Chapter 4

Hera Presentation Generator

The Hera Presentation Generator (HPG) is the integrated development environment that supports the Hera methodology. It is based on a number of software tools created for the Hera methodology that we integrated into one common environment. The HPG fills the existing gap for development environments supporting SWIS design. There are two versions of HPG: HPG-XSLT and HPG-Java. HPG-XSLT corresponds to the static variant of the Hera presentation generation phase and HPG-Java corresponds to the dynamic variant of the Hera presentation generation phase. A comparison of the two implementations based on their advantages and disadvantages is given. We also present a distributed architecture for the HPG based on Web Services.

4.1 Introduction

The success of a WIS design methodology is often depending on the existence of software tools that support the proposed methodologies. As shown in Chapter 2, many of the model-driven methodologies for SWIS design do not provide integrated development environments (IDE) similar to ones found for WIS design (e.g., RMCase, WebRatio) in order to help the design and construction of SWIS. An IDE has the advantage of supporting all design steps of a methodology from a single tool.

In Chapter 3 it was presented the presentation generation phase of Hera, a SWIS design methodology. The presentation generation phase has two variants: a static variant in which the user is unable to influence the generated hypermedia presentation, and a dynamic variant that considers user input before each hypermedia page is generated. The static variant uses XSLT data transformations and the dynamic variant uses Java data transformations.

The Hera Presentation Generator (HPG) is an IDE to support the development of SWIS using the Hera methodology. Based on the two Hera implementation variants, two versions of the HPG were realized: HPG-XSLT which corresponds to the static variant, and HPG-Java which corresponds to the dynamic variant.

The remainder of this Chapter is structured as follows. Section 4.2 describes HPG-XSLT. Section 4.3 presents HPG-Java. The two version of HPG are compared in Section 4.4 Section 4.5 shows a Web Service-Oriented Architecture for HPG. Section 4.6 concludes the chapter and presents future work.

4.2 HPG-XSLT

HPG-XSLT is an IDE that assists the designer of the static variant of the Hera presentation generation phase. It integrates several tools built during the last couple of years in the Hera project into one common environment. Besides its practical purpose, HPG-XSLT has also an explanatory purpose as it offers an explicit view over the data flow in the Hera presentation generation phase.

HPG-XSLT has the following graphical interfaces: *CM design interface*, *AM design interface*, *PM design interface*, *UP design interface*, and *implementation interface*, that correspond to the design steps in the presentation generation phase of Hera. For building (and visualizing) the CM, AM, and PM several Visio solutions were implemented. We chose to build the Hera models using Visio because: (1) Visio is widely used in industry, (2) it provides a graphical interface to build model specific shapes, (3) it is based on a simple programming language, i.e., Visual Basic, which makes it easy for one to define the behavior of the application. A solution is composed of a stencil (which has the model shapes) and a template (which has the load/export feature for the RDF/XML serialization of models). At the current moment the adaptation conditions are not supported by the Visio solutions, the designer has to insert them after a model is exported.

Each model needs to fulfill a set of model constraints. In case that the designer uses the HPG-XSLT interfaces to develop models these constraints are automatically fulfilled as they are enforced during model construction. Nevertheless, in case that the designer uses a different tool to build models, the resulted specifications need to be checked if they fulfill their associated constraints. For this purpose a separate Java program based on Jena [Hewlett-Packard Development Company, LP, 2005] was developed for checking the constraints of a model.

4.2.1 CM Design Interface

HPG-XSLT provides a graphical interface for building CM. Figure 4.1 shows a snapshot of the CM design interface. On the left-hand side there is the stencil that contains shapes for all CM elements. On the right-hand side there is the drawing frame in which a CM is built. For all drawn shapes one can set specific properties to them (e.g., for a concept relationship shape there are attributes that specify the inverse and cardinality of this relationship).

Some of the CM constraints that this interface enforces are: (1) only the media types defined in the media vocabulary can be used for attributes, (2) all concept relationships need to have a domain and a range , (3) every concept relationship needs to have its inverse

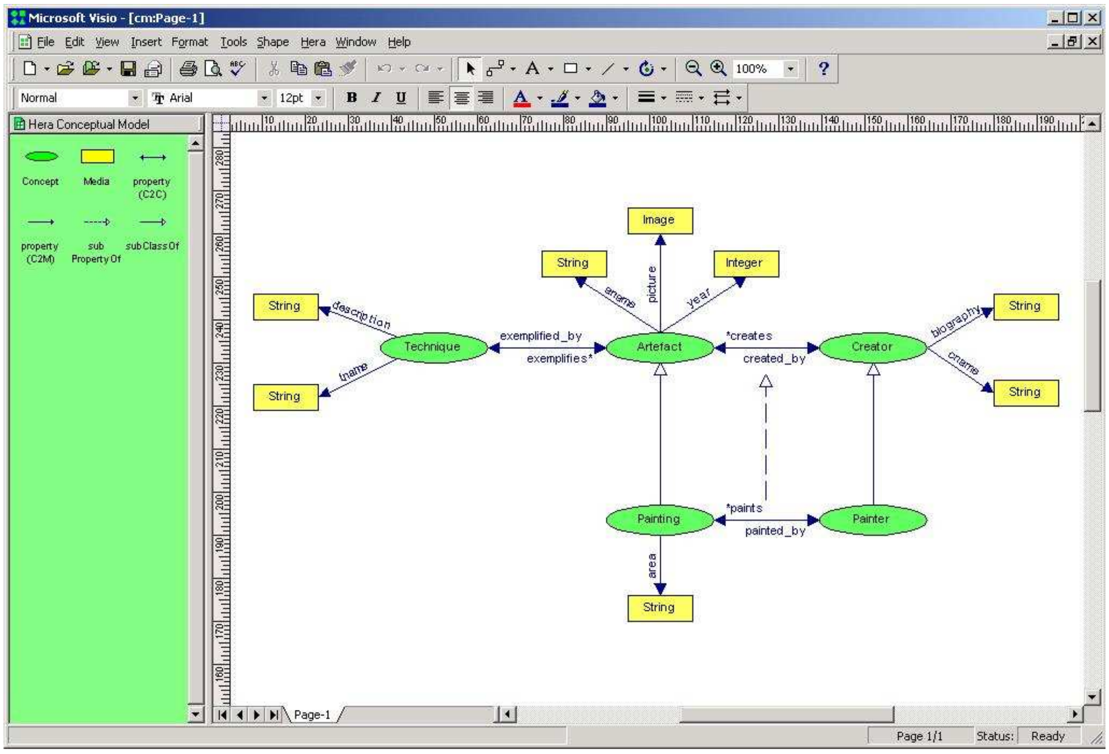


Figure 4.1: Conceptual model interface.

relationship defined, (4) every concept relationship needs to have its cardinality specified, etc.

4.2.2 AM Design Interface

HPG-XSLT provides a graphical interface for building AM. Figure 4.2 shows a snapshot of the AM design interface. On the left-hand side there is the stencil that contains shapes for all AM elements. On the right-hand side there is the drawing frame in which an AM is built. For all drawn shapes one can set specific properties to them (e.g., for a slice shape there is an attribute that specifies the name of the owner concept).

Some of the AM constraints that this interface enforces are: (1) only concepts defined in the associated CM can be used as owners of slices, (2) all region relationships need to have a source and a destination, (3) slices related by slice aggregation relationship need to specify a valid concept relationship between the slice owners, if the owners are different, (4) all slices can be reached from the start main slice, etc.

4.2.3 PM Design Interface

HPG-XSLT provides a graphical interface for building PM. Figure 4.3 shows a snapshot of the PM design interface. On the left-hand side there is the stencil that contains shapes

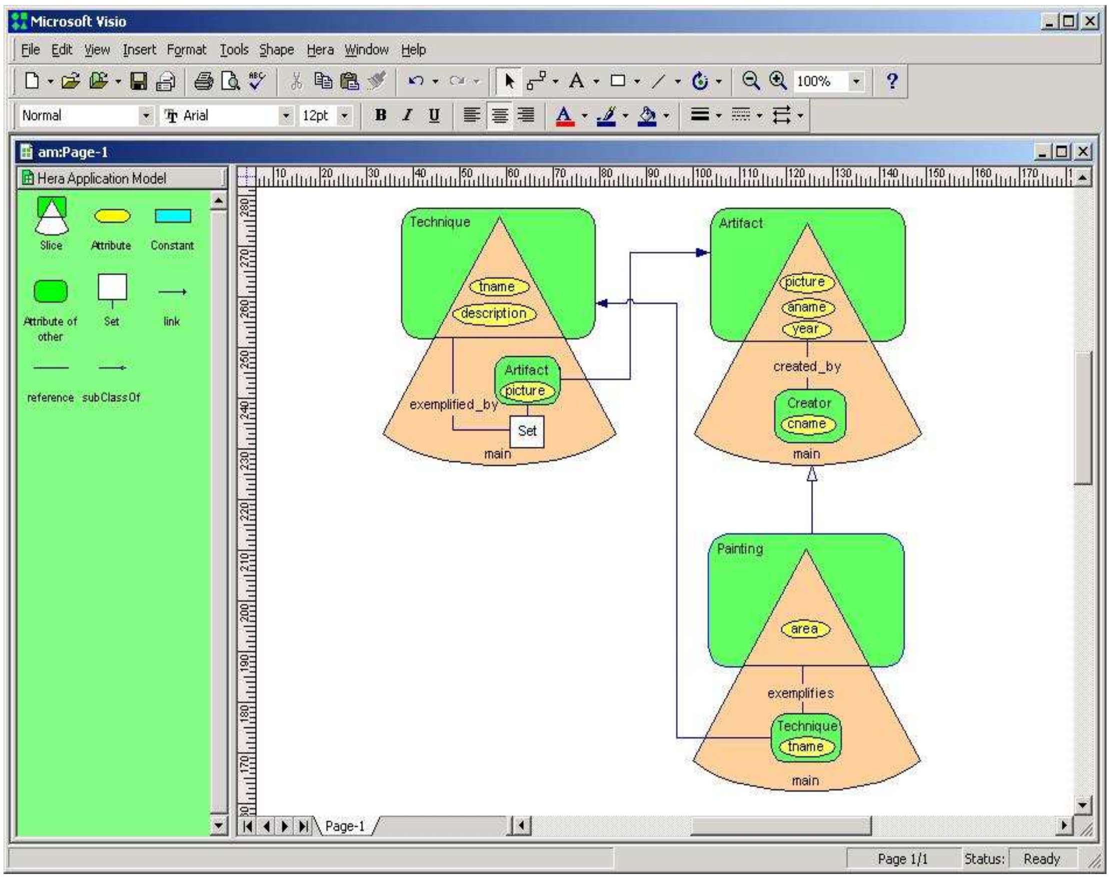


Figure 4.2: Application model interface.

for all PM elements. On the right-hand side there is the drawing frame in which a PM is built. For all drawn shapes one can set specific properties to them (e.g., for a region shape there is an attribute that specifies the name of the owner slice). The layout and style information are associated to a region by means of shape attributes.

Some of the PM constraints that this interface enforces are: (1) only the slices defined in the associated AM can be used as owners of regions, (2) all region relationships need to have a source and a destination, (3) complex slices need to have the layout information specified, (4) all regions need can be reached from the start region, etc.

4.2.4 UP Design Interface

HPG-XSLT supports also the specification of a UP definition and of a UP instantiation. As shown in the previous chapter, the UP is used in the adaptation conditions in AM and PM. Taking this in consideration the UP was split in two parts, one relevant for AM and another part relevant for PM.

The profile definition is a CC/PP vocabulary [Klyne et al., 2004]. It defines three components: *HardwarePlatform*, *SoftwarePlatform*, and *User* (preferences). Each component

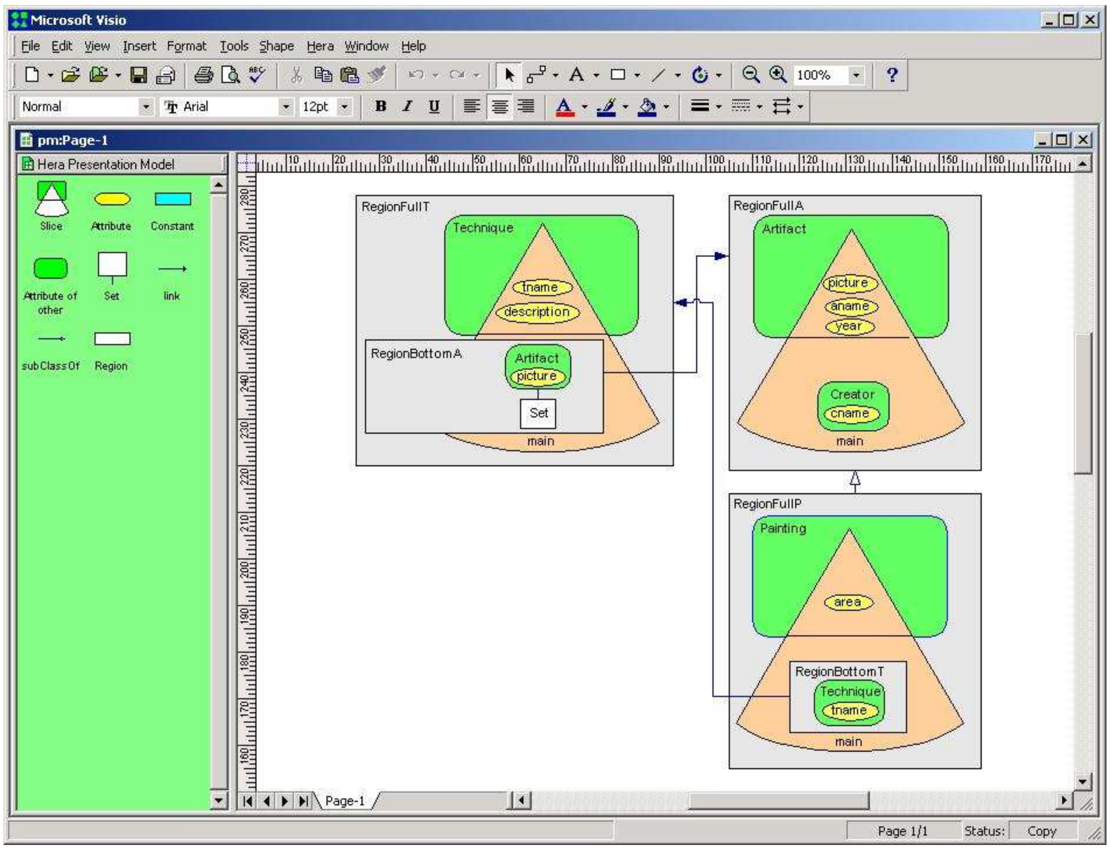


Figure 4.3: Presentation model interface.

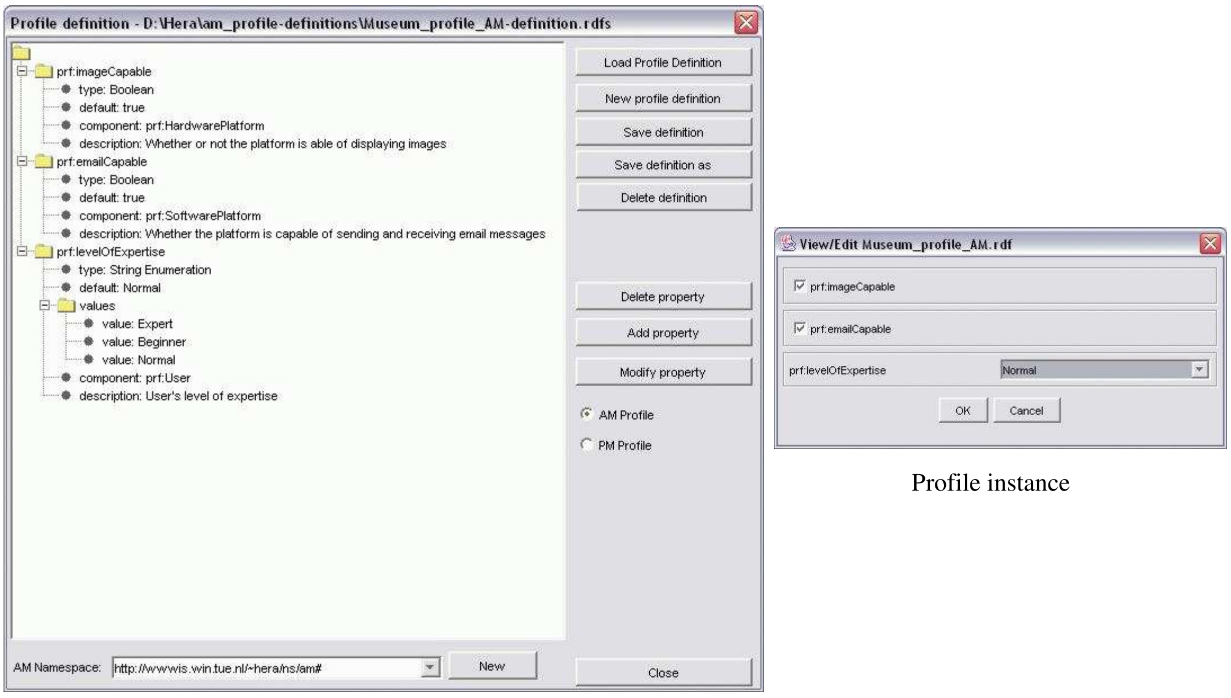
has a number of attributes for which the types are also specified. Among the supported types are *Boolean*, *Integer*, *String*, and *Enumeration*. The instantiation of this profile is done by means of an interface automatically generated from a profile definition.

Figure 4.4 shows the UP definition and instantiation for AM. In this UP, *imageCapable* is defined as a *HardwarePlatform* attribute with an *Boolean* type. For the instantiation of this attribute one can check the associated radio button to indicate the value *True* (otherwise the value is *False*). In the current example this attribute was set to *True*.

Figure 4.5 shows the UP definition and instantiation for PM. In this UP, *client* is defined as a *HardwarePlatform* attribute with an enumerated type *PC*, *PDA*, or *WAP phone*. For the instantiation of this attribute one can select only one of the three values of the attribute type. In the current example this attribute was set to *PC*.

4.2.5 Implementation Interface

Figure 4.6 shows the advanced-designer view in HPG-XSLT. Inexperienced designers will be presented with another interface which follows the popular wizard paradigm (in which the more complex user interface is split into a sequence of smaller, easy-to-use interfaces). As one can notice from Figure 4.6, this advanced view shows two important parts: a left-



Profile definition - D:\Hera\am_profile-definitions\Museum_profile_AM-definition.rdfs

```
prf:imageCapable
    type: Boolean
    default: true
    component: prf:HardwarePlatform
    description: Whether or not the platform is able of displaying images
prf:emailCapable
    type: Boolean
    default: true
    component: prf:SoftwarePlatform
    description: Whether the platform is capable of sending and receiving email messages
prf:levelOfExpertise
    type: String Enumeration
    default: Normal
    values
        value: Expert
        value: Beginner
        value: Normal
    component: prf:User
    description: User's level of expertise
```

AM Namespace: http://wwwis.win.tue.nl/~hera/ns/am#

[✓ AM Profile] PM Profile

View/Edit Museum_profile_AM.rdf

- [✓] prf:imageCapable
- [✓] prf:emailCapable
- prf:levelOfExpertise: Normal

Profile instance

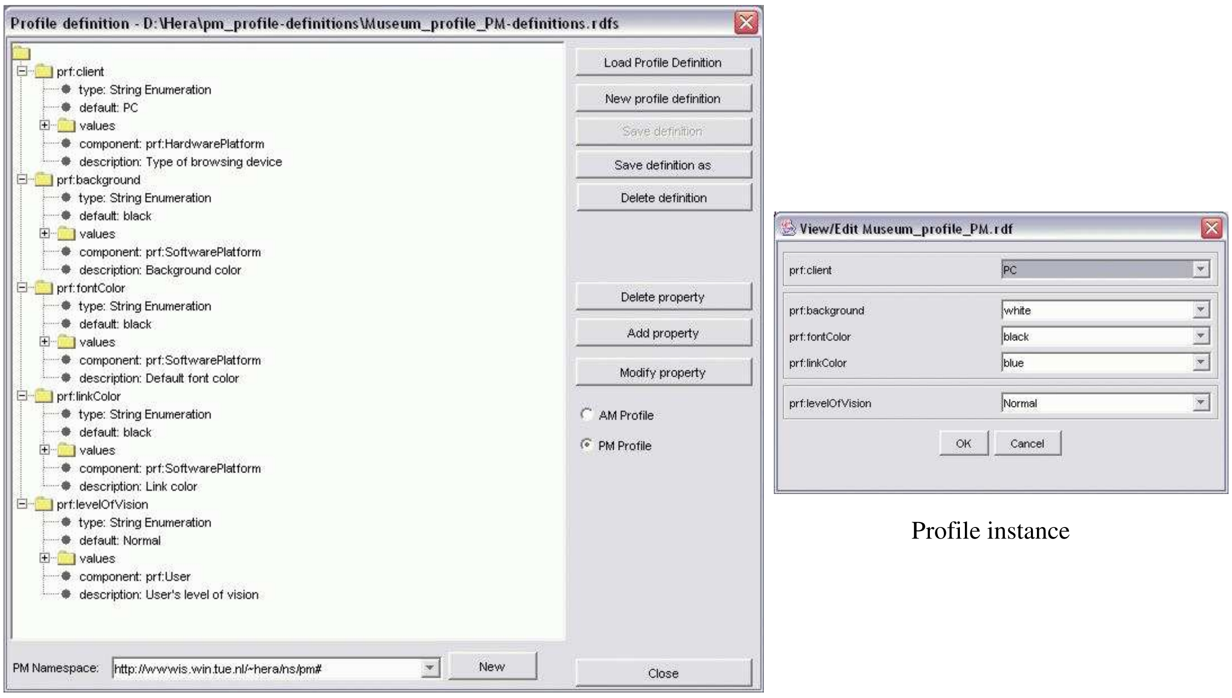
Profile definition

Figure 4.4: The user/platform profile for AM.

hand side responsible for converting a CM instance into an AM instance based on the AM and a right-hand side accountable for converting this AM instance into a Web presentation based on a PM.

Each step in this advanced HPG view has associated with it 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 an transformation (bottom) represent the output. The transformation steps that can be triggered at a given moment (all inputs are present) have their buttons enabled while the inhibited transformation steps (not all inputs are present) have their buttons disabled. These visual cues in the advanced view are extremely useful for the understanding and good functioning of the whole transformation process.

All models are represented in RDFS and model instances are represented in RDF; both models and model instances have corresponding RDF/XML serializations. In HPG-XSLT we use XSLT transformations in order to convert one RDF/XML file into another RDF/XML file. The XSLT stylesheet that drives such a transformation process is one of the transformation's inputs. All models and transformation specifications are available for



prf:client
- type: String Enumeration
- default: PC
- values
- component: prf:HardwarePlatform
- description: Type of browsing device

prf:background
- type: String Enumeration
- default: black
- values
- component: prf:SoftwarePlatform
- description: Background color

prf:fontColor
- type: String Enumeration
- default: black
- values
- component: prf:SoftwarePlatform
- description: Default font color

prf:linkColor
- type: String Enumeration
- default: black
- values
- component: prf:SoftwarePlatform
- description: Link color

prf:levelOfVision
- type: String Enumeration
- default: Normal
- values
- component: prf:User
- description: User's level of vision

PM Namespace: http://wwwis.win.tue.nl/~hera/ns/pm#

[✓ PM Profile]

prf:client	PC
prf:background	white
prf:fontColor	black
prf:linkColor	blue
prf:levelOfVision	Normal

Profile instance

Profile definition

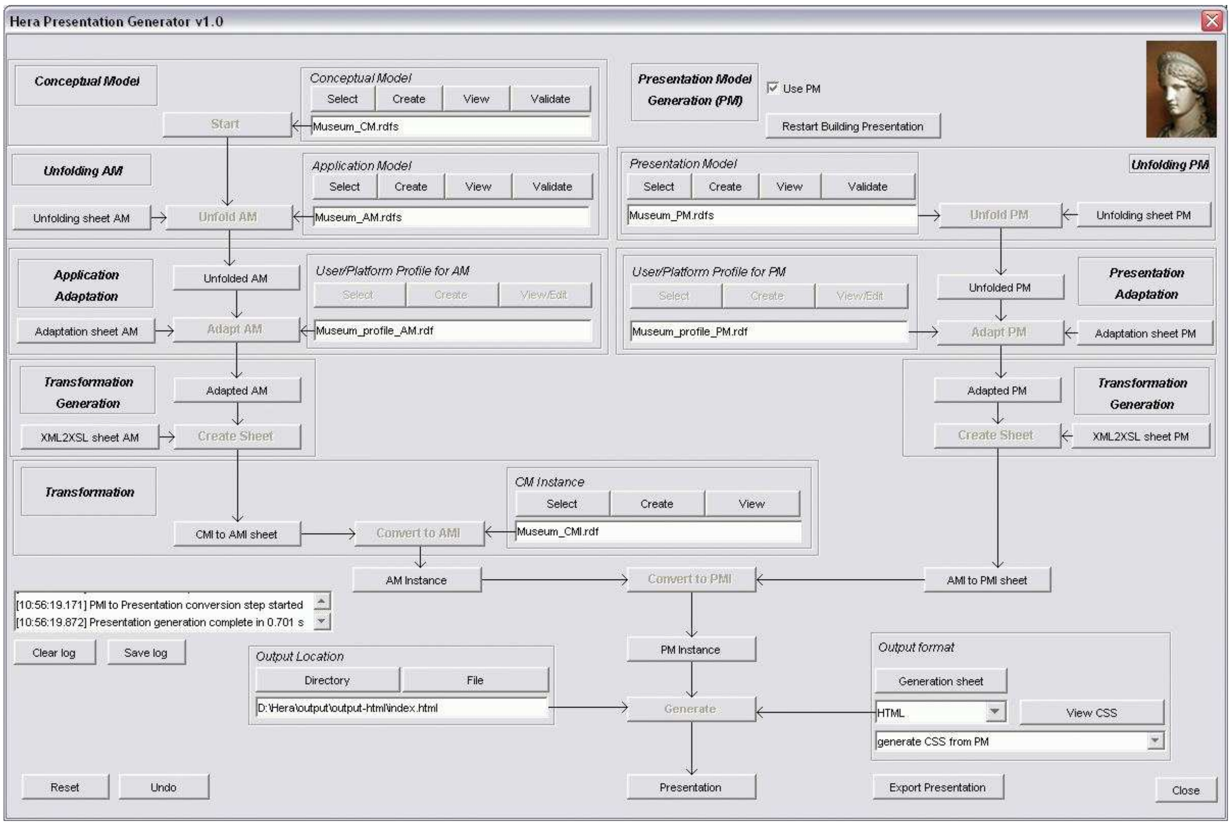
Figure 4.5: The user/platform profile for PM.

inspection: the *View* button is used to display models, or specific buttons labeled with the name of the model (*AM Instance*, *PM Instance*, etc.) or the name of the transformation (e.g., *Unfolding sheet AM*, *Adaptation sheet AM*, *XML2XSL sheet AM*, etc.) are used to display models or transformations. The basic inputs for HPG are a CM, an AM, a PM, and UP (input specifications), and a CM instance (input data). The output is a Web presentation for the input data that fulfills all the input specifications.

The transformation process starts with the selection of a CM. In case that such a CM doesn't exist the designer can create one using the Visio solution as described in Section 4.2.1. After selecting a CM, the user can choose an AM from the available AMs that correspond to the chosen CM. Again, if such an AM doesn't exist the designer is offered the possibility to build one using the Visio solution presented in Section 4.2.2. The unfolding step is a preparation step in the sense that it restructures the AM in a format more fit (than the original AM) for the next transformation step.

Based on the UP for AM selection, the original AM is adapted. Slices with conditions invalid are discarded and the hyperlinks (slice relationships) referring to these slices are disabled. For example, if the user is not an *Expert* he will not see the painting technique *description*.

All transformations that we have seen so far are generic. Unless otherwise specified a transformation refers to a generic transformation. Based on the adapted AM one can use a generic transformation to produce a specific transformation (*CMI to AMI sheet*) that will



Museum_CM.rdfs
Museum_AM.rdfs
Museum_PM.rdfs
Museum_profile_AM.rdf
Museum_profile_PM.rdf
Museum_CMI.rdf
[10:56:19.171] PMI to Presentation conversion step started
[10:56:19.872] Presentation generation complete in 0.701 s
D:\Hera\output\output-html\index.html
HTML
generate CSS from PM

Figure 4.6: The HPG-XSLT interface.

convert a CMI to an AMI.

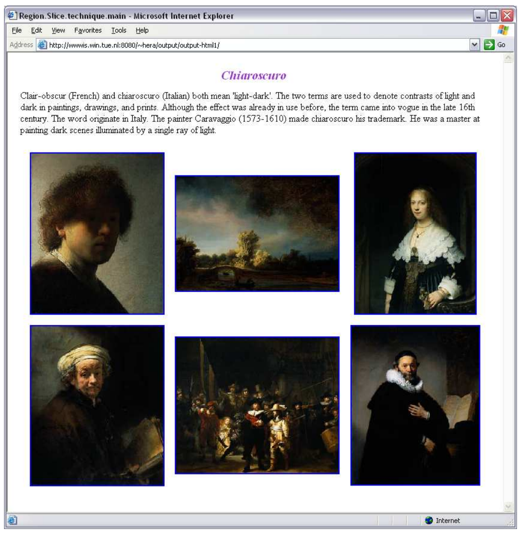
Until now we have presented the transformations at AM level (left-hand side of Figure 4.6). Similar to the above transformations, there are PM-driven transformations (right-hand side of Figure 4.6). Again as a technical convenience, the unfolding mechanism is also used with the PM.

Based on the UP for PM selection, the original PM is adapted. For example, for presenting an index of paintings images one would use a *TableLayout* for a *PC* and a *BoxLayout* for a *PDA* or *WAP phone*. The PM has not only information on the layout but also on the style of the Web presentation. If the user has a *poor* vision, a style with *large* fonts will be used instead of the default style with *medium*-size fonts.

In a similar way as for the AM but this time based on the adapted PM one can use a generic transformation to produce a specific transformation (*AMI to PMI sheet*) that will convert a particular AMI to a PMI.

The last transformation generates code in the format suitable to the user's browser (HTML, HTML+TIME, SMIL, or WML). If the browser supports CSS also a CSS stylesheet is generated according to the style given in the adapted PM. The designer is offered the choice of specifying the directory where the Web presentation will be generated. Note that such a presentation can include thousands of files that might require a lot of disk memory.

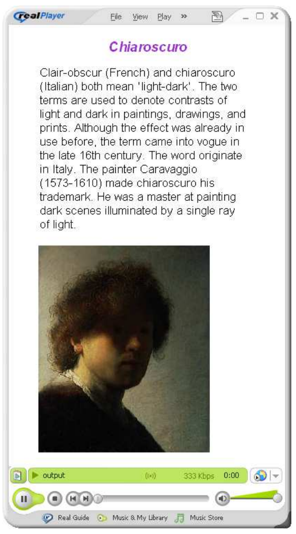
Figure 4.7 presents the four different snapshots for four different browsing platforms: HTML for PC, SMIL for PC, HTML for PDA, and WML for WAP phone. These presentations are in accordance with the adaptation with respect to the user browsing device, i.e., the *client* in the UP, as given in the design specifications. Note that, as described in the PM adaptation, the paintings images have a *TableLayout* for HTML on PC, are arranged using *TimeLayout* for SMIL, and a *BoxLayout* (on the vertical axis) for the HTML for PDA and WML presentations. According to the media adaptation, the PDA and the WAP phone use a shorter text version for the painting technique description compared with the one on the PC. Also, the WAP phone presentation doesn't contain pictures. In a similar way, the presentation can be further adapted by considering other attributes from UP, e.g., the level of expertise of the user, the user visual capabilities, etc.


Region.Slice.technique.main - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://wwwis.win.tue.nl:8080/~hera/output/output-html1/
Go
Internet

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 originate 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.

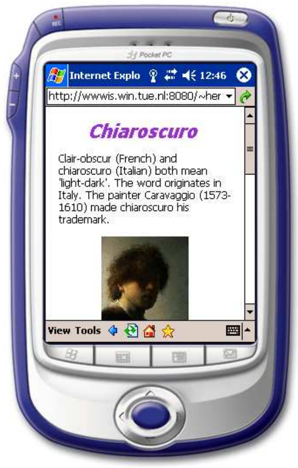
HTML for PC



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 originate 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.

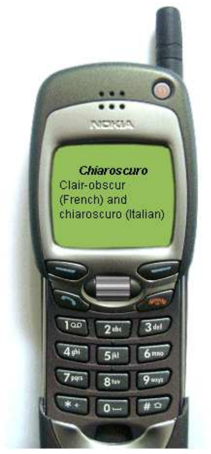
SMIL



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.

HTML for PDA



Chiaroscuro
Clair-obscur (French) and chiaroscuro (Italian)

WML

Figure 4.7: Presentations in different browsers.

The first XSLT processor used to carry out the transformations specified by the different XSLT stylesheets was Xalan 1.2D02 [Apache Software Foundation, 2004]. Since this processor supported only XSLT 1.0 [Clark, 1999] it was replaced with Saxon [Kay, 2005a], a more powerful XSLT processor that supports XSLT 2.0 [Kay, 2005b]. In order to speed-up the execution of these stylesheets (note that the used museum data has about 1000 art objects with their relations) several XSLT keys have been defined.

4.3 HPG-Java

HPG-Java supports the development of the dynamic variant of the Hera presentation generation phase. The design tools integrated in HPG-XSLT for the CM, AM, PM, and UP building can be used also for HPG-Java. HPG-Java doesn't have a graphical user interface similar to the implementation interface of HPG-XSLT.

One of the disadvantages of HPG-XSLT was the fact that it used XSLT stylesheets to transform RDF models. In this way it was difficult to make use of full semantics of a model as given by the model's RDFS-closure. HPG-Java eliminates this shortcoming by defining Java transformations based on Jena [Hewlett-Packard Development Company, LP, 2005]. For querying and updating models it is used the (Se)RQL implementation of Sesame [Aduna, BV, 2005].

4.3.1 Designing HPG-Java

Being a dynamic system able to react to user actions, HPG-Java was developed based on Java servlet technology. It runs as a Java servlet on an Apache Tomcat Web server. Figure 4.8 shows an excerpt of the class association diagram of HPG-Java.

The main class that receives user requests is the HeraServlet, which extends the Java *HttpServlet* class.

In order to build robust and flexible applications we used several design patterns. A servlet specific pattern is the delegation event model. All request handlers implement the *RequestHandler* interface. The request handlers are registered in the HeraServlet. Based on the value stored in a hidden field for the *GET* request, the HeraServlet is able to identify the particular request handler responsible for this event. Examples of events are login, logout, link following, each one having a corresponding request handler. In this way one avoids the building of complex *RequestHandler*s able to solve all request.

The façade pattern was used to hide from the HeraServlet the complexity of the different data transformations. Four classes were defined to perform data transformations: *AMController*, *SessionUpdater*, *PMController*, and *PresentationConverter*. The *AMController* and *PMController* are responsible for adapting and creating instances of the AM, respectively PM. The *AMController* has associated the *SessionUpdater* which manages the *Session*. The *Session* class is an extension of the Java *HttpSession* class. The *Session* class stores information that persist between user requests: the navigational data model,

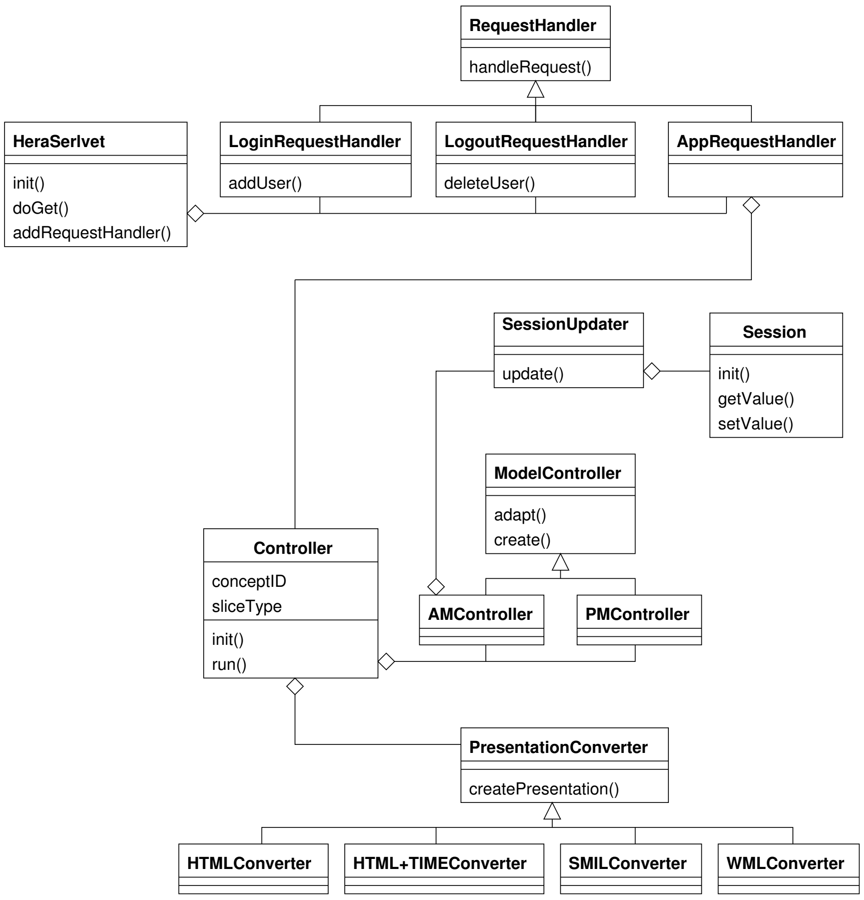


Figure 4.8: HPG-Java class association diagram.

form models, variables (including the user name and password). *SessionUpdater* is used for updating the *Session* data as part of the process of generating the new slice instance.

As both slices and regions have a recursive structure we used the composite pattern to define them. In order to build a slice or region the *create()* function is used. Based on the UP the AM and PM will be adapted using the *adapt()* function. Based on the same UP a specific *PresentationGenerator* is chosen. The *createPresentation()* function is used to produce a presentations interpretable by the user browsing platform.

Figure 4.9 presents the exchange of messages between different class instances in response to a user query.

Suppose the *HeraServlet* receives a *doGet()* function call. In case that the request is originating from a link-following request, the *handleRequest()* function of *AppRequestHandler* is called. Next, the Controller *run()* function is called to manage the whole data transformation process.

Depending if the current session is a new session, the AM and PM are adapted by calling the *adapt()* function of their associated controllers. Note that at the present moment we support only static adaptation of the system, that is why the adaptation is performed at

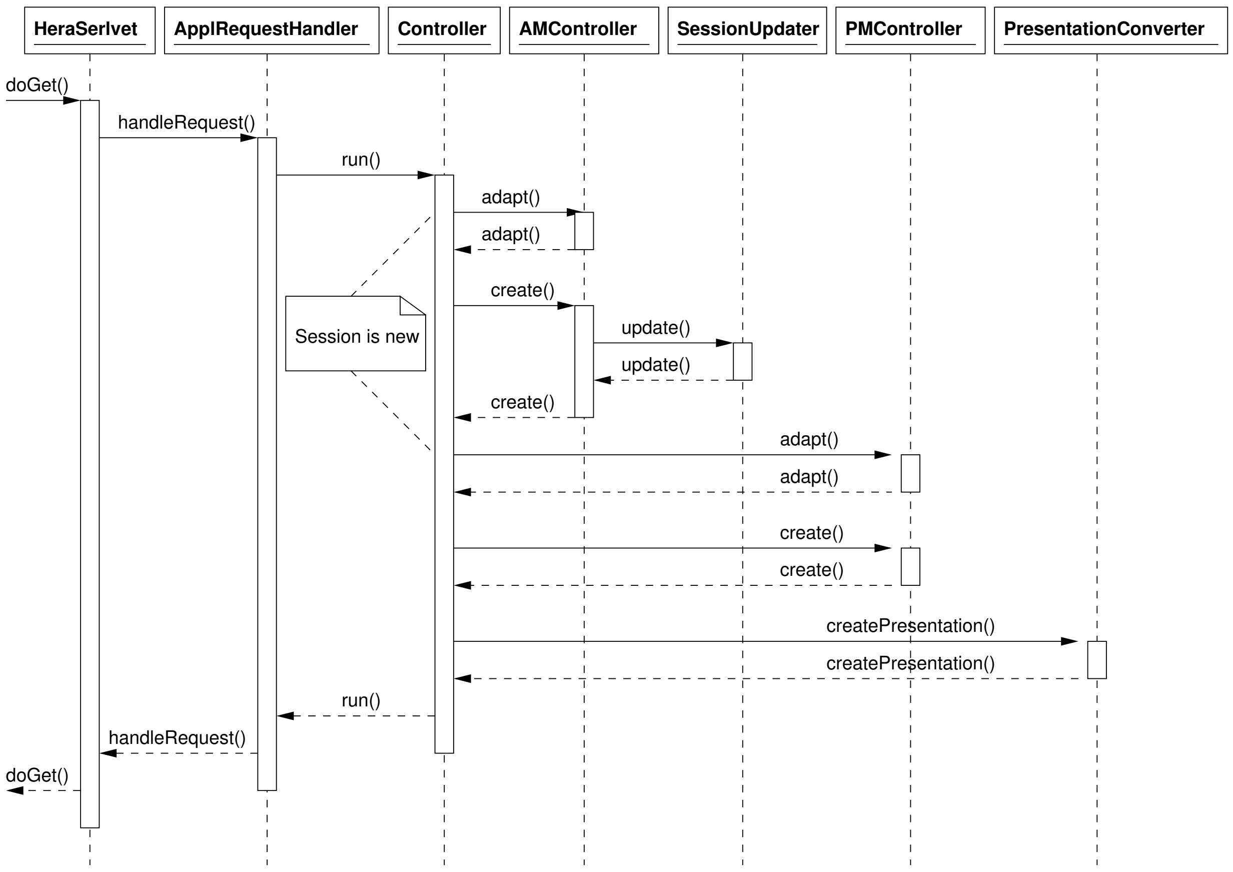


Figure 4.9: HPG-Java message sequence chart.

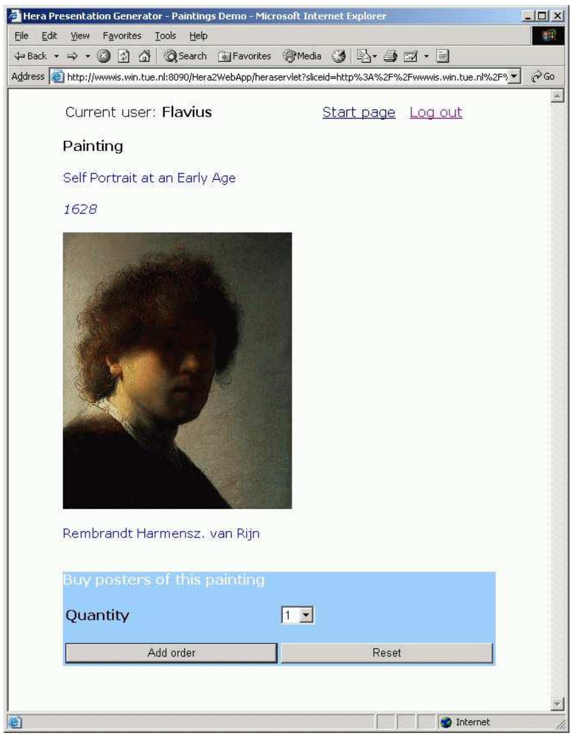
the beginning of the user session. With the AM and PM adapted the *create()* function calls will build a new slice instance and a region instance, respectively for the current page. Note that in the *create* function of the *AMController* the *update()* function of the *SessionUpdater* is called in order to update the *Session* content. The region instance is based on the previously generated slice instance. The last function is the *createPresentation()* which transforms the region instance into a Web page.

4.3.2 Using HPG-Java

Several applications were built using HPG-Java: a review system for the Hera papers, a shopping site for vehicles, a portal for a virtual paintings museum (without user interaction), and a shopping site for posters depicting paintings (with user interaction)[1]. We will use the example of the shopping site for posters in order to better illustrate the HPG-Java page generation process. Figure 4.10 (left) shows one page generated with HPG-Java.

Suppose the user wants to buy a poster of the shown painting. Based on such a user

[1]Examples of applications built using HPG-Java are available from `http://wwwis.win.tue.nl:8090/Hera2WebApp`.


Hera Presentation Generator - Paintings Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://wwwis.win.tue.nl:8090/Hera2WebApp/heraservlet?sliceid=http%3A%2F%2Fwwwis.win.tue.nl%2F%
Go
Internet

Current user: **Flavius** Start page Log out

Painting

Self Portrait at an Early Age

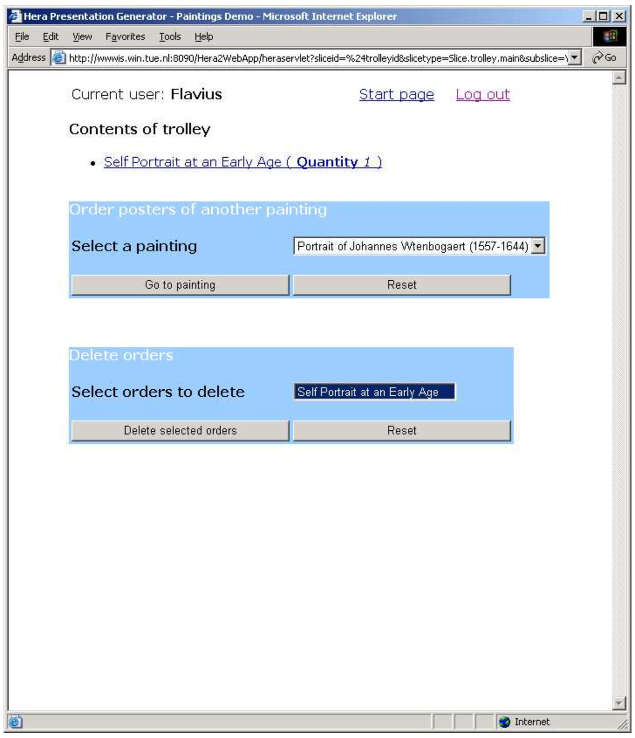
1628

Rembrandt Harmensz. van Rijn

Buy posters of this painting

Quantity 1

Add order Reset


Hera Presentation Generator - Paintings Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://wwwis.win.tue.nl:8090/Hera2WebApp/heraservlet?sliceid=%24trolleyid&slicetype=Slice.trolley.main&subslice=
Go
Internet

Current user: **Flavius** Start page Log out

Contents of trolley

- Self Portrait at an Early Age (**Quantity** *1*)

Order posters of another painting

Select a painting Portrait of Johannes Wtenbogaert (1557-1644)

Go to painting Reset

Delete orders

Select orders to delete Self Portrait at an Early Age

Delete selected orders Reset

Previous slice Current slice

Figure 4.10: Pages generated by HPG-Java.

request, the next page to be displayed is computed. Let us have a closer look at what happens when the user presses the *Add order* button.

In order to be able to construct the next slice instance one needs to know the concept instance identifier that owns this slice instance and the slice type of this slice instance. These two elements, the concept instance identifier and the slice type are encoded in the user request.

In our example, there are two queries (attached to the current slice) that are triggered when the user presses the *Add order* button. The first query creates a new order. The second query fills the order properties and adds the order to the trolley. Based on the user request and AM specifications, a new slice instance is produced. This slice shows the list of ordered paintings and contains two forms: one for selecting the next painting and one for deleting an order from the trolley. The current slice instance is converted to a region instance by adding layout and style information to the slice as specified in the PM. In the last step, the region instance is converted to the next page (in this case a HTML page) to be presented to the user. Figure 4.10 (right) shows the next generated page.

4.4 HPG-XSLT vs. HPG-Java

HPG-XSLT and HPG-Java have both their advantages and disadvantages. Figure 4.1 compares the characteristic features of HPG-XSLT and HPG-Java.

	HPG-XSLT		HPG-Java
	generation of full Web presentation		generation of one page at-at-time
+	user interface	−	no user interface
+	deployable on any Web server	−	can be deployable only on Web servers supporting Java servlets
−	no form-support	+	form-support

Table 4.1: HPG-XSLT vs. HPG-Java.

The generation of the full presentation in HPG-XSLT requires usually a long time for computing the whole presentation. If one decides to deploy the resulted pages on a Web server this high computational time does not influence the system response time to a user. The user can browse the presentation at a reasonable speed if his network connection allows it because there is no computation performed on the server. The generation of one page at-a-time in HPG-Java has as consequence a longer response time than for a presentation generated with HPG-XSLT. Nevertheless if the HPG-XSLT presentation is built at run-time, the time needed for computing the whole presentation is higher than the computing of only one page at the beginning of the browsing process.

HPG-XSLT has a user interface that helps the designer for building models. It also allows the generation and execution of the data transformations based on previously defined models. At the current moment HPG-Java does not have such interfaces. It is planned in the future to build a graphical console for HPG-Java which will look similar to the implementation interface of HPG-XSLT.

The resulted Web pages from HPG-XSLT can be deployed on any Web server. Due to its dynamic nature, HPG-Java can be deployed only on Web servers that support Java servlets. Modern Web server (like Apache) do support Java servlets.

HPG-XSLT has no support for user interaction besides simple link-following. The user of a generated presentation cannot influence the content of the presentation. HPG-Java does allow for more advanced forms of user interaction (e.g., forms) as a way to let the user influence the content of the presentation. This is an extremely useful feature if the built application will be used for example, as a shopping site or as a review system.

Performing the data transformations in XSLT or Java have both advantages and disadvantages. Figure 4.2 compares the characteristic features of XSLT transformations and Java transformations used in HPG-XSLT and HPG-Java, respectively.

XSLT is a declarative programming language and Java is an imperative programming language. Depending on the programmer's preference one way of programming can be

	XSLT stylesheets		Java code
	declarative		imperative
+	loosely-coupled, changing the system can be done by changing one stylesheet	−	strongly-coupled, internal dependencies makes the system harder to change
+	easy to understand	−	more difficult to understand
−	no IDEs	+	many IDEs
−	limited exploitation of models' RDFS semantics	+	full exploitation of models' RDFS semantics
−	XSLT processor offers little support for optimization leading to poorer performance	+	custom software can be optimized better leading to better performance
−	introduces extra steps	+	no need of extra steps

Table 4.2: XSLT stylesheets vs. Java code.

easier than the other one. As for many declarative languages, XSLT is supported by interpreters. Java is supported by many compilers that generate code interpretable directly by the machine. As such the execution time of the code generated by compilers is smaller than the time required by an interpreter to perform the same computations.

Changing the system can be done by changing only one XSLT stylesheet due to the nice separation of concerns provided by stylesheets. A similar change done for Java code might be harder to achieve due to the internal dependencies between software components. The use of design patterns can alleviate this problem in the Java code.

XSLT stylesheets are easy to learn, one can write fairly complex transformations after learning some of the basic XSLT concepts. Java code is harder to produce, the learning curve is usually higher than for XSLT programming.

At the current moment there is a lack of IDEs to assist the XSLT programmer. For Java, a more mature language, there is a huge number of developing IDEs that provide very advanced debugging facilities.

XSLT is a language for transforming XML documents. As there is no data transformation language for RDF it was decided to use XSLT for transforming the RDF/XML serialization of RDF models. Clearly these XSLT transformations have limitations as they are not able to exploit the full RDFS semantics of a model as given by the model RDF(S)-closure. The Java code is based on Jena and Sesame, two Java libraries that can fully exploit the RDFS semantics of models.

While developing HPG-XSLT it was noticed that there is very little support to optimize a data transformation. By optimization it is seeked the reduction of the time needed by a data transformation. The performance of HPG-XSLT is based on the performance of the data structures used by the XSLT processor. The Java code offers more room for optimizing the data transformations as one can define its own data structures and processing facilities.

HPG-XSLT introduces several extra steps, for unfolding models. These steps were developed in order to prepare an RDF/XML document in an XML format suitable for the next transformation step. These steps were not needed for HPG-Java as the Java code directly operates on RDF models.

4.5 A Web Service-Oriented Architecture for HPG

HPG integrates several software components that together form one centralized application. In this section it is presented a distributed architecture for HPG in which components are mapped to Web Services (WS). The loosely coupled Hera WSs realize the plug-and-play software vision in the context of SWISs. For example, it is possible to generate a SWIS by composing a WS which provides up-to-date data, a WS that knows how to present this data, and a WS that is able to perform adaptation of the presentation based on user preferences and device capabilities.

It was decided for a WS solution for realizing the distributed Hera architecture because WSs have clear advantages compared to their predecessors CORBA, J2EE, and DCOM [O'Toole, 2003]. First of all WSs are based on the XML document paradigm, a human readable language that abstracts from the implementation details. WS interfaces are specified in a universally accepted Web Service Description Language (WSDL) [Christensen et al., 2001], an XML-based language. Last but not least, Web services use the popular HTTP protocol as the carrier of exchanged messages.

In the rest of this section we will mainly focus on HPG-XSLT, but one can implement similar services for HPG-Java that produces a single Web page at-a-time instead of a full Web presentation as is the case for HPG-XSLT. For this reason we will not distinguish between the two HPGs and we will refer to them with only one term, i.e., HPG.

Figure 4.11 presents a Web service-oriented architecture (WSOA) for HPG based on two WSs: the Data Service and the Presentation Service. The Data Service is responsible for delivering up-to-date data for which the Presentation Service will make a hypermedia presentation. A Client placed at a Web Server location will orchestrate the communication with the two services. The proposed WSOA has a star topology, with the Client in the middle. The communication between Client and services is done at SOAP [Box et al., 2000] level which resides on top of HTTP while the communication between the Web Browser and the Web Server is done in plain HTTP.

First the Client asks the Data Service to provide the data. Once the data is received it is passed to the Presentation Service. After receiving the data, the Presentation Service constructs a hypermedia presentation which is passed back to the Client. The Web Server that hosts the Client uses this presentation in providing pages to the Web Browser. The underlying assumption here is that the Data Service and the Presentation Service share the same CM.

Note that a service-based solution provides a lot of flexibility for such a system. Different Presentation Services (with different AMs and PMs) can be plugged into the system

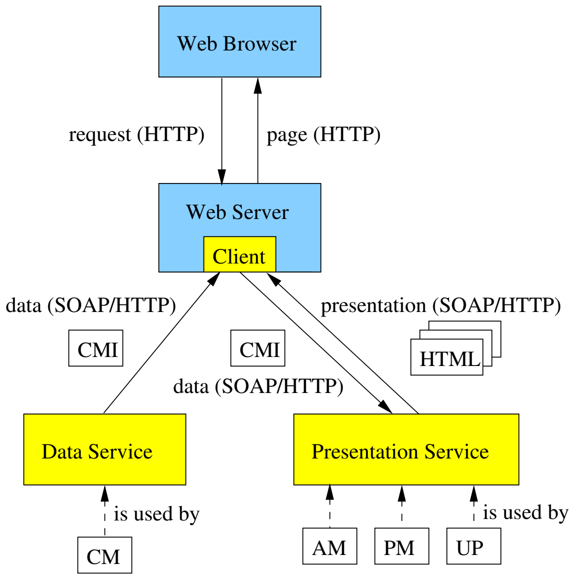


Figure 4.11: Web service-oriented architecture.

to produce different presentations for the same data. Moreover one Presentation Service may produce an HTML presentation, while another one can provide a WML presentation, ensuring thus the ubiquity of the built SWIS. Also different Data Services can be used in the same manner (assuming the fact that they agree with the Presentation Service on the CM). In this way data that comes from two different sources but sharing the same domain can benefit from presentation capabilities from the same Presentation Service.

4.5.1 Web Service Descriptions

The interface of a WS is given in a WSDL specification. In addition to the service interface, such specifications give also the data types used by the service messages and the location of the service. In this subsection it is described only the interface as we only use existing XML Schema Datatypes [Thompson et al., 2001; Biron and Malhotra, 2001] (we did not need to define our own types) and the service location can be defined anywhere on the Web.

Figure 4.12 depicts an excerpt from the Data Service WSDL specification. First it specifies which are the messages, their embeddings, and the type of data that messages will carry. The `getDataRequest` message is an empty message. This message is used just to trigger the response from the service. The `getDataResponse` message has a `<wsdl:part>` containing the requested data string. The `<wsdl:portType>` associates the request and response messages with the `getData` operation of the Data Service. It also specifies the type of the message as input or as output for the operation. The data string returned is the CMI that the Data Service holds.

Figure 4.13 depicts an excerpt from the Presentation Service WSDL specification. The

```
<wsdl:message name="getDataRequest">
</wsdl:message>
<wsdl:message name="getDataResponse">
     <wsdl:part name="getDataReturn"
                type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="DataService">
     <wsdl:operation name="getData">
          <wsdl:input message="impl:getDataRequest"
                      name="getDataRequest"/>
          <wsdl:output message="impl:getDataResponse"
                       name="getDataResponse"/>
     </wsdl:operation>
</wsdl:portType>
```

Figure 4.12: Excerpt from Data Service WSDL.

request message `getPresentationRequest` has a `<wsd:part>` named `in0` (this is the default naming convention used by our SOAP server for the operation arguments) containing one string. The response message `getPresentationResponse` will contain also a string. The `<wsdl:portType>` associates the request and response messages with the `getPresentation` operation of the Presentation Service. As for the Data Service, it also specifies the type of the message as input or as output for the operation. The input data string is the CMI and the data string returned from the operation is the encoded (in one string) presentation.

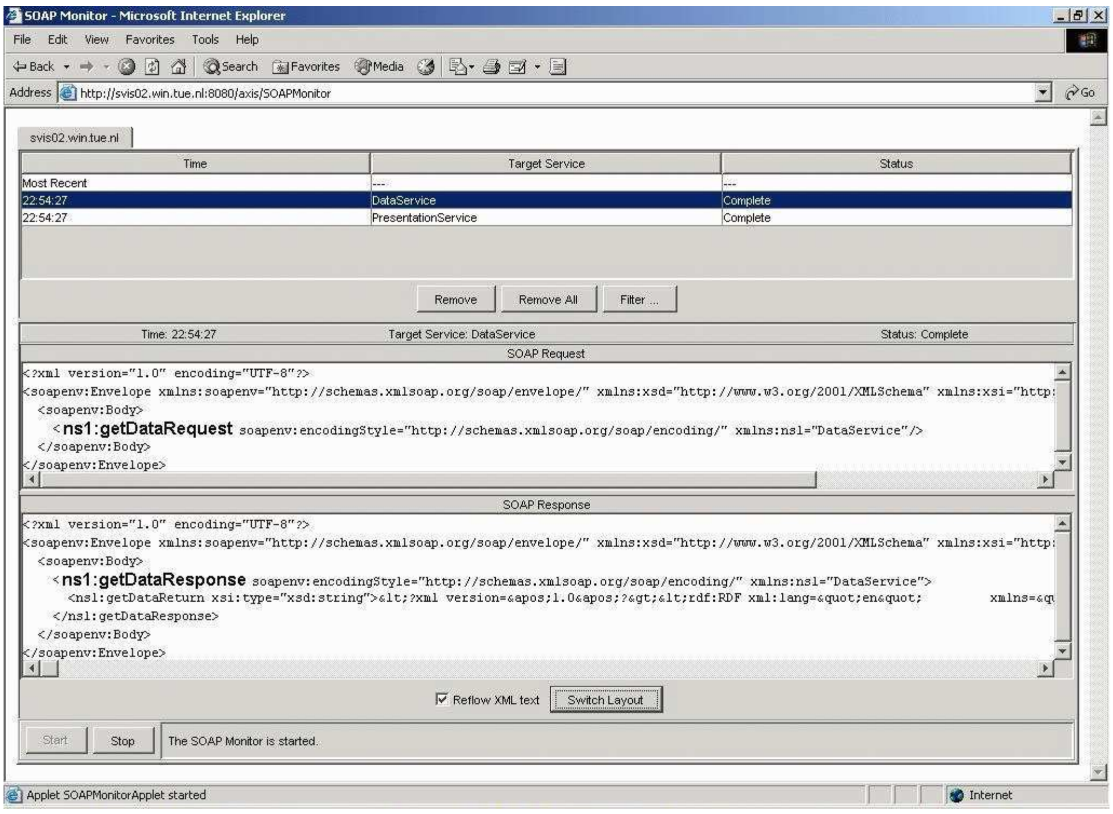
```
<wsdl:message name="getPresentationRequest">
    <wsdl:part name="in0"
               type="xsd:string"/>
</wsdl:message>
<wsdl:message name="getPresentationResponse">
    <wsdl:part name="getPresentationReturn"
               type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="PresentationService">
    <wsdl:operation name="getPresentation"
                    parameterOrder="in0">
        <wsdl:input message="impl:getPresentationRequest"
                    name="getPresentationRequest"/>
        <wsdl:output message="impl:getPresentationResponse"
                     name="getPresentationResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 4.13: Excerpt from Presentation Service WSDL.

4.5.2 SOAP Messages

After we have defined the service interface we can have now a closer look at the actual representation of the service messages. Despite its name the Simple Object Access Protocol (SOAP) is not a classic (communication) protocol. It is rather a one-way message exchange paradigm (or some others prefer to say a lightweight protocol to exchange information in a distributed system). A SOAP message is an XML message containing a SOAP envelope. A SOAP envelope has an optional SOAP header and a required SOAP body. It is the SOAP body that contains the data carried in a message. The current implementation uses SOAP RPC which means that all message communication is done synchronously.

Figure 4.14 presents a snapshot of Client-services communication, namely the SOAP messages exchanged between the Client and the Data Service. The `SOAP Request` window displays the `getDataRequest message`, an empty message as we already saw from the interface description. The `SOAP Response` window shows the `getDataResponse` message containing in its `getDataReturn` part an actual CMI. Note that `<`, `>` are escaped as we encoded the data as a string.


SOAP Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://svis02.win.tue.nl:8080/axis/SOAPMonitor
Go
svis02.win.tue.nl
Remove
Remove All
Filter ...
Reflow XML text
Switch Layout
Start
Stop
Applet SOAPMonitorApplet started
Internet

Time	Target Service	Status
Most Recent	---	---
22:54:27	DataService	Complete
22:54:27	PresentationService	Complete

Time: 22:54:27 Target Service: DataService Status: Complete

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http:
  <soapenv:Body>
    <ns1:getDataRequest soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="DataService"/>
  </soapenv:Body>
</soapenv:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http:
  <soapenv:Body>
    <ns1:getDataResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="DataService">
      <ns1:getDataReturn xsi:type="xsd:string">&lt;?xml version=&apos;1.0&apos;?&gt;&lt;rdf:RDF xml:lang=&quot;en&quot;        xmlns=&q
    </ns1:getDataResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

The SOAP Monitor is started.

Figure 4.14: SOAP messages.

4.5.3 Tools

In order to experiment with the proposed architecture a Java-based Hera tool was developed. Tomcat 4.1 [Apache Software Foundation, 2005a] was used as the Web server that supports servlets. On this Web server we installed Axis 1.1 (Apache eXtensible Interaction System) [Apache Software Foundation, 2005b], a SOAP 1.1 engine. By SOAP engine we mean a tool that supports both a SOAP server and SOAP clients. We did deploy on the SOAP server two services Data Service and Presentation Service. For their deployment we used appropriate Axis Web Service Deployment Descriptors. The SOAP Client that communicates with these services was installed on the Web server, outside the SOAP server. The WSDL specifications were generated by the Java2WSDL emitter. Both Java2WSDL and the SOAP Monitor are part of the Axis distribution kit. Tomcat and Axis are Java-based and freely available from the Apache Software Foundation. The services and the client were written in Java. All software is running on the Java 1.4 platform.

It is important to notice that when developing WSs with Axis, the programmer doesn't need to bother about making WSDL interfaces or the actual encoding of the SOAP messages. All these will be automatically done by the system. Making all WS details transparent to the programmer enables him to focus only on the application logic implementation in Java and makes thus the system less error-prone.

4.5.4 Adaptation in HPG Web Service-Oriented Architecture

Figure 4.15 presents a WSOA based on four services: Data Service, Presentation Service, Profile Service, and Adaptation Service. Note that this architecture doesn't have a star topology as the Client only communicates with three of the four services. In order to denote the order in which the messages will be passed we added a label to the continuous arrows. This label should be read in the increasing number order or alphabetical order. The two sequences (1, 2, 3) and (a, b) can be done in parallel. Step 4 is performed after completion of steps 3 and b.

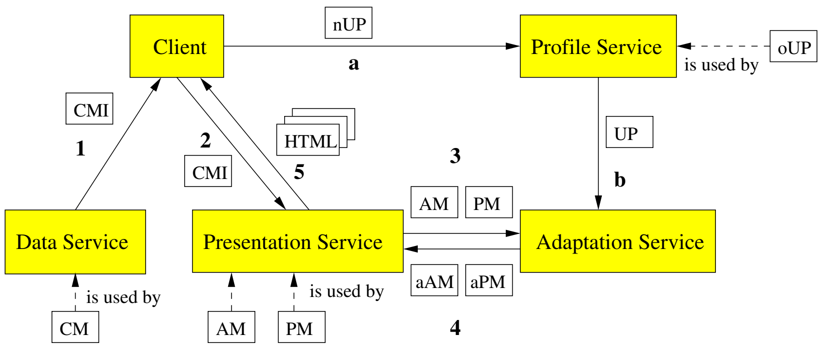


Figure 4.15: Extended Web service-oriented architecture.

The steps 1, 2, and 5 were already discussed in the beginning of this section. In step a the nUP (‘n’ stands for new) provided by the Client is sent to the Profile Service. The Profile Service can be viewed as a shared memory service for user profiles. By a shared memory service we simulate shared memory between services using one service. The profile attributes that are not defined in the nUP will be taken from the oUP (‘o’ stands for old), the old profile of the user, and result in a merged UP. In the request message sent to the Profile Service the user may also specify if an update of the oUP with the UP should be done. The UP is sent to the Adaptation Service. Also, the Adaptation Service receives the AM and PM from the Presentation Service. After receiving these two messages (3 and b) the Adaptation Service computes the aAM and aPM (‘a’ stands for adapted) and sends them to the Presentation Service. In this WSOA the Presentation Service will use the aAM, instead of the AM, and the PM, instead of the aPM, to compute the presentation.

4.6 Conclusions

The Hera tool suite aims at supporting the design of SWISs using the Hera methodology. There are two versions of the software tools for the presentation generation in Hera: HPG-XSLT and HPG-Java. The XSLT stylesheets from HPG-XSLT are replaced in HPG-Java with Java code able to better cope with the RDF(S) semantics of the Hera models. Compared with HPG-XSLT, HPG-Java extends the functionality of a generated WIS with user interaction support. Nevertheless, HPG-Java lost the declarativity, simplicity, and reuse capabilities of the XSLT transformation templates. A declarative transformation language dedicated to RDF(S), which to our knowledge doesn’t exist at the present moment, would probably be the best option for Hera transformations.

HPG has also a distributed architecture based on WS. We chose for a WS-oriented solution due to the popularity and easy-to-implement features of WSs. In this way WISs can be seamlessly built by composing appropriate WSs. The Axis distribution kit proved to be a very flexible set of tools to support WS development, deployment, and monitoring. We have also shown examples of WSDL specifications used to describe WSs and of SOAP messages exchanged with WSs.

As future work, a user interface (servlet console) will be developed for HPG-Java, very similar to the user interface in HPG-XSLT, in order to better trace and configure the Hera servlet activities. In both HPGs, the user interfaces for designing the CM, AM, and PM will be extended with adaptation specification support (for the appearance conditions). Depending on the future existence of a declarative RDF transformation language the Java code will be replaced with RDF transformation templates which will combine the best features of the two HPG versions: declarativity, simplicity, and reuse of templates as in HPG-XSLT, and the full RDF(S) semantics exploitation for the Hera models as in HPG-Java.

We would also like to extend the Web service-oriented architecture of the HPG with new services like a data query service, a data integration service, or a service able to generate adaptive hypermedia presentations.

Bibliography

Aduna, BV (2005). openrdf.org ... home of sesame. `http://www.openrdf.org/`.

Apache Software Foundation (2004). Xalan-java. `http://xml.apache.org/xalan-j/`.

Apache Software Foundation (2005a). Apache tomcat. `http://jakarta.apache.org/tomcat/`.

Apache Software Foundation (2005b). Webservices - axis. `http://ws.apache.org/axis/java/user-guide.html`.

Biron, P. V. and Malhotra, A. (2001). Xml schema part 2: Datatypes. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-2/`.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple object access protocol (soap) 1.1. W3C Note 08 May 2000.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web services description language (wsdl) 1.1. W3C Note 15 March 2001.

Clark, J. (1999). Xsl transformations (xslt) version 1.0. W3C Recommendation 16 November 1999. `http://www.w3.org/TR/xslt`.

Hewlett-Packard Development Company, LP (2005). Jena - a semantic web framework for java. `http://jena.sourceforge.net/`.

Kay, M. (2005a). Saxon (the xslt and xquery processor). `http://saxon.sourceforge.net`.

Kay, M. (2005b). Xsl transformations (xslt) version 2.0. W3C Working Draft 11 February 2005. `http://www.w3.org/TR/xslt20/`.

Klyne, G., Reynolds, F., Woodrow, C., Hidetaka, O., Hjelm, J., Butler, M. H., and Tran, L. (2004). Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. W3C Recommendation 15 January 2004.

O'Toole, A. (2003). Web service-oriented architecture: The best solution to business integration. Cape Clear Software. `http://www.capeclear.com/clear_thinking/Web_Service_Oriented_Architecture2.pdf`.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2001). Xml schema part 1: Structures. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-1/`.