
3. **Composability** of reusable document parts to aggregates (composites) that again act as reusable units and can be subject to further composition.
4. **Ease of configuration and adjustment** of parts of Web presentations of different granularity through well-defined interfaces described by appropriate descriptive metadata.
5. **Inherent adaptation support** by built-in language constructs allowing to refer to user and context model parameters. Provision of generic facilities for defining conditional alternatives (of different concerns such as content, structure, navigation, presentation) depending on the actual client device, user preferences, and the entire usage context.
6. **Platform and device independence** by abstraction from concrete implementation platforms, client devices, Web browsers, and specific Web output formats.
7. **Support for media integration** allowing to incorporate both existing and future media and internet document formats.
8. **Support for data-driven Web presentations** based on component templates that can be extended (i.e. “filled”) with dynamically retrieved data at run-time.
9. **Interoperability** with existing internet standards by extensive usage of approved XML technologies.
10. **Extensibility and flexibility** support through modularity as well as well-defined metadata interfaces.

4.2 A Component-based Document Model and its XML Description Language

As discussed above, the proposed document model is based on the notion of declarative *document components*. These are instances of a specific XML-grammar that represent different application and adaptation concerns on various abstraction levels and can thus be configured, aggregated, and interlinked to complex adaptive Web presentations.

In order to support platform independence and interoperability, the document format was specified based on XML-technology. It is defined by a set of XML Schema documents

[4]Note that these requirements also served as the basis for comparing existing component-based and document-oriented approaches in Section 3.2.10

[Fallside and Walmsley 2004], each specifying a separate aspect of the document model, such as composition, metadata, adaptation, presentation, etc. Note that XML Schema allows for more powerful definitions than a DTD by supporting namespaces, type safety, reuse of type definitions, more exact expressions of cardinality, self-documentation as well as type inheritance.

The document model allows to define components on different abstraction levels (also referred to as component levels), each responsible for a given application concern. In Figure 4.1 the corresponding level-based architecture is illustrated. It is constituted of *media components*, *content unit components*, *document components*, and the *hyperlink view*. Based on a number of representative examples, the following sections introduce each component level in more detail.

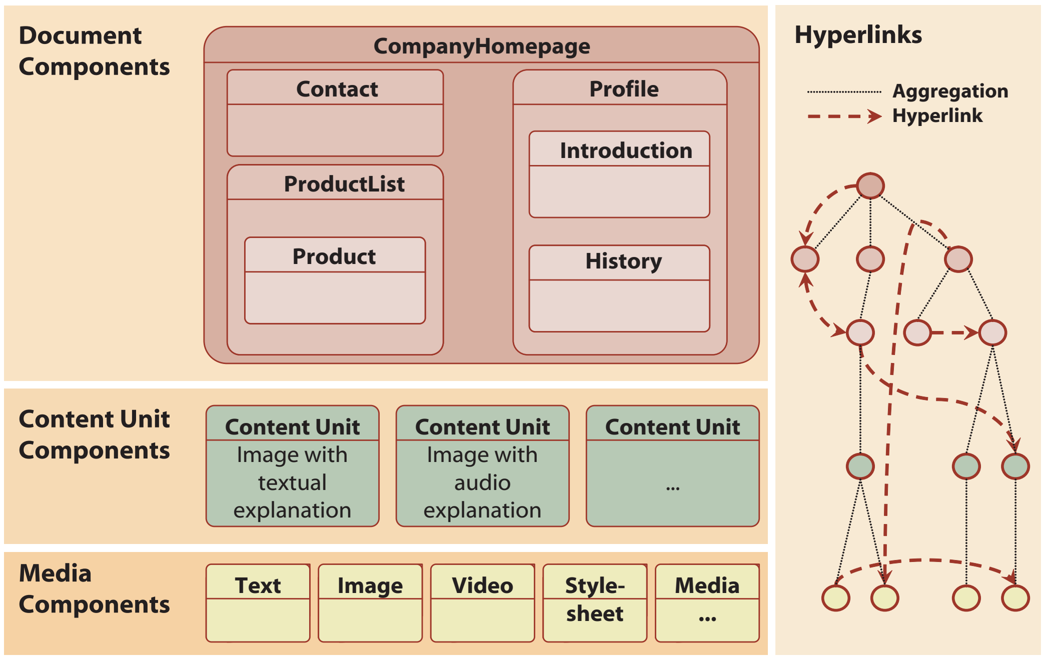

Figure 4.1: A concern-oriented component model for adaptive Web sites [Fiala et al. 2003a]

4.2.1 Media Components

On the lowest level of the component hierarchy there are *media components* that encapsulate particular media assets by describing them with specific XML-based metadata. The set of supported media assets comprises text, toggle-text, structured text (e.g. HTML or XML code fragments), images, sound, video, Java applets, Flash and Director presentations, but may be extended arbitrarily. Dynamically created media components (e.g. HTML fragments or pictures generated on the fly) are also supported, provided that the corresponding metadata is delivered, as well. Furthermore, even whole documents (such as HTML pages) might be wrapped to media components in order to optimize page generation and support flexible authoring.

As an example, the code snippet in Listing 4.1 describes a simple image component. It is unequivocally identified by its *name* attribute which is unique for all components. The *layer* attribute (see line 1) dictates that it is a component from the level of media components.

The namespace *aco* identifies the XML schema definition AmaComponent.xsd which specifies the different levels, types as well as the composition hierarchy of components. Similarly, the namespace *amet* identifies the XML schema document AmaMetaInformation.xsd aimed at defining the possible metadata attributes of components. The metadata attributes of media components (aimed at describing their technical properties) were inspired by the appropriate descriptors of the MPEG-7 standard [Manjunath et al. 2002]. In the presented example they describe the image object's size dimensions (*width* and *height*) as well as its source location.

```xml
<aco:AmaImageComponent name="myImage" layer="Media">
  <aco:MetaInformation>
    <amet:ImageMetaData>
      <amet:source>images/myimage.jpg</amet:source>
      <amet:width>500</amet:width>
      <amet:height>300</amet:height>
    </amet:ImageMetaData>
  </aco:MetaInformation>
</aco:AmaImageComponent>
```

Listing 4.1: Simple media component example

In order to support for form-based user interactions, the component-based document format allows to define Web form elements (such as input fields, select lists, check boxes, etc.) as media components [Hoja 2005]. This can be done either by using a *structured text component* that contains a form description in a specific Web output format (e.g. HTML) or by using a dedicated *XForms component* encapsulating a form description based on the XForms standard [Dubinko 2004]. By separating the data model, the behavior and the presentation aspects of Web forms, XForms allows for the specification of interaction elements in a device independent way.

4.2.2 Content Unit Components

On the second component level, media components are grouped to so-called *content units*. The purpose of such groupings is the explicit combination of media elements that belong together concerning their content[5] and thus should not be handled separately. For example, an image with its appropriate textual description can constitute a content unit. Further predefined content unit types are *audio component with text component* or *collection of media components*, but arbitrary extensions are supported. Note that the definition of such collections of media objects is a key factor of component reuse.

As an example, the code snippet in Listing 4.2 illustrates a simple content unit containing an image and a text component (both from the media component layer). Whereas in this case these subcomponents are "physically" aggregated to a content unit (see the *SubComponents* tag in line 5), note that it is also possible to include subcomponents by reference from another XML document. The latter mechanism allows to efficiently reuse components in different composition scenarios.

Since the media components constituting a content unit belong together, they also have to be presented together in a Web presentation. Therefore, appropriate metadata describing their layout, i.e. their relative spatial arrangement on the generated hypermedia pages is needed. For this purpose content unit components contain additional layout descriptions (as part of their own meta data information). Inspired by the layout manager mechanism of

[5]e.g. in order to represent a given concept of an application domain

```
<aco:AmaImageTextComponent name="myUnit" layer="ContentUnit">
  <aco:MetaInformation>
    ...
  </aco:MetaInformation>
  <aco:SubComponents>
     <aco:AmaTextComponent name="myText" layer="Media">
        ...
     </aco:AmaTextComponent>
     <aco:AmaImageComponent name="myImage" layer="Media">
        ...
     </aco:AmaTextComponent>
  </aco:SubComponents>
</aco:AmaImageTextComponent>
```

Listing 4.2: Simple content unit example

the Java language, these properties describe a size- and client-independent layout allowing to abstract from the exact resolution of the display or the browser's window. A detailed description of these layout descriptors will be given in Section 4.3.2.

4.2.3 Document Components

The uppermost component level contains so-called *document components*. These are meaningful presentation units of a Web application that typically play a certain semantic role, such as a news column, a product presentation, a navigation bar or even a whole Web page. Take for example a document component bearing the semantic role "company product". It could aggregate (and thus be represented by) the content unit from Listing 4.2 that again contains an image component and a text component.

Document components can not only contain content units, they can also aggregate other document components. This aggregation results in an arbitrary deep hierarchy of document components which describes the logical structure of a component-based Web document. The root component (element) of a component-based adaptive Web document has to be always a component from the document component level. Such a top-level document component contains all the information to be shown on the user's display at a particular moment. According to the component hierarchy example depicted in Figure 4.1, Listing 4.3 demonstrates the corresponding aggregation of components[6].

The document components constituting a Web presentation typically portray some meaningful concepts that are represented by (a set of) content units or aggregated document components. Still, the component-based document format does not prescribe to explicitly specify the semantics of document components by unequivocally assigning them to parts (i.e. concepts or attributes) of a specific conceptual model or domain ontology. The component hierarchy of a Web page is specified by component authors in the authoring process which will be described in detail in Chapter 5. There it will be also shown how component-based Web applications presenting information on a well-defined application domain can be systematically developed.

In analogy to content units, the presentation of document components on a Web page is specified by layout properties defining the spatial adjustment of their aggregated document

[6]Similar to content unit components, document components can not only "physically" contain their subcomponents (as aggregated XML subelements), they can also include subcomponents contained in separate XML documents by reference.

```
<aco:AmaCompanyHomepageComponent name="Company" layer="DocumentComponent">
  <aco:SubComponents>
    <aco:AmaProductListComponent name="ProdList" layer="DocumentComponent">
      <aco:SubComponents>
        ...
        <aco:AmaProductComponent name="Product" ...>
          <aco:SubComponents>
             <aco:AmaImageTextComponent ... layer="ContentUnit">
               <aco:SubComponents>
                 <aco:AmaImageComponent name="myImage" layer="Media">
                  ...
                 </aco:AmaImageComponent>
                 <aco:AmaTextComponent name="myText" layer="Media">
                   ...
                 </aco:AmaTextComponent>
               </ac:SubComponents>
             </aco:AmaImageTextComponent>
          </aco:SubComponents>
        </aco:AmaProductComponent>
        ...
      </aco:SubComponents>
    </aco:AmaProductListComponent>
     ...
    <aco:AmaContactComponent name="Contact" layer="DocumentComponent">
     ...
    </aco:AmaContactComponent>
    ...
  </aco:SubComponents>
</aco:AmaCompanyHomepageComponent>
```

Listing 4.3: Document component composition example

components. However, the layout properties of a given component only describe the presentation of its immediate subcomponents which encapsulate their own layout information in a standard component-based way. Thus, in order to provide reuse and configuration, each composite component stores and manages its layout information on its own. The concept of adaptive layout managers will be introduced in detail in Section 4.3.2.

4.2.4 Hyperlink Components

Whereas aggregation relationships between components are expressed on the level of content unit components and document components, navigational relationships between components are defined by so-called *hyperlink components*, each defining an (optionally typed) directed link between two "non-hyperlink" components[7]. Two kinds of of hyperlink components exist: simple hyperlink components and hyperlink list components. While a simple hyperlink component constitutes a directed (and optionally typed) navigational relationship between two components (that act as its *source* and *destination* anchors), a hyperlink list component is a collection of simple links aimed at the easier definition of index-like navigation structures.

For the sake of efficient document reuse, hyperlinks are always defined within the scope

[7]Even though hyperlinks are also considered as components, the document model does not allow to specify hyperlinks between hyperlinks, i.e. the end points of a hyperlink may be only media components, content unit components or document components.

of a document component. The fact that a hyperlink component is specified within the scope of a document component means that at least its source anchor is contained in (the subcomponent hierarchy of) that document component. Consequently, the reuse of that document component in another composition scenario also implies the reuse of its associated hyperlinks.

The specification of hyperlink components is based on the XPath [Berglund et al. 2004] and XPointer [DeRose et al. 2002] standards of the W3C. The source anchor of a hyperlink component is unequivocally described by the identifier of its source component and an optional offset. The source component of a hyperlink component is (if not further specified) either the document component in the scope of which it was defined, or an arbitrary component of its subcomponent hierarchy. Furthermore, an (optional) XPointer expression defining an offset of the source link anchor in that component can be also specified. Typically, this offset is needed in order to define a hyperlink anchor that is assigned only to a fragment of a text component.

The destination of a hyperlink component is either a component in the subcomponent hierarchy of the document component in which it was defined, or an arbitrary component in another component-based Web document. Besides, external link destinations pointing to arbitrary URIs are also allowed, thus facilitating to reference any external content.

Finally, hyperlink components can be optionally assigned a *type* and/or a *class* attribute, as well. The former one allows for the specification of typed navigational relationships between components. However, the possible hyperlink types are not prescribed by the document format and can thus be specified by component authors (e.g. by exploiting existing link type classifications [Casteleyn and De Troyer 2002]). The latter one (class attribute) aims at assigning a presentation class to hyperlinks. Links with different class attributes can be visualized differently (by the definition of appropriate CSS media components attached to their containing components) and thus be used for realizing link adaptation techniques such as link annotation or link disabling (see Section 2.2.3).

The code snippet depicted in Listing 4.4 shows a simple hyperlink component defining a navigational relationship between two components. It was defined in the scope of the document component that was visually shown in Figure 4.1, representing a company homepage. The source anchor of this link is the component called “Introduction” (see the *From* element in line 5), its destination is the component called “Contact”. Since in this case both components are subcomponents of “Company”, its reuse in another composition scenario also implies the reuse of this hyperlink component. Note that in this example no link type was defined.

4.3 Adaptation Support

The component-based document format supports a separation of concerns by distinguishing between different abstraction levels of components as well as their interlinking. Important aspects of a Web presentation, such as content, structure, semantics, navigation, etc. are handled on different component levels, thus enabling a better reuse and configurability of (even parts of) a Web presentation. Furthermore, this level-based model also facilitates the efficient separation of different adaptation targets (i.e. parts or aspects of a Web presentation to be adapted). The following paragraphs mention typical adaptation concerns to be considered on the different component levels.

Adaptation of Media Components: Adaptation on the level of media components pri-

```
<aco:AmaCompanyHomepageComponent name="Company">
  <aco:Hyperlinks>
   ...
    <aco:AmaComponentLinkComponent name="Link_1" layer="Hyperlink">
      <aco:From component="Introduction"/>
      <aco:To component="Contact"/>
    </aco:AmaComponentLinkComponent>
   ...
  </aco:Hyperlinks>
  <aco:SubComponents>
     ...
  </aco:SubComponents>
</aco:AmaCompanyHomepageComponent>
```

Listing 4.4: Link component example

marily concerns media quality and is required to consider various device capabilities or other technical constraints. For instance, in a device independent Web presentation it is necessary to provide different instances of a certain picture (image component) with variable size, color depth or resolution in order to automatically adapt to various display types. Similarly, a video component should be provided with different bit rates to be adjusted to the currently available bandwidth.

Adaptation of Content Unit Components: Adaptation on the level of content units concerns the type and number of the included media components (that actually constitute the structure of a content unit) and can be defined for different purposes. On the one hand, technical properties of client devices can be taken into account. For example, on a company homepage a user with a high performance client could be shown a short video clip of the presented article, while others with low-performance terminals would be presented an image and a textual description of that product. On the other hand, the adaptation of content units may also consider semantic user preferences. Consider again the case of two customers, one of them preferring detailed textual descriptions, the other visual information. While the presentation for the first user might include content units primarily referring to textual objects, the other could be shown multimedia information, respectively. These kinds of adaptation were approved as very profitable e.g. in the TELLIM project [Jörding 1999, Hölldobler 2001] focusing on electronic shopping applications.

Adaptation of Document Components: Adaptation of document components concerns the overall component hierarchy, i.e. the way document components are nested. This results in different variations of component trees (and thus Web page structures) depending on user preferences and client properties. For instance, consider the component hierarchy shown in Figure 4.1 that represents a company homepage. Depending on the interests and previous knowledge of users, the resulting Web page could be generated differently. For internal users working as employees of the company, the document component presenting the company's history should not be inserted in the generated Web page.

Nevertheless, note that the adjustment of document components may also depend on other parameters, e.g. client capabilities. Let us consider the Web portal of a railway company as an example. Whereas the desktop PC version of such a Web site might include numerous parts such as timetable, online-shop, online travel agency, etc., the

WAP version is mostly restricted to a minimal set of services, e.g. merely an online timetable.

Adaptation of Hyperlink Components: Finally, adaptation on the level of hyperlink components implies the adjustment of hyperlink targets as well as navigation structures according to information describing the actual usage context. It might concern the usage of different adaptive navigation techniques, such as the conditional inclusion or annotation of hyperlinks, the offering of navigation alternatives, etc. This adjustment of hyperlink structures can allow a personalized navigation through a component-based Web presentation.

In order to realize these adaptation scenarios, the component model facilitates two generic adaptation mechanisms. First, it is possible to specify context-dependent *adaptation variants* for (different aspects of) components. Second, a facility is provided for describing the *adaptive layout* of components by means of client-independent layout descriptors that can be automatically adapted to different output formats. The rest of this section describes these two mechanisms in more detail and illustrates them by representative examples.

4.3.1 Describing Adaptation Variants

To provide generic adaptation support, components (but also their parameters) may include a number of (conditional) alternatives. As an example, the definition of an image component might include two variants for color and monochrome displays. Similarly, the number, structure, spatial arrangement, and linking of subcomponents within a composite component can also vary depending on the current usage context. The decision, which alternative is selected, is made during document generation (see Section 4.5) according to a *selection method* which is also encapsulated by the component.

Such selection methods are chosen by component developers at authoring time and can represent arbitrarily complex conditional expressions parameterized by context model parameters. These parameters describe the user's actual usage context (e.g. his knowledge, characteristics, preferences, client device, location, etc.) and will be described in more detail in Section 4.5.3. The XML-grammar for selection methods was specified as an XML Schema definition and allows for the declaration of user model parameters, constants, variables, and operators, as well as complex conditional expressions (such as IF-THEN-ELSE or SWITCH-CASE) of arbitrary complexity [Fiala et al. 2003a, Fiala et al. 2003c]. The appropriate reconfiguration (adjustment) of a component's adaptation logic allows to reuse it in different adaptation scenarios.

The example code in Listing 4.5 demonstrates the definition of a media component's variants as well as a corresponding selection method. It is taken from a Web presentation offering different versions of media elements for different end devices (in high quality for desktop computers and low quality for mobile clients), i.e. the appropriate variant is chosen based on the actual client device. The possible variants to be selected are defined in the *Variant* elements (see the lines 21 and 24). Again, the the namespace *aada* references the XML schema definition AmaAdaptation.xsd which specifies the grammar for describing variants and their corresponding selection methods.

The selection method is contained in the *Logic* tag (line 5), which is in this particular example an IF-THEN-ELSE construct. The condition to be evaluated is specified within the *Expr* element. Based on the Polish Notation (PN)[8], it is an arithmetic expression consisting

[8]The PN-based definition of arithmetic expressions allows for their proper validation, as well as evaluation

```
<aco:AmaImageComponent name="myAdaptiveImage">
  <aco:MetaInformation>
    <amet:ImageMetaData>
      <aada:Variants>
        <aada:Logic>
          <aada:If>
            <aada:Expr>
              <aada:Term type="=">
                <aada:UserParam>Device</aada:UserParam>
                <aada:Const>Desktop</aada:Const>
              </aada:Term>
            </aada:Expr>
            <aada:Then>
                <aada:ChooseVariant>HQ_Picture</aada:ChooseVariant>
            </aada:Then>
            <aada:Else>
               <aada:ChooseVariant>LQ_Picture</aada:ChooseVariant>
            </aada:Else>
          </aada:If>
        </aada:Logic>
        <aada:Variant name="HQ_Picture">
            ...
        </aada:Variant>
        <aada:Variant name="LQ_Picture">
            ...
        </aada:Variant>
      </aada:Variants>
    </amet:ImageMetaData>
  </aco:MetaInformation>
</aco:AmaImageComponent>
```

Listing 4.5: Describing adaptive variants

of a simple term (*Term*) that compares the context model parameter *Device* (referred to as `<UserParam>Device</UserParam>`) with the constant "Desktop"[9]. The elements *Then* and *Else* specify either (like in this case) the appropriate variants to be selected if the condition holds or not, or contain another IF-THEN-ELSE or SWITCH-CASE construct. However, the definition of the else-branch is optional. Whenever it is missing, the selection logic describes *conditional inclusion*, i.e. the addressed variant is presented if and only if the condition holds.

Note that such a component containing adaptation variants acts as an adaptive and reusable "Web site building block". Based on its internal adaptation logic, it can automatically adjust (or reconfigure) itself to the current usage context. Furthermore, since both this logic and the corresponding adaptation variants are inherently contained by it, it can be reused as an adaptive unit of composition in different Web documents. As a consequence, it even fulfills the definition of a *self-adaptive software* provided by [Oreizy et al. 1999][10].

While the adaptation logic described in Listing 4.5 is based on a simple conditional expression in the IF-THEN-ELSE style, Listing 4.6 shows another example that uses a SWITCH-

by XSLT stylesheets [Kay 2004].

[9]The children elements of a Term element might be again Term elements, thus allowing to define arbitrarily complex arithmetic expressions.

[10]According to Oreizy et al., "self-adaptive software modifies its own behavior in response to changes in its operation environment" [Oreizy et al. 1999].

CASE construct allowing to easily select from more than two different alternatives. It is taken from an eLearning example and aims at the selection of an appropriate document component based on the expertise level of the actual user (Beginner, Advanced or Expert). Note the optional *Default* element in line 19 aimed at determining the variant to be selected if none of the other cases holds. It can be used to guarantee that the resulting component is not empty.

```
1   <aada:Variants>
2    <aada:Logic>
3      <aada:Switch>
4       <aada:Expr>
5        <aada:Term>
6         <aada:UserParam>Expertise</aada:UserParam>
7        </aada:Term>
8       </aada:Expr>
9       <aada:Cases>
10       <aada:Case value="Beginner">
11         <aada:ChooseVariant>Beginner_Version</aada:ChooseVariant>
12       </aada:Case>
13       <aada:Case value="Advanced">
14         <aada:ChooseVariant>Advanced_Version</aada:ChooseVariant>
15       </aada:Case>
16       <aada:Case value="Expert">
17         <aada:ChooseVariant>Expert_Version</aada:ChooseVariant>
18       </aada:Case>
19       <aada:Default>
20         <aada:ChooseVariant>Beginner_Version</aada:ChooseVariant>
21       </aada:Default>
22      </aada:Cases>
23     </aada:Switch>
24   <aada:Logic>
25  </aada:Variants>
```

Listing 4.6: Describing adaptation variants (Example 2)

As a matter of course, adaptation variants may be defined on all component levels. When adjusting complex document structures to the current usage context the appropriate selection methods are processed recursively, i.e. in a top-down-manner (beginning from the top-level document component). The run-time evaluation process of such selection methods will be described in detail in Section 4.5.

Parameter Substitution While the concept of adaptation variants allows to adjust various component properties to parameters of the actual usage context, in some cases it is also required to include the values of those parameters into the generated Web presentation. The reason for this can be the intention to inform the user about his/her context or the need for a better personalization of the Web application. For this reason the component-based document format allows to utilize *parameter substitution*. As an example, the code snippet shown in Listing 4.7 depicts a text component, the content of which is parameterized by the context parameter *LastName* denoting the surname of the current user. When substituted by the surname of the author of this thesis following text would be created: “You are logged in as Mr/Ms Fiala.”.

Component (variants) combined with selection methods and parameter substitution allow to describe the adaptation behavior of parts of a Web presentation in a declarative component-

```
<aco:AmaTextComponent name="GreetingText" layer="Media">
  <aco:MetaInformation>
    <amet:TextMetaData>
      <amet:value>
       You are logged in as <aada:UserParam>Salutation</aada:UserParam>
       <aada:UserParam>LastName</aada:UserParam>.
      </amet:value>
    </amet:TextMetaData>
  </aco:MetaInformation>
</aco:AmaTextComponent>
```

Listing 4.7: Context parameter substitution example

wise manner. Nevertheless, the complexity of their underlying XML grammar makes it very difficult to manually author such logical expressions and calls for intuitive visual tools facilitating the graphical definition of appropriate XML documents. For this purpose the AMACONTBuilder, a visual authoring tool for component-based Web documents will be presented in Section 5.2.

4.3.2 Describing Adaptive Layout

In order to describe the spatial arrangement (layout) of components, the component-based document format allows to attach XML-based layout descriptions [Fiala et al. 2003a] to them. Inspired by the layout manager mechanism of the Java language (AWT and Swing) and the abstract user interface representations of UIML [Abrams and Helms 2002] or XIML [Puerta and Eisenstein 2002], they describe a client-independent layout allowing to abstract from the exact resolution of the browser's display. Note that layout managers of a given component only describe the presentation of its immediate subcomponents which encapsulate their own layout information in a standard component-based way.

The available layout managers are depicted in Figure 4.2. `OverlayLayout` allows to present components on top of each other. `BoxLayout` lays out multiple components either vertically or horizontally. `BorderLayout` arranges components to fit into five regions: north, south, east, west, and center. It is especially useful to specify the layout of Web presentations containing a header, a footer, sidebars, and a main content area. Finally, `GridTableLayout` enables to lay out components in a grid with a configurable number of columns and rows. Though it can be also realized by nested `BoxLayouts`, it was implemented separately because Web applications often present dynamically retrieved sets of data in a tabular way.

The code snippet in Listing 4.8 depicts a possible layout description of the content unit from Listing 4.2 based on the layout manager `BoxLayout`. The contained image component (aligned right) and the text component (aligned left) are arranged above each other, taking 30 and 70 percent of the available vertical space. Note that the *alay* namespace references the XML schema definition AmaLayout.xsd which specifies the possible layout managers and their specific attributes.

Layout managers are formalized as XML tags with specific attributes [Fiala et al. 2004a]. Two kinds of attributes exist: *layout attributes* and *subcomponent attributes*. Layout attributes declare properties concerning the overall layout and are defined in the corresponding layout tags. As an example the `axis` attribute of `BoxLayout` (see line 5 in Listing 4.8) determines whether it is laid out horizontally or vertically. Subcomponent attributes describe

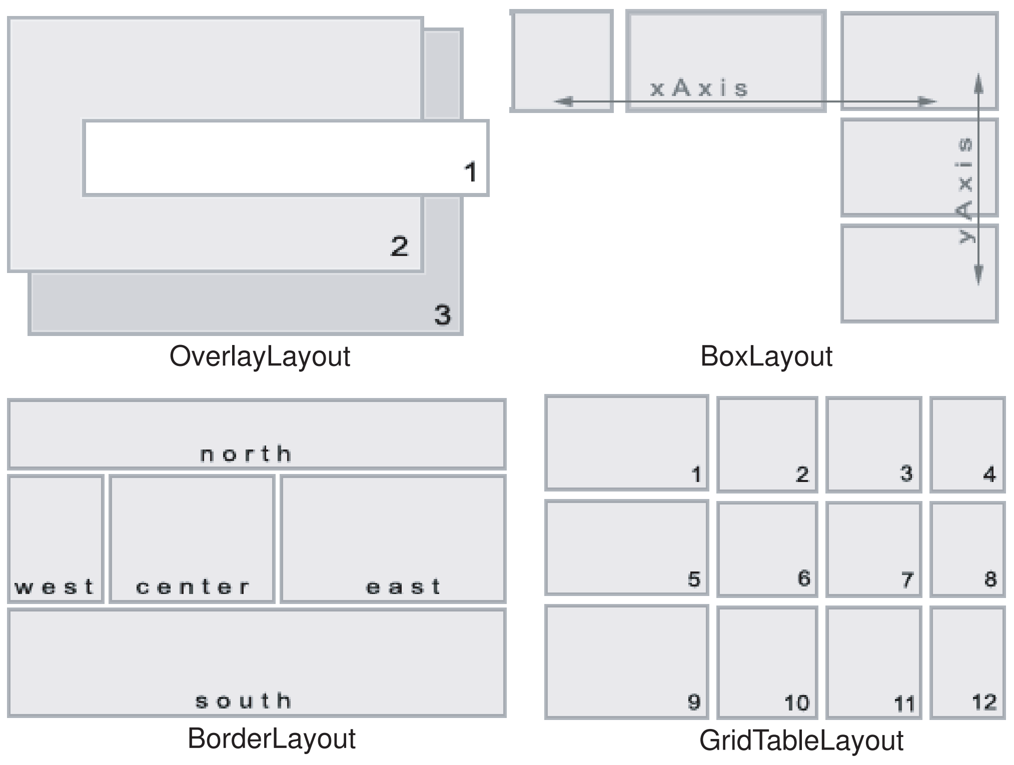

Figure 4.2: Abstract layout managers

how each referenced subcomponent has to be arranged in its surrounding layout. For instance, the `align` attribute of `myImage` declares it to be right-justified. Table 4.1 summarizes the possible attributes of the layout manager `BoxLayout` by describing their names, role, usage (required or optional), and possible values.

Layout Attributes	Meaning	Usage	Values
`axis`	Orientation of the BoxLayout	req.	xAxis|yAxis
`space`	Space between subcomponents	opt.	int
`width`	Width of the whole layout	opt.	string
`height`	Height of the whole layout	opt.	string
`border`	Width of border between subcomponents	opt.	int
Subcomponent Attributes	Meaning	Usage	Values
`align`	Horizontal alignment of subcomp.	opt.	left|center|right
`valign`	Vertical alignment of subcomp.	opt.	top|center|bottom
`ratio`	Space taken by subcomponent	opt.	percentage
`wml_visible`	Should be shown on same WML card?	opt.	boolean
`wml_desc`	Link description for WML	opt.	string

Table 4.1: BoxLayout attributes [Fiala et al. 2004a]

```
<aco:AmaImageTextComponent name="myUnit" layer="ContentUnit">
  <aco:MetaInformation>
    <amet:LayoutProperties>
      <alay:LayoutManager>
        <alay:BoxLayout axis="yAxis" border="1">
          <alay:ComponentRef ratio="30%" align="right">
              myImage
          </alay:ComponentRef>
          <alay:ComponentRef ratio="70%" align="left">
              myText
          </alay:ComponentRef>
        </alay:BoxLayout>
      </alay:LayoutManager>
    </amet:LayoutProperties>
  </aco:MetaInformation>
  <aco:SubComponents>
    ....
  </aco:SubComponents>
</aco:AmaImageTextComponent>
```

Listing 4.8: Layout manager example

Even though most attributes are device independent, two platform-dependent attributes were also added in order to consider the specific card-based structure of WML presentations. Note the optional attribute `wml_visible` that determines whether in a WML presentation the given subcomponent should be shown on the same card. If not, it is put onto a separate card that is accessible by an automatically generated hyperlink, the text of which is defined in `wml_description`. This mechanism of content separation is used since the displays of WAP-capable mobile phones are very small.

The exact rendering of media objects is done at run time by XSLT stylesheets that transform components with such abstract layout properties to Web document fragments in a specific output format (see Section 4.5). A number of stylesheets for converting those descriptions to formats such as XHTML (and its different modules), cHTML, WML, etc. have been developed [Fiala et al. 2003a, Hinz et al. 2004].

While adaptive layout managers support the automatic adaptation of abstract layout descriptions to different output formats, component authors can also use them in combination with adaptation variants, thus being able to specify even more precise layout adaptations that are explicitly parameterized by the current usage context. For example, while the presentation of a content unit containing a list of images could be realized as a GridTableLayout on a browser with sufficient horizontal resolution, another client device with a small display width should render it according to a vertical BoxLayout. The code snippet in Listing 4.9 depicts such combined layout adaptation definition. The appropriate layout manager is selected according to the horizontal resolution (denoted by the context parameter *InnerSizeX*) of the current browser window. Thus, the layout of a document component can be adapted independently of its other aspects, such as its subcomponent, hyperlinks, etc.

Again, the complexity of the XML code shown in Listing 4.8 and Listing 4.9 makes it obvious that the manual creation of layout manager descriptions with a text or XML editor might be a cumbersome task. For the visual authoring of adaptable layouts Section 5.2.4 will introduce the Layout Editor module of the authoring tool AMACONTBuilder.

```
<alay:LayoutManager>
  <alay:Variants>
    <aada:Logic>
      <aada:If>
        <aada:Expr>
          <aada:Term type="gt">
             <aada:UserParam>InnerSizeX</aada:UserParam>
             <aada:Const>600</aada:Const>
          </aada:Term>
        </aada:Expr>
        <aada:Then>
            <aada:ChooseVariant>GridTableLayout_Variant</aada:ChooseVariant>
        </aada:Then>
        <aada:Else>
            <aada:ChooseVariant>BoxLayout_Variant</aada:ChooseVariant>
        </aada:Else>
      </aada:If>
    </aada:Logic>
    ...
  </alay:Variants>
  <aada:Variant name="GridTableLayout_Variant">
      ...
  </aada:Variant>
  <aada:Variant name="BoxLayout_Variant">
      ...
  </aada:Variant>
</alay:LayoutManager>
```

Listing 4.9: Combined context dependent layout adaptation

4.4 Document Component Templates

The document components introduced above are static, i.e. they represent a concrete piece of (adaptable) Web content, such as a specific instance of an image (as a media component instance with optional quality alternatives) or a chapter in an eLearning course (as a document component instance). Still, in order to provide support for data-driven Web applications, like online-shops, product presentations, e-galleries, etc., there is a need for components that are created from dynamic data sources on-the-fly.

For this purpose so-called document component templates have been introduced. These are component skeletons (i.e. component instances containing placeholders) that declare the structural, behavioral and layout aspects of components independent of their actual content [Fiala et al. 2004b]. At run-time, component templates are extended (i.e. filled) with content that is dynamically queried (retrieved) from a data source. Therefore, they are associated with a query that can be parametrized by arbitrary request and/or context model parameters. As an example, the XML-code in Listing 4.10 describes a simple media component template[11]:

The namespace *t_aco* dictates that the component acts as a component template. The query associated with it is described in the *query* attribute of its starting tag. In this particular case this is an SQL expression querying a table of a relational database that contains

[11] As a matter of course, the same mechanism is applicable for content unit component templates, document component templates, and hyperlink component templates.

```
<t_aco:AmaImageComponent name="productimage" type="template"
                         query="SELECT source, width, height
                                FROM productimages
                                WHERE ID=substitute(id)">
  <aco:MetaInformation>
    <amet:ImageMetaData>
      <amet:source><t_temp:query field="source"/></amet:source>
      <amet:width><t_temp:query field="width"/></amet:width>
      <amet:height><t_temp:query field="height"/></amet:height>
    </amet:ImageMetaData>
  </aco:MetaInformation>
</t_aco:AmaImageComponent>
```

Listing 4.10: Simple component template example

images of a company's products described by appropriate metadata[12]. The expression *substitute(id)* references the *id* request parameter of the actual HTTP request. The values from the corresponding result set are referred to as `<t_temp:query field="myname"/>`, where *myname* is the name of a given field. As an example, the resulting media component's *source* attribute is substituted by the value of the database field *picturesource*.

While in this example all metadata attributes of the image component are dynamically retrieved, note that it is also possible to define selected attributes as constants so that they remain unchanged for all instantiations. Furthermore, it is also possible to parameterize a template's query by arbitrary context model attributes. In such a case these parameters are substituted by their corresponding values before the query is executed, i.e. the data to be inserted is queried in a personalized way.

The above example describes a single media component template, the actual content of which is delivered by a dynamic data source. Still, in a data-driven Web application it is not only required to dynamically retrieve single content (component) elements, but also component sets, such as all books of a given author (in an electronic book store), or all employees of a department (in an institutional Web site). For such cases the component-based document format allows to define so-called *iterative component templates*. Again, a simple example of a content unit containing a dynamic set of image components is depicted in Listing 4.11.

The content unit component template defined in this example contains (as its subcomponent) a dynamically iterated image component (see the *iterate* attribute in line 16). Consequently, this media component is iterated (repeated) according to the size of the result set delivered by the template's query (line 3) so that each iteration is parameterized by the corresponding result. To ensure that the resulting image components have unequivocal *name* attributes the *idField* attribute denoting a unique identifier field in the query's result set is used (see line 1). It dictates that for each repetition (iteration) the name attribute of the iterated image component is complemented with a unique postfix (in this case the 'id' field of the query). This explicit definition of the result set's unique identifier field is necessary since the component-based document format is not by definition associated with a given underlying data model. On the other hand, the flexibility of the template mechanism allows to refer to arbitrary data sources, and even to different ones within the same component-based

[12]The concept of component templates was realized for SQL-based queries on relational databases but can be easily extended for other data sources such as XML or RDF databases in a straightforward manner. In Section 5.3.2 it will be shown how component instances can be automatically generated based on RDF data.

```
<t_aco:AmaListComponent name="productimagelist"
                        type="iterativeTemplate"
                        query="SELECT id,source,width,height FROM
                               productimages"
                        idField="id">
   ...
     <alay:LayoutManager>
       <alay:BoxLayout axis="yAxis">
         <t_alay:ComponentRef iterate="yes">
              picture
         </t_alay:ComponentRef>
       </alay:BoxLayout>
     </alay:LayoutManager>
   ...
  <aco:SubComponents>
      <t_aco:AmaImageComponent name="productimage" iterate="yes">
        <aco:MetaInformation>
          <amet:source><t_temp:query field="source"/></amet:source>
          <amet:width><t_temp:query field="width"/></amet:width>
          <amet:height><t_temp:query field="height"/></amet:height>
        </aco:MetaInformation>
      </t_aco:AmaImageComponent>
  </aco:SubComponents>
</t_aco:AmaListComponent>
```

Listing 4.11: Iterative component template example

document.

As the resulting content unit contains a set of image components, their spatial arrangement has to be specified, as well. However, since the number of these subcomponents is not known at authoring time, only the layout managers *BoxLayout* (with an initially undefined number of cells) and *GridTableLayout* (with only one predefined dimension) are allowed and the missing dimensions have to be automatically computed at run time when evaluating the template's query. In the particular example shown in Listing 4.11 a vertical *BoxLayout* is used.

At run time (see Section 4.5), component templates are dynamically filled with content according to their queries as well as the actual state of the corresponding request parameters. Since component templates might aggregate other component templates, this evaluation process is performed recursively and results in dynamically generated component instances, i.e. the "placeholders" in the original templates (component skeletons) are substituted by the actual query's specific results. Thus, after being evaluated, component templates can be treated in the same way as "conventional" static components. Furthermore, while the examples in Listings 4.10 and 4.11 contain select queries, it is also possible to define update queries in component templates. In this case these queries can be used to add (or manipulate) data to (in) a database.

Component templates provide an effective means for the creation of data-driven component-based Web presentations. For their intuitive creation and manipulation a visual authoring tool called the AMACONTBuilder will be introduced in Section 5.2.

4.5 Document Generation

The component-based document format allows to compose adaptive Web documents by creating, configuring, aggregating, and interlinking reusable components (or component templates) on different abstraction levels. When requested by a particular user, such document structures have to be automatically adjusted to his current usage context and delivered to his end device in an appropriate Web output format. For this purpose a modular document generation architecture was developed [Fiala et al. 2003a, Hinz and Fiala 2004, Hinz and Fiala 2005].

The document generation architecture serves several purposes: 1) the automatic translation of component-based documents to Web presentations according to the actual usage context, 2) the storage of this context information, 3) as well as its continual updating based on user's navigation and interaction history. While not being the central focus of this work, the rest of this section describes this functionality in more detail[13].

4.5.1 Pipeline-based Document Generation

As illustrated in the lower part of Figure 4.3, the process of document (presentation) generation is based on a stepwise pipeline concept[14]. According to the component-based document format introduced above, its inputs are complex document component instances or document component templates. They are retrieved from a component repository (or another source aimed at dynamically generating components) according to a user request that is optionally parameterized by a number of HTTP request parameters. Still unadapted, they encapsulate all variants concerning their content, layout, structure, and interlinking.

In the document generation pipeline document components are processed by a series of transformations. Each transformation deals with a given application (or adaptation) concern and produces output for the next transformation step until a final Web presentation is generated. Note that the possibility to use such a staged architecture is a natural consequence of the clean separation of concerns, a basic design principle of the component-based document format. While the modularity of the document generation pipeline allows for different transformer configurations, Figure 4.3 depicts a typical one.

First, possible component references (to components in other XML documents) are resolved and hierarchical component structures are created. Second, whenever the processed documents contain component templates, these are filled with instance data that is dynamically retrieved by the on-the-fly execution of their appropriate queries. Subsequently, the resulting component instances are subdued to a number of adaptation transformers. Parameterized by the current state of the context model (see Section 4.5.2), each of them considers a certain adaptation aspect by the selection, configuration, or device-specific rendering of component variants.

In Section 4.3 two mechanisms for specifying adaptation were mentioned: one for describing adaptation variants and another one for describing adaptive layout. Consequently, the adaptation of document instances is also performed in two main stages. First, a transformer aimed at processing components containing adaptation variants is invoked. It handles all appropriate selection methods in a recursive (top-down) manner and keeps only the selected

[13]The document generation architecture, its context modeling framework, as well as the investigation of its performance issues is a primary research focus and contribution of Michael Hinz. This section describes these topics as detailed as required to understand the overall context of the work presented in this thesis. For more information we refer to the corresponding publications.

[14]Note that this stepwise pipeline concept corresponds to the architectural style "staged architectures", proposed by Aßmann [Aßmann 2005] for active documents (see Section 3.2.9)

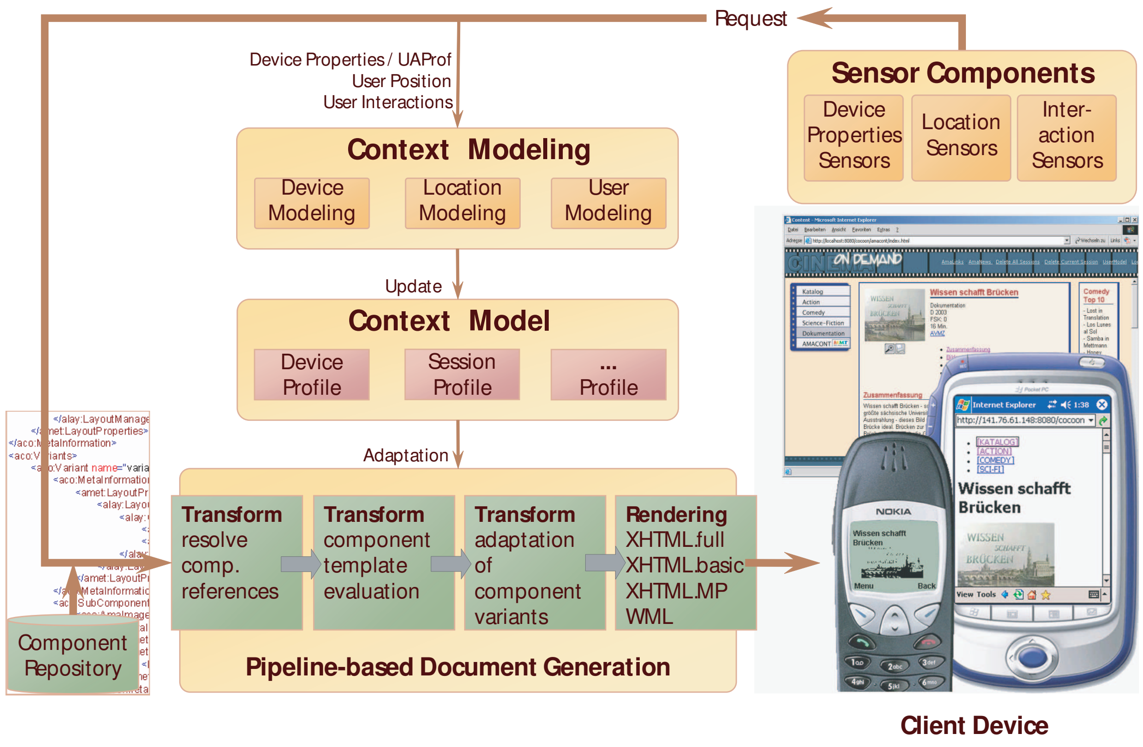

Figure 4.3: Overview of the document generation architecture

content variants in the processed XML stream, i.e. all other variants are omitted from the processed document. The result of this transformation step is a final component hierarchy without adaptation variants.

After all component variants and conditions are evaluated, the last transformer creates a Web page in an output format supported by the user's browser device. For instance, a BoxLayout in XHTML is realized by means of a table (and its specific attributes) with either one column or one row. However, not all layout managers can be visualized properly on all devices. As an example, since PDAs or WAP phones have very small displays, a horizontal BoxLayout is automatically converted to a vertical arrangement of subcomponents on those devices. This kind of adaptation is performed by the system, i.e. no explicit specification from the author is needed. The document generation pipeline was realized based on the Cocoon publishing framework [Ziegeler and Langham 2002].

4.5.2 The Context Model

The information describing the actual user and his usage context is stored in the extensible context model (see the middle part of Figure 4.3). It is represented in XML and consists of a set of context profiles[15], each maintaining up-to-date data on a given user/context feature. The structure of context profiles relies on CC/PP (Composite Capability/Preference Profiles [Klyne et al. 2003]), an RDF grammar for describing device capabilities and user preferences in a standardized way. Still, while the original CC/PP specification defines a

[15]While some literature uses the notion of a *user profile* or *context profile* for describing usage data that is static with regard to a Web session, the profiles of the context model may contain both static and dynamic information. The update process of context information will be described in Section 4.5.3.

profile as a flat two-level hierarchy of *components* and their *attributes* that are represented as (sets of) literals, it is allowed to utilize arbitrary deep XML structures for describing component attributes. As a general grammar, CC/PP makes no specific assumptions on concrete context characteristics. Therefore, for each profile a corresponding schema (e.g. expressed by an XML Schema or RDFS declaration) has to be provided.

Since the component-based document model is not bound to a specific application domain, the context information used in a given adaptation scenario is also typically very application specific. Therefore, the context model can be arbitrarily extended by the introduction of new context profiles, each specified by a given schema definition. Still, there are also some predefined profiles that can be generally used for a broad range of ubiquitous and context-aware adaptive Web presentations. The following list gives a representative overview of them.

- The **Identification Profile** provides basic personal information about the user, such as his name, login ID, email address, age, etc. This data is typically static with regard to a given browsing session and can be acquired by explicitly asking the user e.g. in the beginning of a Web session. For example, the login ID of the user is determined when he starts his browsing session. As a matter of course, the usage of this profile for adaptation purposes is optional.

- The **Device Profile** contains technical information describing the user's client device. It is represented on the basis of the WAP User Agent Profile (UAProf [Wir 2001]), a common CC/PP vocabulary aimed at describing WAP devices. However, to support a broader range of mobile devices (e.g. PDAs) specific extensions of UAProf have been made [Hinz et al. 2004].

  While parts of the Device Profile (such as the device's hardware platform) are static with regard to a Web session, note that it contains also parameters that can change according to user interactions. For example, when the user resizes his/her browser window, the appropriate context parameter should be updated, respectively. The acquisition and update process of context model parameters according to user interactions will be described in Section 4.5.3.

- The **Location Profile** stores the physical (geographical) location of the user. This information is described both by means of physical coordinates as well as semantic location information describing the semantic meaning of a location (e.g. the Multimedia Chair of the Dresden University of Technology) and is retrieved from a landmark store. For more information on the appropriate location context descriptors the reader is referred to [Hinz and Fiala 2005].

- The **Session Profile** collects information on the actual user's browsing and interaction history within a component-based Web presentation. Its main goal is to track user access information, such as the number of the user's previous sessions in a given application, the unique identifiers of the document components he already visited as well as the interactions he performed on selected media components (e.g. starting a video, enlarging an image component, etc.). This profile is automatically updated at each user request and is an important basis for the utilization of user modeling mechanisms facilitating dynamic adaptation (adaptivity).

The example code in Listing 4.12 illustrates a CC/PP-based context model description. Note that while in this case only a small excerpt from the Identification Profile and the Device Profile is shown, the actual context description might contain an arbitrary number

```
<ContextModel>
 ...
 <IdentificationProfile>
  <ccpp:component>
   <UserData>
    <ID>fiala</ID>
    <Title>Mr.</Title>
    <Firstname>Zoltan</Firstname>
    <Lastname>Fiala</Lastname>
    <Age>29</Age>
   </UserData>
  </ccpp:component>
  ...
 <IdentificationProfile>
 ...
 <DeviceProfile>
  <ccpp:component>
   <HardwarePlatform>
    <ColorCapable>Yes</ColorCapable>
    <TextInputCapable>Yes</TextInputCapable>
    <ImageCapable>Yes</ImageCapable>
     ...
   </HardwarePlatform>
  </ccpp:component>
  ...
  <ccpp:component>
   <SoftwarePlatform>
     <CcppAccept-Language>de</CcppAccept-Language>
     ...
   </SoftwarePlatform>
  </ccpp:component>
   ...
 </DeviceProfile>
 ...
</ContextModel>
```

Listing 4.12: Extract from an example context model

of profiles. For more detailed information on the utilized context model and its profiles the reader is referred to [Hinz et al. 2004, Hinz and Fiala 2005].

4.5.3 Support for Context Modeling and Interaction Processing

The pipeline-based document generation process supports adaptability (or static adaptation) by adjusting complex document structures to available information describing the actual usage context. However, to facilitate adaptivity (i.e. dynamic adaptation based on user's browsing behavior), there is a need for additional mechanisms. First, user interactions (or other external events, such as bandwidth fluctuations) have to be acquired that might influence specific parts of the usage context and thus lead to a dynamic reconsideration of the presentation. Second, the context model has to be updated based on this acquired information, respectively.

To facilitate these mechanisms the document generation architecture provides a context modeling framework that allows to utilize an extensible set of sensor components and context modeling components [Hinz et al. 2006] (see the upper part of Figure 4.3). Whereas the sensor components aim at acquiring user interactions (such as following links, or interacting document with components) device capabilities (e.g. information on the user's client device type or current browser window size) or other kinds of context information (e.g. location), the context modeling components perform updates of the context model according to this information on the server side. As a matter of course, the usage of a given context modeling (or user modeling) strategy is typically strongly dependent on the given application scenario. Still, the document generation architecture provides a number of generic facilities that can be efficiently used for acquiring and processing interactions in a broad range of component-based adaptive Web presentations.

4.5.3.1 Acquiring User Access Information and Device Capabilities

Whenever a user follows a link or submits a form in the generated Web presentation, the request sent to the server contains standard HTTP request information (in form of request parameters). Furthermore, the document generation architecture allows to automatically extend this information by more detailed data gathered both on the user's interactions as well as his actual context (e.g. device capabilities and location information).

In order to gather user access in a component-based adaptive Web presentation, component authors can configure selected components as "observed" by setting their *watched* attribute to *true* at authoring time. Whenever such an observed component is included in the resulting Web presentation (i.e. if it is not omitted by a certain selection method), the generated Web page is automatically enriched with sensor components based on client side code fragments aimed at tracking user interactions on those components. The fact that a component was presented on the user's browser is tracked as a "trivial" interaction with that component (meaning the user saw that component). However, some media components allow for more "complex" interactions (such as starting a video, enlarging an image, etc.).

When requesting another component-based Web document, the identifiers of the observed components as well as the interactions performed on them are automatically sent to the server side where the session profile is updated, respectively. Note that the utilization of such user access information for adaptation purposes is a basic facility in adaptive hypermedia and Web-based systems and was successfully applied in different application scenarios [Jörding 1999, De Bra et al. 2002, Casteleyn 2005].

The acquisition of client device capabilities happens in a similar way as the acquisition of user interactions, i.e. by the insertion of device capability sensors in form of client-side code fragments. Again, the appropriate code fragments are automatically inserted into the generated Web presentation and collect up-to-date information about the user's browser device (such as its type, supported media types, and plug-ins, current browser window size, etc.) and location. The gathered information is encoded in a UAProf like representation and integrated in the HTTP request by a client/server communication component for processing that information on the server. For more information on the technical realization of these mechanisms the reader is referred to [Hinz and Fiala 2005].

4.5.3.2 Context Modeling

As described above, the HTTP requests originating from the client contain besides standard request parameters additional information describing user interactions, device characteristics, etc. Before the next hypermedia page is generated, this information has to be processed and the context model updated, respectively. For this purpose the document generation architecture allows utilizing an extensible set of context modeling components. Providing a well-defined interface for accessing the request parameters (incl. the information delivered by the sensor components) and manipulating the context model, these components can be programmed or configured to perform arbitrary context model updates based on the newly acquired information. Furthermore, they can also be used to implement server-side application logic, such as performing a database query, invoking a Web service, etc.

The currently existing repertoire of context modeling components comprises modules for device modeling (aimed at updating the device profile), location modeling (for storing the exact geographical position of the user in the context model) as well as a number of solutions for user modeling [Hinz and Fiala 2005] that can be configured and activated depending on the current application scenario. As an example, in a prototypical component-based Web presentation for an online video store a user modeling algorithm based on the incremental learning algorithm CDL4 [Shen 1996, Hinz et al. 2004] was successfully utilized, allowing to predict user's preferences based on their interactions with media objects. Nevertheless, in order to support a broad number of context modeling mechanisms, it is also possible to implement and easily integrate new context modeling components.

When the process of context modeling was performed a new component-based document is retrieved and put through the presentation generation pipeline, respectively. This might be an already existing component-based Web document (or document template) from the component repository, but it is also possible to redirect the request to a backend application that dynamically generates such a document. Thus, a component-based Web presentation can be also effectively used as the adaptive front-end for a more complex back-end application. Such a scenario will be described in more detail in Section 5.3.

4.6 Summary and Model Benefits

This chapter a presented concern-oriented component model for dynamic adaptive Web documents. The concept of declarative document components was introduced and a corresponding XML-based component-description language was presented. The different component layers addressing both different application concerns and adaptation facilities were explained by a number of examples. Furthermore, a pipeline-based document generation architecture for the on-the-fly publishing of component-based Web presentations was also briefly described.

Before turning to the authoring process of component-based adaptive Web presentations and its tool support in Chapter 5, the rest of this section summarizes selected important aspects and characteristics of the document model. First, Section 4.6.1 describes its main analogies and differences to the already presented hypermedia reference models Dexter and AHAM. Then, Sections 4.6.2 to 4.6.5 recapitulate a selection of its main benefits, among them component reuse and configurability, adaptation support, extensibility, as well as support for Web annotations[16].

[16]In Section 3.2.10, a number of requirements towards component models for adaptive Web applications were mentioned, which also served as the basis for the design of our own model. Note, however, that this section recapitulates only a selection of those aspects. Other issues (e.g. the separation of concerns, device independence, or template support) were already in detail discussed throughout this chapter and thus do not need further emphasis.

4.6.1 The Component Model vs. Dexter and AHAM

In Chapter 2, the two reference models Dexter and AHAM were introduced to capture the main characteristics of (adaptive) hypermedia systems. Since the concern-oriented component model presented in this chapter is an approach aimed at implementing adaptive Web applications, this section summarizes its main analogies and differences to those reference models.

First, we note that the concept of document components corresponds to the component concept of Dexter. Nevertheless, by the introduction of different component layers, our model provides a more explicit typing as well as a fine-grained consideration of different concerns involved in a Web or hypermedia application. Furthermore, whereas Dexter keeps the Within-Component layer unspecified, the media component layer of our model specifies in detail the supported atomic content elements and the specification of their attributes. Moreover, while Dexter considers components to be static with regard to their content (and is thus mainly applicable for static hypermedia presentations), the concept of component templates (in our model) supports even data-intensive applications.

Another similarity of our model to Dexter (and AHAM) is the consideration of hyperlinks as components. However, in contrast to both reference models, hyperlinks with several end points and/or bidirectional references are not supported. The reason for these restrictions is the goal to explicitly consider the specific characteristics of the World Wide Web as a hypermedia system (see Section 2.1.3).

The concept of abstract layout descriptions (layout managers) corresponds to Dexter's presentation specifications. Yet, instead of being separated from the actual components (e.g. as part of a specific Run-Time layer), it is one of their inherent properties.

Finally, similar to the Teaching Model of AHAM, the concern-oriented component model also sets a great store by supporting adaptation. However, whereas the adaptation rules of AHAM are stored as separate entities, the concern-oriented component model considers them as parts of components allowing for their adaptation in a component-based manner. Moreover, whereas AHAM (and its reference implementation AHA!) mainly focus on the conditional inclusion/exclusion or the annotation of components, our model allows to implement a broader range of adaptation techniques. As will be described in Section 4.6.4, the combination of component templates, abstract layout descriptions, and their conditional variants allows to implement most of the adaptation techniques introduced by Brusilovsky (see again Section 2.2).

4.6.2 Support for Component Reuse and Configurability

A very important aspect of effectively engineering Web sites is the reuse of formerly developed artefacts. However, the current coarse-grained document-oriented implementation model of the Web makes it difficult for authors to identify and efficiently reuse configurable content fragments of a Web presentation [Gaedke et al. 2000]. The component-based document model presented in this chapter tries to solve this problem by defining fine-granular Web components for creating Web applications. By encapsulating their properties and functionality in a component-wise manner, they can be easily reconfigured and thus be utilized in different application scenarios.

The level-based structure of the document model supports the effective reuse of components of a certain level in components on higher levels. For instance, an adaptive image component being capable to adjust itself to the current screen size can be reused as a "black-box" in different content units. Similarly, a dynamic content unit arranging a list of pictures in a tabular way might be easily reused by being filled with different image components. Furthermore, the recursive nature of document components facilitates the reuse of component trees of arbitrary depth and granularity. Since each component encapsulates its structure (subcomponents and links), presentation (layout managers), and adaptation behavior (in form of selection methods) in an inherent way, all this functionality is automatically "carried with" when applying the component in another composition scenario.

As a matter of course, a crucial issue of the efficient reuse of Web implementation artefacts is their ease of (re)configuration. Therefore, the description language of document components provides a clear separation of independent concerns (content, structure, presentation, navigation, adaptation) by utilizing separate descriptors (metadata interfaces) for configuring all these different issues. Whereas e.g. in an HTML document all these aspects are "interwoven" to a coarse-grained file-based resource, the component-based document model enables component authors to configure or manipulate them separately. As an example, a component developer might define two different layout variants for a component (e.g. one for a desktop PC and another one for a handheld device) without influencing its content or interlinking.

4.6.3 Extensibility Support

As described in Section 4.2, the component-oriented document model rests upon a level-based architecture. That is to say, all possible components are derived from the basic (abstract) component types *media component*, *content unit component*, *document component*, and *hyperlink component*. Each of these abstract types has a number of predefined concrete derived types (e.g. in the case of media components these are image component, text component, structured text component, audio component, etc.). However, it is also easily possible to introduce new component types on each abstraction level.

As an example, the introduction of a new media component type requires the extension of the schema definition AmaComponent.xsd with a new component type (that inherits from the abstract type AmaMediaComponent based on the substitution group mechanism of XML Schema [Fallside and Walmsley 2004]) and the specification of its metadata properties in the schema definition AmaMetaData.xsd. Furthermore, the existing layout stylesheets for appropriately transforming the new media component descriptions into a given Web output format (e.g. HTML, cHTML, WML) have to be adjusted, accordingly. All other functionality (e.g. the ability to define adaptation variants, layouts, hyperlinks) is defined for the abstract component definitions, i.e. it can be utilized by each instance of the new component type, as well.

Besides component types, it is also easy to extend the component description languages with new layout managers and adaptation logics. For instance, the definition of a new layout manager type implies the extension of the schema definition AmaLayout.xsd by declaring its *layout attributes* and *subcomponent attributes* (see Section 4.3.2). Furthermore, the appropriate stylesheets for transforming abstract layout descriptions to a given output format have to be extended. However, this concerns only the rendering of format specific container elements (e.g. tables, lists, etc.) aimed at the presentation of the actual content represented by the embedded media objects, the rendering of media components is not depending of the actual layout manager.

4.6.4 Adaptation Support

As discussed in Section 4.3, the component-based document format supports two basic adaptation facilities: the possibility to define component alternatives (on different component levels), as well as to describe the layout of components in an abstract and implementation independent way. Though being simple adaptation mechanisms, note that these facilities can be effectively utilized to realize a number of adaptation techniques. According to the already mentioned taxonomies of Brusilovsky [Brusilovsky 1996, Brusilovsky 2001] as well as Paterno and Mancini [Paterno and Mancini 1999], the following lists comprise the supported content-level, link-level, and presentation-level adaptation facilities and their component-based realization:

Content-Level Adaptation

- **Support for page variants and fragment variants** by the definition of content alternatives on different abstraction levels
- **Conditional inclusion of fragments** by defining conditional variants for sub-components
- **Adaptation of media content** by offering media components with quality alternatives
- **Adaptation of modality** by providing content units with varying types of included media elements

Link-level Adaptation

- **Link Disabling** based on the conditional inclusion/exclusion of hyperlink components
- **Link Removal** by the conditional inclusion of both hyperlink components and the media components serving as their anchors
- **Link Annotation** based on the conditional assignment of style classes to hyperlink components
- **Link Hiding** based on the conditional assignment of style classes to hyperlink components
- **Link Generation** by the dynamic inclusion of hyperlink components based on context-dependent component templates
- **Link Sorting** by the dynamic inclusion and ordering of hyperlink components based on context-dependent component templates

Presentation-Level Adaptation

- **Support for layout variants** based on conditional alternative layout manager definitions
- **Adaptive styling of pages and page fragments** by utilizing alternative CSS components

As can be seen, the adaptation support provided by our model go far beyond the capabilities of related component-based and document-oriented approaches (see Section 3.2.10). Furthermore, as was described in Section 4.5, it also supports the automatic generation of Web presentations in different output formats, among them (X)HTML, cHTML, or WML.

4.6.5 Support for Web Annotations

Annotating Web pages is an important aspect of asynchronous communication on the WWW. Authors and visitors of Web applications attach notes to certain pieces of Web content in order to remember things better, to communicate with each other or to manage information more intelligently. Typical scenarios of Web annotations are Web-based learning systems allowing students and tutors to communicate, distributed authoring environments supporting the concurrent editing of content, and even product presentations, where users give feedback to the system via personal remarks.

Existing annotation systems, like ComMentor [Röscheisen et al. 1995], CritLink [Yee 1998], CoNote [Davis and Huttenlocher 1995], YAWAS [Denoue 1999], iMarkup [@imarkup], Annotator [Ovsiannikov et al. 2000], WebWise [Grønbæk et al. 1999], etc. mainly focus on annotating static Web pages which do not change their content, structure, and layout temporally. In general, an annotation is clearly defined by the URL of the Web page containing it and some anchor points within that page [Denoue and Vignollet 2000]. A significant disadvantage of these tools is the lacking support for separation of content, structure, and layout. Annotations are not attached to the contents itself, rather to the Web pages containing them. Notes go lost, when the same content is presented on a different page, in a new context or with a changed layout. Thus, this mechanism is not suitable for dynamic Web documents generated at runtime for which no persistent state and no constant layout exists.

Abstracting from the coarse-grained model of current Web implementation languages, the proposed document model and its document generation architecture support the creation of annotations to reusable components. That is to say, the smallest objects to be annotated are not whole Web pages but document fragments, i.e. media components, content units or document components. When a user marks a Web page generated on the basis of Web components, his annotations can be reversely mapped to the fine-granular content components and stored as specific component metadata. This reverse mapping is supported by automatically enriching the generated Web page source code (e.g. HTML) by appropriate semantic markup and client-side Java script code fragments. Annotation anchors are unequivocally located by the identifier of the corresponding component and some offset coordinates within it. When the same component is used in another presentation - possibly on a different client or in a different context - the attached annotations can be reused, too.

While not being a central issue of this thesis, we note that concept of attaching fine-granular annotations to reusable, adaptive components was prototypically implemented in an annotation system called DynamicMarks. For more detailed information on it the reader is referred to [Fiala and Meissner 2003].