Chapter 5

The Authoring Process and its Tool Support

"We shall not fail or falter; we shall not weaken or tire. Give us the tools and we will finish the job."[1]

The document model introduced in Chapter 4 allows to create adaptive Web presentations from reusable implementation artefacts (components) that encapsulate adaptable content, navigation, and layout on different abstraction levels. Still, even though component-based reuse is crucial to Web Engineering, the development of adaptive Web applications of such components is typically a complex task that requires systematic process models and appropriate tool support [Fiala et al. 2004b]. Therefore, this chapter deals with the authoring process of component-based adaptive Web presentations and its tool support[2].

Independent of a given application domain, the proposed document model supports different Web application scenarios. Consequently, the resulting authoring process should not be bound to a fixed process model or workflow, rather adjusted to the specific requirements of the targeted application area. These requirements may vary depending on various factors, such as the application's type (e.g. static adaptive hypermedia presentation vs. data-driven dynamic Web application), its targeted user group, size, complexity, etc. As a trivial example, consider the case of a Web author aimed at the rapid development of a small set of Web pages presenting static content on different end devices. He could be best suited by an ad-hoc authoring process allowing to visually create adaptable content components and "plug them together" to a set of Web pages. On the contrary, the development of a dynamic Web Information System providing different features of adaptive navigation and presentation is a significantly more complex Web engineering task. It should be based on a systematic process model that considers separate concerns of the planned application (data, navigation, presentation, personalization, device and context dependency, etc.) in a structured way. Thus, to facilitate different development scenarios, there is a need for flexible authoring support.

The first part of this chapter (Section 5.1) deals with the structured authoring process of component-based adaptive Web presentations. However, instead of suggesting an own methodology, the chosen strategy is the adoption of existing hypermedia design methods for this purpose. The main reason behind this approach is the observation that, given the abstraction gap between high-level hypermedia design models and low-level implementation entities (document components), even different methodologies can be utilized to develop component-based adaptive Web applications [Fiala et al. 2004b]. This thesis focuses on an

[1]Winston Churchill (1874-1965)

[2]As discussed in Section 3.1, the overall life-cycle of adaptive Web applications encompasses different activities, such as requirements engineering, design, implementation, testing, or maintenance. While the author is aware of the importance of all related activities, the focus of this dissertation (and thus this chapter) is on the model-based design and component-based implementation of adaptive Web sites.

important development scenario: the engineering of data-driven Adaptive Web Information Systems (AWIS) from reusable components. Therefore, it adopts and extends the model-based Hera Web design method [Vdovjak et al. 2003] to the context of component-based Web engineering. The resulting design methodology and engineering process is called Hera-AMACONT and supports the structured development of adaptive Web information systems from reusable components. Considering the steps identified by the Hera design models as a guideline, it is shown how component authors can systematically create, configure, aggregate, and link document components (and templates) to complex adaptive Web presentations. It is illustrated how different design issues concerning data, navigation, presentation, as well as their related adaptation concerns can be taken into account at implementation in a structured way. Thus, a possible model-based authoring process is proposed for the developers of component-based adaptive Web presentations.

In order to efficiently put a given design or authoring process (such as the one dictated by Hera-AMACONT) into practice, component authors need appropriate tools for creating, configuring and aggregating components. To this end, the second part of this chapter (Section 5.2) introduces the AMACONTBuilder, a modular authoring tool for the developers of component-based Web applications. Based on an extensible set of graphical editor modules, it allows to visually create, configure, and aggregate adaptive document components on different abstraction levels. Moreover, it also facilitates the creation of component templates, thus allowing to author data-driven adaptive Web presentations. Independent of a specific methodology, it is shown how it can facilitate different authoring workflows. Finally, selected implementation and extensibility issues are also briefly presented.

While the AMACONTBuilder facilitates flexible component authoring (implementation) independent of a specific design method, in some cases it is desirable to take a further step from model-based component authoring to model-driven component generation and to add automation to the overall process of design and component-based implementation. Therefore, the third part of this chapter (Section 5.3) deals with the research question of how a component-based implementation can be automatically generated on basis of a high-level design specification in a model-driven way. Again, this is illustrated by example of the Hera-AMACONT methodology. After identifying main automation requirements, an RDF(S)-based formalization of the presentation design phase of Hera-AMACONT is provided. According to this formalization, high-level model specifications can be automatically mapped to a component-based implementation, thus exploiting its flexible presentation and adaptation capabilities. The resulting multi-stage development process and document generation architecture are described in detail and exemplified by a prototype application.

Finally, Section 5.4 summarizes the resulting multi-stage Web engineering process and provides a representative overview of already realized component-based adaptive Web applications.

5.1 Hera-AMACONT: Model-based Component Development based on a Hypermedia Design Method

In recent years, different methodologies facilitating the structured design of complex Web applications have been developed. A detailed overview of the most significant existing approaches was given in Chapter 3. As discussed there, most of them distinguish between the conceptual design, the navigational design, and the presentation design of a Web application. Furthermore, some of them even explicitly address selected issues of personalization and adaptation.

This section adopts the model-based Hera design method [Vdovjak et al. 2003] to the context of component-based Web engineering. The resulting design methodology and engineering process is denoted as Hera-AMACONT and supports the structured development of data-driven adaptive Web applications (e.g. online-shops, e-galleries, etc.) from reusable implementation artefacts. Note that Hera is suitable for this undertaking for different reasons. First, its main focus lies on the specification of different kinds of adaptation in a Web Information System [Frasincar et al. 2002]. Besides aspects of adaptability (static adaptation), issues of adaptivity (dynamic adaptation) are also concerned [Frasincar 2005]. Second, Hera uses Semantic Web technologies (RDF and RDFS) to explicitly formalize model descriptions. Such a Semantic Web-based approach has a number of benefits: a more explicit description of model semantics, better interoperability and (possibly) model verification support, as well as the possibility to integrate existing ontologies. Furthermore, due to the usage of XML-based models, an automatic translation of high-level Hera design models to a component-based implementation appears to be also possible. Finally, in contrast to several other methodologies, Hera foresees to specify the presentation aspects (layout, look-and-feel) of a Web application at model level.

Therefore, based on a small running example, different phases of designing and implementing component-based adaptive Web presentations are described. Considering the steps identified by the (extended) Hera design models as a guideline, it is shown how component authors can apply those concepts to systematically develop adaptive Web presentations out of reusable document components. The main focus is on the question of how different adaptation issues (both static and dynamic) can be targeted in each design and implementation step.

5.1.1 Conceptual Design

The first step of the Hera design method is the so-called *conceptual design* aimed at representing the application domain using conventional conceptual modeling techniques. It results in the *conceptual model* (CM) consisting of a hierarchy of concepts, their attributes, and relationships. A concept represents a certain entity in a particular application domain. It is further characterized by concept attributes, each being typed. Besides basic types (e.g. `Integer` and `String`), multimedia types (e.g. `Image`, `Audio`, `Video`) are also allowed, thus enabling to assign representative media items to concept attributes. The CM can be expressed both graphically and using RDFS [Brickley and Guha 2003]. For the graphical specification of conceptual models, Hera provides appropriate modeling tools [Frasincar 2005].

The example application used throughout this chapter is a (small part of a) Web Information System providing information on painters, their paintings, and painting techniques [Fiala et al. 2004b, @ICWE2004Demo]. An excerpt of its underlying conceptual model is depicted in Figure 5.1.

The concepts constituting the application domain are illustrated as dark ellipses. They contain concept attributes denoted as light ellipses. As an example, the concept *technique* (representing painting techniques) has two attributes: a *name* and a *description*. Concepts are related to each other by typed concept relationships. For instance, a painting *technique* is associated to a set of *paintings*, all inheriting from the concept *artifact*. This is an 1:n relationship of the type *exemplified_by*. A painting (since inheriting from the concept artifact) is characterized by its *name*, the *year* of its creation, and a corresponding *picture*. Furthermore, via the relationship *painted_by*, *paintings* are associated with *painters* that again inherit from the abstract class *creator*.

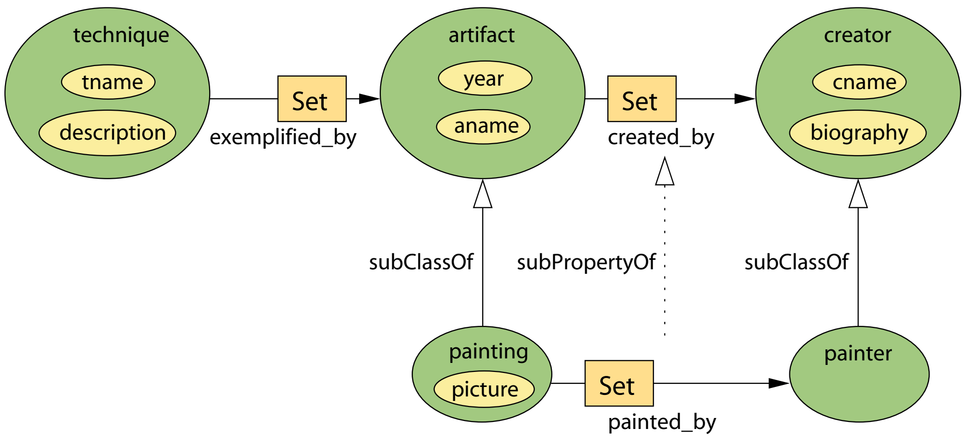


Figure 5.1: CM example [Fiala et al. 2004a]

The media types associated to the concept attributes are described in the *Media Model* (MM), a submodel of CM [Fiala et al. 2004a][3]. It is a hierarchical model composed of media types. The most basic media types are: Text, Image, Audio, and Video. Figure 5.2 shows an excerpt of the MM for the running example. The media types are depicted in dark rectangles.

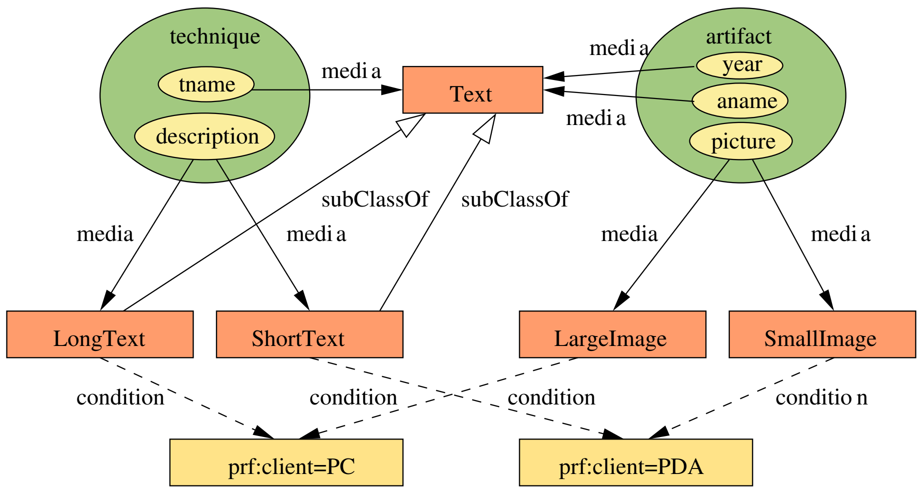


Figure 5.2: MM example [Fiala et al. 2004a]

5.1.1.1 Adaptation at Conceptual Design

The Media Model aims not only at the assignment of media types to concept attributes, it also allows to define media adaptations. These are based on simple Boolean expressions (depicted as light rectangles in Figure 5.2) that reference attributes from the current usage context and dictate the conditional usage of different media types.

In the running example, two conditions addressing the limited screen size constraints of mobile client devices are used. One condition requires to use a long text for the technique

[3]Frasincar also refers to the Media Model as the so-called *media vocabulary* [Frasincar 2005].

description and a large image for the artifact pictures on PCs. The other condition stipulates that a short text for the technique description and a small image for the artifact pictures should be used for PDAs. While in this particular example both conditions specify adaptability (i.e. adaptation based on context parameters that are typically static with regard to a Web session), note that media adaptations can be performed dynamically, as well. As an example, the designer could specify different quality alternatives for a video depending on the currently available bandwidth. Thus, in case of possible bandwidth fluctuations (e.g. in a mobile environment), the usage of a given alternative could be dynamically reconsidered even during a single browser session.

We note, however, that this adaptation of the media representation of concept attributes concerns not only the data presented by a Web application (which is the actual focus of conceptual design) but also its presentation. The reason for this is the fact that a media object (e.g. an image) represents not only content but also inevitably presentation. As a consequence, the media model of a Web application might be reconsidered or extended after specifying its presentation design (in a later design step, see Section 5.1.5). Therefore, we claim that the design of a Web application might be an iterative approach, allowing to refine the separate design models in several turns.

5.1.2 Realization with Document Components

When developing adaptive Web presentations from document components, the conceptual design step as proposed by Hera has to be accompanied by the creation or retrieval of media instances that represent the identified concept attributes. These media instances (as well as the metadata attributes required by the component-based document format for describing their media properties) have to be stored in a structured data store so that they can be dynamically presented in the resulting adaptive Web application. The structure of this data source is to be derived from the CM, respectively. Furthermore, the appropriate media component templates facilitating the dynamic presentation of the created media instances have to be created.

In order to realize different media adaptations, component authors also have to reason about alternative media instances with different quality (e.g. concerning their formats, bandwidth, color depth, bit rate, size, etc.). According to Section 4.3.1, these alternatives have to be defined as media component (template) variants with their corresponding selection methods. For instance, a media component representing a painting's picture should have two variants, one for desktop devices and another one for handhelds.

5.1.3 Application Design

The application design step of Hera is the most important design phase dealing with the logical, structural, and navigational aspects of a Web application. Similar to the navigational models of other methodologies, its main goal is the specification of the overall hypermedia structure of the resulting application, i.e. the design of navigational units (hypermedia nodes and pages), their relationships (aggregation and interlinking), as well as corresponding adaptation issues[4].

[4]Recently, Hera's application model was extended by mechanisms for the specification of form-based user interactions [Frasincar 2005]. Still, the concepts described here focus on the basic Hera models, as also published in [Fiala et al. 2004b, Fiala et al. 2004a]. It is claimed that the mentioned extensions are adoptable for the context of component-based adaptation engineering, which is subject to ongoing cooperation.

In order to model navigational units, Hera uses the notion of *slices*. As in the case of RMM, a *slice* is a meaningful presentation unit that fulfills a certain communication purpose [Isakowitz et al. 1995], i.e. it represents an abstract view over the content described in the conceptual model that should be shown on a hypermedia node.

A slice is always associated with its owner concept which denotes the data (i.e. the concept from the CM) portrayed by it. Furthermore, there are two types of slice relationships: *slice navigation* (a hyperlink abstraction between two slices) and *slice aggregation* (a slice including another slice). An aggregation relationship between two slices with different owner concepts needs to specify the concept relationship between those concepts from the CM that made such an embedding possible. In the case that the cardinality of this concept relationship is one-to-many, the *Set* construct needs to be used. The most primitive slices represent concept attributes and are also referred to as *simple slices*. Slices that aggregate other slices are called *complex slices*. The most complex ones (called *top-level slices*) correspond to pages, which contain all the information presented on the user's display at a particular moment. The creation of AMs is provided by graphical tool support. For a more thorough introduction to Hera's application model vocabulary the reader is referred to [Frasincar 2005].

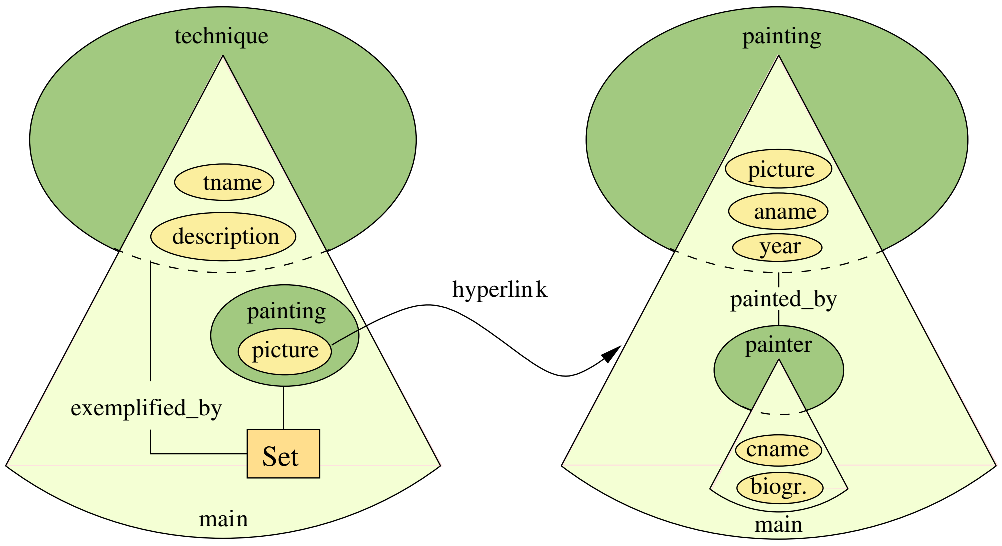


Figure 5.3: AM example

Figure 5.3 depicts (a small excerpt of) the application model of the running example in a graphical way. It consists of two top-level slices, one presenting painting *techniques*, the other *paintings*. As indicated by the underlying dark ellipse, the left top-level slice is associated with the concept *technique*. It shows two attributes of that concept: its name (*tname*) and description (*description*). Furthermore, it also contains a link list pointing to the *paintings* exemplifying the given painting technique. As depicted in the picture, these links are based on the concept relationship *exemplified_by*. The starting anchor of each link is represented by the *picture* attribute of the referenced *painting* concept. Since there are several paintings associated with a technique, a Set construct is used. When following a link, the user can navigate to the corresponding *painting* slice, a composite (top-level) slice presenting the concept *painting*. This slice presents the actual painting's *picture*, its *name*, the *year* when it was painted, as well as (from the bottom slice) some information about its *painter*. Note that besides the graphical representation (called the Application Diagram) there exists also an RDFS-based formalization of the AM [Frasincar et al. 2002].

5.1.3.1 Adaptation at Application Design

Adaptation at application design concerns the adjustment of the Web presentation's logical and navigational structure to the user and his usage context. Generally, different adaptation issues can be considered in the AM. First, it is meaningful to adjust the coarse navigational structure to varying device capabilities (e.g. desktop computer, PDA, cell phone, etc.) or user profiles (preferences, interests, knowledge). Depending on this information the designer can decide which concepts should be presented at all and how they should be assigned to different interlinked slices. Second, the population of each specified slice with concept attributes (or media types) can be adjusted, too. According to the preferences and/or used devices of different users, different media types for presenting the same concept can be utilized. As an example, take the case of two visitors, one of them preferring multimedia content, the other rather textual information. When presenting a painter's biography, the first one could be shown a video and an audio sequence, the second one a detailed textual description. Furthermore, dynamic adaptation (adaptivity) can also be targeted at this step. For example, different versions of a painter's biography could be presented in accordance with the user's changing knowledge on that painter: a long version at the user's first visit and a short one at his later visits. As a matter of course, these are only possible adaptation examples, the consideration of a certain adaptation concern (device dependency, personalization, security, etc.) depends on the designer's choice.

In order to specify adaptation, Hera prescribes that one associates so-called *appearance conditions* to slices [Frasincar et al. 2002, Frasincar 2005]. These are Boolean conditions using attribute-value pairs from the current usage context. Two kinds of AM adaptation are enabled: conditional inclusion of slices and link hiding. Conditional inclusion means that a slice is included (and therefore visible) when it has a valid condition. Similarly, link hiding refers to the mechanism that a link is only included when its destination slice is valid.

Figure 5.4 depicts another version of the application model shown above which is enriched by adaptation definitions. Note that it contains three appearance conditions. The first one (mentioning ExpertiseLevel) supports adaptability by including the painting technique's *description* only for Experts. The second one (mentioning imageCapable) refers to the device profile and dictates that the pictures of paintings should be only shown on devices being capable of presenting images. The third one (mentioning biography) defines adaptivity by presenting different versions of a painter's biography depending on the user's knowledge on that painter.

5.1.4 Realization with Document Components

There are important analogies between (the concepts of) Hera slices and adaptive document components. Both represent meaningful presentation units bearing also some semantic role (e.g. painting, painting technique, newspaper article) and are recursive structures enabling an arbitrary deep hierarchy. Moreover, both top-level slices and top-level document components correspond to hypermedia pages to be presented on the user's display. Furthermore, both may contain adaptation issues according to context model parameters.

Nonetheless, there are also significant differences. First, in contrast to slices, document components also contain information describing their layout. However, as application design concentrates on the navigation and does not deal with presentation issues, the specification of these layout properties can (and should) be postponed to a later stage of the development process (see Section 5.16). Second, whereas AM slices define the structure of presented concepts on the schema level (i.e. independent of concrete instances of those concepts), document

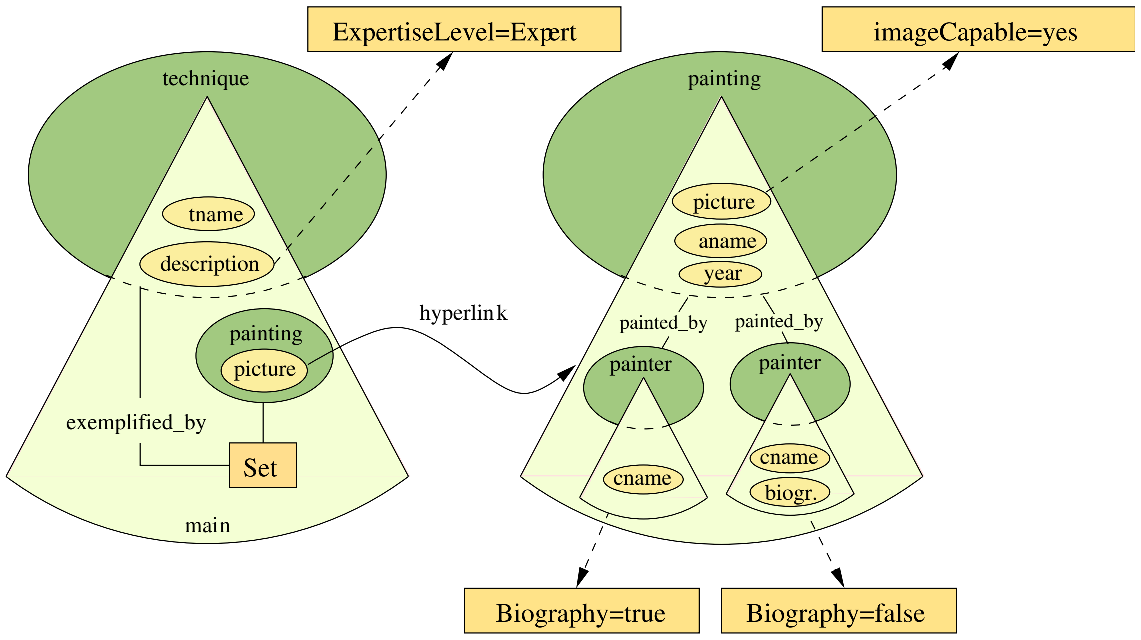


Figure 5.4: AM example with appearance conditions

components represent reusable implementation entities on the instance level. Still, note that this gap can be bridged by considering the notion of component templates. As described in Section 4.4 in detail, these are component skeletons declaring the structural, behavioral, and layout aspects of components independent of their concrete content. For example, a component author might create a component template for presenting dynamic information on painters. Such a template can be instantiated for specific painters by dynamically querying the Web application's underlying data source, which was created accompanying its conceptual design.

The mentioned analogies allow component authors to specify the aggregation hierarchy of component templates in accordance to a given AM design. First, top-level slices have to be mapped to top-level document component templates. Second, by unfolding slice aggregation relationships in a top-down manner, subslices have to be mapped to "sub document components" (or templates). In the case of simple slices, the media items (components) representing the concept attributes associated with those slices have to be additionally considered. Furthermore, both Set structures (of simple and composite slices) as well as slice navigation relationships (e.g. link lists) have to be taken into account. While there are different possibilities to map slice hierarchies to component template structures, we mention a straightforward and easily automatable one:

1. A complex slice that contains other (complex or simple) subslices should be mapped to a document component template. For its aggregated subslices, this mapping process should be performed recursively. As already mentioned, top-level slices correspond to top-level document components.

2. A simple slice (representing a concept attribute) has to be mapped to a document component template that additionally contains a content unit that again contains one or more media component templates. These media components correspond to the media items representing the slice's owner concept attribute, their types have to be determined

by the media type(s) associated with that concept attribute in the Media Model (MM). *Integer* and *String* attributes have to be mapped to text components, media attributes to corresponding media components (image, audio, video, etc.).

3. Whenever a slice (complex or a simple) is part of a Set construct, the appropriate component template that was assigned to it should be defined as an iterative template.

4. Finally, slice navigation relationships between two slices have to be mapped to hyperlink components between the appropriate component templates. Again, when a slice navigation relationship is based on a 1:n concept relationship, an iterative hyperlink list has to be configured.

Figure 5.5 depicts this possible mapping process in a graphical (schematic) way by example of the slice representing painting techniques. The types of the created component templates are denoted by the abbreviations MC (media component), CU (content unit component) and DC (document component). Furthermore, the mappings are illustrated as arrows, each labeled by a number indicating the appropriate mapping rule from the above list.

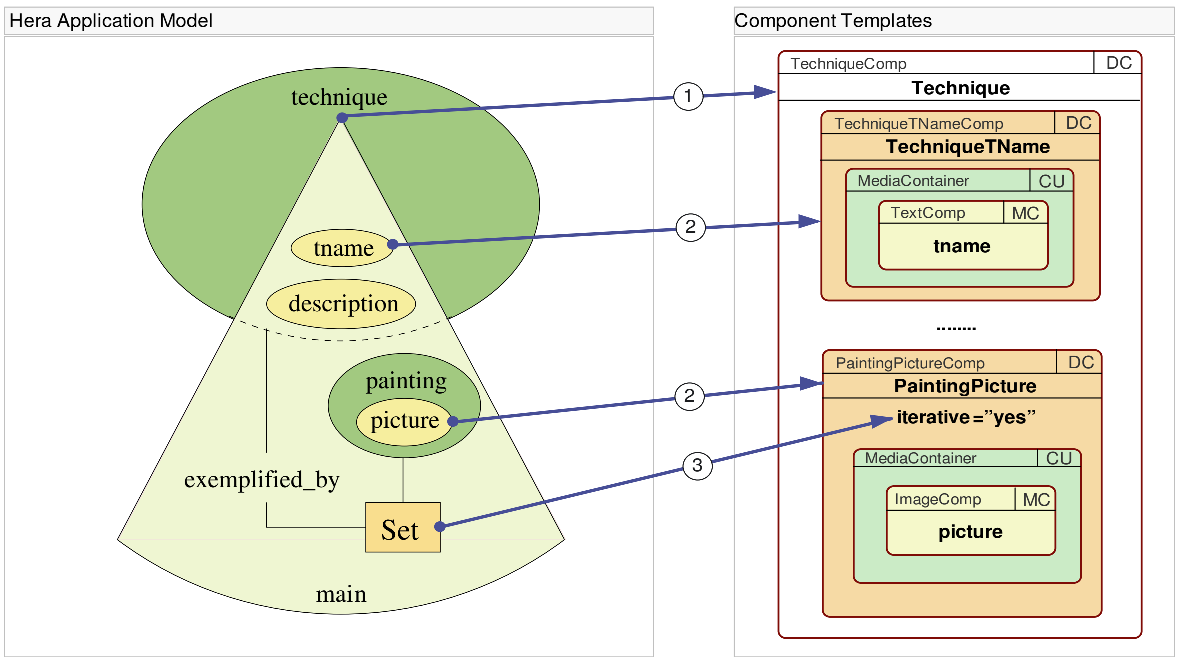


Figure 5.5: Slice to component template mapping

If the AM specifies adaptation aspects via appearance conditions, these have to be expressed in form of corresponding component variants and their selection methods (see Section 4.3.1). Again, this can be done in a straightforward way. Whenever a slice is provided with a Boolean appearance condition, the component associated to it has to be made a variable component containing only one variant. Moreover, according to the slice condition a selection method in the IF-THEN style has to be composed (see Section 4.3.1). Note, however, that the concept of adaptation variants supported by the component-based document model allow for more sophisticated kinds of adaptations than simple appearance conditions attached to navigational elements. To optimally address different client capabilities or user preferences, component authors might flexibly define different variants of the created component templates on all abstraction levels.

Furthermore, whenever a given adaptation specification concerns adaptivity, component authors also have to reason about how to update context model parameters according to users' interactions. For this purpose the context modeling components of the document generation architecture can be utilized (see Section 4.5.3). For instance, the number of a user's requests to a document component instance (e.g. representing a painter's biography) can be easily tracked in the Session Profile (see Section 4.5.2). This information can be utilized to define the appropriate component variants and their corresponding selection methods.

5.1.5 Presentation Design

The presentation design step of Hera bridges the logical level and the actual implementation by introducing the implementation independent Presentation Model (PM). Complementary to the AM, where the designer is concerned with organizing the Web application's overall structure and identifying which concept attributes from the entities of the application domain should be included in slices, the PM specifies how and when the identified slices should be displayed.

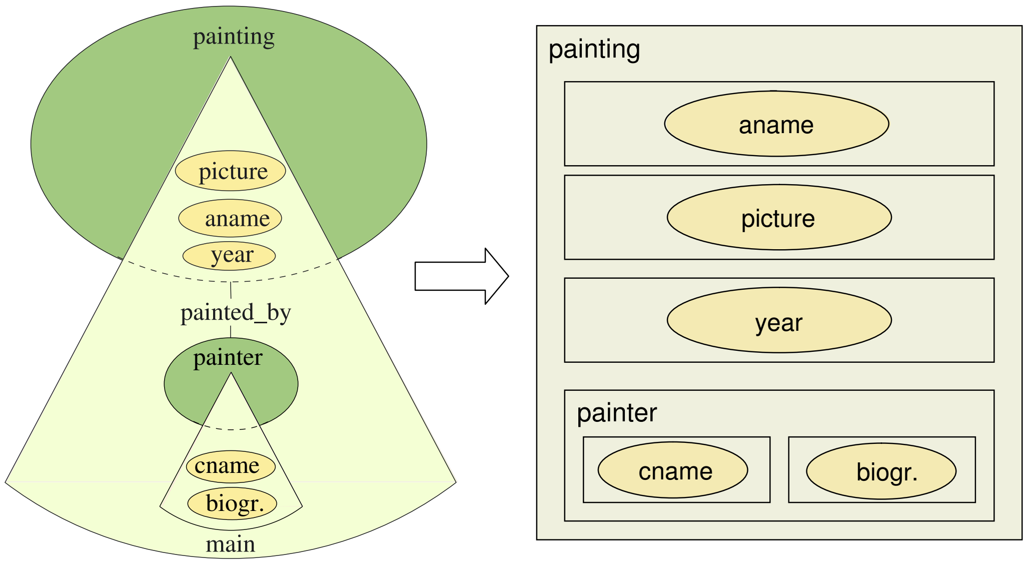


Figure 5.6: Presentation diagram (PD) example: assigning regions to slices

The PM is described by a presentation diagram (PD) consisting of regions and their relationships. A region is an abstraction for a rectangular part of the display area where the content of a slice is to be displayed. During presentation design, the slices introduced in the AM are mapped to regions (and subregions). The PD specifies the organization of regions in an informal, graphical way by means of region relationships that describe their relative position (e.g. above, below, left/right from) to each other [Frasincar et al. 2001]. As an example, Figure 5.6 depicts a possible presentation diagram assigned to the *painting* slice (see Figure 5.3). It dictates to display a painting's name, picture, year, and painter below each other and arranges the name and biography attributes of the painter in a horizontal manner.

Note that at the time when the work described in this chapter was carried out there was no RDF-grammar for expressing PDs in a formal way, nor were style design and adaptation considered in the PM. These issues were addressed in form of the Hera-AMACONT model extensions as part of the work presented in this thesis [Fiala et al. 2004a, Fiala et al. 2004b] and will be described here, respectively. In the following subsection different issues of presentation layer adaptation as well as a possible component-based realization of the presentation design step are discussed. An RDF(S)-based formalization of the corresponding Hera-AMACONT PM will be provided later in Section 5.3.1.

5.1.5.1 Adaptation at Presentation Design

As personalization and adaptation become prominent issues of Web engineering, it is inevitable to address adaptation at presentation design. Still, while adaptation has been extensively considered in navigation design (see Chapter 3), adaptation in the presentation layer has not been a central issue of Web design methodologies, yet. Nevertheless, it is in a lot of scenarios necessary to adjust a Web application's presentation aspects. As the most significant issues the following can be mentioned:

1. An important adaptation target is the *spatial placement* (layout) of the content elements of a Web page. Depending on varying user preferences and/or device characteristics (e.g. screen size, supported document formats, interaction techniques, etc.), they should be displayed differently. Possible layout adaptation techniques are:

   Reorganization: In this case the arrangement of content elements is adapted. Most typically, the purpose of this adjustment is to optimally fit a Web presentation to the varying display sizes of different client devices. Whereas for example the tabular arrangement of content may look well on conventional desktop computers, it could cause a lot of undesirable horizontal scrolling when being browsed on handhelds with limited display size. Similarly, the writing scheme of a Web presentation's language (e.g. left-to-right, right-to-left, horizontal, or vertical) can also significantly influence the arrangement of content elements like headers, footers, navigation bars, etc. [Evers and Day 1997].

   Exclusion: Information being unsuitable for a particular browser (e.g. a picture gallery for a monochrome mobile phone) or content elements without an important semantic meaning (e.g. company logos in an online shop) can be excluded from Web presentations on mobile devices with small displays or low bandwidth connections.

   Separation: As a (less strict) form of exclusion, it can be advantageous to put certain content pieces onto separate pages and automatically create hyperlinks to them. This mechanism is very useful to keep the structure of Web pages while providing a lot of information easily understandable on handhelds [Hwang et al. 2002].

2. A further adaptation target is the corporate design (i.e. the "look-and-feel") of a Web application, which is typically determined by decorative elements such as logos, background colors, font parameters (size, color, type), buttons, etc. Though not influencing the logical structure of a Web site, such design elements are important to appropriately convey the published information to the user. Consequently, presentation designers might provide alternative style variants in order to address different user properties and usage contexts. As possible adaptation aspects the following can be mentioned:

   Style preferences: The visitors of a Web presentation might have different style preferences based on their interests, education, and/or age. While e.g. a site addressing little children typically utilizes vivid colors and decoration elements [Nielsen 2002], Web applications targetting a more “serious” audience are characterized by rather modest and simple layouts.

Cultural background: The cultural background of the targeted audience is also an important presentation adaptation feature. Barber and Badre provide an overview of so-called *cultural markers*, i.e. “interface design elements that are prevalent, and possibly preferred, within a particular cultural group” [Barber and Badre 1998]. As an example, specific color combinations, symbols, or decoration elements can have varying interpretations in different countries and cultures.

Accessibility issues: A further adaptation aspect to be considered at style design is accessibility. In order to appropriately address users with visual impairments (limited level of sight, color blindness, etc.), it requires to offer Web presentations with different color schemes, font types, and sizes. The World Wide Web Consortium (W3C) tackles this issue by offering a number of Web Content Accessibility Guidelines [Chisholm and Vanderheiden 1999] for Web designers and developers.

Environment characteristics: Web applications developed for mobile scenarios might adapt their visual appearance based on selected characteristics of the current environment. As an example, the contrast or brightness of a Web page might be adjusted to whether the user is situated in an indoor or anoutdoor context.

Specific events or time periods: Finally, the corporate design of a Web site might be also adjusted to specific events or time periods, such as seasons, anniversaries, or festivals. As a typical example, Web-based online shops or communities are often decorated with a dedicated layout at festivals like Halloween or Christmas, at Valentine’s day, or even during sport events like e.g. football world championships.

3. Third, the qualitative adjustment of the media objects included in a Web site (e.g. to the display capabilities of different end devices) is also an important adaptation issue to be considered at presentation design. Still, since the assignment of media types to the application’s conceptual model is specified in the already presented Media Model (see Section 5.1.1.1), these adaptations should be also defined there, respectively. As already mentioned, the adaptation definitions concerning the media model might be modified or extended at presentation design.

4. The aforementioned examples represent static adaptation. However, in some cases it is meaningful to consider dynamic adaptation, i.e. adaptation according to parameters that may change while the Web presentation is being browsed. As a possible scenario (in presentation design) we mention the dynamic reorganization of presentation elements on a page when the user resizes his browser window or the automatic reconfiguration of a Web presentations color scheme (colors, contrasts, brightness) when a mobile user enters a differently illuminated area (e.g. changing from an outdoor to an indoor context).

5.1.6 Realization with Document Components

The aggregation hierarchy of component templates was determined at application design. Now, based on the guidelines of the graphical presentation diagram, component authors are expected to specify the layout attributes of those component templates as well as the corporate design of the resulting presentation. Furthermore, to address the possible adaptation issues mentioned above, they also have to consider layout and design style variants.

As mentioned in Section 4.3.2, the component-based document format provides an XML-based mechanism for specifying the spatial adjustment of subcomponents within their container components in a size and client-independent way. Those abstract layout definitions support the automatic conversion of component structures to Web presentations in different output formats and are well suitable for implementing an abstract presentation design consisting of a hierarchy of rectangular regions. Consequently, the spatial relationships between regions defined in the PD have to be mapped to such component layout descriptions.

Again, this mapping can be performed in a straightforward way. Beginning at top-level document component templates and visiting their subcomponents recursively, one has to declare for each component template how its immediate subcomponents are arranged. As an example, the component templates containing the concept attributes describing a painting (see in Figure 5.6) can be arranged according to a vertical *BoxLayout* scheme. Similarly, the name and the biography attributes of the corresponding painting's painter can be organized based on a horizontal *BoxLayout*. When defining such layout managers, component authors can use their various configuration options (widths, heights, alignments, etc.) which were described in detail in Section 4.3.2.

The corporate design of a component-based presentation can be specified by the creation and configuration of corresponding CSS media components. The CSS standard of the W3C [Bos et al. 2006, Lie 2005] allows for the definition of a Web presentation's design and style elements (background colors, font sizes and types, link colors, etc.) and is thus perfectly suitable for this purpose. A CSS media component contained by a composite document component specifies the corporate design of the content it displays. Whenever there are several CSS components in a component-based Web document, they can redefine each other's style definitions by the order of their occurrence in the overall component hierarchy.

In order to cope with the adaptation issues described in Section 5.1.5.1, component authors might create different layout variants for components, each bound to a specific adaptation condition. For instance, the typical small display size and horizontal resolution of handheld devices would require to present not only the attributes of a painting, but also the names and the biographies of their painters below each other (i.e. according to a vertical *BoxLayout* scheme). Similarly, the corporate design of a component-based presentation can be also adapted by specifying different CSS media component variants and their selection methods.

After this authoring step, the content, the structure, and also the layout layout managers of the resulting components (templates) are fully specified. They manifest a component-based implementation of the corresponding design models.

5.1.7 Summary

This section exemplified the development process of component-based adaptive Web presentations according to the design phases dictated by the model-based Hera design method. Based on a small example application, it was shown how during the phases of design and implementation different application concerns (content, navigation, presentation) as well as corresponding adaptations (both static and dynamic) can be taken into account systematically. That is to say, a possible model-based authoring process for the developers of component-based adaptive Web presentations was introduced[5]. As a short summary, Table 5.1 recapitulates the identified design steps, their "implementation recipe" based on adaptive Web document components, but also the adaptation issues to be addressed in each development phase.

[5]A summary of component-based Web application prototypes realized based on the authoring process and tool support described in this chapter will be given in Section 5.4.2.

	Design & Modeling	**Component-based Implementation**
Conceptual Modeling	specification of the application's domain model	creation, retrieval and structured storage of media components representing concept attributes
CM Adaptation	adaptation of media quality	creation of media component variants with quality alternatives
Application Modeling	design of the application's navigational structure by slices and slice relationships	creation and interlinking of composite components (content units, document components) and templates
AM Adaptation	adaptation of slice aggregation and navigation	definition of alternatives for subcomponents and hyperlink structures
Presentation Modeling	design of the user interface based on regions and style definitions	definition of components' layout managers and CSS styling
PM Adaptation	design of layout and style adaptation	definition of alternative layout managers and CSS components

Table 5.1: Summary of design and implementation phases

The mentioned development steps facilitate a structured design and implementation process for component-based adaptive Web presentations. The resulting component templates constitute a dynamic component-based hypermedia presentation realizing the different design (and adaptation) issues expressed by the corresponding design models. Consequently, they can be used as the input of the pipeline-based document generator introduced in Section 4.5. As described there, for each user request the corresponding component template is retrieved and instantiated with the requested data. According to the current usage context, it is then subdued to a series of transformations, each considering a certain adaptation aspect. The resulting Web presentation is automatically adjusted to the actual usage context.

Note, however, that the development process presented in this section is only one possible approach for data-driven component-based adaptive Web applications. As mentioned before, the abstraction gap between design methods and implementation entities (components) allows to use different methodologies for developing component-based adaptive Web sites.

5.2 A Modular Authoring Tool for Component-based Adaptive Web Applications

As illustrated in the previous section, the component-based document format and its document generation architecture provide a sound basis for the development and publication of adaptive Web presentations. Based on a structured authoring process (e.g. Hera-AMACONT), authors can compose adaptive Web applications from reusable components in a disciplined way, by subsequently taking into account different design concerns, their adaptation issues, as well as their corresponding "implementation recipes". Still, the complexity of the component-based document format's underlying XML grammar calls for an intuitive authoring tool that supports this composition process in a graphical way. Such an authoring tool should provide following functionality:

1. **Visual authoring support:** The authoring tool should offer graphical editor modules for the creation, configuration, and composition of document components. These should hide the low-level details of the XML-based component description language from authors, allowing to create component-based documents in a visual way.

2. **Independent authoring of separate concerns:** The authoring tool should provide a number of specialized editors allowing to separately configure different component properties (such as content, structure, layout, interlinking, adaptation) in different phases of the authoring process.

3. **Support for instance- and template-level authoring:** The authoring tool should facilitate the creation of both component instances and component templates. For the latter case it should allow authors to intuitively access dynamic data sources and configure the appropriate queries.

4. **Preview functionality:** In order to allow authors to test the currently created/edited documents, the authoring tool should provide a flexible preview functionality depending on the actual user, context, and device characteristics.

5. **Flexible authoring workflows:** Instead of being bound to a specific authoring process (e.g. the one presented in Section 5.1), the authoring tool should provide a flexible set of editor modules for the manipulation of different component types and properties. Authors should have the freedom to flexibly use these editors based on the current authoring scenario, thus being able to proceed based on different process models.

6. **Extensibility:** To cope with the flexibility and extensibility of the component-based document format (see Section 4.6.3), the authoring tool should also be based on a modular and extensible architecture. This should facilitate to integrate both alternative editor modules for existing and new editors for future component types.

To fulfill these requirements, a graphical component authoring tool called the AMACONTBuilder was developed [Fiala et al. 2005]. It is based on an extensible set of visual editor plug-ins and allows to graphically create and compose document components on different composition levels. Furthermore, it also supports the configuration of both their adaptation variants and adaptive layout. Note, however, that as a tool designated for component authoring, the AMACONTBuilder is oriented at the phase of (component-based) implementation in the overall Web engineering process (see Section 3.1). That is to say, instead of being bound to a specific methodology (e.g. Hera-AMACONT), it allows to compose adaptive Web components based on different authoring processes[6].

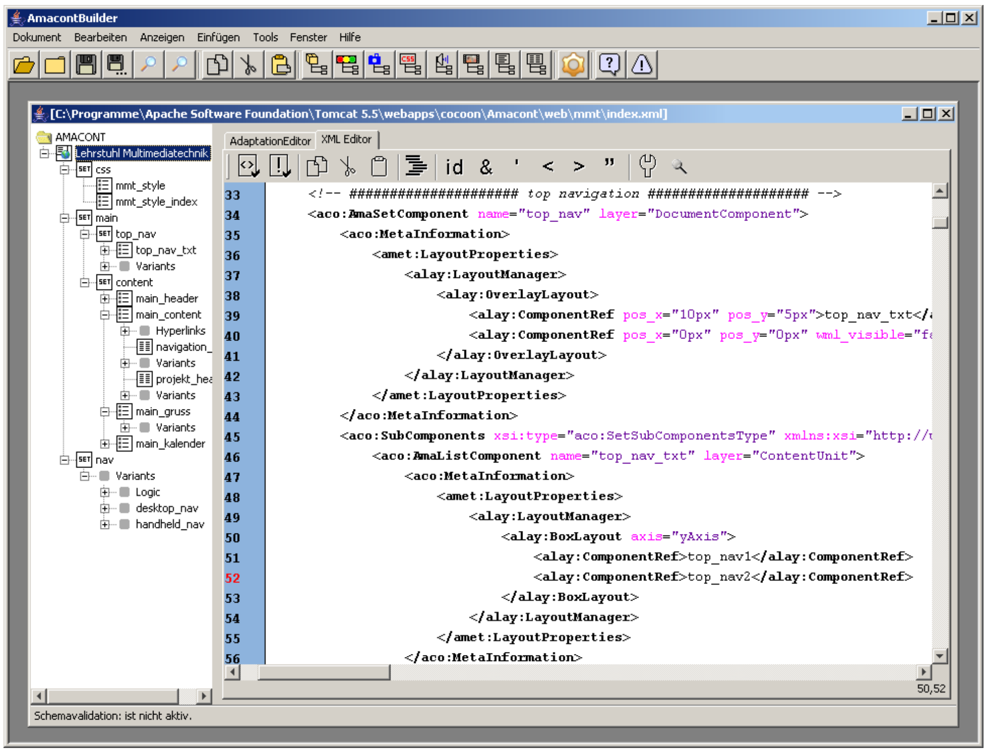
This section gives an introduction to the AMACONTBuilder. First, its basic concepts and main architecture is presented. Then, selected editor modules are described in more detail, supporting different phases of the authoring process of component-based adaptive Web presentations. Finally, a couple of implementation issues are briefly summarized.

[6]The issue of the model-driven generation of component-based adaptive Web applications based on Hera-AMACONT models will be subject to Section 5.3.

5.2.1 AMACONTBuilder: An Overview

The AMACONTBuilder is a modular authoring tool aimed at the visual development of component-based adaptive Web applications. It is based on an extensible authoring framework [Chevchenko 2003] that allows to edit arbitrary XML documents. The framework was developed at the Chair of Multimedia Technology of the Dresden University of Technology and provides generic functionality for parsing XML files into and internal object model, as well as for implementing editor plug-ins dedicated to specific object model (XML) elements. Thus, it can be easily extended by graphical editor modules (plug-ins) to visually author content based on a given XML grammar. Previously, the framework was successfully utilized for the development of courseware in the CHAMELEON project (see Section 3.2.5).

As shown in Figure 5.7, the user interface of the AMACONTBuilder consists of two main parts: the *application frame* and the *document frame*. The *application frame* provides generic functionality for configuration options and file management. It is responsible for parsing XML-based documents to an internal object model, for assigning editor plug-ins to parts of this object model, and for serializing the modified object model to XML, respectively. It contains the *document frame* showing the currently opened (edited) document. This is again divided into two parts: the *navigation frame* and the *editor frame*.



```
<!-- ##################### top navigation ##################### -->
<aco:AmaSetComponent name="top_nav" layer="DocumentComponent">
    <aco:MetaInformation>
        <amet:LayoutProperties>
            <alay:LayoutManager>
                <alay:OverlayLayout>
                    <alay:ComponentRef pos_x="10px" pos_y="5px">top_nav_txt</
                    <alay:ComponentRef pos_x="0px" pos_y="0px" wml_visible="f
                </alay:OverlayLayout>
            </alay:LayoutManager>
        </amet:LayoutProperties>
    </aco:MetaInformation>
    <aco:SubComponents xsi:type="aco:SetSubComponentsType" xmlns:xsi="http://
        <aco:AmaListComponent name="top_nav_txt" layer="ContentUnit">
            <aco:MetaInformation>
                <amet:LayoutProperties>
                    <alay:LayoutManager>
                        <alay:BoxLayout axis="yAxis">
                            <alay:ComponentRef>top_nav1</alay:ComponentRef>
                            <alay:ComponentRef>top_nav2</alay:ComponentRef>
                        </alay:BoxLayout>
                    </alay:LayoutManager>
                </amet:LayoutProperties>
            </aco:MetaInformation>
```

Figure 5.7: AMACONTBuilder overview [Fiala et al. 2005]

The *navigation frame* on the left provides a tree-based view on the node (component) structure of the currently edited document. When navigating through this component hierarchy, the specific editors assigned to the appropriate component types are automatically activated in the *editor frame* (shown on the right). While in Figure 5.7 the *navigation frame* shows the overall component hierarchy of the edited document, it is possible to use predefined filters. For instance, an author aimed at the creation and configuration of image components has the possibility to display only the corresponding components and filter out all other ones.

Finally, the *editor frame* provides space for the actual editor plug-ins associated to different node (or component) types. It displays the editors associated to the currently edited component (chosen in the navigation frame). A component type might be associated with several editor modules. As an example, content unit components and document components have both editors for manipulating the aggregation of their subcomponents as well as their layout. In this case all corresponding editors associated with the actual author role are shown as separate panes and can be activated by the author, respectively. The assignment of editors modules to component types is determined by an XML-based configuration file that can be individually set for different authoring scenarios.

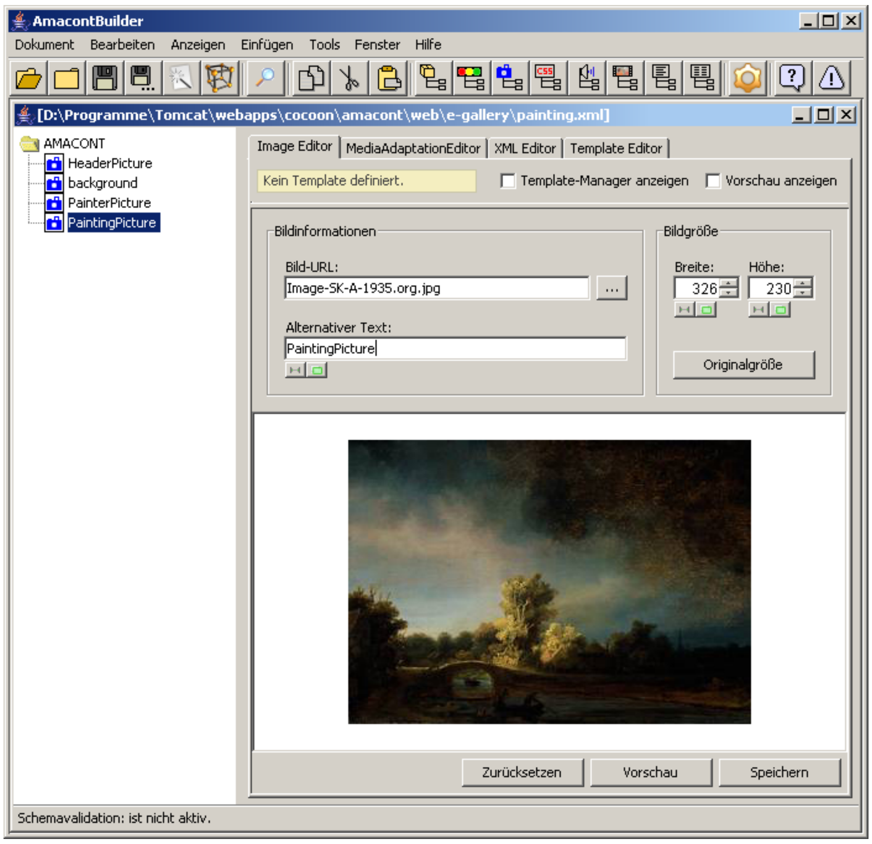
While there are editor modules being applicable to all kinds of XML content (e.g. the XML code editor shown in Figure 5.7), most modules are assigned only to specific node (component) types and are thus activated at well-defined phases of a given authoring process. The following sections give a short overview of the most important existing editor modules. According to the main steps of the authoring process (and the running example) described in Section 5.1, selected editors for content, navigation, and presentation authoring are presented.

5.2.2 Editors for Content Authoring

For the graphical creation of media components different visual editors (*text editor*, *image editor*, *CSS editor*, etc.) have been created. By example of a picture representing a painting, Figure 5.8 presents the *image editor*. It allows to upload images in different formats (e.g. jpeg, gif, bmp, png), to edit their properties, and to save them as image components. While most image metadata properties can be configured by appropriate input fields, some editing operations can be also performed visually. As an example, the size of an image component can be simply altered by mouse dragging.

To support for adaptation, the media editors (but also all other kinds of component editors) were extended with a generic mechanism for creating content alternatives. For instance, in the image editor it is possible to provide an alternative text for browsers that are not able to present images. Furthermore, image variants with different quality alternatives can be added. Such variants can be created in three ways: by uploading alternative images, by reconfiguring (e.g. resizing) the current image and save it as a new variant, or by generating new images automatically. In the latter case the author can predefine the properties (e.g. pixel size, color depth, image format) of an arbitrary number of variants to be created. According to this configuration, the alternative media instances are generated automatically. This feature was implemented by using the Java API of ImageMagick [@ImageMagick].

After creating media component alternatives, component authors can define their adaptive behavior by attaching adaptation conditions to each variant. As discussed in Section 4.3.1, these conditions are Boolean expressions referencing parameters from the CC/PP-based context model. For the configuration of adaptation conditions the *profile browser* was developed (see Figure 5.9). It allows authors to visually navigate through the hierarchy of profiles, to choose the appropriate parameters, and to insert them into adaptation conditions. The profile browser can be configured by an RDFS document defining the current application's context model. The example in Figure 5.9 declares to use the current picture for browsers with less then 8 bits per pixel color depth and less than 400 pixel horizontal resolution. As can be seen, authors can "click together" complex logical expressions by visually choosing the appropriate parameters from the pop-up window presenting the context model's hierarchical structure.



[D:\Programme\Tomcat\webapps\cocoon\amacont\web\e-gallery\painting.xml]

Kein Template definiert.

Template-Manager anzeigen ☐ Vorschau anzeigen ☐

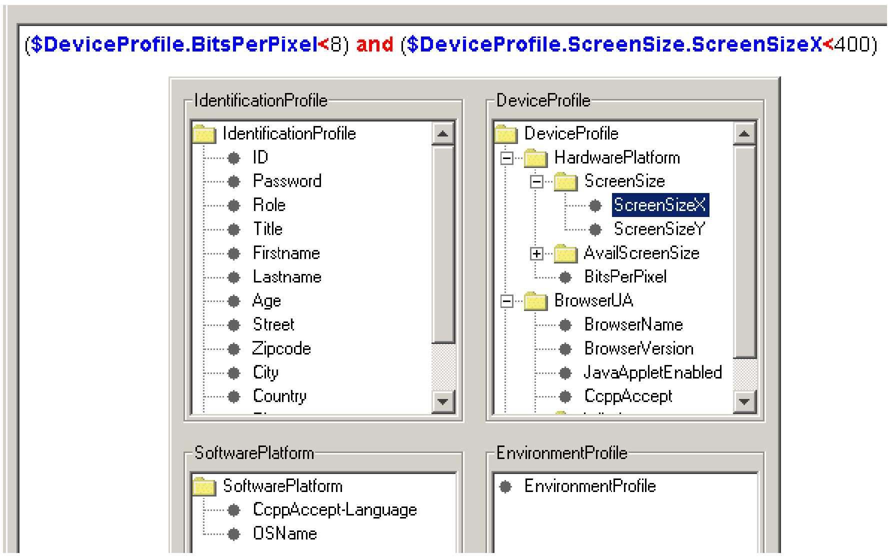
Bild-URL: Image-SK-A-1935.org.jpg

Alternativer Text: PaintingPicture

Breite: 326 Höhe: 230

Schemavalidation: ist nicht aktiv.

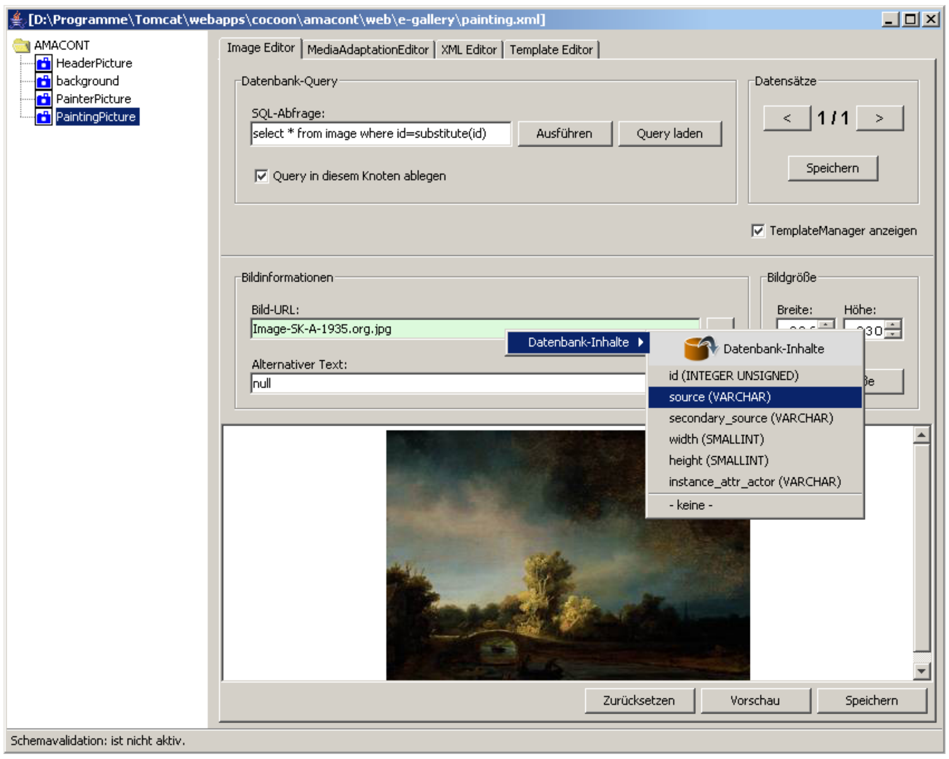
Figure 5.8: Image editor



($DeviceProfile.BitsPerPixel<8) and ($DeviceProfile.ScreenSize.ScreenSizeX<400)

Figure 5.9: Defining adaptation conditions with the profile browser

The mechanisms described above aim at authoring adaptable content (media component) instances. Still, in order to support for data-intensive Web applications, the editor tools can be switched from this "instance mode" to the so-called "template mode", i.e. authors can assign a data source query to the currently edited component [Tietz 2006]. The result is a component skeleton (component template) that can be filled with dynamically retrieved data on-the-fly.



[D:\Programme\Tomcat\webapps\cocoon\amacont\web\e-gallery\painting.xml]

SQL-Abfrage: select * from image where id=substitute(id)

☑ Query in diesem Knoten ablegen

Datensätze: 1 / 1

☑ TemplateManager anzeigen

Bild-URL: Image-SK-A-1935.org.jpg

Alternativer Text: null

Schemavalidation: ist nicht aktiv.

Figure 5.10: Template editor for image components

Figure 5.10 depicts this mechanism by example of the image editor. As can be seen, in this particular case the author defined a simple query retrieving images (of paintings) as well as their metadata from a relational data source. Note, however, that it is also possible to access the results of a query which was defined on a higher level in the component hierarchy. After defining a query, the author has the possibility to assign specific fields of its result set to the attributes of the image component. The drop-down list shown in Figure 5.10 illustrates how he assigns the *source* field of the query's result set to the source attribute of the image component template. Consequently, such dynamic attributes will be filled with the appropriate values on-the-fly. As a matter of course, it is allowed to define some attributes as constants. Furthermore, it is also possible to define adaptation operations on template level by creating a variant of the component template, assigning an alternative query field (containing e.g. the PDA variants of images) to it and defining a corresponding selection method. For more information on the AMACONTBuilder's media editors the reader is referred to [Fiala et al. 2005].

5.2.3 Editors for Hypertext Authoring

Whereas the editors for content authoring facilitate the creation of media components (or templates) and their adaptation variants, the editor modules for hypertext authoring focus on structuring and interlinking components to complex hypermedia structures. They are further divided into two groups: 1) editors for composing component hierarchies and 2) editors for defining hyperlink structures between those hierarchies.

5.2.3.1 Editors for Creating Component Hierarchies

The visual creation of component hierarchies is facilitated by two modules, the *structure editor* and the *subcomponent editor* [Niederhausen 2006]. While the former one aims at specifying the overall composition structure of a component-based Web document, the latter one allows to manipulate the immediate subcomponents (i.e. child components) of a composite component in more detail.

The main application scenario of the *structure editor* (see Figure 5.11) is the creation of a component-based Web document "from scratch". Starting from an empty document component, authors can easily specify its internal structure by defining its subcomponents (and the subcomponents of those subcomponents) in a visual way. The available component types to be included (created) are visualized on the bottom part of the editor and can be placed into a container component (or moved from one container to another) by using "Drag&Drop" mechanisms. During this composition process, the integrity constraints dictated by the component-based document format are strictly taken into account. For instance, a media component has to be always contained by a content unit component.

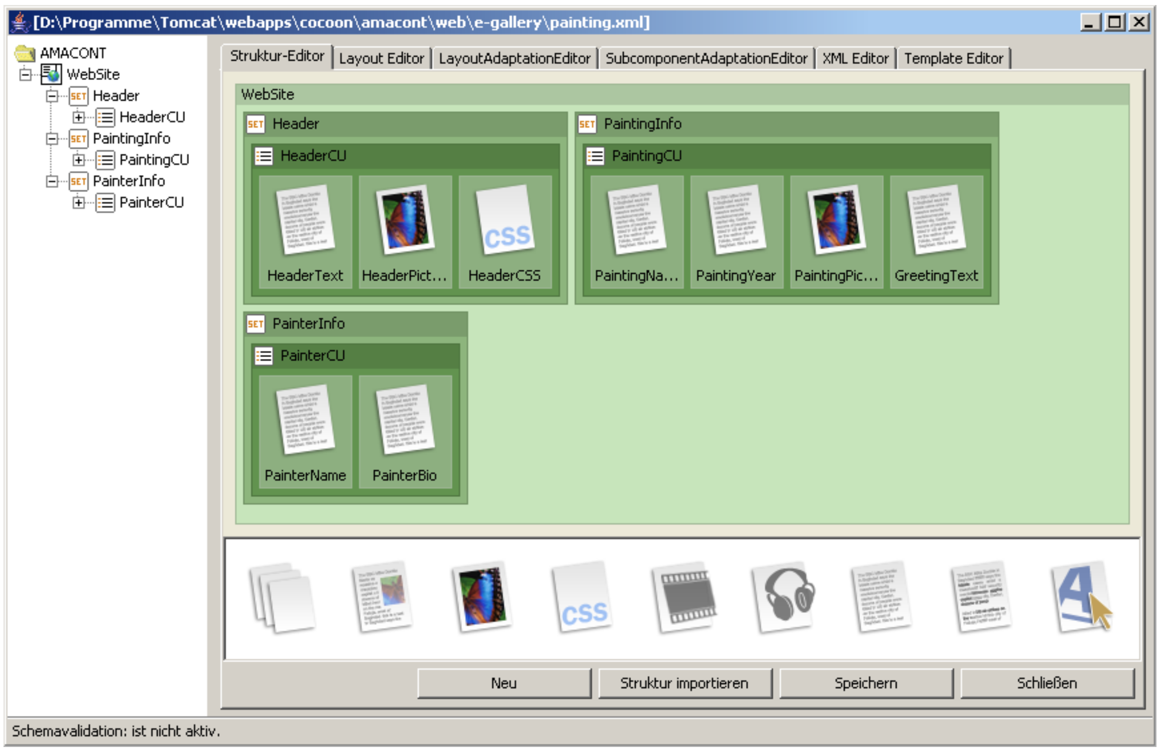


Figure 5.11: Structure editor

As a matter of course, the structure editor is also ideal for visualizing (or manipulating) the composition hierarchy of an already existing component hierarchy. Furthermore, it allows authors to directly access any component contained in a component-based Web document. By double-clicking on an arbitrary component the appropriate editors assigned to it are automatically activated. As an example, the activation of an image component invokes the image editor (shown in Figure 5.8) in a modal editor window. Moreover, the structure editor is not only associated with top-level document components, but also with all kinds of composite components. In the latter case it only visualizes the subcomponent tree of the currently selected composite component, thus providing partial views on (fragments of) complex component hierarchies.

Whereas the structure editor is ideal for creating and visualizing component hierarchies of arbitrary depth, there are also cases when component authors would like to deal only with the immediate subcomponents of a composite component. A typical use case is the population of a component structure with concrete media components or the definition of adaptation variants: the subcomponent structure of a component might vary according to a given user model or context model parameter. For this reason the *subcomponent editor* shown in Figure 5.12 has been developed. It is associated with composite components (both document components and content unit components) and visualizes their immediate subcomponents as an unordered list.

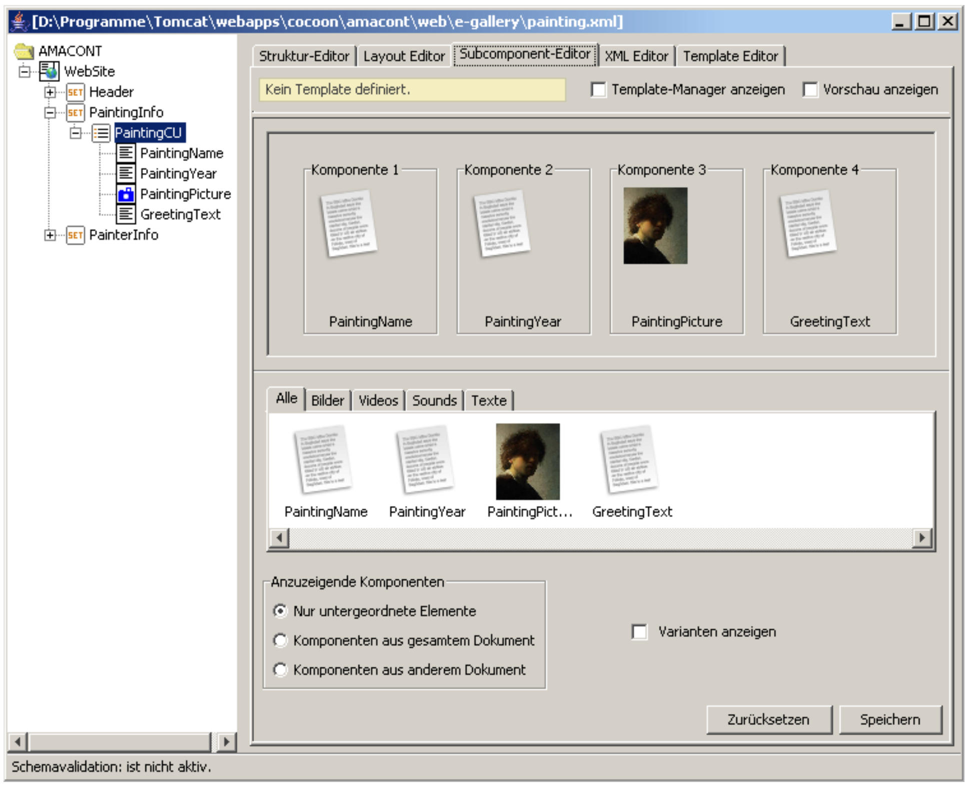


Figure 5.12: The subcomponent editor

According to the running example described in this chapter, the composite component shown in Figure 5.12 contains information on a painting. In this case it is put together from several subcomponents: the painting's name, textual description, creation year, and a greeting text addressing the user. As a matter of course, the subcomponent editor can be also switched to template mode. Taking the example, one could connect it to a database query in order to present dynamic information on paintings selected at run-time. What is more, a composite component may aggregate both static and dynamic (i.e. template-based) subcomponents. For instance, while the media items describing the actual painting could be dynamically retrieved, the greeting text addressing the user is constant for all paintings and is therefore a static component instance. Finally, similar to media editors, the subcomponent editor also facilitates the creation of alternative component structures and selection methods.

As an example, the author might provide additional information on paintings for expert users, or insert some multimedia material for devices capable of presenting it.

5.2.3.2 Editors for Creating Hyperlink Structures

The AMACONTBuilder provides two editor modules for graphically authoring hyperlinks and hyperlink structures [Niederhausen 2006]. While the *graph editor* facilitates the visualization of a component-based Web presentation's overall hypermedia structure, the *hyperlink editor* supports the creation of single hyperlinks or hyperlink lists.

The graph editor shown in Figure 5.13 presents the hypermedia structure of a component-based Web presentation in a graph-like way[7]. The nodes of the graph correspond to top-level document components. A directed edge between two such nodes means that there is at least one hyperlink component connecting them. Still, for better readability, "parallel hyperlinks" between two documents are merged to one edge, i.e. only the connectivity of the corresponding nodes is represented. Furthermore, author can use "filter functions" to display only hyperlinks of a given type (e.g. typed links, template-based links or adaptive links). While mainly serving for visualization purposes, the graph editor is an ideal starting point for further authoring operations. When clicking on a node, the AMACONTBuilder opens the appropriate document that can be then edited in more detail.

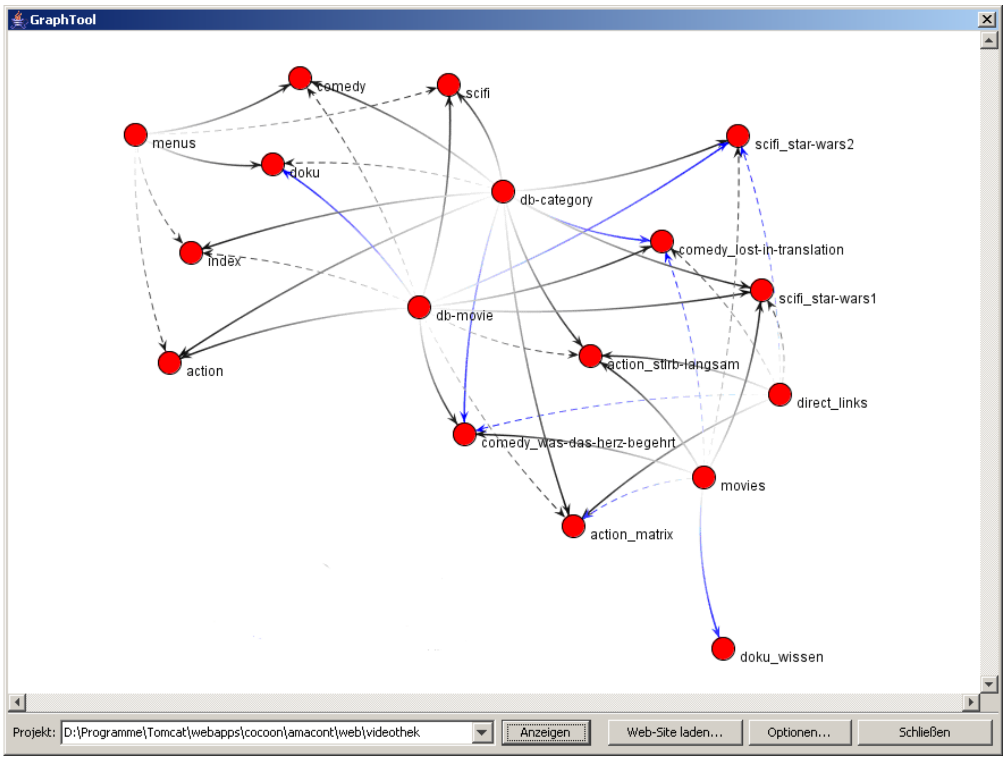


Figure 5.13: The graph editor

In order to create and manipulate single hyperlinks or hyperlink lists the so-called *hyperlink editor* was developed. However, instead of being a stand-alone module for authoring abstract

[7]In order to present the graph editor based on a larger example, Figure 5.13 illustrates the navigation structure of a component-based video rental shop.

hyperlink components independent of the underlying content, it is used in combination with the editor modules aimed at the creation of components that serve as the starting anchor of a hyperlink component. As an example, the author of a text component can mark an arbitrary text fragment and define a hyperlink starting from that component. Furthermore, it is also possible to visually create complex navigation bars (i.e. composite components consisting of a number of anchors and corresponding links) in form of specific document components. As a matter of course, the resulting structures are stored as separate content components (e.g. media component) and hyperlink components. However, this integrated view supports for a more intuitive way of working for component authors.

Similarly to other editor modules, the hyperlink editor can be also switched to the template mode. Again, different parameters of a hyperlink (such as its target, anchor text, or even the request parameters attached to it) can be customized by appropriate data queries. Furthermore, it is also possible to specify the adaptive behavior of a hyperlink component. Four basic adaptation techniques are supported: *link hiding*, *link removal*, *link disabling*, and *link annotation*. The usage of a given adaptation technique can be bound to a condition that references the context model. Again, such conditions can be visually defined by using the profile browser (see Figure 5.9).

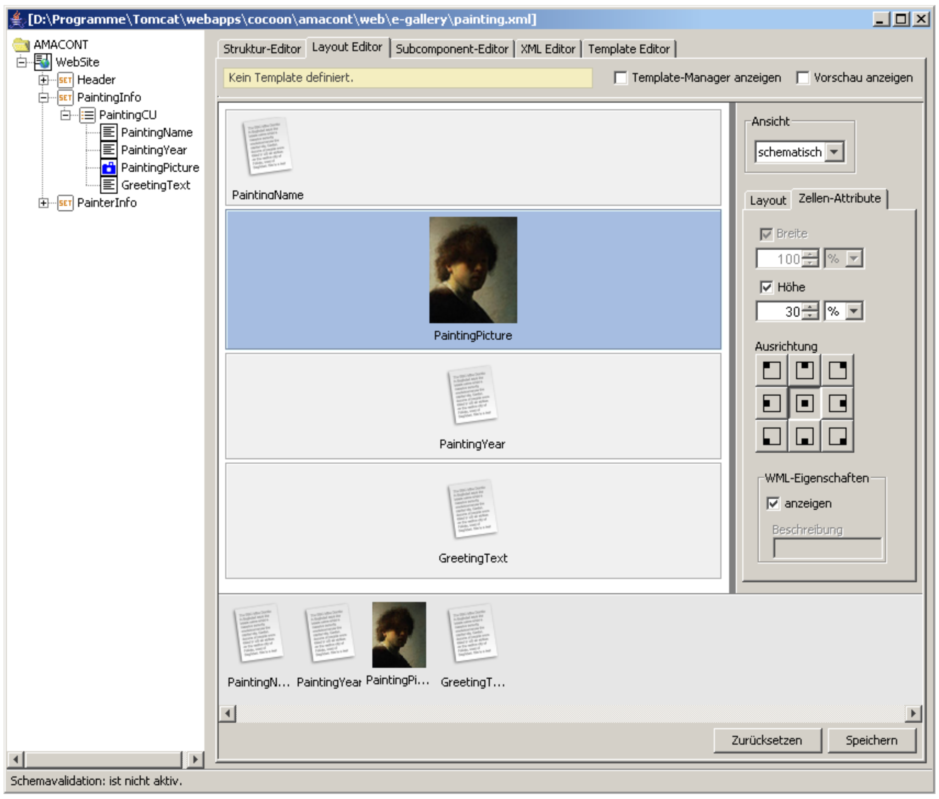
5.2.4 Editors for Presentation Authoring

Finally, a number of editor modules for configuring the presentation layout of component-based Web documents have been developed. As discussed in Section 5.1 they serve two purposes: the definition of a component's abstract layout, and the configuration of its styling by using CSS. These tasks are facilitated by the *layout editor* and the *CSS editor*, respectively.

Figure 5.14 shows a screenshot of the layout editor. It facilitates the assignment of layout managers to components and to configure their various attributes in an intuitive visual way. Layout managers are visualized by means of a grid that can be filled by icons representing subcomponents. Various mouse dragging and "Drag&Drop" techniques have been realized in order to perform most operations graphically, such as resizing the grid, placing subcomponents into grid cells, changing their alignment, etc. Besides, various input fields for fine-tuning all possible layout attributes (both layout attributes and subcomponent attributes) can be found on the right editor pane. Furthermore, a preview function for testing the current layout in XHTML has been developed, too.

Figure 5.14 depicts a possible abstract layout for the component presenting paintings (see Section 5.2.3). Based on vertical BoxLayout, the content pieces describing a painting are arranged in a linear structure. Note that even though the figure depicts a concrete "painting instance", this editor can be switched to template mode, as well. However, as far as iterative templates (i.e. templates with an unpredictable number of subcomponents) are concerned, only the layout managers GridTableLayout (with only one predefined dimension) or BoxLayout (with an undefined number of subcomponents) can be utilized. Furthermore, layout adaptations can be easily specified by the creation of appropriate layout alternatives and the definition corresponding selection methods.

On the other hand the CSS editor aims at defining the design of the resulting application. It is a simple (media component) editor module allowing for loading CSS files as well as for manipulating their style definition entries. It allows authors to select different elements of the respective output format and configure their layout attributes (e.g. font sizes, colors, text decorations, etc.). The resulting definitions are stored as CSS media components. Of course, similar to the other editors, component authors can again define alternative CSS variants


[D:\Programme\Tomcat\webapps\cocoon\amacont\web\e-gallery\painting.xml]

Kein Template definiert.

Ansicht: schematisch

Breite: 100 %

Höhe: 30 %

WML-Eigenschaften: anzeigen

Schemavalidation: ist nicht aktiv.

Figure 5.14: Layout editor

and assign them to a specific context model parameter.

5.2.5 The XML editor

The main goal of the AMACONTBuilder is to provide a set of *visual* editor modules that allow component authors to abstract from the the component-based document format's underlying XML grammar. Still, in order to exploit the full range of language features (e.g. the ones for which there are no graphical authoring plugins available, yet) or to debug the XML code generated by other visual editor modules, it is also required to provide "low-level" source code editing support. For this purpose, the built-in XML editor provided by the AMACONTBuilder's underlying plug-in architecture can be utilized[8]. Inspired by the functionality offered by professional XML tools (e.g. XMLSpy [@XMLSpy]), it provides a number of useful features, such as:

- **syntax highlighting** with adjustable font types and sizes, as well as configurable color schemes for different parts of XML documents, like tag names, attribute names (and values), comments, DOCType declarations, etc.
- **partial views** on complex XML documents by displaying only the XML subtree that belongs to the node (component) being currently selected in the navigation frame
- **automatic indentation** of XML tags to an easily readable "pretty-print" layout

[8]Note that the XML editor was already presented in Figure 5.7.

- **well-formedness checking** as well as **validation** of XML documents based on associated XML schemas with appropriate textual feedback on found errors
- **schema-based code completion** for XML (sub)elements and attributes based on well-formedness rules as well as default values provided by a corresponding XML schema

While being primarily used to edit XML documents based on the component-based document format, the XML editor is a generic tool that is applicable for arbitrary XML grammars.

5.2.6 Implementation Issues

The AMACONTBuilder was implemented in Java and is a modular authoring tool allowing for creating and editing arbitrary XML documents. As mentioned above, it is based on a generic framework [Chevchenko 2003] that can be extended by graphical editor plug-ins for visually authoring specific types of XML content. In order to provide programmatic access to all kinds of XML data, the AMACONTBuilder utilizes a flexible internal object model which is based on JDOM [@jdom]. This generic object model was extended by specific classes that provide an API for efficiently manipulating adaptive Web components. That is to say, component-based adaptive Web documents are automatically parsed into a hierarchy of component specific objects when they are opened by the AMACONTBuilder.

The UML diagram shown in Figure 5.15 depicts the most important classes of the object model hierarchy that are specific to the component-based document format. The root of this object hierarchy is the class *AmacontNode* providing a number of generic methods for component manipulation. The specific classes representing concrete component types inherit from it and declare their (additional) specific attributes and methods, respectively. As an example, the developer of an editor plug-in dealing with image components can utilize predefined methods of the class AmaImageComponent for getting and setting image metadata, for creating image component variants and selection methods, etc.

Note that the utilization of such an object model has different advantages. First, plug-in programmers can use a high-level API for manipulating adaptive Web components and do not have to bother about their concrete underlying XML-based format. Second, this solution provides also more robustness regarding to modifications of the utilized XML languages. During the "evolution" of this dissertation different changes to the component model's XML-based description language were made, especially in order to provide less redundant descriptions and better performance in the document generation process. Still, as the plug-ins of the AMACONTBuilder work on an internal object model, it was sufficient to adjust the mappings between that model and the XML-based formats, not needing to modify the application logic of specific plug-ins.

The editor modules (plug-ins) of the AMACONTBuilder have to implement a corresponding interface class[9]. It specifies a number of generic methods, e.g. for accessing the underlying object model, to set up the editor's graphical user interface, to check if the editor performed modifications on the object model, to write back these modifications to the object model, etc. Furthermore, whenever an editor should be extended with the capability to create and manage adaptation variants of the edited component type, it can inherit from a specific predefined class[10] that already implements this functionality in a generic way.

The assignment of editor modules to a specific component type is managed by an XML-based configuration file called `amaplugins.xml`. Listing 5.1 depicts a "fragment" of this

[9]de.tudresden.inf.amacont.plugins.EditorPlugin [Müller et al. 2005]

[10]de.tudresden.inf.amacont.plugins.AbstractAdaptableEditor [Niederhausen 2005a]

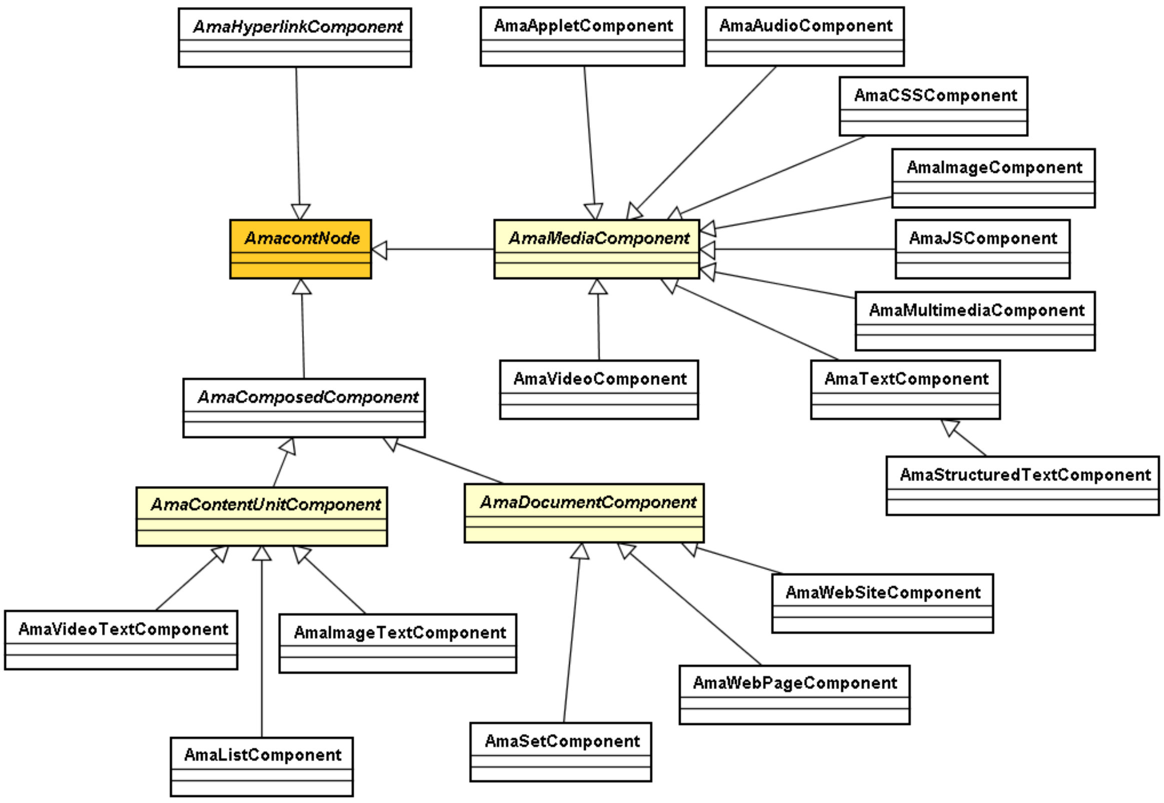


Figure 5.15: AMACONTBuilder object model

file aimed at associating the Java class implementing the image editor with all objects of the type AmaImageComponent (from the AMACONTBuilder's object model). The configuration specifies the plug-in's name, its version number, author, a short textual description, the class that implements it, and the elements from the object model to which it is assigned. While in this case the image editor is associated only with the elements of the type aco:AmaImageComponent, note that one can also associate an editor plug-in (e.g. the XML editor or the structure editor) with a number of classes of the object model.

```xml
<Plugin type="editor">
  <Name>amacont.imagecomponent.editor</Name>
  <Version>1.0</Version>
  <Author>Matthias Niederhausen</Author>
  <Description>Image Editor</Description>
  <Class>de.tudresden.inf.amacont.plugins.ImageEditor</Class>
  <Element name="aco:AmaImageComponent" />
</Plugin>
```
Listing 5.1: Assignment of an editor module to a component type

For further detailed information about the AMACONTBuilder's architecture, internal object model, configuration, and the implementation of its various editor modules the reader is referred to [Fiala et al. 2005, Chevchenko 2003, Niederhausen 2005b, Niederhausen 2006, Tietz 2006].

5.3 From Component Authoring Towards Automatic Model-Driven WIS Generation

The first part of this chapter introduced the structured Hera-AMACONT methodology for the development of component-based adaptive Web applications. According to the design steps identified by Hera, it was shown how those concepts can be applied to systematically implement adaptive Web applications by creating, configuring, and aggregating components (or component templates). During the phases of design and implementation, a main focus was put on the consideration of different adaptation issues (concerns). Thus, a possible model-based authoring process for component developers was provided.

Basically, there are two possibilities to put such an identified process model into practice. In the first case component authors can use the existing (and already introduced) modules of the AMACONTBuilder. Considering the different designs and keeping in mind the identified authoring steps, they can then build complex adaptive Web applications by creating, configuring, and composing reusable components (or component templates). The advantage of this "manual mapping" approach is the utilization of a graphical authoring tool that allows for visually editing component properties in detail. Furthermore, in a similar way, it is possible to proceed according to the steps identified by another design or process model. However, such a manual mapping also means that the applied design model serves mainly as a "guideline" (or documentation) for component authors, i.e. its semantics is not explicily exploited when creating component-based adaptive Web applications.

The second possibility is take a further step from model-based to model-driven component engineering and add automation to the overall process of design and implementation. The reason for this is the fact that, besides graphical representations (in form of diagrams), high-level design models can be also expressed in a formal way. As an example, we again consider Hera that provides RDF(S)-based specifications of its different design issues. By explicitly describing model semantics, such specifications can be used for the automatic model-driven generation of a corresponding implementation. This approach is also pursued by the Hera Presentation Generator (HPG [Frasincar et al. 2005]), a tool aimed at creating and implementing Hera models. However, prior to the work described in this dissertation, Hera's presentation model was not formalized, nor was adaptation at the presentation level addressed and implemented in the Hera tools. Moreover, HPG uses conventional Web document formats (such as HTML) as its implementation model, not allowing to reuse the generated implementation artefacts in a component-based manner. Thus, this section aims at the automatic, model-driven generation of component-based adaptive Web presentations from high-level design model specifications. This will allow to combine the modeling power of the (extended) Hera design method with the flexible reuse, presentation, and adaptation capabilities provided by the component-oriented document format and its publication architecture.

To achieve this goal, two requirements have to be fulfilled. First, all design models describing an adaptive Web application have to be expressed in a formal way. Second, a series of model-driven transformation steps is needed to automatically map these model descriptions to a component-based implementation. To meet these requirements, this section provides a facility for the (currently missing) RDF(S)-based formalization of a Web application's presentational aspects (as well their as corresponding adaptation issues) at model level. Bridging the gap between the application model and the actual implementation, this formalization will be utilized to automatically generate an adaptive component-based presentation.

The rest of this section is structured as follows. In Section 5.3.1 the concept of abstract layout managers (from the component-based document format) is adopted to the

Hera-AMACONT presentation model, and its RDFS-based description is provided. According to this formalization, Section 5.3.2 describes how high-level model specifications can be automatically transformed to an implementation utilizing the component-based document format and its document generation architecture. The XML-based transformation steps are explained in detail, and the resulting methodology is exemplified by a prototype application. Furthermore, selected aspects of dynamic adaptation provided by the overall presentation generation process are also discussed.

5.3.1 RDFS-based Specification of the Hera-AMACONT PM

In order to formalize the Hera-AMACONT presentation model, an RDFS-based specification of the PM schema was developed [Fiala et al. 2004a]. The basic idea behind it was to transfer (i.e. adopt) the concept of abstract layout managers from the component-based document format to the model level. The layout manager concept was already introduced in detail in Section 4.3.2. As mentioned there, layout managers aim at describing the spatial arrangement of components in a client-independent way, thus allowing to abstract from the exact presentation capabilities (e.g. window size) of a concrete browser display.

Note that in Section 5.1.4 significant analogies between document components and Hera slices were mentioned. Furthermore, a "recipe" for the mapping of slices to components was also introduced. Taking advantage of these analogies (and the fact that both slices and components rest upon XML technologies), it is thus straightforward to transfer the concept of layout managers to the model level. That is to say, the basic idea is the assignment of abstract layout descriptors to Hera slices in order to specify the arrangement of their subslices in an implementation-independent way. As a consequence, the RDFS-based PM formalization supports two mechanisms: 1) the definition of model-level layout managers and 2) their assignment to AM slices. A slice with an associated layout manager constitutes a so-called *region*: an abstraction for a rectangular part of the display area where the content of that slice will be displayed. These mechanisms will now be explained based on the running example used throughout this chapter.

Figure 5.16 depicts a schematic graphical presentation diagram (PD) of the running example's starting page (presenting painting techniques). Note that it is based on the corresponding application diagram (see Figure 5.3) which is extended by additional presentation specific information, i.e. the presentation diagram acts as an *overlay* of that application diagram aimed at specifying its layout[11]. As an example, the dark rectangle "behind" the top-level slice depicts the top-level region representing its contents. It utilizes the layout manager instance BoxLayout1 for the spatial arrangement of the corresponding subslices (i.e. of the regions assigned to them). The simple RDF code snippet for specifying this layout assignment is shown in Listing 5.2.

```
<Slice rdf:about="#Slice.technique.main">
    <layout rdf:resource="#BoxLayout1"/>
</Slice>
```

Listing 5.2: Layout assignment to a slice

The specific attributes of this layout (BoxLayout1) are also schematically shown in the diagram by means of arrows that are labeled with their names and corresponding values. Both

[11]Similar to the AM aimed at grouping the concepts of the CM to slices, the PM leans itself on the AM by further defining the spatial arrangement of those slices.

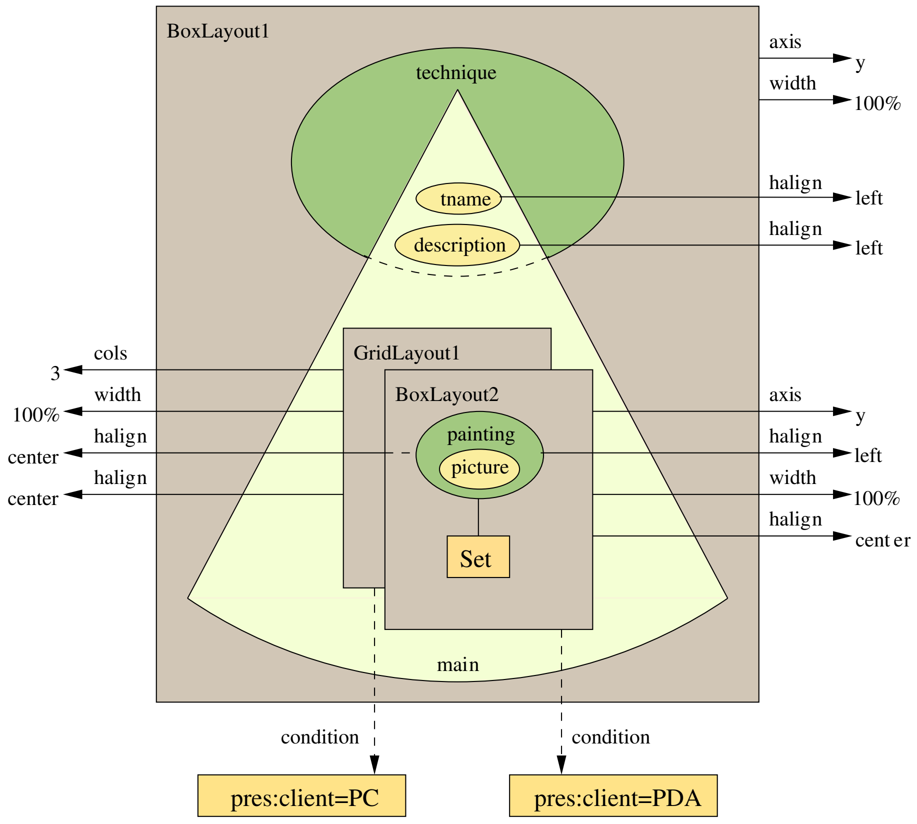


Figure 5.16: A Hera-AMACONT PM example [Fiala et al. 2004a]

attributes describing the overall layout and attributes specifying the arrangement of each referenced subslice (subregion) can be defined. Since these attributes were taken from the component-based document format, the reader is referred to Section 4.3.2 for more detailed information.

In this particular case the subslices (subregions) of the top-level slice (top-level region) are arranged in a vertical way. Concretely, these are the three subregions associated with the subslices `tname`, `description` as well as the link list pointing to the paintings representing the actual painting technique. The RDF-based representation of this layout definition is shown in Listing 5.3.

As already mentioned, layout descriptions of a given region describe only the spatial arrangement of its immediate subregions. Whenever these subregions also contain nested subregions, their appropriate layouts have to be additionally specified. In Figure 5.16 this is the case for the link list (`set-element`) pointing to the associated painting slices. The corresponding layout assignment is specified by the RDF code shown in Listing 5.4.

Note the attribute `pres:condition` that allows to declare simple *adaptation conditions* that reference parameters from the usage context. Whereas for example the paintings exemplifying the presented painting technique are arranged on a desktop in a tabular way (GridTableLayout1), the small screen size of PDAs requires to adjust them below each other (BoxLayout2). In Figure 5.16 these conditional layout assignments are visualized by the two overlapping regions as well as the two dashed arrows pointing to the rectangles containing their conditions.

```
<BoxLayout rdf:ID="BoxLayout1">
  <axis>y</axis>
  <width>100%</width>
  <subregion-ref>
       <subregion pres:align="left">
           <slice-ref rdf:resource="#Slice.technique.tname"/>
       </subregion>
  </subregion-ref>
  <subregion-ref>
       <subregion pres:align="left">
          <slice-ref rdf:resource="#Slice.technique.description"/>
       </subregion>
  </subregion-ref>
  <subregion-ref>
       <subregion pres:align="center" pres:valign="top">
          <set-element-ref rdf:resource="#SetOfLinks_1"/>
       </subregion>
  </subregion-ref>
</BoxLayout>
```

Listing 5.3: High-level BoxLayout definition example

```
    <Set-element rdf:about="#SetOfLinks_1">
       <layout rdf:resource="#GridTableLayout1"
             pres:condition="pres:client='Desktop'"/>
       <layout rdf:resource="#BoxLayout2"
             pres:condition="pres:client='PDA'"/>
    </Set-element>
```

Listing 5.4: Layout assignment to Set elements

Finally, Listing 5.5 presents the RDF code specifying the `GridTableLayout1` layout manager. According to this, the painting pictures (acting as the link anchors to the painting slices) are arranged in a tabular way.

```
  <GridTableLayout rdf:ID="GridTableLayout1">
    <rows>2</rows>
    <width>100%</width>
    <height>80%</height>
    <space>10</space>
    <border>0</border>
    <header_align>xAxis</header_align>
    <subregion-ref>
       <subregion pres:align="center" pres:valign="center" ... >
         <slice-ref rdf:resource="#Slice.painting.picture"/>
       </subregion>
    </subregion-ref>
  </GridTableLayout>
```

Listing 5.5: High-level GridTableLayout definition example

Note that in contrast to static components defined at instance level, Hera-AMACONT layout assignments have to be specified at *schema level*. Due to the dynamic nature of WIS applications, this means that the number of items in an access element (e.g. the number of paintings exemplifying a given painting technique) is not known at design time. In such cases one should use either a `BoxLayout` with an undefined number of cells or (as shown in our particular example in Listing 5.5) a `GridTableLayout` so that only one of its dimensions (`columns` or `rows`) is predeclared. The missing dimensions (in this particular example the number of columns in the resulting table) are automatically computed at run time (see later in Section 5.3.2.2).

5.3.2 Automatic Generation of a Component-based Implementation

After the RDF(S)-based specification of the Hera-AMACONT PM, all design models can be expressed in form of RDF(S)-based specifications. Given these specifications, it is now shown how they can be utilized to automatically generate a component-based adaptive Web application that can then be published (and adapted) for different device, user, and context profiles. Furthermore, a prototypical implementation of this automatic hypermedia generation process is presented.

First, the general transformation architecture is described in overview. Then, the automatic model-driven generation of adaptive document component structures as well as their dynamic publishing process based on the actual usage context are explained. Finally, selected issues of dynamic adaptation (adaptivity) provided by the resulting component-based implementation are illustrated.

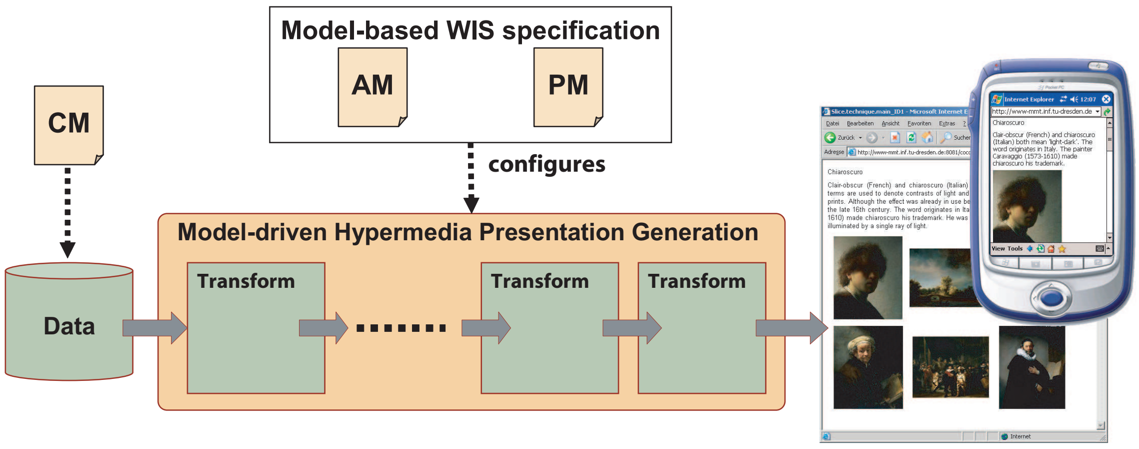

Model-based WIS specification
AM
PM
CM
configures
Model-driven Hypermedia Presentation Generation
Data
Transform
Transform
Transform

Figure 5.17: Model-driven WIS generation process overview

Figure 5.17 gives a general overview of the targeted model-driven Web presentation generation process. As depicted there, this general architecture consists of a series of transformations that convert some input data to a hypermedia (Web) application. The input data represents structured information that corresponds to the application domain (here defined by the CM). The transformations are configured by a number of models that dictate the application's navigational and presentational behavior. In the case of Hera-AMACONT, these are the AM and the PM, each being "enriched" with corresponding adaptation definitions[12]. We note, however, that this general architecture is also characteristic for other model-driven approaches.

Figure 5.18 depicts the concrete envisioned data transformation process in more detail. Its

[12]Note that the Hera project provides graphical tool support for creating conceptual and application models [Frasincar 2005].

input is a conceptual model instance (CMI), an RDF document (or repository) that contains all the data (among others references to the the media objects) underlying the conceptual model (CM) of a Web application. Thus, as already described in Section 5.1.2, the specification of a Web application's conceptual model has to be accompanied by the creation or retrieval of media instances that represent the identified concept attributes, i.e. constitute the application's underlying data. To cope with the specifics of the component-based document model, it is assumed that those media instances are annotated with appropriate metadata.

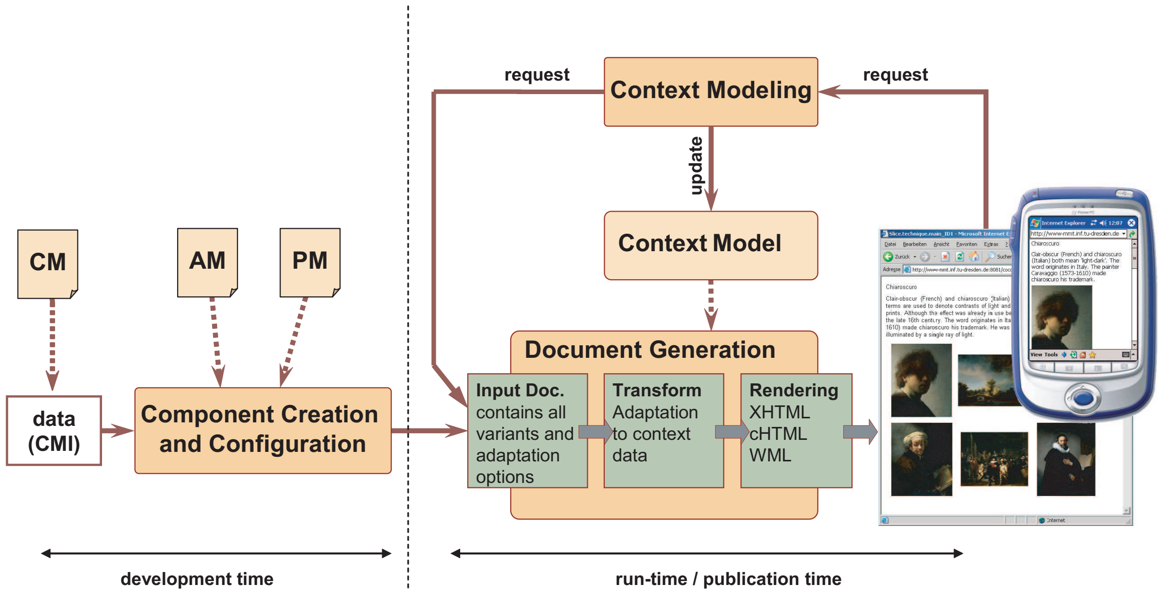

request
Context Modeling
request
update
Context Model
CM
AM
PM
Document Generation
data (CMI)
Component Creation and Configuration
Input Doc. contains all variants and adaptation options
Transform Adaptation to context data
Rendering XHTML cHTML WML
development time
run-time / publication time

Figure 5.18: Component configuration and publication

In order to deliver Web users an adaptive hypermedia presentation on top of this data, two tasks have to be performed. First, a component-based Web presentation has to be created (see the left part of Figure 5.18). During this phase of "component creation and configuration", the appropriate models (i.e. the AM and the PM) describing the application have to be taken into account. Second, according to the actual user's request and current usage context, the created component structures have to be sent to the document generation pipeline (see the right part of Figure 5.18). While the first task is performed only once (i.e. once for each application), the second one is executed for each user request. However, both tasks can be performed automatically, i.e. no additional participation of the Web designer/developer is needed. The following sections describe the corresponding transformation steps in detail.

5.3.2.1 Model-driven Component Creation and Configuration

As described above, the first part of the overall transformation process aims at the model-driven generation of a component-based adaptive Web application. It was designed and prototypically implemented in a static and a dynamic variant, indicating whether the generated adaptive component structure consists of component instances or component templates. The first variant creates a static component-based adaptive hypermedia presentation, i.e. a network of adaptive document component instances for all the underlying data (e.g. in our running example for all instances of painting techniques, paintings, or painters) at once. Still, the term "static" refers only to the data offered by the resulting hypermedia presentation, it can be still dynamically adjusted to different usage contexts at the later document generation process (see Section 5.3.2.2). While providing better performance for document generation, the shortcoming of this variant is that the underlying data can not be altered at run-time. On the other hand, the dynamic variant creates a structure of adaptive document component templates (each corresponding to a top-level slice) that are subsequently dynamically filled with volatile data at each request only during document generation. Since (apart from small deviations) the two transformation variants are quite similar, we describe here the static one.

This “component creation and configuration” process is parameterized by the application model (AM) and the presentation model (PM) and consists thus of two phases. The first phase takes the AM into account to convert the original data (CMI) to a component-based Web document structure (still without layout descriptors). This phase consists of two substeps.

In the first substep, a so called application model instance (AMI) is generated. It is an RDF document, an instantiation of the application model (AM) with the data available from the original conceptual model instance (CMI). For this transformation substep existing modules from the Hera Presentation Generator (HPG [Frasincar et al. 2005]) provided by the Hera project have been also utilized. However, in this particular scenario, the delivered application model instance is still unadapted, i.e. it contains appearance conditions referencing (both static and dynamic) parameters from the context model. As described later, this enables to use the various (dynamic) adaptation mechanisms provided by the component-based document format and its document generation architecture.

In the second substep, the incoming application model instance is automatically converted to a document component structure. Based on the mapping “recipe” described in Section 5.1.4, the corresponding XSLT transformation stylesheets take the analogies between slices and adaptive document components automatically into account. Whenever the AM specified appearance conditions, the resulting component structure also contains adaptation variants and selection methods (referencing the context model). Still, as the presentation model has not been considered at this stage, it does not contain layout specific attributes, yet.

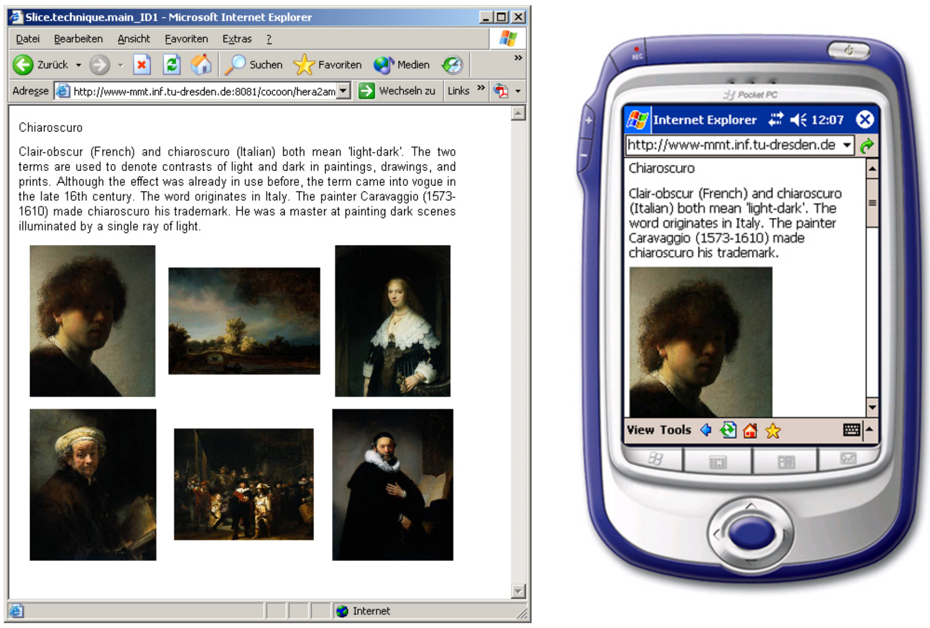
In the next phase of the “component creation and configuration process”, the layout attributes of the created component structure are configured according to the actual application’s presentation model (PM) description. Beginning at top-level document components and visiting their subcomponents recursively, the appropriate layout descriptors are added to the meta-information section of each component’s header. Since the layout manager attributes of the Hera-AMACONT PM rest upon the layout concepts of the component-based document format, this mapping is a straightforward process [Fiala et al. 2004a]. Yet, for set elements (containing a variable number of subelements depending on the actual size of the CMI) the concrete dimensions of `BoxLayout` or `GridTableLayout` layout managers have to be computed at run time.

Whenever the PM contains adaptation conditions, these are translated to component layout variants and corresponding selection methods. Thus, for each layout assignment condition defined in the PM a separate layout manager variant is created. Furthermore, a selection method according to the switch-case or the if-then-else mechanism is composed (according to Section 4.3.1). Again, all transformations are implemented as XSLT stylesheets.

Note that the output of this component creation and configuration process is a document component structure (or network) containing both adaptation variants as well as as adaptive layout descriptors.

5.3.2.2 Document Generation

As shown in the right part of Figure 5.18, the generated component structures manifest a component-based implementation of the designed hypermedia application and serve as the input data for the document generation pipeline. As described in detail in Section 4.5, they are subdued to a series of data transformations that are triggered by the user's actual request, parameterized by the current context model, and result in an adapted hypermedia presentation. Based on these transformations, Figure 5.19 shows two versions of the generated hypermedia presentation, one for desktop browsers and another one for PDAs. Note that (as specified above) the limited display size of the handheld does not allow for a tabular arrangement of painting pictures, i.e. they are displayed in a linear way.


Slice.technique.main_ID1 - Microsoft Internet Explorer
Datei
Bearbeiten
Ansicht
Favoriten
Extras
?
Zurück
Suchen
Favoriten
Medien
Adresse
http://www-mmt.inf.tu-dresden.de:8081/cocoon/hera2am
Wechseln zu
Links
Internet
Pocket PC
Internet Explorer
12:07
http://www-mmt.inf.tu-dresden.de
View
Tools

Chiaroscuro

Clair-obscur (French) and chiaroscuro (Italian) both mean 'light-dark'. The two terms are used to denote contrasts of light and dark in paintings, drawings, and prints. Although the effect was already in use before, the term came into vogue in the late 16th century. The word originates in Italy. The painter Caravaggio (1573-1610) made chiaroscuro his trademark. He was a master at painting dark scenes illuminated by a single ray of light.

Chiaroscuro

Clair-obscur (French) and chiaroscuro (Italian) both mean 'light-dark'. The word originates in Italy. The painter Caravaggio (1573-1610) made chiaroscuro his trademark.

Figure 5.19: Generated hypermedia presentation [Fiala et al. 2004a]

5.3.3 Adaptivity Support

The document generation process described in Section 5.3.2.2 provides for static adaptation (or adaptability) by taking the user's actual usage context into account. However, it also supports selected issues of dynamic adaptation (adaptivity), i.e. the kind of adaptation included in the generated adaptive hypermedia presentation.

As defined in Section 2.2.1, adaptivity is the capability of a hypermedia presentation to dynamically reconfigure itself according to a dynamic usage context that is continually changing during the user's browsing session. These changes can originate from different "events", such as user interactions or even changes to the user's environment. To implement adaptivity, such events have to be acquired, the usage context has to be updated, and the Web presentation has to be regenerated, respectively.

Based on our running example application, Table 5.2 summarizes a number of characteristic examples for adaptability and adaptivity to be considered in the different models specifying a Web application. To be more accurate, it presents factors (context parameters) that can be the basis for adaptation. While the parameters in the left column (describing the state of the user, his device, and environment) are constant for a single user session, the parameters in the right column can change (within the session) according to the user's interaction behavior. For instance, whereas the color depth of the user's device or its capability to present images influence the data to be presented statically, the available bandwidth can fluctuate and lead to a dynamic adaptation of media instances. Similarly, while the user's expertise level might be considered as constant, his knowledge on specific painters might change dynamically when browsing through the presentation described throughout this session. Finally, while the user's preferences for design elements like font types, sizes, or colors can be viewed as static factors, the current screen size can be influenced dynamically during a user session.

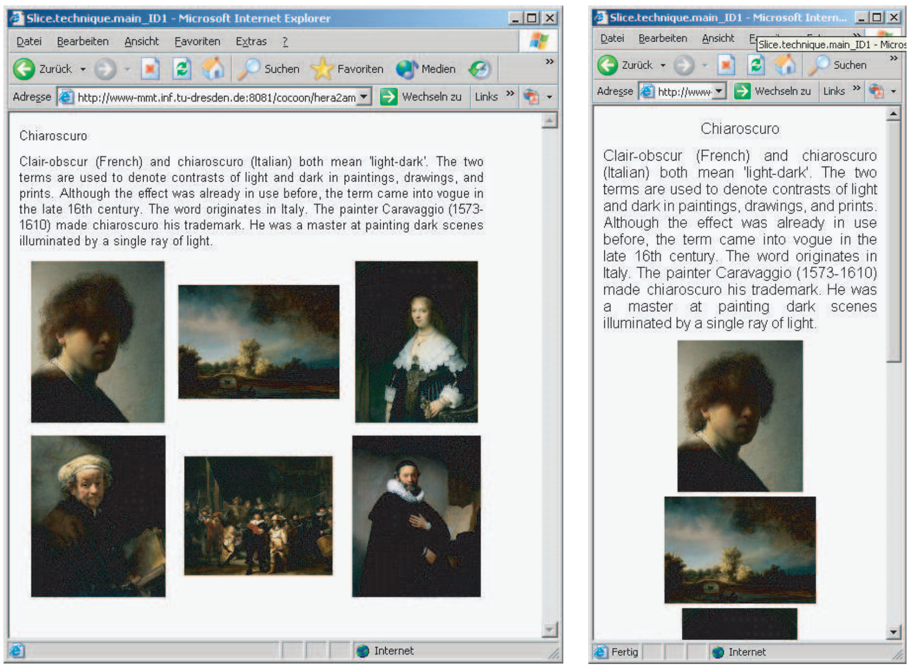
	Adaptability	**Adaptivity**
CM/MM	device type (`Device`) media types supported by end device (`ImageCapable`)	dynamically changing bandwidth (`Bandwidth`)
AM	user's expertise (`ExpertiseLevel`)	user's changing knowledge on painters (`Biography`)
PM	user's layout preferences (`PreferredCSS`)	resizing the browser's window (`InnerSizeX`)

Table 5.2: Adaptability/adaptivity examples across the design and implementation phases

As described in Section 4.5.3, the document generation architecture of the component-based document format utilizes an extensible context modeling framework. Providing different kinds of context modeling components (e.g. for device modeling, location modeling, or for user modeling), it allows to automatically update selected parts of the context model [Hinz and Fiala 2005]. As the focus of this chapter was put on the RDF-based specification of the Hera-AMACONT presentation model, it is now explained how dynamic adaptation specified in the PM can be realized.

When the user resizes his browser window, a JavaScript function aiming at determining the new dimensions and sending them to the server is executed. It is part of a set of client-side scripts for acquiring device capabilities and interactions which is automatically included in the presentation during document generation. Via the next HTTP request (initiated by the user's navigation) this data is sent to the server where the device model is appropriately updated. For this purpose the corresponding device modeling component utilizes the profile-diff mechanism of UAProf implemented by DELI [Butler 2003], an API provided by the Apache Web server for maintaining and updating device and user profiles based on CC/PP. For more information of this device modeling issue the reader is referred to [Hinz et al. 2006].

After updating the device model, the request is further processed and the hypermedia presentation is regenerated (see Figure 5.18). According to the steps described in Section 5.3.2.2, a new component instance is taken and subdued to the document generation pipeline so that a new presentation according to the updated context is generated. Figure 5.20 shows how the generated XHTML presentation is dynamically updated when the user resizes his browser window during browsing.


Slice.technique.main_ID1 - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Medien
Adresse http://www-mmt.inf.tu-dresden.de:8081/cocoon/hera2am Wechseln zu Links
Internet
Slice.technique.main_ID1 - Microsoft Intern...
Fertig

Chiaroscuro

Clair-obscur (French) and chiaroscuro (Italian) both mean 'light-dark'. The two terms are used to denote contrasts of light and dark in paintings, drawings, and prints. Although the effect was already in use before, the term came into vogue in the late 16th century. The word originates in Italy. The painter Caravaggio (1573-1610) made chiaroscuro his trademark. He was a master at painting dark scenes illuminated by a single ray of light.

Figure 5.20: Presentation layer adaptivity

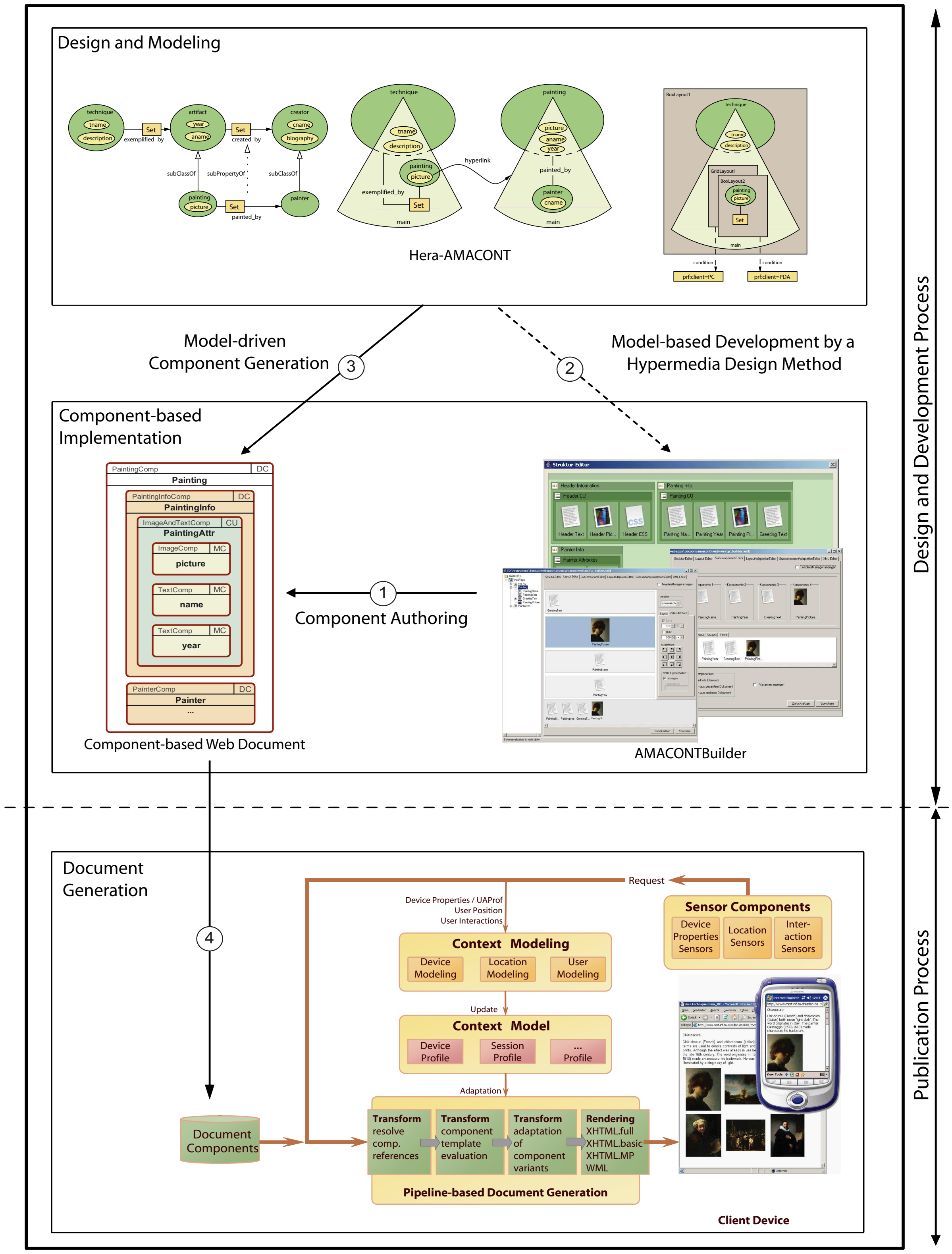
5.4 Summary and Realized Applications

This chapter dealt with the engineering process of component-based Web applications. Different application scenarios were briefly discussed, but the main focus was put on the structured development of data-driven adaptive Web presentations. It was shown how during the phases of design and implementation different aspects of adaptation can be dealt with. Furthermore, tools and mechanisms for authoring or generating component-based adaptive Web applications were explained and demonstrated. This section gives a summary of the proposed multi-stage development process and gives an overview of already realized component-based adaptive Web applications.

5.4.1 Summary of the Multi-stage Development and Document Generation Process

As a summary, Figure 5.21 recapitulates in a graphical way the different levels and possible activities involved in the development and publication process of component-based Web applications. It distinguishes between three main "levels": *design and modeling*, *component-based implementation*, and *document generation*. The numbered arrows represent selected Web engineering activities or processes. In the following they are discussed in more detail.

1. **Component Authoring:** The main focus of the activities described in this chapter lies on the development of adaptive Web applications from reusable implementation entities (document components) in a component-based way. Such a component-based Web document is schematically depicted on the left side of the second (middle) box in Figure 5.21. Though it could be created or edited by using a simple text or XML editor, the complexity of the component-based document format's underlying XML grammar (see Chapter 4) calls for visual authoring support. For this purpose the



Design and Modeling
Hera-AMACONT
Model-driven Component Generation
3
2
Model-based Development by a Hypermedia Design Method
Component-based Implementation
Component-based Web Document
1
Component Authoring
AMACONTBuilder
Document Generation
4
Document Components
Request
Device Properties / UAProf
User Position
User Interactions
Sensor Components
Device Properties Sensors
Location Sensors
Interaction Sensors
Context Modeling
Device Modeling
Location Modeling
User Modeling
Update
Context Model
Device Profile
Session Profile
... Profile
Adaptation
Transform resolve comp. references
Transform component template evaluation
Transform adaptation of component variants
Rendering XHTML.full XHTML.basic XHTML.MP WML
Pipeline-based Document Generation
Client Device
Design and Development Process
Publication Process

Figure 5.21: Overview of the multi-stage development process

authoring tool AMACONTBuilder (shown on the right side of the second box) was introduced. It offers a number of graphical editor modules for the implementation of adaptive Web applications by the visual creation, configuration, and interlinking of document components. In Figure 5.21 this activity of “Component Authoring” is depicted by the horizontal arrow in the second box (Nr. 1).

2. **Model-based Development by a Hypermedia Design Method:** As a format-specific tool aimed at the component-based implementation of adaptive Web applications, the AMACONTBuilder is not bound to a given process model or authoring workflow. Quite the opposite, it can be flexibly used in different development scenarios based on the requirements of the targeted application area. While for smaller Web presentations an ad-hoc approach is obviously suitable, the development of more complex adaptive Web applications necessitates to systematically take into account different application (and adaptation) concerns. As argued earlier, in this latter case component authors should proceed in a structured way, guided by an appropriate high-level model-based design of the hypermedia application.

   In Figure 5.21 this model-based component development process is depicted by the dashed arrow (Nr. 2) pointing from the level of “Design and Modeling” to the AMACONTBuilder. Considering the lines identified by a high-level model-based design methodology (in this case Hera-AMACONT) as a guideline, component authors implement adaptive Web applications by utilizing the appropriate modules of the AMACONTBuilder in a systematic way. The advantage of this approach is the usage of a graphical authoring tool that facilitates to manipulate component properties in detail. Furthermore, in a similar way it is possible to proceed according to the steps identified by another design method. Note that this approach is also typical for today’s software engineering practice. Guided by a number of (mostly UML-based) models specifying the targeted software application, software developers utilize visual development platforms and selected implementation (programming) languages for the realization of the required functionality.

3. **Model-driven Component Generation:** As discussed above, the AMACONTBuilder is a flexible authoring tool that facilitates to implement component-based Web applications independent from a given design or process model. Still, the observation can be made that by exploiting the explicit semantics described in a high-level design model it might be also possible to add automation to the process of design and implementation. That is to say, the resulting development process is not only model-based but also *model-driven*. This process of “Model-driven Component Generation” is depicted by the arrow Nr. 3 and was illustrated by example of the Hera-AMACONT methodology in Section 5.3 of this chapter. Based on the RDF-based formalization of the PM, it was shown how a component-based Web application can be automatically generated based on a sequence of design models aimed at describing the Web application’s semantic, navigation, and presentation behavior in a formal high-level way. The resulting Web application still contains all adaptation variants and can be thus later published for specific users, devices, and contexts.

   The main benefit of this approach is the specification of a Web application on a high-level of abstraction independent of its actual implementation. The required implementation-specific knowledge is integrated into the “model-to-component transformation process”, i.e. the automatic mapping guarantees that the semantics of the design models is appropriately incorporated in the generated component-based implementation.

Moreover, the usage of Semantic Web technologies in the models also provides a number of facilities (to be investigated in the future), such as efficient model reuse, interoperability, model checking, and validation, etc.

On the other hand, the specification of an adaptive Web application in form of high-level design models does not allow to describe its implementation and presentation aspects as detailed as a "lower-level" implementation-oriented authoring tool. Thus, as also depicted in Figure 5.21, a combined approach is also possible: the model-driven generation of a component-based Web document (arrow Nr. 3) and its further "refinement" with an implementation-centric authoring tool (arrow Nr. 1).

4. **Document Generation:** While the activities mentioned above aimed at the creation or model-driven generation of component-based Web documents, the arrow Nr. 4 depicts the process of their publication to a specific Web output format. For this purpose the document generation architecture presented in Section 4.5 is utilized, that acts as a "player" of the component-based document format. The documents created (or generated) on the higher levels serve as the input of this architecture. It automatically generates a Web presentation from this input, based on available information on the current user as well as his entire usage context. The resulting presentations are delivered to the user's browser in an appropriate Web output format, such XHTML, cHTML, or WML.

   Note that this document generation process can be also considered as a specific kind of model-driven transformation. Starting from a platform-independent (and context-independent) description of a Web application based on the proposed concern-oriented component model, it generates a Web page in a platform-specific (i.e. device-specific) Web implementation format. Thus, the overall authoring and publication framework depicted in Figure 5.21 can be viewed as a multi-stage model-driven transformation process between the three abstraction levels of design and modeling, component-based implementation, and format-specific Web presentations. Yet, while the transformation process between the Hera-AMACONT models and their component-based implementation (arrow Nr. 3) is performed only once for each application, the transformation from this component-based implementation to a specific Web output format is executed for each user request.

5.4.2 Realized Applications

This chapter exemplified the design and implementation process of component-based adaptive Web presentations based on a rather small example. However, during the "evolution" of the work presented in this thesis a number of component-based Web presentations have been developed. They vary in size and complexity, address different application scenarios, and support different kinds of both static and dynamic adaptation. This section provides a representative overview of them.

Component-based MMT homepage prototype: As one of the first demonstrators selected pages of the Web site of the author's research group were realized in a device independent component-based way. Figure 5.22 depicts two versions of the research group's welcome page for desktop browsers and PDAs, respectively. The demonstrator provides mainly presentation adaptation based on the adjustment of the utilized layouts. Furthermore, the inserted media elements are also adapted (regarding their

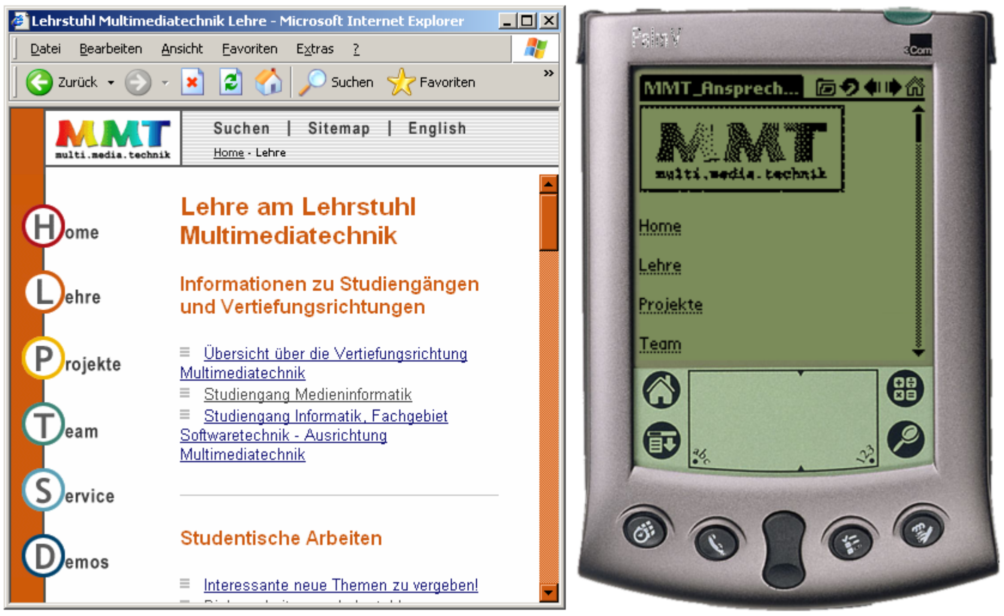

Lehrstuhl Multimediatechnik Lehre - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten
MMT multi.media.technik
Suchen | Sitemap | English
Home - Lehre
Home
Lehre
Projekte
Team
Service
Demos
Lehre am Lehrstuhl Multimediatechnik
Informationen zu Studiengängen und Vertiefungsrichtungen
Übersicht über die Vertiefungsrichtung Multimediatechnik
Studiengang Medieninformatik
Studiengang Informatik, Fachgebiet Softwaretechnik - Ausrichtung Multimediatechnik
Studentische Arbeiten
Interessante neue Themen zu vergeben!
Palm V
MMT_Ansprech...
MMT multi.media.technik
Home
Lehre
Projekte
Team

Figure 5.22: Component-based MMT homepage prototype

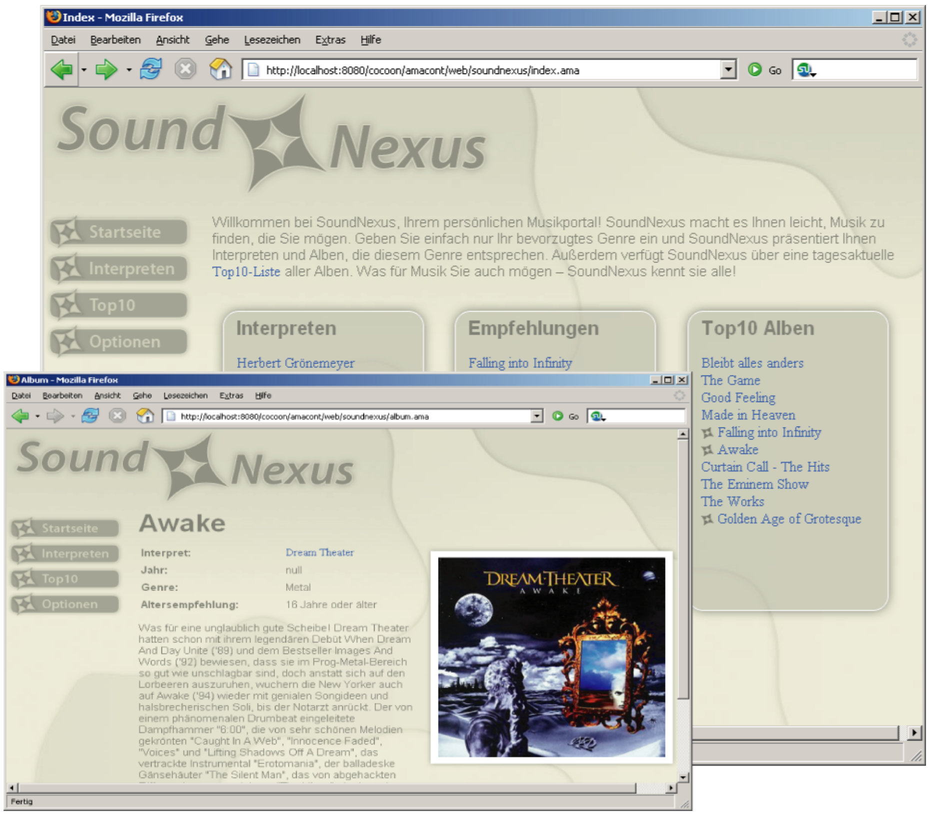
size, resolution, and other quality attributes) based on the capabilities of the appropriate end device. The component-based MMT homepage prototype was authored with conventional XML editors.

SoundNexus Prototype: The SoundNexus prototype is a data-driven Web presentation providing online information on music genres, bands (performers), and their albums. It was completely authored with the AMACONTBuilder to demonstrate its various editor modules and its support for creating component templates [Niederhausen 2006, Tietz 2006].

The prototype offers different kinds of content, navigation, and presentation adaptation. As an example, the list of albums shown to the user is generated dynamically, and is adjusted to his age, genre preferences, and personal voting. Furthermore, a TOP 10 list of the most popular albums (based on the votings of other users) is also provided. As also depicted in Figure 5.23, the presentation of this TOP 10 list makes use of the adaptation technique *link annotation*, i.e. the albums belonging to the current user's favorite genre are highlighted with a special icon.

Finally, the presentation of genres, bands, and albums is also adjusted to the device capabilities of the current user, i.e. different media elements with different modality (image vs. video) and media quality (large resolution image vs. small resolution image) can be used, respectively. For further information on the SoundNexus prototype the reader is referred to [Niederhausen 2006, Tietz 2006].

Model-driven Painting Gallery Prototype: This prototype demonstrates the automatic generation of a component-based adaptive Web presentation based on high-level (Hera-AMACONT based) model specifications, and is the actual implementation of the concepts described in Section 5.3. The user has the possibility to choose from a number of possible presentation model specifications and view the generated Web pages, accordingly.


Index - Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://localhost:8080/cocoon/amacont/web/soundnexus/index.ama
Go
Sound Nexus
Startseite
Interpreten
Top10
Optionen
Album - Mozilla Firefox
http://localhost:8080/cocoon/amacont/web/soundnexus/album.ama
Fertig
DREAM THEATER
AWAKE

Willkommen bei SoundNexus, Ihrem persönlichen Musikportal! SoundNexus macht es Ihnen leicht, Musik zu finden, die Sie mögen. Geben Sie einfach nur Ihr bevorzugtes Genre ein und SoundNexus präsentiert Ihnen Interpreten und Alben, die diesem Genre entsprechen. Außerdem verfügt SoundNexus über eine tagesaktuelle Top10-Liste aller Alben. Was für Musik Sie auch mögen – SoundNexus kennt sie alle!

Interpreten

Herbert Grönemeyer

Empfehlungen

Falling into Infinity

Top10 Alben

Bleibt alles anders
The Game
Good Feeling
Made in Heaven
Falling into Infinity
Awake
Curtain Call - The Hits
The Eminem Show
The Works
Golden Age of Grotesque

Awake

Interpret:	Dream Theater
Jahr:	null
Genre:	Metal
Altersempfehlung:	16 Jahre oder älter

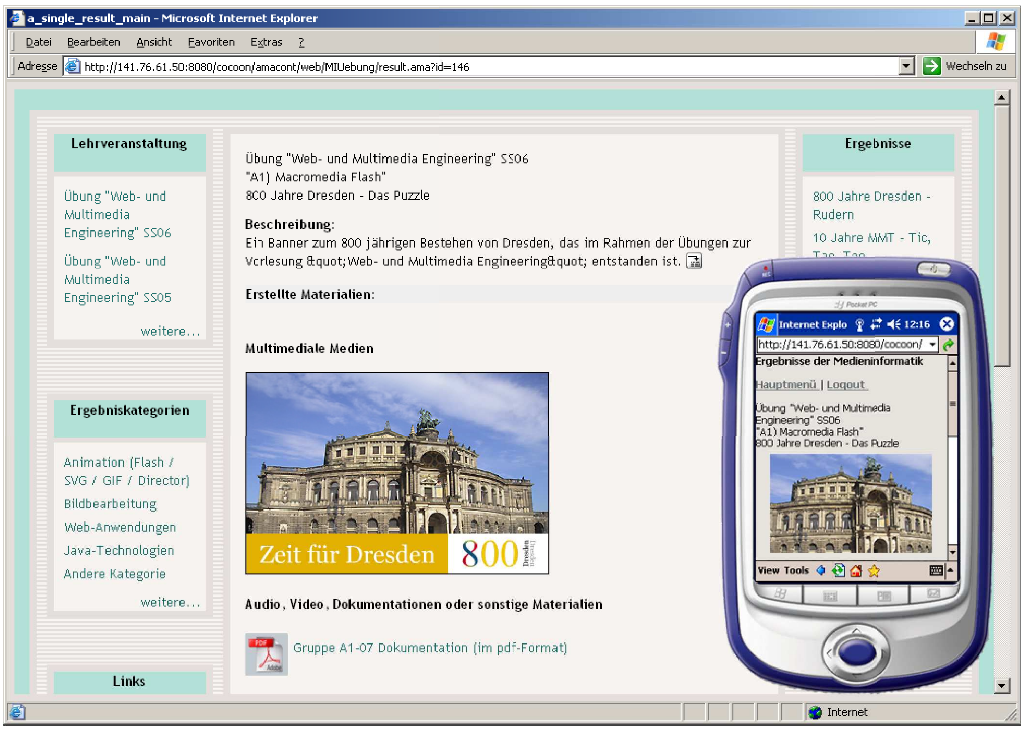
Was für eine unglaublich gute Scheibe! Dream Theater hatten schon mit ihrem legendären Debüt When Dream And Day Unite ('89) und dem Bestseller Images And Words ('92) bewiesen, dass sie im Prog-Metal-Bereich so gut wie unschlagbar sind, doch anstatt sich auf den Lorbeeren auszuruhen, wuchern die New Yorker auch auf Awake ('94) wieder mit genialen Songideen und halsbrecherischen Soli, bis der Notarzt anrückt. Der von einem phänomenalen Drumbeat eingeleitete Dampfhammer "6:00", die von sehr schönen Melodien gekrönten "Caught In A Web", "Innocence Faded", "Voices" und "Lifting Shadows Off A Dream", das vertrackte Instrumental "Erotomania", der balladeske Gänsehäuter "The Silent Man", das von abgehackten

Figure 5.23: SoundNexus prototype

The prototype was presented in 2004 at the 4th International Conference of Web Engineering (ICWE04 [Fiala et al. 2004a]) and is available online at [@ICWE2004Demo]. Selected screenshots of the generated presentations were already presented in Figure 5.19 and Figure 5.20 of this chapter.

Adaptive Web Information System for presenting student works: In order to demonstrate the capabilities of the component-based document model by example of a larger Web application, an adaptive Web information system aimed at the presentation of students' works at the author's university was designed and developed [Starke 2005]. It allows students of the multimedia technology study program to upload multimedia material created in different classes and courses (e.g. pictures, video and audio material, flash presentations, etc.), as well as to navigate through this information in an adaptive way.

Besides student works, the application offers information on the current and past semesters, their courses, as well as the persons responsible for them. The application supports for presentation layer adaptation based on its users' end devices and layout preferences by adjusting the the quality (e.g. different image resolutions), the type (image vs. flash presentations), as well as the spatial arrangement of the included media elements. Furthermore, it also adapts the structure and the interconnection of pages based on security aspects (different versions internal vs. external visitors), the


a_single_result_main - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Adresse
Wechseln zu
Internet
Internet Explo 12:16
http://141.76.61.50:8080/cocoon/
View Tools
Pocket PC
Zeit für Dresden 800

http://141.76.61.50:8080/cocoon/amacont/web/MIUebung/result.ama?id=146

Lehrveranstaltung

Übung "Web- und Multimedia Engineering" SS06

Übung "Web- und Multimedia Engineering" SS05

weitere...

Ergebniskategorien

Animation (Flash / SVG / GIF / Director)

Bildbearbeitung

Web-Anwendungen

Java-Technologien

Andere Kategorie

weitere...

Links

Übung "Web- und Multimedia Engineering" SS06
"A1) Macromedia Flash"
800 Jahre Dresden - Das Puzzle

Beschreibung:
Ein Banner zum 800 jährigen Bestehen von Dresden, das im Rahmen der Übungen zur Vorlesung &quot;Web- und Multimedia Engineering&quot; entstanden ist.

Erstellte Materialien:

Multimediale Medien

Audio, Video, Dokumentationen oder sonstige Materialien

Gruppe A1-07 Dokumentation (im pdf-Format)

Ergebnisse

800 Jahre Dresden - Rudern

10 Jahre MMT - Tic, Tac, Toe

Ergebnisse der Medieninformatik

Hauptmenü | Logout

Übung "Web- und Multimedia Engineering" SS06
"A1) Macromedia Flash"
800 Jahre Dresden - Das Puzzle

Figure 5.24: AWIS for presenting student works

visiting students' experience (their actual semester), etc. The application was successfully utilized at different courses at the author's research group. Figure 5.24 presents two screenshots of the application, one for desktop PCs and another one for PDAs. For more information on its design and realization the interested reader is referred to [Starke 2005, @kpss05].

Note that all mentioned prototype applications are available at the AMACONT project's homepage [@AMACONT].