Chapter 10

# HERA

[1,2]Geert-Jan Houben, [1]Kees van der Sluijs, [1]Peter Barna, [1,3]Jeen Broekstra, [2]Sven Casteleyn, [4]Zoltán Fiala, and [5]Flavius Frasincar
*[1]Technische Universiteit Eindhoven, PO Box 513, 5600 MB Eindhoven, The Netherlands, +31 40 247 2733, {g.j.houben, k.a.m.sluijs, p.barna, j.broekstra}@tue.nl; [2]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium, +32 2 629 33 08, {Geert-Jan.Houben, Sven.Casteleyn}@vub.ac.be; [3]Aduna, Prinses Julianaplein 14b, 3817 CS Amersfoort, The Netherlands, +31 33 465 9987, jeen@aduna.biz; [4]Technische Universität Dresden, Mommsenstr. 13, D-01062, Dresden, Germany, +49 351 463 39062, zoltan.fiala@inf.tu-dresden.de; [5]Erasmus Universiteit Rotterdam, PO Box 1738, 3000 DR Rotterdam, The Netherlands, +31 10 4081340, frasincar@few.eur.nl.*

Abstract: This chapter considers an approach for designing adaptive Web information systems, called Hera. The approach distinguishes models for specifying the domain, the navigation over the content from the domain, and the context for navigation. Hera is characterized by the use of the Semantic Web language RDF to express the models for the domain and context data and for the adaptive navigation. This chapter presents the main elements from this approach and illustrates also some of the tools for constructing and executing the models. The running example is expressed in terms of Hera models that use RDF query expressions to retrieve the data from RDF repositories for content and context data.

Key words: Hypermedia, adaptation, personalization, RDF, SeRQL, Semantic Web.

## 1. INTRODUCTION

This chapter illustrates a method for Web information systems (WIS) design that found its origins in an approach for hypermedia presentation generation. It was also this focus on hypermedia presentation generation that gave the first engine complying with this method its name HPG[1]. The method distinguishes three main models that specify the generation of hypermedia presentations over available content data. With a model for the

content, a model for the hypermedia navigation construction, and a model for the presentation construction, the method enables the creation of a hypermedia-based view over the content. Originally, in the first generation of the method and its toolset, the models specified a transformation from the content to the presentation. The engine that was compliant with this definition was based on XSLT and is therefore known as HPG-XSLT.

One of the characteristic aspects that HPG-XSLT supported was adaptation. As an illustrative example, we show in figure X-1 how different presentations could be produced by the engine out of a single design in which the "translation" to formats such as HTML, SMIL, and WML was dealt with generically.

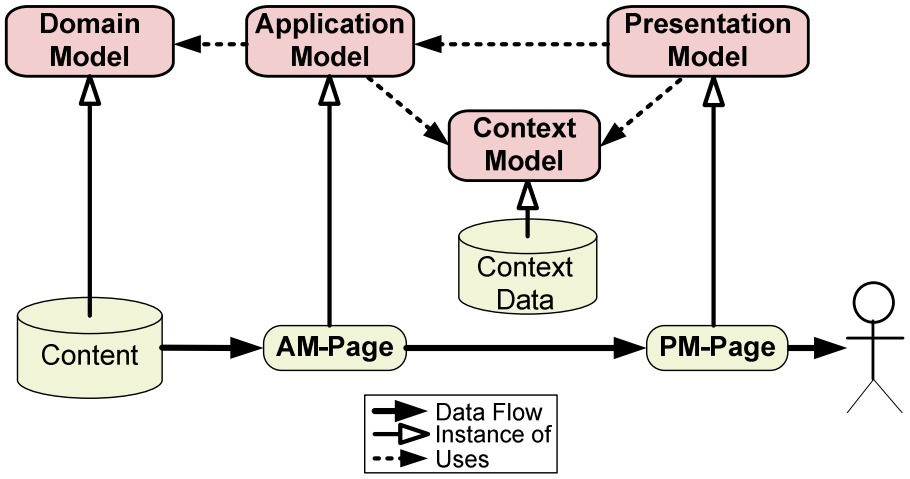


*Figure 10-1.* Hera Models

Characteristic for the Hera models was not only their focus on user- and context-adaptation support, but also the choice to base the models on the Resource Description Framework (RDF)[2].and RDF Schema (RDFS)[3]. The use of Web standards such as RDF and RDFS as a modeling paradigm facilitates easy deployment on very heterogeneous data sources: the only assumption made is that a semi-structured description (in RDF) of the domain is available for processing. Not only is such a representation less costly to develop than any alternative, it also enables reuse of existing knowledge and flexible integration of several separate data sources in a single hypermedia presentation.

During the further research into the development of the method, the support was extended for more advanced dynamics. Where the first XSLT-

based approach primarily transformed the original content data into a hypermedia document, with which the user could interact through following links with a Web browser, the subsequent engine version allowed the inclusion of form processing, which led to the support of other kinds of user-interaction, while retaining the hypermedia-based nature. Out of this effort, also a Java-based version of the engine became available which used RDF-queries to specify the data involved in the forms.

The experience out of these HPG-based versions and the aim for further exploitation of the RDF-based nature of the models have led to a further refinement of the approach in what is now termed Hera-S. The Hera-S-compliant models do combine the original hypermedia-based spirit of the Hera models with more extensive use of RDF-querying and storage. Realizing this RDF data processing using the Sesame framework[4] and its query language SeRQL[5] caters for extra flexibility and interoperability.

In the current version of the Hera method that we present in this chapter we aim to exemplify also the characteristic elements included in the method. As we mentioned before there is the RDF-based nature of the models. There is certainly also the focus on the support for adaptation in the different model elements. Adapting the data processing to the individual user and the context that the user is in (in terms of application, device etc.) is a fundamental element in WIS design and one that deserves the right attention: managing the different design aspects and thus controlling the complexity of the application design is crucial for an effective design and implementation.

In this chapter we first address the main characteristics of the method, before we explain the models, i.e. the main design artifacts, for the book's running example. We present the implementation of the hypermedia presentation generation process induced by the models. We also consider some extensions to the basic approach that can help the design process in certain scenarios.

## 2. METHOD

In this chapter of the book we discuss the Hera approach and illustrate it by means of examples from the Hera models for the running example (in this case we use Hera-S-compliant versions of those models). In this section we will capture the main elements of the key models used in the example before we go into details in the next section.

The purpose of Hera is to support the design of applications that provide navigation-based Web structures (hypermedia presentations) over semantically structured data in a personalized and adapted way. The design approach centers on *models* that represent the core aspects of the application

design. Figure X-2 gives an overview of these models. With the aid of a tool for executing those models (e.g. HPG-XSLT or Hera-S) we can also generate the application, as depicted in this figure. Thus the appropriate pipeline of models captures the entire application design, leaving room for the designer to change or extend the implementation where desired. In this section, we only give a short overview over the different models and associated modeling steps, before each of them is presented in more detail in the subsequent sections.

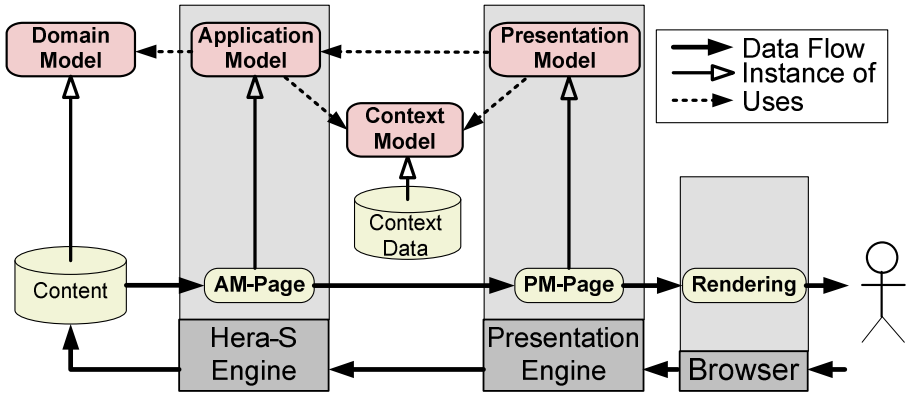


*Figure 10-2.* Hera-S Models and Tool Pipeline

Before we can create a model to specify the core design of the application, we need in Hera as a starting point a *Domain Model* (DM) that describes the structure of the content data. The sole purpose of the DM is to define how the designer perceives the semantical structure of the content data: it tells us what we need to know about the content over which we want the application to work. Based on this DM, the designer creates an *Application Model* (AM) that describes a hypermedia-based navigation structure over the content. This navigation structure is devised for the sake of delivering and presenting the content to the user in a way that allows for a (semantically) effective access to the content.

In turn, this effective access can imply the personalization or adaptation that is deemed relevant. Hera allows dynamic personalization and adaptation of the content. For this purpose, context data is maintained (under control of the application) in a so-called *Context Model* (CM). This context data is typically updated based on the (inter)actions of the user as well as on external information.

So, on the basis of DM and CM the AM serves as a recipe that prescribes how the content is transformed into a navigational structure. To be more

precise, instantiating the AM with concrete content results in AM (instance) *pages* (AMP). These AMP's can be thought of as pages that contain content to be displayed and navigation primitives (based on underlying semantic relations from the DM) that can be used by the user to navigate to other AMP's and thus to semantically "move" to a different part of the content. An AMP itself is not yet directly suitable for a browser, but can be transformed into a suitable presentation by a presentation generator, i.e. an engine that executes a specification, for example a Presentation Model (PM) of the concrete presentation design in terms of layout and other (browser-specific) presentation details. In Section 7 we demonstrate that both proprietary and external engines can be used for this task. For the Hera method this presentation generation phase itself is not specific and it may be done in whatever way is preferred. So, the AM specifies the (more conceptual or semantical) construction of the navigational structure over the content, while the subsequent presentation phase, possibly specified by a PM, is responsible for the transformation of this structure into elements that fit the concrete browsing situation.

AMP creation is conceptually *pull-based*, meaning that a new AMP is only constructed in the Hera pipeline at request (in contrast to constructing the whole instantiation of the AM at once, which was done in for example the implementation by the HPG-XSLT engine). Through navigation (link-following) and forms submission the user triggers the feedback mechanism, which results in internally adapting (updating) the website navigation or context data and the creation of a new AMP.

As indicated in the introduction, Hera models use RDF(S) to represent the relevant data structures. In the next sections we will see this for the specification of the data in DM, CM and AM. In the engines these RDF(S) descriptions are used to retrieve the appropriate content and generate the appropriate navigation structures over that content. In HPG-XSLT the actual retrieval was directly done by the engine itself, whereas in HPG-Java this was done with the aid of expressions that are based on SeRQL[5] queries. In Hera-S the actual implementation exploits the fact that we have chosen to use RDF(S) to represent the model data and allows to use native *RDF querying* to access data, for example the content (DM) and context (CM) data. For this Hera-S allows the application (AM) to connect to the content and context data through the Sesame RDF framework. This solution, combining Hera's navigation design and Sesame's data processing by associating SeRQL queries to all navigation elements, allows to effectively applying existing Semantic Web technology and a range of its solutions that is becoming available. We can thus include background knowledge (e.g. ontologies, external data sources), we can connect to third-party software (e.g. for business logic), and connect to services through the RDF-based

specifications. We can also use the facilities in Sesame for specific adaptation to the data processing and to provide more extensive interaction processing (e.g. client-side, scripting). The dynamics and flexibility required in modern Web information systems can thus be met by accommodating the requirements that evolve from an increasing demand for personalization, feedback, and interaction mechanisms. We point out that Hera-S models in principle are query and repository independent. We only require a certain type of functionality, and if a repository fulfils these requirements it can be used for implementation of Hera-S models.

## 3. DATA MODELING

Before the application design can consider the personalized navigation over the domain content, the relevant data needs to be specified. As a necessary first step in the approach the *data modeling* step leads to the construction of the data models for the domain content and the context of the user.

The modeling of the domain content uses RDF(S) and is primarily targeted towards capturing the semantical structure of the domain content. With the Hera-S engine we even allow the model to be an OWL[6] ontology (without restrictions). If we look at the UML representation for the IMDb-example that is used throughout the book, this can be easily modeled as an RDFS or OWL definition. In this case this could be done by using a UML to OWL conversion process (several papers have been written on the relations between UML and OWL[7]). We could however also create this model ourselves, e.g. by using an ontology editor like Protégé[8]. Figure X-3 contains a screenshot of the UML model translated into an RDFS hierarchy together with its properties and OWL-restrictions in Protégé. We divided the UML model in four parts. One part, with the prefix ‘imdb’, contains the “core” of the movie domain, describing the movies and the persons involved in those movies. Another part, with the prefix ‘cin’, models the cinemas that show the movies that are modeled in the ‘imdb’ part.

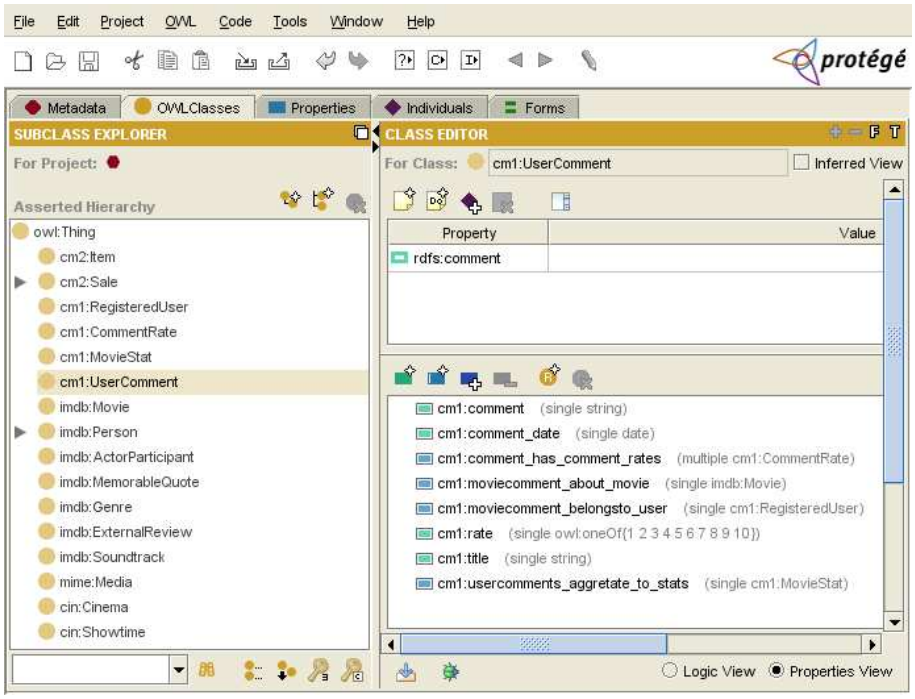


*Figure 10-3.* Protégé screenshot for the IMDb data modeling

In this figure we also see prefixes starting with ‘cm’. They relate to the *context* modeling. The CM is modeled and implemented in a similar way as the DM. The main difference between the two is that the content data is meant to be presented to the user, while the context data is meant to support the context-dependent adaptation of the application. So, the content typically contains the information that in the end is to be shown to the user, while the context data typically contains information used (internally) for personalization and adaptation of content delivery. This distinction might not always be strict, but as it is only a conceptual distinction in Hera-S, the designer may separate content and context in whatever way he desires. As a consequence, we assume that context data is under direct control of the engine, while the content often is not. In the IMDb example the context model is modeled in the same way as the domain. We first maintain a model, with the prefix ‘cm1’, that contains users and their comments on movies in the ‘imdb’ part. The second part, with the prefix ‘cm2’, contains a description of tickets bought by the user for a particular movie showing in a particular cinema.

Considering the role and function of the context data, we can identify different aspects of context. We will come back to context data later when we discuss adaptation in the AM, but we now address the context data

modeling. Even though the designer is free to choose any kind of context data, we in general discern three types: session data, user data and global data.

- *Session* data: Session data is data relevant to a certain session of a certain user. An example of such data is the current browsing context, such as the device that is used to access the Web application or the units browsed in the current session.
- *User* data: Data relevant to a certain user over multiple sessions (from initial user data to data collected over more than one session). User (profile) data can be used for personalization (even at the beginning of a new session). Note that for maintaining this user data over time the application needs some authentication mechanism.
- *Global* data: Usage data relevant to all users over all sessions. Global data typically consists of aggregated information that gives information about groups of people. Examples include "most visited unit" or "people that liked item *x*, also browsed item *y*".

In figure X-4 we show part of an RDF graph representation of the domain data model that we will use as a basis for the examples of this chapter. It shows the main classes and a selection of relationships between those classes, while omitting their datatype properties.

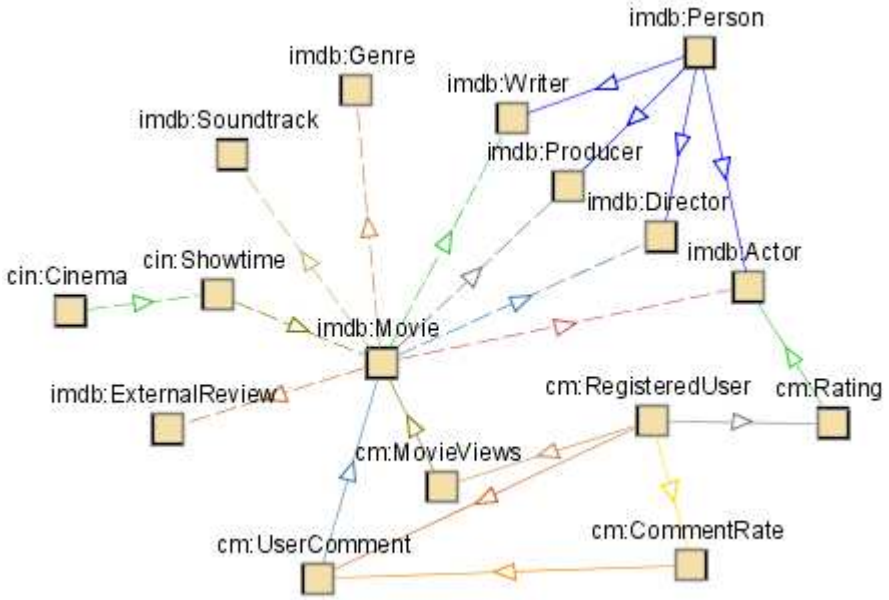


*Figure 10-4.* RDF-graph representation of IMDb domain and context data

Both the DM and CM data are in Hera-S implemented using a Sesame repository. For the CM which is under direct control of the application, this allows the application to manage and update the context as it perceives this context. Next to this, it also provides the means for other processes to use and update (parts of) this information. The context data could for instance be manipulated by business logic software for the sake of adaptation, or by external user profiling software.

Another great advantage of using Sesame is the possibility to combine several data sources (both content and context data) at the same time. In this way, designers can couple additional data sources to the already existing ones and can thus easily extend the domain content. This also offers possibilities to exploit additional knowledge when performing a search. Currently, we are involved in the exploitation of the WordNet ontology[9,10] time ontologies[11] and geographic ontologies[12,13]. In this way, a keyword-search can be extended with synonyms extracted from the WordNet ontology, or a search for a city can be extended with surrounding cities from a geographic ontology. By supporting unrestricted RDFS or OWL DM's, Hera-S is particularly suited to (re-)use existing domain ontologies. Moreover, many existing data sources that are not yet available in RDFS or OWL format can be used via Semantic Web wrapping techniques[14,15]. In the latter case Sesame can be used as a mediator between such a data source and Hera-S.

As we will see in detail in the next section, the access to the data from the DM or CM is part of the application definition. It means that the access to the RDF data is part of the model. In principle, we assume that the concepts from the DM and CM are associated with concrete data elements in the data storage structure. As we use the Sesame RDF Framework as our back-end repository, this data can be exploited and reasoned upon. Accessing the content data in Hera-S will be done via explicit SeRQL queries and by making them explicit in the models we support customizable access via customizable SeRQL queries. Thus, the full potential of the SeRQL query language can later be used in the AMP creation. For the purpose of defining the content and context, we can abstract from the SeRQL queries, but for the support of different types of adaptation, we benefit from making this SeRQL access explicit.

## 4. APPLICATION MODELING

Based on the domain definition, application modeling results in the Application Model (AM) that specifies the navigational behavior of the Web application. The AM enables designers to specify how the (navigational)

access to the data (dynamically retrieved from the domain) is structured by describing which data is shown to the user and what Web pages the user can navigate to. At the same time, the AM allows this specification to be dynamic, such that the navigational access to the data can be personalized to a user and adapted for a specified context.

Since in the AM we use Turtle and SeRQL syntax, we first highlight in Section 4.1 the most relevant elements from those languages. In Section 4.2 we present the basic AM constructs and exemplify them based on the IMDb example as discussed in the previous section. In Section 4.3 we give examples of adaptation expressed in the AM. Section 4.4 contains a number of more advanced modeling primitives. In Section 4.5 we illustrate a model builder that offers designers a visual tool to help create the domain models and the AM and produce the correct RDF serialization for those graphical representations.

## 4.1 Queries and Syntax

### 4.1.1 Turtle

Turtle[16] (Terse RDF Triple Language) is an RDF syntax format designed to be compact, easy to use, and easy to understand by humans. Although not an official standard, it is widely accepted and implemented in RDF toolkits.

In Turtle, each part of the triple is written down as a full URI or as a qualified name (using namespace prefixes). In our examples, we will mostly use the latter form, for brevity. For example,

```
my:car rdf:type cars:Volvo .
```

denotes the RDF statement “my car is of type Volvo”. ‘my:’ and ‘cars:’ in this example are namespace prefixes that denote the vocabulary/ontology from which the term originates. Turtle also introduces the predicate ‘a’ as a shortcut for the ‘rdf:type’ relation.

```
my:car a cars:Volvo .
```

denotes the same RDF statement as the first example.
In order to list several properties of one particular subject, we can use the semi-column to denote branching.

```
my:car a cars:Volvo ;
       my:color “Red” .
```

denotes two statements about the car: that it is of type ‘Volvo’, and that its color is red (denoted by a string literal value in this example).

When the value of a property is itself an object with several properties, we can denote this by using square brackets (‘[‘ and ‘]’). In RDF terms, such brackets denote a blank node (which can be thought of as an existential qualifier). For example:

```
my:car a cars:Volvo ;
       my:hasSeat [
          a my:ComfortableSeat ;
          my:material "Leather"
       ] .
```

This denotes that the car has a seat which is “something” (a blank node) which has as its type ‘my:ComfortableSeat’ and has leather as the material.

In the next sections, we will regularly use Turtle syntax forms in various examples.

### 4.1.2 SeRQL

SeRQL[4,5] (Sesame RDF Query Language) is an RDF query and transformation language that uses graph templates (in the form of path expressions) to bind variables to values occurring in the queried RDF graph. It is an expressive language with many features and useful constructs.

A SeRQL query consists of a set of clauses (SELECT, FROM, and WHERE). Like in SQL, the SELECT clause describes the projection, i.e., the ordered set of bound values that is to be returned as a query result. The FROM clause describes a graph template that is to be matched against the target graph, and the WHERE clause specifies additional Boolean constraints on matching values. A simple example query that selects all instances of the class ‘Volvo’ is:

```
SELECT aCar
FROM {aCar} rdf:type {cars:Volvo}
```

As one can see, the path expression syntax bears a strong resemblance to Turtle syntax, except that in SeRQL, each node in the graph is surrounded by ‘{‘ and ‘}’. In the above query, ‘aCar’ is a variable that is to be matched in the target graph against all statements that conform to the pattern, i.e., which have ‘rdf:type’ as their predicate, and ‘cars:Volvo’ as their object.

Like in Turtle, SeRQL paths can be branched (using a semi-column), and they can also be chained. For example, in the following query we use both chaining and branching to select a car, its color, its owner, and the address of that owner:

```
SELECT aCar, color, owner, address
FROM {aCar} rdf:type {cars:Volvo} ;
            my:color {color} ;
            my:owner {owner} my:address {address}
```

Additionally, we can use the WHERE clause to specify additional constraints on the results. To adapt the above query to only return results for those cars whose color is red, we could add a WHERE clause:

```
WHERE color = "Red"
```

## 4.2 Units, Attributes and Relationships

Now we will discuss the constructs that we provide in our AM. We will start in this section with the basic constructs that are sufficient to build basic Web applications and move on afterwards to more complex constructs for realizing richer behavior.

The AM is specified by means of *navigational units* (shorthand: units) and *relationships* between those units. The instantiation of units and relationships is defined by (query) expressions that refer to the (content and context) data as explained in Section 3.

The unit can be used to represent a "page". It is a primitive that (hierarchically) groups elements that will together be shown to the user. Those elements shown to the user are called attributes and so units build hierarchical structures of attributes.

An attribute is a single piece of information that is shown to the user. This information may be constant (i.e. predefined and not changing), but usually it is based on information inside the domain data. If we have a unit for a concept *c*, then typically an attribute contained in this unit is based on a literal value that is directly associated with *c* (for example as a datatype property). Note that literals may not only denote a string type, but also other media by referring to a URL. Furthermore, we offer a built-in media class (denoted by 'hera:Mime') that can be used to specify an URL and the MIME-type of the object that can be found at the URL. This can be used if the media type is important during later processing.

Below we give an example of the definition of a simple unit, called 'MovieUnit', to display information about a movie. We mention two elements in this definition:

- From the second until the seventh line, we define which data instantiates this unit. This data is available as input ('am:hasInput') from the environment of this unit, e.g. passed on as link parameter or available as global value. In this case we have one variable 'M': the fourth line specifies the (literal) name of the variable, while the fifth line indicates the type of the variable. In this case, a value from the 'imdb:Movie' class concept from the domain will instantiate this unit.
- In the eighth until the fourteenth line, starting with 'am:hasAttribute', we decide to display an attribute of this movie, namely a title. We label this attribute with 'Title' (such that later we can refer to it), and we indicate (with 'am:hasQuery') how to get its value from the data model. This query uses the 'imdb:movieTitle' (datatype) property applied to the value of 'M'. Note that in the query '$M' indicates that 'M' is a Hera variable, i.e. outside the scope of the SeRQL query itself. The output of the SeRQL query result is bound to the Hera variable 'T' (implicitly derived from the SELECT list).

In our RDF/Turtle syntax the definition looks as follows:

```
:MovieUnit a am:NavigationUnit ;
   am:hasInput [
      am:variable [
         am:varName "M" ;
         am:varType imdb:Movie
      ]
   ] ;
   am:hasAttribute [
      rdfs:label "Title" ;
      am:hasQuery
         "SELECT T
         FROM {$M} rdf:type {imdb:Movie};
                   imdb:movieTitle {T}"
   ] .
```

For the attribute value instead of the simple expression (for which we can even introduce a shorthand abbreviation) we can use a more complicated query expression, as long as the query provided in 'am:hasQuery' returns a datatype property value.

Relationships can be used to link units to each other. We can use relationships to contain units within a unit, thus hierarchically building up

the "page" (we call these aggregation relationships), but we can also exploit these relationships for navigation to other units (we call these navigation relationships).

As a basic example, we include in the unit for the movie not only its title but also a (sub)unit with the name and photo of the lead-actor and a navigational relationship that allows to navigate from the lead-actor information to the full bio-page (unit) for that actor. Note that from now on we omit in our text namespaces when they appear to be obvious.

- We have separated here the definitions of 'MovieUnit' and 'ActorUnit' (which allows later reuse of the 'ActorUnit'), but we can also define subunits inside the unit that contains them.
- In the definition of the 'MovieUnit' one can notice compared to the previous example, that we have an additional subunit with its label 'LeadActor', with its type 'ActorUnit', and with the query that gives the value with which we can instantiate the subunit.
- In the definition of the 'ActorUnit' one can notice its input variable, two attributes, and a navigation relationship. This navigation relationship has a label 'Actor-Bio', targets a 'BioUnit', and with the query based on the 'imdb:actorBio' property it determines to which concrete 'BioUnit' this 'ActorUnit' offers a navigation relationship. Note that in this case the variable '$B' is passed on with the navigational relationship (it is also possible to specify additional output variables that are passed on with the relationship).

```
:MovieUnit a am:NavigationUnit ;
   am:hasInput [ am:variable [ am:varName "M";
                               am:varType imdb:Movie]] ;
   am:hasAttribute [ rdfs:label "Title" ; … ] ;
   am:hasUnit [
      rdfs:label "LeadActor" ;
      am:refersTo :ActorUnit ;
      am:hasQuery
         "SELECT L
         FROM {$M} rdf:type {imdb:Movie};
         imdb:movieLeadActor {L} rdf:type {imdb:Actor}"
      ] .

:ActorUnit a am:NavigationUnit ;
   am:hasInput [ am:variable [ am:varName "A" ;
                               am:varType imdb:Actor]] ;
   am:hasAttribute [
      rdfs:label "Name" ;
```

```
      am:hasQuery
         "SELECT N
         FROM {$A} rdf:type {imdb:Actor};
                   imdb:actor_name {N}" ] ;
   am:hasAttribute [
      rdfs:label "Photo" ;
      am:hasQuery
         "SELECT P
         FROM {$A} rdf:type {imdb:Actor};
                   imdb:actorPhoto {P}" ] ;
   am:hasNavigationRelationship [
      rdfs:label "Actor-Bio" ;
      am:refersTo :BioUnit ;
      am:hasQuery
         "SELECT B
         FROM {$A} rdf:type {imdb:Actor};
                   imdb:actorBio {B}"
   ] .
```

So, in these examples we see that each element contained in a unit, whether it is an attribute, a subunit, or a navigational relationship, has a query expression ('hasQuery') that determines the value used for retrieving (instantiating) the element.

Sometimes we know that in a unit we want to contain subunits for each of the elements of a set. For example, in the 'MovieUnit' we might want to provide information for all actors from the movie (and not just the lead actor). Below we show a different definition for the 'MovieUnit' that includes a *set*-valued subunit element ('am:hasSetUnit'). In its definition, one can notice

- the label "Cast" for the set unit,
- the indication that the elements of the set unit are each an 'ActorUnit', and
- the query that determines the set of concrete actors for this movie to instantiate this set unit, using the 'imdb:movie_actor' object property.

```
:MovieUnit a am:NavigationUnit ;
   am:hasInput [ am:variable [ am:varName "M" ;
                               am:varType imdb:Movie]] ;
   am:hasAttribute [ rdfs:label "Title" ; … ] ;
   am:hasSetUnit [
      rdfs:label "Cast";
      am:refersTo ActorUnit ;
```

```
   am:hasQuery
      "SELECT A
      FROM {$M} rdf:type {imdb:Movie};
                imdb:movieActor {A}"
] .
```

So, we see that a set unit is just like a regular unit, except that its query expression will produce a set of results, and this will cause the application to arrange for displaying a set of (in this example) 'ActorUnits'.

Likewise, we can have set-valued query expressions in navigational relationships, and with 'am:tour' and 'am:index' we can construct guided tours and indexes respectively. With these the order of the query determines the order in the set and with the index an additional query is used to obtain anchors for the index list.

## 4.3 Adaptation Examples

Adaptation and personalization are important aspects within the Hera methodology. For this purpose the query expressions can be used to include conditions that provide control over the instantiation of the unit. Typically, these conditions use data from the context model (CM) and thus depend on the current user situation. For example, we can use 'U' as a (global) variable that denotes the current (active) user for this browsing session (typically this gets instantiated at the start of the session). Let us assume that in the CM for each 'Actor' there is a 'cm:actorRating' property that denotes 'U's rating of the actor (from 1 to 5 stars) and that the user has indicated to be only interested in actors with more than 3 stars. We could then use this rating in adapting the cast definition in the last example:

```
am:hasSetUnit [
   rdfs:label "Cast";
   am:refersTo ActorUnit ;
   am:hasQuery
      "SELECT A
      FROM {$U} cm:actorRating {} cm:stars {V};
               cm:ratingOnActor {A} imdb:playsIn {$M}
      WHERE V > 3"
] .
```

Here we see how we can influence (personalize) the input to an element (in this case a set) by considering the user context in the query that determines with which values the element gets constructed. To be precise, in

this example we state inside the movie unit what actors of the cast this user will be provided with, i.e. which values we "pass on".

Another user adaptation example would be that the user has indicated not to be interested in photos from actors. We could then change the query for the photo attribute accordingly:

```
:ActorUnit a am:NavigationUnit ;
   am:hasInput [ am:variable [ am:varName "A" ;
                               am:varType imdb:Actor]] ;
   …
   am:hasAttribute [
      rdfs:label "Photo" ;
      am:hasConditionalQuery [
         am:if "SELECT *
            FROM {$U} cm:showElement {}
                      cm:showAbout {imdb:actorPhoto}

         am:then "SELECT P
                  FROM {$A} imdb:actorPhoto {P}"
        ]
     ] ;
     … .
```

Here we see that with 'am:hasConditionalQuery' the attribute becomes "conditional", i.e. the photo attribute is only shown when the condition ('am:if') query produces a non-empty result. We can also add an 'am:else' part here and for example display the string "no photo displayed". We point out that this query can be written in one single (nested) SeRQL query, but for the clarity of adaptation specification we use this syntax sugaring.

Finally, we present a more complex example of adaptation. Consider again the 'ActorUnit' from the previous section, which showed an actor's name, his picture, and a link to his bio. Now imagine we would like to add the list of movies in which the actor played. However, because some movies are age-restricted, we would like to restrict this list so that adult-rated movies are only shown to registered users that are 18 or older. As in the previous adaptation examples, this adaptation can be achieved by tweaking the SeRQL query that computes the list of movies:

```
:ActorUnit a am:NavigationUnit ;
   am:hasInput [ am:variable [ am:varName "A" ;
                               am:varType imdb:Actor]] ;
   …
```

```
am:hasSetUnit [
   rdfs:label "Movies Played In";
   am:refersTo MovieUnit ;
   am:hasConditionalQuery [
   am:if "SELECT *
         FROM  {$U} cm:age {G}
         WHERE G > 17"
   am:then "SELECT M
           FROM {$A} imdb:actorMovie {M},
                {M} rdf:type {imdb:Movie}"
   am:else "SELECT M
            FROM {$A} imdb:actorMovie {M},
                 {M} rdf:type {imdb:Movie};
                      imdb:mpaaRating {R}
        WHERE R != "NC-17""
   ]] .
```

First of all, notice in the code-excerpt the 'am:hasSetUnit', which represents the list of movies for the active actor ('A'). This list is defined by a conditional query, in which it is verified whether the active user ('U') is registered and his age is over 17 ('am-if'). If this condition holds ('am:then'), all movies of the particular actor are computed. If the condition does not hold ('am:else'), the computed movie list is restricted to movies which are not MPAA (Motion Picture Association of America)-rated as "NC-17" (No Children Under 17 Admitted).

## 4.4 Other constructs

Before, the basic constructs were explained. In this section we will look at some additional features of Hera-S that also allow designers to use some more advanced primitives in order to construct richer applications.

### 4.4.1 Update Queries

For the sake of adaptation we need to maintain an up-to-date context model. In order to do so, we need to perform updates to this data. For this, we have the functionality to specify an 'am:onLoad' update query and an 'am:onExit' update query within every unit, that are executed on loading (navigating to) and exiting (navigating from) the unit. Furthermore, we allow attaching an update query to a navigation relationship so that the update query is executed when a link is followed. In all cases, the designer may also specify more than one update query.

In our example we could for instance maintain the number of page views (visits) of a certain 'movieUnit', and update this information if the 'movieUnit' is loaded using the 'onLoad' query:

```
:MovieUnit a am:NavigationUnit ;
   …
   am:onLoad [
      am:updateQuery
         "UPDATE {V} cm:amount {views+1}
         FROM {$U} rdf:type {cm:RegisteredUser};
                cm:userMovieViews {V} cm:amount {views};
                cm:viewsOfMovie {$M}"
   ];
   … .
```

### 4.4.2 Frame-based Navigation

We explained earlier that units can contain other units. The root of such an aggregation hierarchy is called a top-level unit. The default semantics of a navigational relationship (that is defined somewhere inside the hierarchy of a top-level unit) is that the user navigates from the top-level unit to the top-level unit that is the target of the relationship. In practice this often means that in the browser the top-level unit is replaced by the target unit. However, we also allow specifying that the navigation should only consider the (lower-level) unit in which the relationship is explicitly specified, so that only that unit is replaced while the rest of the top-level unit remains unchanged.

This behavior is similar to the frame construct from HTML. We specify this behavior by explicitly indicating the source-unit for the relationship. Inspired by the HTML frame construct we allow the special source-indications "_self" (the unit that contains the relation), "_parent" (the unit that contains the unit with the relation) and "_top" (the top-level unit – the default behavior). Alternatively, relations may also indicate another containing unit by referring to the label of the contained unit. An example of a navigational relationship with source indication looks like:

```
am:hasNavigationRelationship [
   …
   am:source am:_self ;
   … ]
```

### 4.4.3 Forms

Besides using relationships (links) for navigation, we also support applications that let the user provide more specific feedback and interact. For this we provide the form unit. A form unit extends a normal unit with a collection of input elements (that allow the user to input data into the form) and an action that is executed when the form is submitted. In a form a navigational relationship typically has a button that activates the submission.

Below we give an example of a form that displays the text "Search Movie:" (line three) with one text input-field (line five to eleven) to let the user enter the movie he wants to browse to. If the user enters a value in this field, it is bound to the variable 'movieName' (line nine). After submitting the form via a button with the text "Go" (line fourteen and fifteen), the user navigates to the 'MovieUnit' (line fourteen) that will display the movie for which the name was entered in the input-field, which is specified in the query (starting in line twenty) using the variable 'movieName'.

```
:MovieSearchForm a am:FormUnit ;
   am:hasAttribute [
      am:hasValue "Search Movie: "
   ];
   am:formElement [
      rdfs:label "Search Input";
      am:formType am:textInput;
      am:binding[
                am:variable [am:varName "movieName" ;
                             am:varType xsd:String ]]
   ];
   am:formElement [
      rdfs:label "Submit Button";
      am:formType am:button;
      am:buttonText "Go";
      am:hasNavigationRelationship [
         rdfs:label "Search Form-Movie" ;
         am:refersTo :MovieUnit ;
         am:hasQuery
            "SELECT M
            FROM {M} rdf:type {imdb:Movie};
                     imdb:movieTitle {X}
            WHERE X = $movieName"
      ]
   ].
```

### 4.4.4 Scripting objects

Current Web applications offer users a wider range of client-side functionality by different kinds of scripting objects, like Javascript and VBscript, stylesheets, HTML+TIME timing objects etc. Even though WIS methods like Hera concentrate more on the creation of a platform-independent hypermedia presentation over a data domain, and these scripts are often (but not always) browser/platform specific, we still provide the designer a hook to insert these kind of scripting objects.

The designer can specify within a scripting object whatever code he wants, as this will be left untouched in generating the AMP's out of the AM. Furthermore, the designer can add an 'am:hasTargetFormat' property to specify one or more target-formats for format-specific code, e.g. HTML or SMIL. This allows later in the process to filter out certain format-specific elements if these are not wanted for the current presentation. The scripting objects can use the variables that are defined within the scope of the units. Scripting objects can be defined as an element within any other element (i.e. units and attributes). Furthermore, it can be specified if the script should be an attribute of its super-element or not (e.g. similar to elements in HTML that have attributes and a body). The need to place some specific script on some specific place is of course decided by the designer.

### 4.4.5 Service Objects

An application designer might want to use additional functionality that cannot be realized by a client-side object, but which involves the invocation of external server-side functionality. Therefore, we provide so-called service objects ('am:serviceObject') to support Web services in the AM. The use of a service object and the reason to provide support for it is similar to that of scripting objects. The designer is responsible for correctness and usefulness of the service object.

Think of utilizing a Web service from a Web store selling DVDs in order to be able to show on a movie page an advertisement for buying the movie's DVD. A service object needs three pieces of information:

- a URL of the Web service one wants to use,
- a SOAP message that contains the request to the Web service, and
- a definition of the result elements.

A service object declaration can be embedded as a part of every other element. If a unit is navigated to ("created"), first the service objects will be executed. The results of the service object will either be directly integrated

into the AM and treated as such, or the result can be bound to variables. Service objects can use unit variables in their calls.

## 4.5 Model Builders

In most RDF serializations it can become difficult to see which structures belong together and what the general structure of the document is; especially if the documents get larger. This also applies to the Hera models and has the consequence that manually creating them can become error-prone. It is therefore beneficial to offer the designer tool-support for creating those models graphically. Based on a given HPG version, Hera Studio (figure X-5) contains a domain, context and application model editor in which the designer can specify the models in a graphical way. All these models can subsequently be exported to an RDF serialization that can be used by Hera.

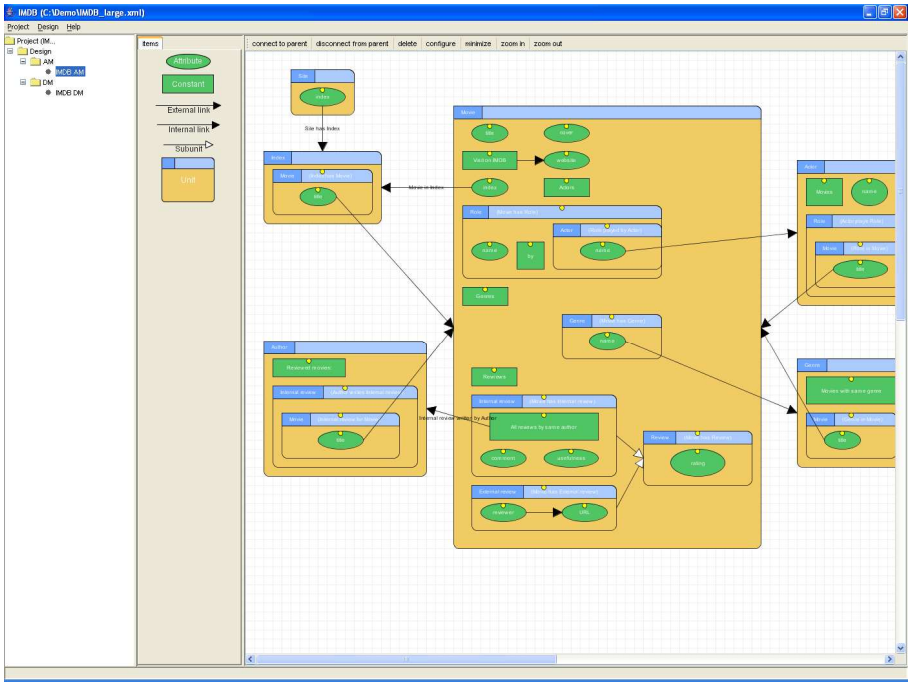


*Figure 10-5.* Hera Studio

Note that Hera Studio is not a general purpose OWL or RDF(S)-editor like for instance Protégé, rather a custom-made version specialized for Web applications designed through Hera models.

In the DM-editor (figure X-6), designers can define classes and object properties between those classes. For every class a number of datatype properties can be given that have a specified media type (e.g. String, Image etc). Furthermore, inheritance relations for classes and properties can be denoted. In addition, instances of the classes and properties can be specified. Note that if more complex constructs are needed, the designer could also use a general purpose OWL/RDF editor like Protégé.

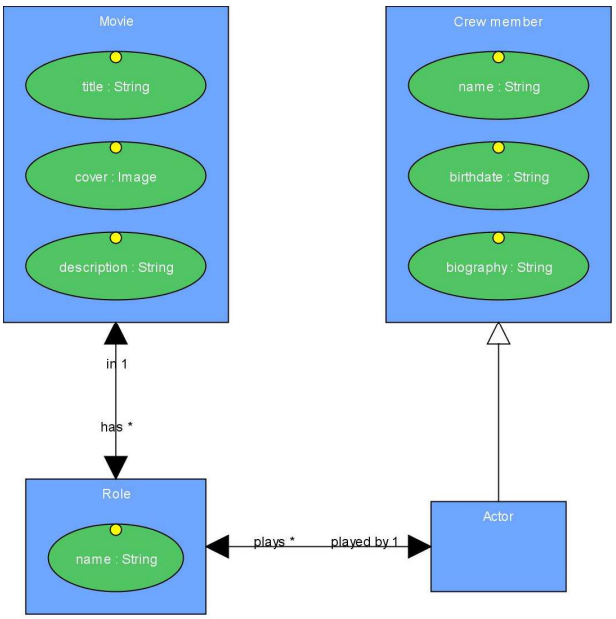


*Figure 10-6.* DM example

The AM-editor provides a graphical way for specifying an AM (figure X-7 gives an example). It specifically allows organizing the units and the relationships between them. Per unit, elements can be defined and displayed. Detailed information like queries is hidden in the graphical view, and can be configured by double-clicking the elements. For the simpler constructs, the editor provides direct help: for example when defining a datatype property, the editor gives the designer a straightforward selection choice out of the (inherited) datatype properties of the underlying context and domain models. However, for the more complex constructs, the designer has the freedom to express his own queries and element-properties. In addition, the designer can control the level of detail of the model to get a better overview of the complete model.

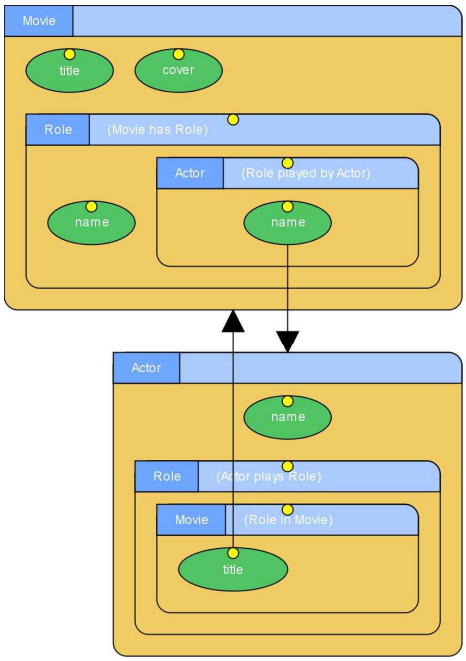


*Figure 10-7.* AM example

Currently, the AM-editor is being extended to support the more specific Hera-S constructs mentioned in this section. In this process, we will also extend the model checking functionality that allows the designer to check if the Web application fulfills certain requirements. Furthermore, we plan extending the builders with an optional lightweight presentation component.

## 5. ASPECT-ORIENTATION IN MODEL DESIGN

Before we continue with presentation design and implementation, we make a side-step and turn to an element of design support that we are currently working on and that uses principles from aspect-orientation.

As described in the previous sections, Hera-S provides conceptual WIS design support on the basis of data contained in an RDF repository like Sesame and which is accessed and manipulated through an RDF query language like SeRQL. In this setting, adaptation is specified by SeRQL queries that, based on (DM and) CM data, conditionally instantiate navigational units in the AM. Examples of such adaptation can be found in Section 4.3. In most cases, the desired adaptation is expressed by

expressions that are embedded in the SeRQL query and that have the explicit purpose of restricting the set of instances; we can call these expressions *adaptation conditions*. We observe that often these adaptation conditions can conceptually be detached from the rest of the SeRQL queries and explicitly specified at (AM) model level. In this way, with each AM modeling element (i.e. units, relationships) we can nicely associate its adaptation conditions that explicitly denote the restriction of this element according to the user model attributes/values.

Typically, in a Web application, several adaptation issues need to be taken into account in parallel (e.g., age-group restriction, accessibility, device-dependence). Adaptation engineering thus constitutes a significant effort in specifying the application's functionality. Moreover, although the adaptation conditions for an adaptation issue can occur at one specific place in the design (e.g. to restrict adult-rated material on a certain page), it is (more) often the case that they cannot be pinpointed to one particular element (e.g., when one wants nowhere on the site to show adult-rated material to minors) and need to be applied at different places in the design (models). Concretely, consider the last example from Section 4.3, which restricts the list of movies starring a particular actor. Obviously, the designer may be required to specify other lists of movies (e.g., in the 'am:CinemaUnit', to denote the movies played in a particular cinema) also at other places in the design. To enforce the age-group restriction policy throughout the application or Web site, the designer thus needs to incorporate the necessary conditions in all SeRQL queries involving the selection of movies (or any other content which may be age-restricted, e.g. adult actors).

A similar observation was made in (regular) software development, when considering different design concerns of an application: some concerns cannot be localized to a particular class or module; instead they are inherently distributed over the whole application. Such a concern is called a *cross-cutting* concern. To cleanly separate the programming code addressing this concern from the regular application code, Aspect-Oriented Programming[17] was introduced. Inspired by the principles of aspect-orientation, Hera-S provides (adaptation) design support to specify, in an aspect-oriented way, the different cross-cutting adaptation concerns by means of an aspect-oriented adaptation specification.

Applying aspect-orientation to extend an AM with different additional adaptation concerns is thus done by modeling each concern as an *aspect*. Each aspect is composed of a number of advice-pointcut pairs. In this setting, the notions of advice and pointcut are as follows:

- *Advice*: An advice specifies a particular transformation in terms of modifications to the different (navigational) elements of the AM. In most

cases, a single modification will add a single adaptation condition to certain navigational units or relationships in the form of a SeRQL query.

- *Pointcut*: A pointcut defines a query on the set of navigational units and relationships of an application model, which specifies exactly the elements to which a certain advice should be applied.

These advice/pointcut pairs can thus be used to inject adaptation conditions to (certain elements of) the AM. It is a current research topic to investigate the limitations of this process of transforming these adaptation conditions in the corresponding SeRQL queries, a restricted form of what is called *weaving* in aspect-terminology.

To exemplify this approach, we illustrate how two additional adaptation concerns, age-group (restriction) and device-dependence, can be specified in an aspect-oriented way over an (existing, in this case non-adaptive) AM. For the first adaptation aspect, namely age-group (restriction), let us express the motivating example mentioned earlier in this section: restrict visibility of all adult-rated, i.e. 'NC-17'-rated, content throughout the application, and only show it when the user's age has been confirmed above 17. In an aspect-oriented way, this adaptation strategy is specified as follows:

```
POINTCUT SET WITH PARENT cm:movie
ADVICE
   SELECT M
   FROM {M} am:MPAA-rating {R}; rdf:type {imdb:Movie}
   WHERE R != 'NC-17'
   OR EXISTS
      (SELECT * FROM  {$U} cm:age {G}
       WHERE G > 17)
```

This pointcut-advice pair specifies first the pointcut: wherever in the AM a navigational unit is used that represents a set of movie elements (i.e. the pointcut part). In all these places, (in the advice) a condition is added in the form of a SeRQL expression, which denotes that the age (an attribute from the CM) should be over 17 to view 'NC-17' rated material. Similar pointcut-advice pairs can be specified to restrict visibility of items with other MPAA-ratings. Note that any movie set, where-ever it appears in the AM, is restricted: the adaptation is not localized to one particular navigational unit, and is thus truly cross-cutting.

The semantics of this condition addition is that this query expression is performed after the one that was defined originally for this element. Concretely, interpretation and execution of the above aspect result in modification of the SeRQL queries instantiating (a set of) movies. Note that the last example of section 4.3 is one particular occurrence of such a set of movies, for which the adaptation was manually specified by the designated

SeRQL query. However, to achieve automatic *weaving* of aspects, a more generic *pipeline approach* is best suited. In this approach, the original (non-adaptive) query expression *Q* producing the set of instances (i.e. movies in this case) is taken as a starting point. Subsequently each adaptation condition $C_i$ specified for this set gives rise to a SeRQL query $Q_i$ which takes as input the result of the previous query $Q_{(i-1)}$, and filters from this result the element according to the adaptation condition $C_i$. For adaptation conditions $C_1 \ldots C_n$ specified for a set, and possibly originating from different adaptation issues, the resulting query will thus be of the form: $Q_n \circ Q_{(n-1)} \circ \ldots \circ Q_1 \circ Q$. Evidently, other approaches, such as query re-writing or query merging are possible.

A second example concerns device-dependence: in order not to overload small screen users, we decide not to show pictures. Therefore, we can specify the following pointcut-advice pair:

```
POINTCUT ATTRIBUTE
ADVICE
   SELECT P
   FROM {P} hera:Mime {} hera:mimeType {T}
   WHERE T != 'image\*'
   OR EXISTS
      (SELECT * FROM {$U} cm:device {D}
        WHERE D != 'pda')
```

In the pointcut, all attributes from the AM are selected. For these attributes, in the advice part, the picture attributes (denoted by the mime-type specification as mentioned in Section 4.2) are filtered out for PDA-users, restricting their visibility for these PDA-users. Note that once again, our example addresses a truly cross-cutting concern: anywhere in the AM where a picture attribute is used, it will be filtered out for PDA-users.

To conclude this section on aspect-oriented adaptation support we would like to point out that the primary way of defining adaptation, namely to manually specify it in the AM by means of SeRQL queries, as was illustrated in Section 4, is still available to the designer. The aspect-oriented support presented here merely offers the designer an additional and alternative means of specifying, in a straightforward and distributed way, the adaptation conditions for (sets of) AM elements.

# 6. IMPLEMENTATION

After illustrating the AM design, we now turn to the engine implementing the model, i.e. generating the hypermedia views over the content according to what is specified in the AM. As explained in Section 1, we distinguish three implementations for the Hera models. The Hera-S implementation is based on our previous experiences with the Hera Presentation Generator (HPG)[1] an environment that supports the construction of WIS using the Hera methodology. Considering the technologies used to implement Hera's data transformations we distinguish two variants of the HPG: HPG-XSLT, which implements the data transformations using XSLT stylesheets, and HPG-Java, which implements the data transformations using Java. In HPG-XSLT we employed as an XSLT processor Saxon, one of the most up-to-date XSLT processors implementing XSLT 2.0. In HPG-Java we used two Java libraries, the Sesame library for querying the Hera models and the Jena library for building the new models. Hera-S resembles in many ways HPG-Java but it is based on the revision of the Hera models presented in this paper.

## 6.1 HPG-XSLT and HPG-Java

First we take a look at the tools for HPG-XSLT and HPG-Java that build the foundation for Hera-S. HPG-XSLT has an intuitive designer's interface, visualizing the Hera pipeline for the development of a Web application (see figure X-8). The user is guided in a sequence of steps to create the complete Web application, which, generated with XSLT stylesheets, results in a concrete, but static, Web site for a given platform. In the interface we see that each step in this advanced HPG view is associated with a rectangle labeled with the step's name (e.g., Conceptual Model, Unfolding AM, Application Adaptation, etc.). In each step there are a number of buttons connected with within-step arrows and between-step arrows that express the data flow. Such a button represents a transformation or input/output data depending on the associated label (e.g., Unfold AM is a transformation, Unfolding sheet AM is an input, and Unfolded AM is an output). The arrows that enter into a transformation (left, right, or top) represent the input and the ones that exit from a transformation (bottom) represent the output. The last step is the generation of the presentation in the end-format (e.g. HTML).

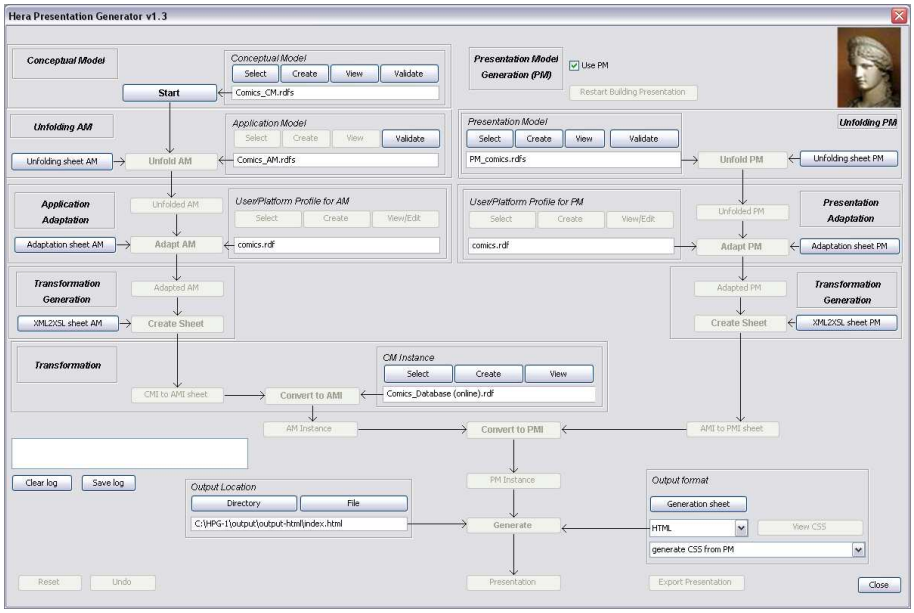


*Figure 10-8.* Screenshot HPG-XSLT

The models for HPG-XSLT can be created by hand but also with the help of Microsoft Visio templates, which provide a graphical environment aiding the correct construction of the different models (see figure X-9 for an example of the corresponding AM-builder).

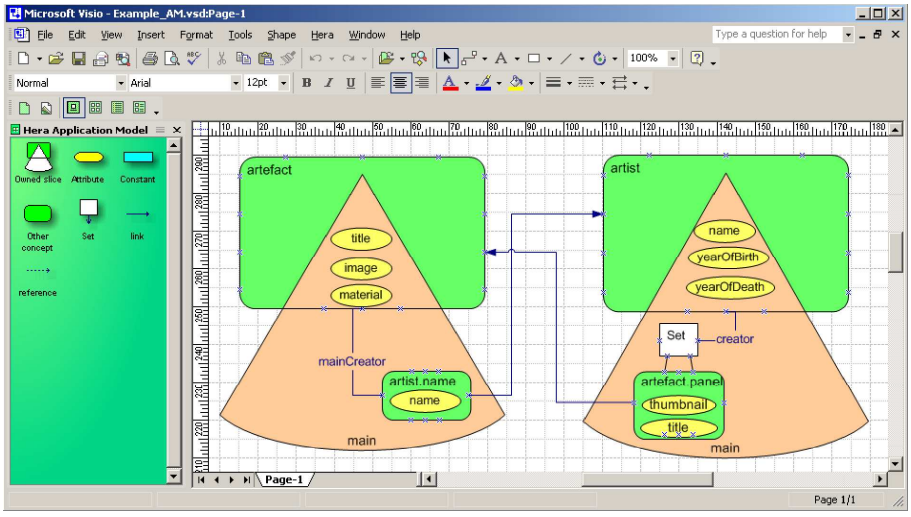


*Figure 10-9.* Visio templates HPG-XSLT

HPG-XSLT is an effective demonstration tool and sufficient for simple Web site applications. However, WIS may require more flexibility and more dynamics, for the application to be able to dynamically change in an ever-changing environment. A stand-alone client creating static Web pages was not enough for this purpose, so we created a server-side engine that evaluates every page request and dynamically creates an adapted (e.g. personalized) page; this engine was called HPG-Java. HPG-Java is a Java Servlet that can be run within a Servlet container Web server like Apache Tomcat. The application can be configured by the Hera models and a basic configuration file that indicates where on the server these models are provided (and some additional database settings). The server-side version allows data to be updated based on the user behavior, providing data for the sake of personalization. HPG-Java does not provide a designer-platform, but the designer can use an adapted version of the Visio templates to create the models.

In order to give the reader some indication of the dynamics (based on queries) provided by HPG-Java we use the example from figure X-10. On the left hand side of the figure there is the current page and on the right hand side the next page is shown that needs to be computed. When the user presses the "Add order" button there are two queries that are executed. The first query builds a new order and the second query adds the currently created order to the trolley. Based on this newly computed data the next page is generated. This new page displays the list of ordered paintings (based on the orders in the trolley) and also provides two forms. The first form allows the user to select the next painting and the second form enables the user to delete an order previously added to the trolley.

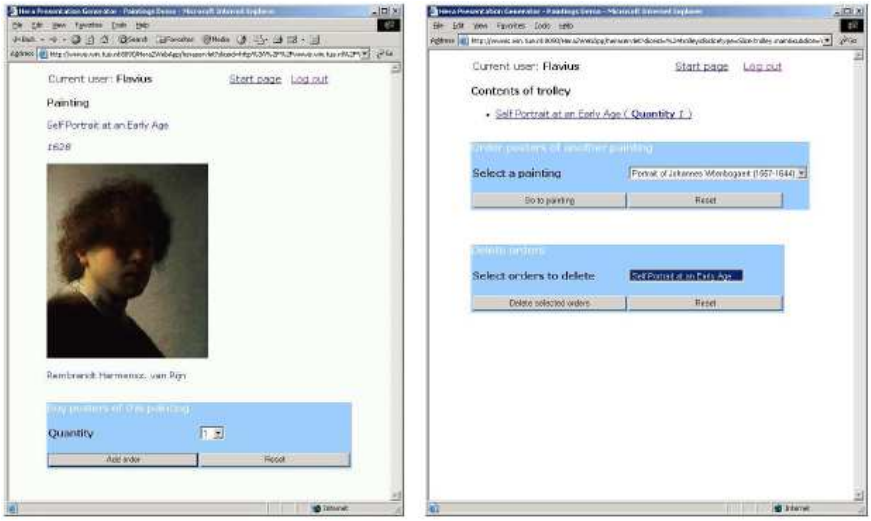


*Figure 10-10.* Web Application served by HPG-Java

## 6.2 Hera-S

As we have explained earlier, the architecture of the Hera-S version is similar to that from the earlier HPG-Java version, but obviously it accomodates the newer Hera-S AMs with their SeRQL queries. A major difference is that the AM is less tightly coupled to the domain, in the sense that the designer has more freedom of selecting elements and concepts in the domain by the use of the SeRQL queries. Furthermore, we allow the domain to be any repository of RDF data (i.e. no proprietary data model). Note that this does not only apply to the domain, but also to the context. The implementation is again a Java servlet, however, storage and manipulation of all the metamodels is now handled by different Sesame repositories.

Furthermore, the Hera-S implementation concentrates on the application model level only. Several presentation modules exist (Section 7 will go into more details) that can configure the presentation of AM data, each with outstanding features that might be more appropiate in different situations. Having separate implementations allows a presentation module to be plugged into the pipeline that fits the situation, and thus also making Hera-S more platform independent.

In figure X-11 we see the main components that make up the Hera-S implementation architecture. The domain model (DM), application model (AM) and all context data are realized as Sesame repositories, exploiting Sesame's capability that enables storage, reasoning and querying of RDF and OWL data.

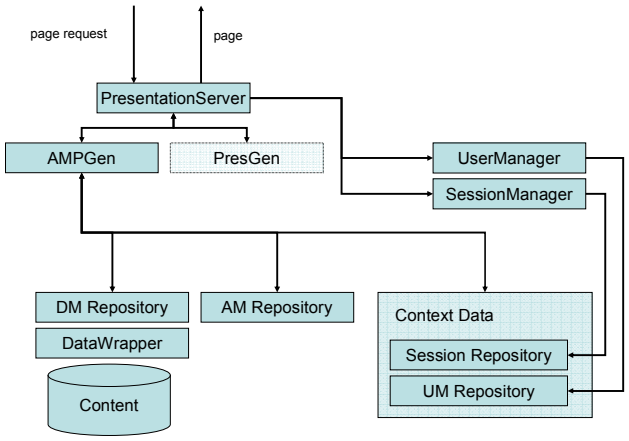


*Figure 10-11.* Hera-S architecture

The content is interfaced to the rest of the system through the DM repository. A major advantage of this approach is that integrated querying of both schema and instance data becomes possible. To enable this, the content has to be represented as RDF statements. For non-RDF content repositories this can be achieved in various ways. The simplest and most straightforward case is an offline translation of the data to RDF and simply storing that RDF in a Sesame repository. However, this approach has a drawback for certain cases by duplicating data which means that updates to the data need to happen in two places. An alternative way of realizing the link between DM and data is by creating a wrapper component around the actual data source that does online back-and-forth translation. The Sesame architecture caters for this scenario by having a storage abstraction layer called the SAIL[4] API. A simple wrapper component around virtually any data source can be realized as a SAIL implementation and then effortlessly be integrated into the rest of the Sesame framework and thus into our Hera-S environment.

The entire system has an event-driven architecture. When a request for a certain page comes in at the Presentation Server component, the request is translated to a request for an AMP. At the same time, the UserManager and SessionManager components are informed of the request. These two manager components can then take appropriate actions in updating the context data repositories (specifically, the UM (user model) Repository, and the Session Repository).

Independently from this, the so-called AMPGen component retrieves the requested part of the AM that contains the conceptual specification which is the basis for the next AMP. It then starts the AMP creation process by following that specification.

The actual AMP is internally implemented as a volatile (in-memory) Sesame repository, which means that all transformation operations on it can simply be carried out as RDF queries and graph manipulations using the SeRQL query language. When the AMP has been fully constructed, it is sent back to the presentation generation component. This presentation generation component can then transform the AMP into an actual page (in terms that a thin client such as a Web browser can understand, e.g. XHTML). The result is then finally sent back to the client (as the response).

In the Hera-S system, SeRQL query expressions are extensively used to define mappings and filters between the different data sources and the eventual AMP. Since all this data is expressed as RDF graphs, an RDF query/transformation language is a natural choice as a mapping tool.

In section 7 we will describe a presentation generation process that uses the AMP as input to generate a presentation for different user platforms. It will also show a screenshot from an application that can be generated with the Hera-S engine in combination with that presentation generation solution.

Via the feedback mechanism, built-in as parameters in the links, user actions will trigger subsequent actions in the Hera-S engine (i.e. presentation generation typically does not interfere with this process).

# 7. PRESENTATION DESIGN

In this section we address one particular approach to presentation design. The presentation design step of Hera bridges the gap between the logical level from the AM and the actual implementation. If needed, the Presentation Model (PM) can specify the details of this transformation. Complementary to the AM, where the designer is concerned with the structure of the information and functionality as it needs to be presented to the user (by identifying navigational units and relationships), the PM specifies how the content of those navigation units is displayed. According to these specifications, AMP's can be transformed to a corresponding Web presentation in a given output format, e.g. XHTML, cHTML, WML etc. We do stress that we foresee multiple alternative ways to render AMP's in specific output formats and the designer is free to choose a way to configure this transformation of AMP's into output. In this section we illustrate one particular way, which uses a PM to detail the presentation design and which is implemented using a particular document format for adaptive Web presentations.

## 7.1 Presentation Model Specification

The PM is defined by means of so-called regions and relationships between regions. Regions are abstractions for rectangular parts of the user display and thus they satisfy browsing platform constraints. They group navigational units from the AM, and like navigation units, regions can be defined recursively. They are further specified by a layout manager, a style, and references to the navigational units that they aggregate. We note that the usage of layout managers was inspired by the AMACONT project's component-based document format[18], adopting its abstract layout manager concept in the Hera PM. As will be explained later (Section 7.2), this enables to use AMACONT's flexible presentation capabilities for the generation of a Web presentation.

Figure X-12 shows an excerpt of the PM for the running example, i.e. the regions associated to the 'MovieUnit' navigational unit and its subregions.

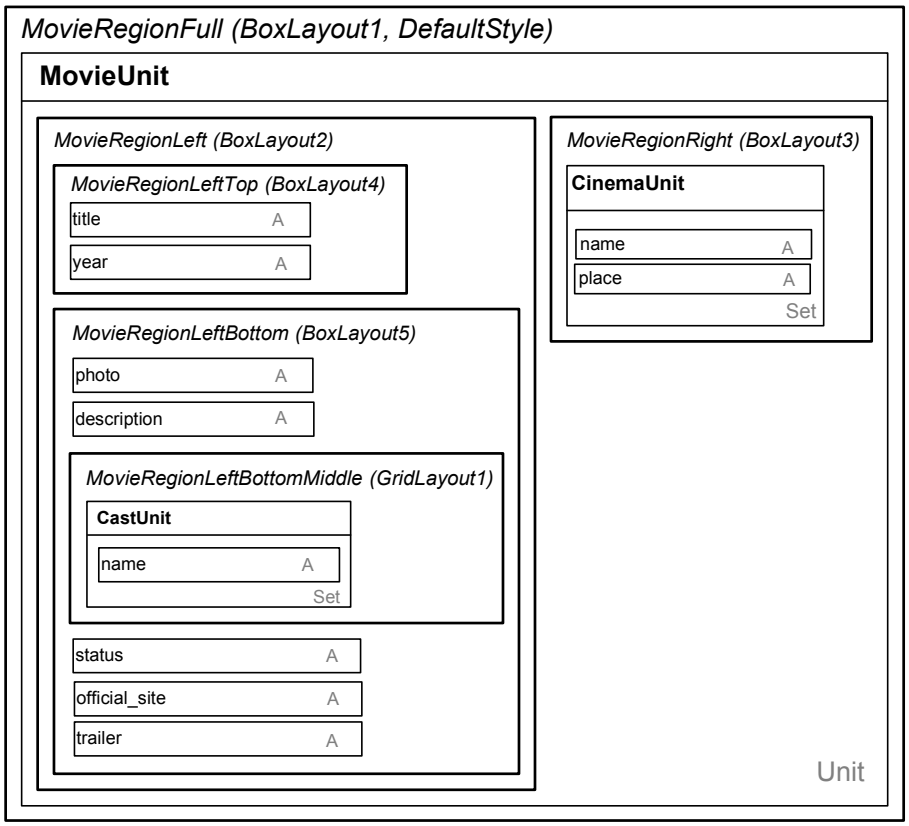


*Figure 10-12.* Presentation Model for the 'MovieUnit' Navigational Unit

The 'MovieUnit' navigational unit is associated with the region called 'MovieRegionFull'. It uses the layout manager 'BoxLayout1' for the arrangements of its subregions ('MovieRegionLeft' and 'MovieRegionRight'), and the style given by 'DefaultStyle'. 'BoxLayout1' is an instance of the layout manager class 'BoxLayout' that allows to lay out the subregions of a region either vertically or (as in this case) horizontally. The style describes the font characteristics (size, color), background (color), hyperlink colors etc. to be used in a region. The definition of styles was inspired by Cascading Style Sheets (CSS)[19]. We chose to abstract the CSS formatting attributes because (1) not every browser supports CSS at the current moment and (2) we would like to have a representation of the style that can be customized based on user preferences.

Both 'MovieRegionLeft' and 'MovieRegionRight' use 'BoxLayout's with a vertical organization of their inner regions. For the 'title' and 'year' attributes 'BoxLayout4' is used, which specifies a horizontal arrangement.

For ‘photo’, ‘description’, the region containing the names in the cast, ‘status’, ‘official_site’ and ‘trailer’ it is used ‘BoxLayout5’ which states a vertical arrangement. The names in the cast are organized using ‘GridLayout1’ (an instance of the layout manager class ‘GridLayout’), a grid with 4 columns and an unspecified number of rows. The number of rows was left on purpose unspecified as one does not know a priori (i.e. before the presentation is instantiated and generated) how many names the cast of a movie will have. The regions that do not have a particular style associated with them inherit the style of their container region. Note that in figure X-12 we have omitted constant units (e.g., ‘(‘, ‘)’, ‘Cast’, etc.) in order to simplify the explanation.

Besides the layout manager classes exemplified in figure X-12, the definition of PM supports additional ones. ‘BorderLayout’ arranges subregions to fit in five directions: north, south, east, west, and center. ‘OverlayLayout’ allows to present regions on top of each other. ‘FlowLayout’ places the inner regions in the same way as words are placed on a page: the first line is filled from left to right and the same is done for the second line etc. ‘TimeLayout’ presents the contained regions as a slide show and can be used only on browsers that support time sequences of items, e.g., HTML+TIME[20] or SMIL[21]. Due to the flexibility of the approach, this list can be extended with other layout managers that future applications might need.

The specification of regions allows defining the application’s presentation in an implementation-independent way. However, to cope with users’ different layout preferences and client devices, Hera-S also supports different kinds of adaptation in presentation design. As an example, based on the capabilities of the user’s client device (screen size, supported document formats etc.), the spatial arrangement of regions can be adapted. Another adaptation target is the corporate design (the “look-and-feel”) of the resulting Web pages. According to the preferences and/or visual impairments of users, style elements like background colors, fonts (size, color, type), or buttons can be varied. For a thorough elaboration of presentation layer adaptation the reader is referred to Fiala et al.[22].

Turning back to our running example, we now consider how to adapt the PM for the ‘MovieUnit’ navigational unit to the typical small display size and horizontal resolution of a handheld device. With respect to this, we aim at replacing the layout managers ‘BoxLayout1’ and ‘GridLayout1’ with ‘BoxLayout’s specifying vertical arrangements for their containment elements. Note that the PM facilitates to specify such adaptations by the assignment of multiple layout or style alternatives (variants) as simple conditions attached to "region-layout manager assignments", in correspondence with the adaptation conditions of the AMACONT document format. These conditions are simple Boolean expressions consisting of

constants, arithmetic and logical operators, as well as references to context model parameters.

## 7.2 Presentation Generation Implementation

After the specification of the PM we now turn to its implementation. As mentioned above, we illustrate it by using the AMACONT project's component-based document model, which is perfectly suited for this task as this PM was based on AMACONT presentation principles in the first place. This approach aims at implementing personalized ubiquitous Web applications by aggregating and linking configurable document components. These are instances of an XML grammar representing adaptable content on different abstraction levels. Media components encapsulate concrete media assets (text, structured text, images, videos, HTML fragments, CSS) by describing them with technical metadata. Content units group media components by declaring their layout in a device-independent way. Document components define a hierarchy out of content units to fulfill a semantic role. Finally, the hyperlink view defines links that are spanned over components. For more details on the AMACONT document model the reader is referred to Fiala et al.[18].

Whereas the AMACONT document model provides different adaptation mechanisms, in this chapter we focus on its presentation support. For this purpose it allows to attach XML-based abstract layout descriptions (layout managers) to components. Document components with such abstract layout descriptions can be automatically transformed to a Web presentation in a given Web output format. As mentioned above, the PM was specified by adopting AMACONT's layout manager concept to the model level. This enables the automatic translation of AMP's to a component-based Web presentation based on a PM specification. The corresponding presentation generation pipeline is illustrated in figure X-13.

In a first transformation step (AMP to Component) the AMP's are translated to hierarchical AMACONT document component structures. Thereby, both the aggregation hierarchy and the layout attributes of the created AMACONT components are configured according to the PM configuration. Beginning at top-level document components and visiting their subcomponents recursively, the appropriate AMACONT layout descriptors (with adaptation variants) are added to each document component. This transformation can be performed in a straightforward way and was already described in detail by Fiala et al.[22]. The automatically created AMACONT documents are then processed by AMACONT's document generation pipeline. In a first step, all adaptation variants are resolved according to the current state of the context model. Second, a Web

presentation in a given Web output format, e.g. XHTML, XHTML Basic, cHTML or WML is created and delivered to the client.

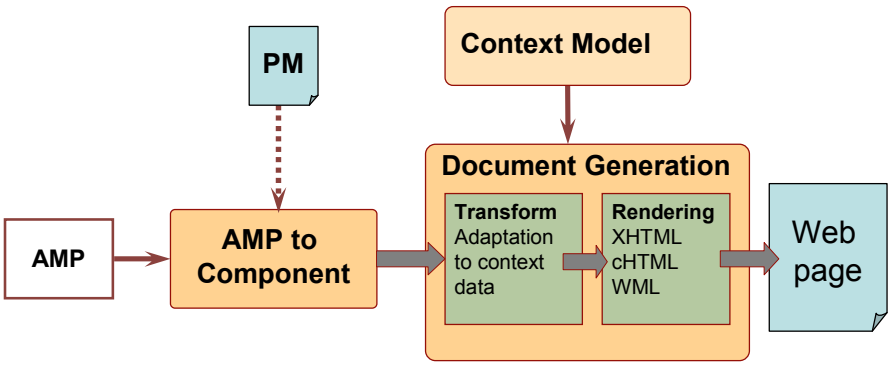


*Figure 10-13*. Presentation Generation with AMACONT

Figure X-14 illustrates the XHTML page generated for the PC version of our running example. It represents an instantiation of the Movie navigational unit with data used for "The Matrix" movie. It also shows the cinemas that are currently playing this movie.

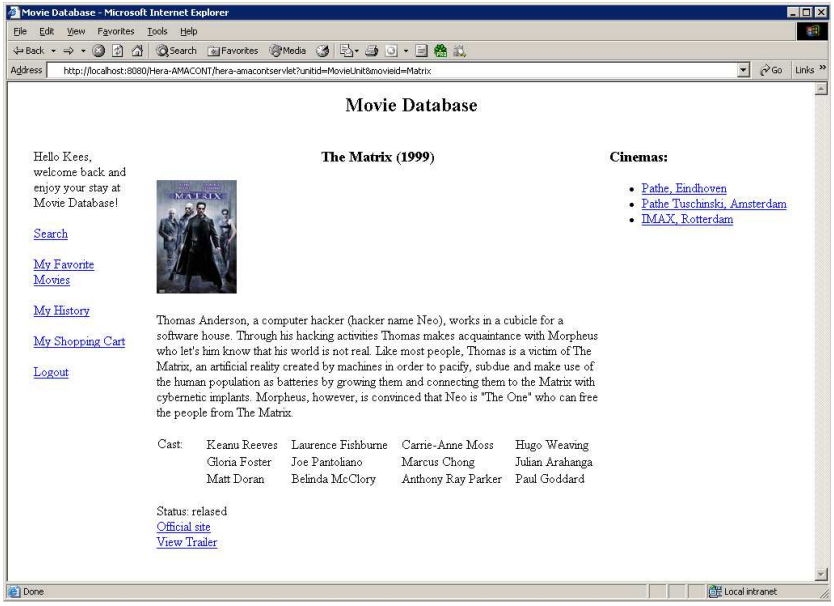


*Figure 10-14*. Presentation on PC

Note that the presentation of content elements corresponds to the PM specification that was illustrated in figure X-12.

As an alternative, figure X-15 shows the same page as presented on a PDA, exemplifying the layout adaptation. As can be seen, the resulting presentation is in correspondence with the PM adaptation specified above, i.e. all content elements are displayed below each other.

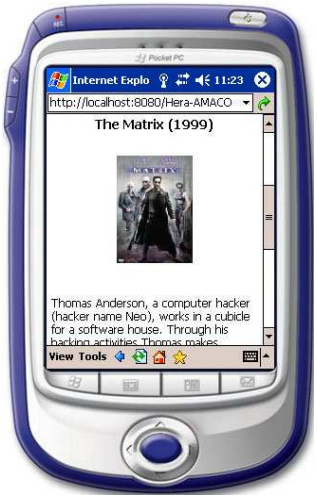


*Figure 10-15*. Presentation on PDA

## 8. SUMMARY

In this chapter we have discussed on the basis of the running example the models and tools that make up the Hera approach to WIS design. This approach is characterized by a focus on adaptation in the navigation design and a number of the facilities are motivated by the goals of this adaptation support. The most characteristic element of the approach is the choice to use RDF as the main language for expressing the domain and context data and the application model (AM) that defines the context-based navigation over and interaction with the content. Since the storage and retrieval of the RDF data involves the manipulation of RDF data we have chosen to use a Sesame-based approach, i.e. making the different RDF data models available as Sesame repositories. As a consequence of this we use SeRQL query expressions in the definition of the AM. With the RDF and SeRQL expressions, we have models that allow a more fine-grained specification of adaptation and context-dependency. Also we can exploit more extensively

the interoperability of RDF data, for example when integrating data sources (e.g. for background knowledge), and interfering with the data processing independently from the navigation. This enables a clean separation of concerns that helps in personalization and adaptation and in the inclusion of external data sources.

## 9. REFERENCES

1. Frasincar, F.: Hypermedia Presentation Generation for Semantic Web Information Systems, PhD Thesis, Chapter 4 (Hera Presentation Generator), pp. 67-87, Eindhoven University of Technology Press Facilities, ISBN 90-386-0594-3, 2005.
2. Klyne, G. and Carrol, J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, February 10 2004. See http://www.w3.org/TR/rdf-concepts/.
3. Brickley, D. and Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, Februrary 10 2004. See http://www.w3.org/TR/rdf-schema/.
4. Broekstra, J., Kampman, A. and van Harmelen, F.: Sesame: An Architecture for Storing and Querying RDF and RDF Schema. In Proceedings of the First International Semantic Web Conference (ISWC 2002), Sardinia, Italy, June 9-12 2002, p. 54-68. Springer-Verlag Lecture Notes in Computer Science (LNCS) no. 2342. See also http://www.openrdf.org/.
5. Broekstra, J.: SeRQL: A Second-Generation RDF Query Language. Chapter 4 in Storage, Querying and Inferencing for Semantic Web Languages. PhD Thesis, Vrije Universiteit Amsterdam (July 2005). ISBN 90-9019-236-0. See also http://www.openrdf.org/doc/SeRQLmanual.html.
6. Dean, M. and Schreiber, G.: The OWL Web Ontology Language Reference. W3C Recommendation, February 10 2004. See http://www.w3.org/TR/owl-ref/.
7. Hart, L., Emery, P., Colomb, B., Raymond, K., Taraporewalla, S., Chang, D., Ye, Y., Kendall, E., Dutra, M.: OWL full and UML 2.0 compared, March 2004, available at: http://www.itee.uq.edu.au/~colomb/Papers/UML-OWLont04.03.01.pdf
8. Protégé, The Protégé Ontology Editor and Knowledge Acquisition System, available at: http://protege.stanford.edu
9. Miller, G.A.: Wordnet: a lexical database for english. Commun. ACM 38(11) (1995) pp. 39-41

10. RDF representation of Wordnet, available at: http://www.semanticweb.org/library/
11. Hobbs, J.R., Pan, F.: An ontology of time for the semantic Web. ACM Transactions on Asian Language Information Processing (TALIP) 3(1) (2004) p.66-85
12. Teknowledge Geographical Ontology, available at: http://reliant.teknowledge.com/DAML/Geography.owl
13. Chipman, A., Goodell, J., Harpring, P., Beecroft, A., Johnson, R., Ward, J.: Getty thesaurus of geographic names: editorial guidelines. Available at: http://www.getty.edu/research/conducting_research/vocabularies/guidelines/tgn_1_contents_intro.pdf. (2005)
14. SIMILE | RDFizers, 2006, available at: http://simile.mit.edu/RDFizers/
15. Thiran, Ph., Hainaut, J.L., Houben, G.J.: Database Wrappers Development: Towards Automatic Generation. in: CSMR'05, Ninth European Conference on Software Maintenance and Reengineering, Manchester, UK, 21-23 March 2005, pp. 207-216, 2005, IEEE CS Press.
16. Beckett, D.:S Turtle: Terse RDF Triple Language. Technical Report. See http://www.dajobe.org/2004/01/turtle/.
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97), Jyväskylä, Finland (1997) 220-242
18. Fiala, Z., Hinz, M., Meißner, K. and Wehner, F.: A Component-based Approach for Adaptive, Dynamic Web Documents; Journal of Web Engineering, Vol.2 No.1&2, (pp058-073), Rinton Press, September, 2003
19. Bos, B., Çelik, T., Hickson, I. and Lie, H.W.: Cascading Style Sheets, level 2 revision 1 CSS 2.1 Specification, W3C Working Draft 13 June 2005
20. Schmitz, P., Yu, J. and Santangeli, P.: Timed Interactive Multimedia Extensions for HTML (HTML+TIME), W3C Note 18 September 1998.
21. Bulterman, D., Grassel, G., Jansen, J., Koivisto, A., Layaïda, N., Michel, T., Mullender, S., Zucker, D.: Synchronized Multimedia Integration Language (SMIL 2.1), W3C Recommendation 13 December 2005
22. Fiala, Z., Frasincar, F., Hinz, M., Houben, G.J., Barna, P., Meissner, K.: Engineering the Presentation Layer of Adaptable Web Information Systems; ICWE 2004, International Conference on Web Engineering , July 28-30 2004, Munich, Germany, Springer-Verlag Berlin Heidelberg 2004, LNCS 3140, pp. 459–472