Hype – A Java Tool for the Rapid Development of Hyperdocument Management Applications for the WWW

Enno Scholz Rainer Lischetzki

DaimlerChrysler Research and Technology
Alt-Moabit 96a, 10559 Berlin, Germany
{enno.scholz,rainer.lischetzki}@daimlerchrysler.com

Abstract.

The paper presents Hype, an integrated development environment for building typed hyperdocument management applications for the World Wide Web. Like Java Server Pages, Hype enables Java code snippets producing dynamic content to be embedded into static HTML pages. Unlike Java Server Pages, Hype covers the complete process of building three-tier applications. This includes the object-oriented modeling of the application schema in a graphical editor, the mapping of the object-oriented application schema to a relational database schema, and the automatic generation of persistent Java classes for accessing the database. Knowing the application schema, Hype can go well beyond Java Server Pages with regard to the typesafeness and modularity it provides for producing dynamic content.

Unlike many commercial tool suites, Hype is not targeted mainly at software professionals. Rather, the design goal of Hype was to enable *power users*, i.e., application domain experts with limited programming expertise, to implement most of a hyperdocument management application without programming. An important means for this (and a hallmark of Hype) is a small set of typed placeholders, which replace the use of embedded Java for many common customization tasks.

1 Introduction

The use of Java applets within HTML pages on the client has made Java the programming language of the World Wide Web. With the advent of *servlets*, Java has recently become increasingly popular also for generating HTML on the server. A servlet is a Java object residing in a web server that is associated with a certain URL. When a browser issues a HTTP requests for that URL, the servlet dynamically generates HTML code and sends it to the browser by writing it to an output stream.

However, in most WWW applications a large part of the HTML text is static and only some parts need to be dynamically generated. Encoding these static parts in Java

quickly becomes tedious, because even small changes require the servlet code to be recompiled. A remedy for this is offered by *Java Server Pages* [Sun 99], which allows static HTML pages to contain snippets of Java code producing HTML dynamically. These hybrid HTML pages are identified by the extension `.jsp`. When the web server gets a request for a `.jsp` page, it executes the code snippets and replaces them by the HTML text they generate.

Still, a number of shortcomings can be observed about Java Server Pages:

- *Lack of modularity:* All code snippets in `.jsp` files reside in a server-global context. They may call methods in servlets or server-side beans, but there is not one-to-one correspondence between `.jsp` files and server objects.
- *No support for WYSIWYG page design:* One advantage of static HTML pages is that a broad variety of tools exist for editing them in a what-you-see-is-what-you-get manner, which boosts productivity and widens the potential range of people able to create and maintain web pages. Unfortunately, `.jsp` files do not coexist benevolently with off-the-shelf WYSIWYG editors, because all snippets must be enclosed in proprietary HTML tags (such as `<jsp:.../>`) ignored by the editor.
- *Support for only a fraction of the total application building process:* The combination of servlets and Java Server Pages addresses only a small part of the complexity of building three-tier web applications, namely, the interface between the client tier and the business tier. The typical range of technologies a prospective web application designer must master is much broader. For the database tier, a basic understanding of relational database design and familiarity with SQL are required. For the business tier, familiarity with Java's database interface, JDBC is required. Finally, for the client tier, at least some experience with JavaScript and Java applets is necessary.
- *Addresses software engineers only:* The above list of shows that it takes experienced software engineers to build three-tier web applications, and Java Server Pages does not change that. Neither do the comprehensive tool suites supporting the complete process of building three-tier applications offered by various database vendors, for instance Oracle and Versant. These tool suites are targeted at software engineering professionals. They offer increased productivity for programmers who already master the full range of web technologies. However, for application domain experts with limited programming background these tool suites are too complex to use.

This paper presents an integrated development environment called *Hype* that – for a certain class of three-tier web applications – enables an application domain expert to implement a large part of the application without any programming. For those parts where programming is inevitable, Hype offers an easy-to-use and safe Java API. Hype can be viewed as a generator for state-of-the-art three-tier application servers.

The primary design goal of Hype is to reduce to a minimum the share of work done by software engineers in the development process, and to maximize the share of application domain experts with limited programming expertise, called *power users*. This is a desirable goal not only because software engineers are presently the critical resource in web development, but also because the extra indirection between end users

and developers is a well-known bottleneck. Thus, empowering non-programming application domain experts to do most of the developing autonomously is helpful for reducing time-to-market, speeding up the feedback loop, and increasing application acceptance.

In technical terms, Hype is an integrated development environment for building typed hyperdocument management applications for the World Wide Web. Conceptually, a Hype application is built in two steps. In step one, an object-oriented *application schema* is designed. It consists of a set of *document types*, each specifying a document's *fields*, and a set of *link types* between document types. In step two, for each document type, a set of HTML *templates* is designed. Using the templates for a given type, documents of that type may be viewed in a web browser. Like `.jsp` files, templates may contain Java code snippets in addition to arbitrary HTML tags. There is an important advantage over Java Server Pages, though: the code snippets are not executed in a server-global context but rather in the context of the current document, which is represented as a persistent, typed Java object. Using this object's methods, the code snippet has typed access to the document's fields and its links to other documents. When the Hype application server receives a request to view a given document using a given template, it replaces all code snippets by the HTML text they generate. The resulting HTML, called a *view* of the document, is served to the browser.

However, the use of Java snippets to customize Hype templates with dynamic content is the exception rather than the rule. For common kinds of customization, for instance for displaying a document's fields or for displaying hyperlinks to documents it is connected with, Hype provides the concept of *placeholders*. Like Java code snippets, they are replaced by dynamic content when a given document is rendered with a given template. A placeholder is replaced by a combination of HTML, JavaScript, and Java applets that serves not only to display a field or a hyperlink. Additionally, it enables the object's properties to be edited directly in the browser.

To purists, defining a placeholder language might seem like a bad move. Why have two languages when one of them, Java, is technically a superset of the other? However, the placeholder language is crucial in order to enable power users to do most of the application customization without programming. The following features are useful for comparing Hype placeholders to those found in other systems, for instance Hyperwave [Maurer 96] and DxML [Knowing 99].

- *Placeholders are powerful*: The placeholder language of Hype is quite powerful. The exact set of placeholders available in a document type's templates depends on the application schema. Templates for a given document type may extend supertype templates using an inheritance mechanism. Moreover, a template may specify that documents linked to the current document are to be shown not simply as hyperlinks but as nested views within the view of the current document.
- *Placeholders are easy to use*: Despite its power, the placeholder language is easy to use for non-programmers. The first reason for this is that the placeholders are designed to harmonize with the use of a WYSIWYG editor. Moreover, the correct use of the placeholders and the absence of type errors is checked by the system. In particular, error messages meaningful to the application developer ("Sorry, there is

no field 'Salry' in Employee") can be given instead of relying on Java compiler errors containing references to dynamically generated code. Finally, placeholders save the application developer the tremendous effort of devising protocol for editing documents.

- *Placeholders are efficient*: Because placeholders are not a complete programming language, they may be statically analysed and object views may be pregenerated and stored on disk. Whenever an object changes, a background thread calculates the set of all affected views of the object itself and of other objects and regenerates them. Therefore, read accesses to a Hype application's documents can be made almost as fast as requests for static HTML pages, because in both cases, they are available on the harddisk and need not be generated on-the-fly.

Hype offers full support for rapid prototyping. Both the application schema and the HTML templates of an application may be modified while the server is running.

Hype has been designed at DaimlerChrysler Research and Technology in the context of an interdisciplinary research group consisting of both social scientists and computer scientists. One Hype application for managing quality-related hyperdocuments is currently in production use at Mercedes-Benz, another one is scheduled for production use in October, and several more are in the process of being implemented.

The following Section 2 gives an overview of the application schema of Hype applications. Section 3 describes the default user interface of Hype applications. Section 4 explores how the user interface of applications may be customized by modifying the default templates generated from the application schema. In particular, the placeholder language defined by Hype is reviewed in some detail. Section 5 presents the ultimate stage of customization, writing code snippets that access the generated Java API classes. Section 6 compares Hype to related work. Finally, Section 7 draws conclusions.

2 The Application Schema

Fig. 1 is a screen shot of Hype's schema editor showing the application schema of a demo application named **`JITDemo`** that will be used throughout the paper. The schema falls into two parts, a standard part comprised of a number of built-in document types and link types which are part of any application, and an application-specific part. Just as writing a Java program can be viewed as extending Java's standard set of classes (defined in **`java.lang`**) with application-specific classes, implementing a Hype application can be viewed as extending the standard application schema with application-specific document types and link types. Just as any Java class must be a subclass of **`java.lang.Object`**, any Hype document type must be a subtype of **`Document`** and thus inherit its fields and links. Hence, every document has a name and a creation time. Link types (and links) in Hype are always bidirectional, similarly to UML associations.

First, consider the standard part of the demo application's application schema. The type **`File`** serves as a wrapper for external unstructured documents like text files or images. It defines two fields, the file's URL on the server and the file's MIME type.

The type **User** serves to represent users that have access to the application. The type **Starting page** has only one instance, the application's starting page. The type **Type folder** has one instance for every document type in the application schema. For instance, the type folder **All Files** contains links to all documents type **File**. When, for example, a new file is created by a user, a link between **All Files** and the new file is created by the system. The link type **Document->Type Folder** is an instance of a *system-maintained* link type which means that instances may only be created and deleted by the system, not by users. Additionally, for each document the system maintains links to the user who is the document's creator and to the users presently viewing the document in a browser.

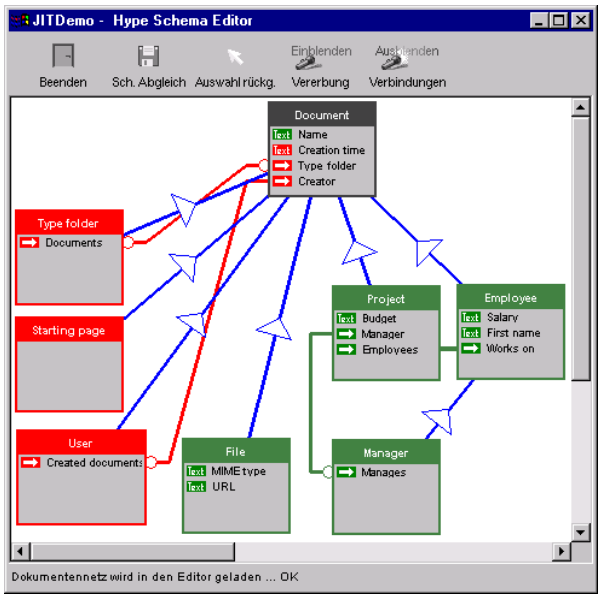

Fig. 1: The application schema of the demo application

The application-specific part of the application schema introduces three document types, **Project**, **Employee**, and **Manager**, and two link types, **Project->Employees** and **Project->Manager**. The document type **Project** has a field **Budget**, the document type **Employee** has a field **Salary** and a field **First name**. These fields are inherited by **Manager**.

3 The Default User Interface of Hype Applications

From the application schema, Hype generates a set of default templates, which define the default user interface of a Hype application. The default user interface of projects in the demo application is shown in Fig. 2 and Fig. 3.

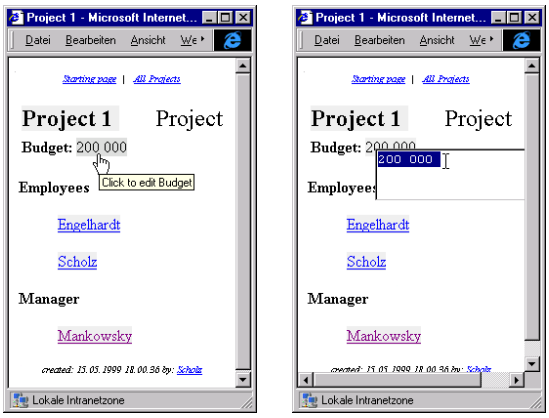

Fig. 2: Editing a field in a document

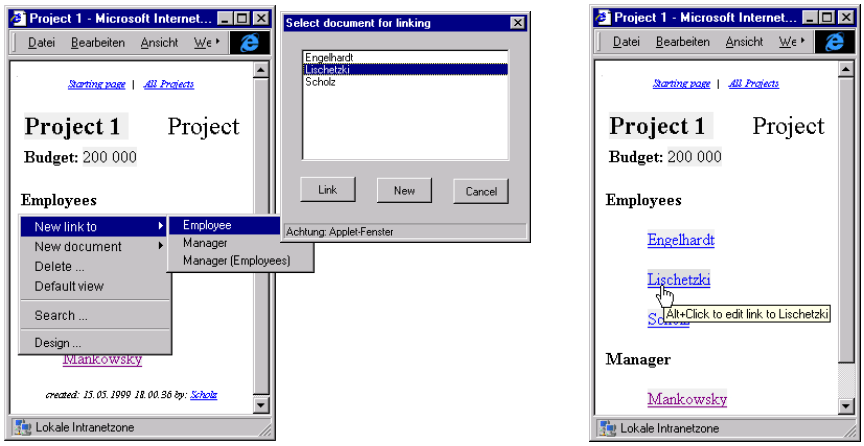

Fig. 3: Adding a link to another document

At the top of the page, each document has hyperlinks to the application starting page and to its type folder. Then the name and the type of the document are given in a large bold font. The background color of the project name is light grey, which indicates that Hype has generated JavaScript and Java elements allowing it to be edited in the browser simply by clicking. Then the remaining slots are shown, in the case of projects this is only the budget. After the slots, the links of each kind are shown, in the case of a project, these are the employees and the manager. Again, the light grey background color of the hyperlinks indicates that they may be edited in the browser. The page bottom shows the creation time and a link to the **`User`** document representing the creator. Both have no grey background because they are not editable but system-maintained.

Fig. 2 shows how the project budget is edited. Fig. 3 shows how the employee *Lischetzki* is added to the project *Project 1*. On the left in Fig. 3, we see a document's standard menu, which pops up when the user clicks anywhere within the document. The menu contains menu items for creating documents (*New document*), creating links

(*New link to*), deleting the current document (*Delete*), viewing the document using a predefined default template (*Default view*), searching the database (*Search*), and bringing up the schema editor in case the user is an administrator (*Design*).When the item *Employee* is chosen from the *New link to* menu, a dialog shown in the middle of Fig. 3 prompts the user to choose from the list of existing employees. The screen dump on the right in Fig. 3 shows that the newly-created link to employee *Lischetzki* appears at the correct place in the alphabetically sorted list of links for role **`Employees`**.

It is important to note that the menu hierarchy shown in the screen shot on the left in Fig. 3 is generated automatically from the application schema. For each role defined or inherited by **`Project`** that is not system-maintained (i.e., for **`Project->Employees`** and **`Project->Manager`** but not for **`Document->Type folder`** and **`Document->Creator`**) a list of menu items is generated for the *New link to* menu. For each role, one menu item is generated for each of the subtypes of the role's target type (including itself). In the example, one menu item is generated for role **`Project->Manager`** and type **`Manager`**, another one for role **`Project->Employees`** and type **`Employee`**, and finally one menu item is generated for role **`Project->Employees`** and type **`Manager`**. The submenu entry is labeled with the target document type's name. Unless the role name is equal to the name of the target document type, the target document type's name in the submenu entry label is followed by the role name enclosed in parentheses.

4 The Hype Placeholders

As mentioned in the introduction, a document's appearance in a browser depends on the template used to view it. All templates for an application are assumed to reside in a special directory, the *template directory*. By convention, the names for template files for a given document type consist of the document type name, followed by '**`@`**', followed by an identifier, followed by the file name extension '**`.html`**'. Here, Hype deposits the default templates, which give rise to the user interface discussed in the previous section. These templates may be customized by the application developer.

Fig. 4 illustrates the relationship between documents, templates, and views. It shows a document of type **`Project`**, a template **`Project@Short.html`** for that type, and the resulting HTML as viewed in a browser.

The HTML text of the template contains two placeholders specific to Hype. Like all placeholders, they are enclosed in square brackets. The placeholder **`[Name]`** is replaced by the name of the document, *Project 1*. The placeholder

```
<a href="Employee@Short.html">[Employees]</a>
```

is replaced by a list of hyperlinks to the employees working in that project. The anchor tag's **`href`** attribute, **`Employee@Short.html`**, denotes the template using which the employees are displayed in the browser when the user follows one of the hyperlinks.

Note that the enclosing anchor tag is an integral part of this placeholder, which is called a *hyperlink placeholder*. At first glance, it may seem strange to choose a

placeholder syntax where the target template is encoded in the attribute of an enclosing HTML tag. Why are not both the placeholder's parameters, the role name and the target template name, encoded within the square brackets? One might imagine a syntax like, say, **`[Employees:Employee@Short.html]`**. The answer is that the chosen syntax is more useful in a WYSIWYG editor for HTML, as demonstrated by Fig. 5.

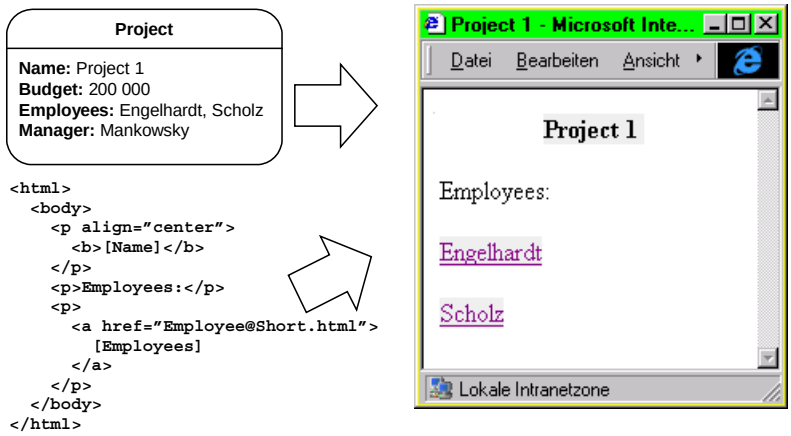

Fig. 4: A document, a template, and the resulting view

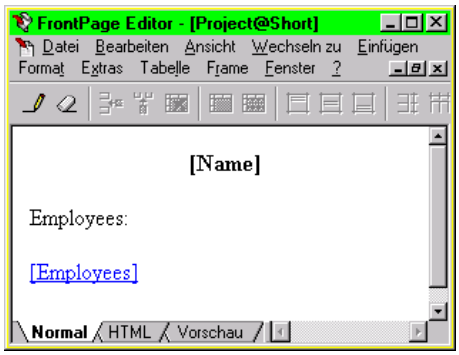

Fig. 5: The template Project@Short.html in a WYSIWYG editor for HTML

Fig. 6: Nesting the views of connected documents within the view of the current document

All WYSIWYG editors known to us provide an easy way for following hyperlinks. For instance, in MS Frontpage pressing CTRL and clicking on the link labeled **`[Employees]`** in the template opens its target template **`Employee@Short.html`**.

The following subsections survey the Hype placeholders in some detail. Section 4.1 presents placeholders for fields and roles. Section 4.2 shows how linked documents may be displayed not as hyperlinks but as nested views within the view of the current document. Section 4.3 shows how a template may refer to itself. Finally, Section 4.4 demonstrates inheritance between templates.

4.1 Placeholders for fields and roles

In Fig. 4, the placeholders for document fields and for hyperlinks to connected documents were demonstrated. These placeholders depend on the application schema and take the form of either a field name or a role name enclosed in square brackets. The placeholder for the list of hyperlinks to connected documents can take an alternative form in the case the application schema allows multiple documents to be connected in a given role. This form allows the separator between hyperlinks to be specified. For instance, the following phrase would cause the hyperlinks to be separated by commas instead:

```
<a href="Employee@Short.html">[Employees]</a>, [_Etc]
```

In this extended form of the link placeholder, the delimiter **`[_Etc]`** is an integral part. The simple form of this placeholder used above, without **`[_Etc]`**, can be seen as a shorthand for using the extended form with the separator **`</p><p>`**.

4.2 Nested views

With a small change in the template **`Project@Short.html`** shown in Fig. 4, the employees connected to a project can be displayed not simply as hyperlinks but in the nested style illustrated in Fig. 6. This is achieved by adding a '**`+`**' after the role name **`Employees`** in the placeholder denoting the list of employees working on a project, i.e., by changing

```
<a href="Employee@Short.html">[Employees]</a>
```

into

```
<a href="Employee@Short.html">[Employees+]</a>
```

Now, the template **`Employee@Short.html`** is interpreted as the template used for rendering the single employee documents in the context of the project.

4.3 Viewing the current document using another template

The template **`Project@Short.html`** does not show a project's budget or its manager. Assuming there is another template **`Project@Long.html`**, the placeholder **`[_This]`** can be used to include a hyperlink to a view of the current document generated with another template:

```
<a href="Project@Long.html">[_This]</a>
  Detailed view
[_End]
```

Here, the hyperlink to the current document is labeled **Detailed view**. In general, arbitrary placeholders may stand between **[_This]** and **[_End]**. For instance,

```
<a href="Project@Long.html">[_This]</a>
  Detailed view of [Name]([Budget])
[_End]
```

causes the label of the hyperlink to be the name of the project, followed by its budget in parentheses.

4.4 Inheritance between templates

Often, a template for a given type should look just like a supertype, except for some small difference. For instance, it would be desirable to define a template **Manager@Detailed.html** such that it looks just like **Employee@Detailed.html**, except that at the bottom there is the list of managed projects. This can be achieved by defining **Manager@Detailed.html** using the placeholder **[_Super+]**:

```
<html><body>
  <a href="Employee@Detailed.html">[_Super+]</a>
  <p><a href="Project@Short.html">[Manages]></a></p>
</body></html>
```

Now consider the following problem. The example above showed that a template could extend a supertype template by means of **[_Super+]**. However, this extension was done by adding something at the front and at the end only. What if we want to define a corporate identity template, which should be reused for each template, but customized in the header and in the body part? The solution would be to define a template for type **Document**, say, **Document@CI.html**. This would include two nested views of the current document, for instance:

```
<html>
... <a href="Document@Header.html">[_This+]</a> ...
... <a href="Document@Body.html"  >[_This+]</a> ...
</html>
```

Here, **Document@Header.html** and **Document@Body.html** are supposed to be empty. Now, by defining **Project@CI.html** as follows

```
<html><a href="Document@CI">[_Super+]</a></html>
```

and by defining **Project@Header.html** and **Project@Body.html** suitably, the goal would be achieved. Note that this technique is similar to calling deferred methods in abstract classes in object-oriented programming.

5 The Java API of Hype Applications

With the means introduced so far, a Hype application can be customized without programming. This section shows how Java program code may be included into templates using a special placeholder **[_Java]**. This Java code, which is executed whenever a document is viewed using a given template, generates HTML that is included in the view. For instance, the following placeholder includes the current time in a document view:

```
[_Java] out.println(new java.util.Date() + "") [_End]
```

This is nothing new over Java Server Pages. The novel aspects of Hype will be presented in the following sections. Section 5.1 shows how views are customized by Java code snippets that have access to typed Java objects representing document of the types defined in the application schema. Section 5.2 demonstrates how updates are customized via a typed interface to HTML forms. Section 5.3 shows how the generated accessor methods for each document type may be refined by the programmer using method overriding. Section 5.4 shows how new Java methods for the document types may be defined by the programmer.

5.1 Customizing views

From the application schema, Hype generates and dynamically loads a set of Java classes which enable server-side code to access the documents in a type-safe manner. For instance, suppose that we want to include into a project view its remaining budget, which is defined as its budget minus 12 times the monthly salary of each employee working on it. This is how it is done:

```
[_Java]
float remainingBudget = this.Budget();
EmployeeVector employees = this.Employees();
for(int i = 0; i < employees.size(); i++)
  remainingBudget -= 12 *
employees.elementAt(i).Salary();out.println(remainingBudge
t + "");
[_End]
```

Note that within the Java code in the placeholder, **this** refers to an instance of the generated API class **Project**. Note that **EmployeeVector** is a customized, typesafe version of **Vector** whose **elementAt()** method returns an **Employee**. Thus, its result needs not to be casted before the invocation of **Salary()**.

5.2 Customizing updates

The standard user interface for Hype applications enables the user to create and delete all types of documents and links that are not system-maintained, and to edit all fields

that are not system-maintained. However, sometimes it is desirable to enable the user to perform customized update operations. There can be two reasons for this: Either the kinds of updates the user can perform should be constrained to certain values, or several separate update operations should be composed into one macro operation. For instance, suppose we want to enhance a project view with a form for allocating multiple academic degrees to each employee working on the project. Suppose furthermore that we want to constrain the academic degrees to BSc, MSc, and PhD. Furthermore, checking a checkbox causes the salaries of all employees to be raised by 1000 Euro. This is achieved as shown in Fig. 7.

Note that here, the **`[_Java]`** appears in a special place, in the action attribute of a form tag. Compared to the use of the **`[_Java]`** placeholder in other places, three things are different here. First, the Java code is not executed when the form is displayed but when the user presses the submit button. Second, there is no stream **`out`** here, because the point of the Java code is not to produce HTML but to perform an update. After the code is executed the current document is reloaded. Third, in the Java code two variables are defined which represent the HTML elements in the form and are typed accordingly. The variable **`titles`** represents the list of options selected by the user. Its type is **`String []`**, because the multiple attribute of the **`select`** element is true. The type of the variable **`raiseSalary`**, which represents the checkbox, is **`boolean`**. In general, for each kind of form element (files, radiobuttons, text area), a Java variable with a corresponding name and appropriate type is in scope in the Java code. Fig. 7 demonstrates the use of the methods like **`void Salary(float)`** for setting object properties as opposed to reading them.

```
<html>
 <body>
  <p><b><big>[Name]</big></b></p>
  <form action='[_Java]
                for(int i = 0; i < Employees(); i++) {
                  Employee e = Employees().elementAt(i);
                  if(raiseSalary)
                    e.Salary(e.Salary() + 1000);
                  for(int j=0; j < degrees.length; j++)
                    e.Name(e.Name() + " " + titles[j]);
                };
                [_End]'>
   <p> Grant acacemic degrees:<br>
       <select name="degrees" multiple="true" size="3">
         <option>BSc</option>
         <option>MSc</option>
         <option>PhD</option></select></p>
   <p> <input type="checkbox" name="raiseSalary">
       Additionally raise salary </p>
   <p> <input type="submit" value="Ok"></p>
  </form>
 </body>
</html>
```

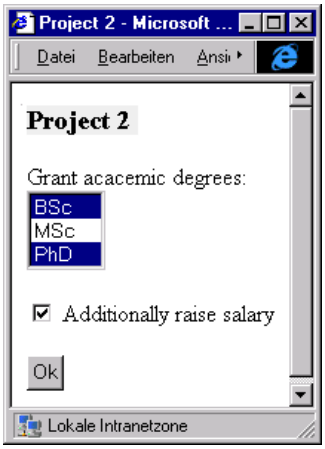

Fig. 7: A template containing form-handling Java, and the corresponding view

5.3 Customizing event handling

Analogously to the way client-side events like **`onClick`**, **`onLoad`**, **`onMouseOver`** can be handled in JavaScript, Hype provides a way for an application to customize server-side events like creating or deleting documents or links or changing attributes.

Hype uses Java inheritance to enable this kind of event handling. The application designer cannot only *use* the typed methods in the classes of the Java API generated from the application schema, she can also *redefine* these methods. For instance, suppose the user should only be able to raise the salaries of employees, not lower them. This can be achieved by redefining the method **`void Salary(float)`** whose use was illustrated in the previous section. In order to do this, the application designer must create a file **`Employee.java`** in the template directory with the following content:

```
public class Employee extends
hype.custom.JITDemo.schema.Employee {
  void Salary(float newSalary) {
    if(newSalary > this.Salary()) {
      super(newSalary);
    };
  };
}
```

Note that this way of handling events is more flexible than simply enabling an event handler to be provided (as for instance an **`onClick`** handler in JavaScript), because arbitrary actions can be performed both before and after the standard event handler is called, and it can even be canceled completely.

5.4 Reusing Server-Side Java Code

Writing Java code directly into templates is fine for short snippets of code but gets tedious when the code gets longer. Moreover, sometimes several templates might need to share code. For instance, the routine defined above for calculating a project's remaining budget might be useful in other places, too, possibly even in templates not belonging to the document type **`Project`**. In Hype, this routine can be defined as a method of class **`Project`** by adding it as to **`Project.java`**. In other words, the **`.java`** files in the template directory cannot only be used for redefining existing methods in the generated Java API but also for adding new methods.

6 Related work

In the field of dynamic generation of HTML, approaches range from generic server-side scripting languages and web-enabled database management systems to specific, more or less WWW-based hyperdocument management systems.

A number of systems exist for server-side execution of code fragments in HTML pages like Sun's *Java Server Pages* [Sun 99], Microsoft's *Active Server Pages* [Weissinger,

Petrusha 99], UNIX-based *PHP3* [PHP3 99], and *DxML* [Knowing 99]. They offer basic programming language features such as loops, conditions, or variables, and provide access to server-side databases, the WWW server and information on the client. They focus on the development of dynamic web pages by experienced programmers.

Another option for programmers is to use the solutions provided by different vendors of database management systems for generating dynamic content. *Oracle 7* [Oracle 98] provides sophisticated support for generating web pages but is limited to relational data modeling. We are in the process of examining its successor *Oracle 8i*. *Jasmine* [Computer Associates 99] is an object-oriented database management system (OODBMS) that is accessed not via a WWW-Browser but a proprietary client. *Versant* [Versant 99] is another OODBMS. It excels at generating classes of persistent Java objects from UML modeling tools. However, its main focus seems to be on supporting smart CORBA and RMI clients, not on generating HTML.

Alternatively, ready-made hyperdocument management systems come complete with a predefined user interface, access control, and features for cooperation support. Similar to Hype, *Fabasoft Components* [Fabasoft 99] enables power users to build document management systems with virtually no programming. However, it is highly integrated with the Windows operating system and has only rudimentary support for WWW access. *Lotus Notes* [Dierker, Sander 97], is another document management system with good support for power users. We are currently investigating its recent release, R5, in which great efforts were made to support a WWW client in addition to the original, proprietary client. In contrast, *BSCW* [Bentley et al. 97] and *Hyperwave* [Maurer 96] are hyperdocument management systems originally conceived for the web. Both have only limited support for application-specific document types. In general, BSCW provides little support for application development and customization. It focuses on collaboration support. Hyperwave, like Hype, distinguishes between a placeholder language (called PLACE) and a general server-side programming language (JavaScript). However, placeholders are not compatible with WYSIWYG editors, the correct use of placeholders is not statically checked, and the server-side programming language is not statically typed.

7 Conclusion

Hype seems to fill a niche: a system enabling power users to develop most of a WWW hyperdocument management system without programming. Existing object-oriented database systems focus on sophisticated applications manipulating persistent objects, not on end users directly viewing, creating, linking, and editing objects, whereas WWW hyperdocument management systems do not seem to attribute much relevance to object-oriented typing of documents. However, Hype demonstrates that having an object-oriented type schema around is a tremendous help for the process of customizing a hyperdocument management application, indeed for minimizing the amount of customization required.

Acknowledgements

Hype is the WWW reincarnation of a Smalltalk system called *HyperCom* developed by Alexander Mankowsky. Alexander's and Sabine Engelhardt's comments were essential in the design of Hype. Winfried Marten at DaimlerChrysler VSE/PQ was an inspiring Customer One. Several students helped implement Hype. In particular, Reimo Tiedemann implemented the schema editor and the graphical search tool. Anne-Grit Gäbler-Wicovsky gave valuable feedback. Steffen Klein was an enthusiastic early adopter; his ongoing comments helped to improve Hype (and this paper) in many ways. Finally, Boris Bokowski made valuable suggestions for some last-minute restructuring of this paper.

References

[Bentley et al. 97] Bentley, R.; Appelt, W.; Busbach U.; Hinrichs, E.; Kerr, D.; Sikkel, J.; Trevor, J.; Woetzel, G.: *Basic Support for Cooperative Work on the World Wide Web*, in: *International Journal of Human Computer Studies* 46, 6 (1997), 827-846; Special Issue on Novel Applications of the WWW, Cambridge: Academic Press, 1997

[Computer Associates 99] *Jasmine – Konzept*. Computer Associates GmbH Darmstadt, 1999 (http://www.cai.com/offices/germany/jas_konz.htm)

[Dierker, Sander 97] Dierker, Markus; Sander, Martin: *Lotus Notes 4.6 und Domino. Integration von Groupware und Internet*, Bonn: Addison-Wesley, 1997

[Fabasoft 99] *Fabasoft Components/Base*, Fabasoft Deutschland Software GmbH, Hallbergmoos, 1999 (http://www.fabasoft.com/COO.1.2010.1.2107)

[Knowing 99] *Der DxML-Wizard – Dynamisierung Ihrer Webseiten*, Knowing GmbH Ulm, 1999

[Lischetzki 99] Lischetzki, Rainer: *Ansichten, Aktionen und Kooperation in WWW-basierten Hyperdokumenten-Management-Systemen*. Diploma Thesis, Institut für Informatik, Freie Universität Berlin, 1999

[Maurer 96] Maurer, Hermann: *Hyper–G (now Hyperwave). The next generation web solution.* Harlow: Addison-Wesley, 1996

[Oracle 98] *Produktangebot Datenbank-Server*. Oracle Deutschland Gmbh, München, 1999 (http://www.oracle.de:80/orcl/dbssi/Main.htm)

[PHP3 99] *PHP3 Manual* (http://www.php.net/manual/)

[Sun 99] *Java Server Pages Specification 1.0*, Sun Microsystems, Inc., Palo Alto: 1999 (http://developer.java.sun.com/developer/earlyAccess/jsp/index.html)

[Versant 99] *Versant ODBMS Release 5*. Versant GmbH Europe, München, 1999 (http://www.versant.de/ ProductEnglish_rel5.htm)

[Weissinger, Petrusha 99] Weissinger, A. K.; Petrusha, R. (Eds.): *ASP in a Nutshell*, Sebastopol, California: O'Reilly, 1999