Pers Ubiquit Comput (2005) 9: 100–107
DOI 10.1007/s00779-004-0314-7

ORIGINAL ARTICLE

Patrick Sauter · Gabriel Vögler · Günther Specht
Thomas Flor

A Model–View–Controller extension for pervasive multi-client user interfaces

Received: 8 June 2004 / Accepted: 8 June 2004 / Published online: 1 October 2004
© Springer-Verlag London Limited 2004

Abstract This paper addresses the implementation of pervasive Java Web applications using a development approach that is based on the Model–View–Controller (MVC) design pattern. We combine the MVC methodology with a hierarchical task-based state transition model in order to achieve the distinction between the task state and the view state of an application. More precisely, we propose to add a device-independent TaskStateBean and a device-specific ViewStateBean for each task state as an extension to the J2EE Service to Worker design pattern. Furthermore, we suggest representing the task state and view state transition models as finite state automata in two sets of XML files. This paper shows that the distinction between an application's task state and view state is both intuitive and facilitates several, otherwise complex, tasks, such as changing devices "on the fly."

Keywords Pervasive computing · Ubiquitous computing · Fluid computing · Model–View–Controller (MVC) · Design patterns · Device independence

1 Introduction

The use of the Model–View–Controller design pattern (MVC, see [1]) is very common in user interface development. Yet, an important practical goal of pervasive application development—maximizing the number of supported devices while minimizing code redundancy—is not necessarily achieved by employing an arbitrary MVC-based development methodology.

In particular, a typical problem of pervasive application development is to provide client-adapted user interfaces [2]. For example, a dialog that fits onto a single Web page on a PC (e.g., a complex form) must be fragmented into several sub-dialogs on a device with a small screen size (e.g., a PDA or a cellphone).

MVC-based approaches allow the support of such adaptations without rewriting the entire user interface code. A result of the decomposition of the architecture into model, view, and controller is that device-independent and device-specific code is decoupled into separated components. In order to support an additional device, only the view (and, for more complex applications, sometimes the controller) must be rewritten or at least adapted. However, this might lead to many "similar" view and controller components with partial code redundancy.

A widely used MVC-based pattern for J2EE applications is the Service to Worker design pattern [3]. It covers many aspects of practical interest when implementing a single-client (mainly desktop browser) Java-based Web application. Although it does suggest using an explicit state transition model that is stored in an XML file, this model does not fully satisfy the needs of pervasive applications with differing screen flow on different devices. Only global transitions such as global JSP state transitions (e.g., "switch from searchmask.jsp to resultlist.jsp") are specified, so a device with a different screen flow would require its very own screen flow model and, thus, probably a different set of JSP files.

It is, therefore, necessary to identify similarities between the screen flow models of the devices and introduce a more abstract *task*-based (as described in [4]) model that all devices have in common. To take on this issue, we suggest using a two-stage hierarchical model of JavaBeans to store information separately about the state of the (device-independent) *task* flow and the state of the (device-specific) *view* flow of the application.

P. Sauter (✉) · G. Specht
Fakultät für Informatik,
Abt. Datenbanken und Informationssysteme,
Universität Ulm, 89069 Ulm, Germany
E-mail: patrick.sauter@informatik.uni-ulm.de
E-mail: specht@informatik.uni-ulm.de

G. Vögler · T. Flor
Software Architectures, DaimlerChrysler,
Research & Technology, Postfach 2360,
89013 Ulm, Germany
E-mail: gabriel.voegler@daimlerchrysler.com
E-mail: thomas.flor@daimlerchrysler.com

This paper extends the Service to Worker design pattern in order to support multi-device Web applications by introducing a task-based two-stage hierarchical state transition model.

The rest of the paper is organized as follows: in Sect. 2, existing work in the range of UI development for pervasive computing is discussed. Section 3 contemplates the task-based development methodology, describes a task-based implementation algorithm for pervasive Web applications, and gives a suitable definition for the term "task." In Sect. 4, we take a look at the Service to Worker design pattern from the viewpoint of pervasive computing. We will then introduce an extension and describe an implementation approach for pervasive applications in greater detail, illustrated by several UML diagrams and code examples. In Sect. 5, we will discuss the additional design-time and run-time benefits of the separation between an application's task state and its view state; more specially, we consider the possibilities for facilitating the development process, the ability to change devices "on the fly," and discuss the applicability of our approach to other related areas of pervasive computing.

2 Related work

The specific requirements of pervasive applications have been widely discussed in [2, 4–7]. In particular, Banavar et al. [4] describe a programming model that strictly treats task logic and user interaction separately. They make the suggestion to start with creating a superior task-based model for program structure that covers the user's abstract interaction and the application logic, and then continue with creating a subordinate navigation model that covers the flow of the view elements. For the time being, the model-based methodology is not yet supported by major programming tools or design patterns. Our suggested development strategy is fundamentally based on this task-driven development methodology.

There are several pervasive computing projects that aim at devising a high-level UI design language for abstract user interaction (see [8, 9] as well as IBM's PIMA project, http://www.research.ibm.com/PIMA/). Any of these approaches proceed similarly: the modeling phase of the abstract user interaction in the respective language is followed by a semi-automatic generation of the device-specific code. Although our paper describes a less automated development process, the two-stage hierarchical state transition model presented in the subsequent section could well be used by any of these languages to support the advanced run-time characteristics described later in this paper. Furthermore, the two JavaBeans specified in Sect. 4 could also be created by the code generator of the respective high-level language.

The IBM Zurich Research Laboratory has devised an approach named "fluid computing" [10] for using a single application with multiple devices simultaneously. Our approach which addresses *changing* devices on the fly can also be extended to support fluid computing behavior (see Sect. 5).

In general, the contribution of this paper is the combination of the well proven MVC methodology with the separation of an application's task state and its view state. The result is a practical task-based implementation guideline for pervasive Java-based Web applications.

3 The task-based development approach

As a general approach for developing pervasive applications with multi-device capabilities, Banavar et al. [4] suggest modeling the task logic prior to the user interaction (i.e., the screen flow). The *task flow* of an application, as we define it, must, in contrast to its screen flow and, without loss of generality, be common to all devices. We therefore need to redefine or at least narrow the definition of the term *task* so that it satisfies our requirements.

A task, as defined by Bergman et al. [11], is a unit of work to be performed by the user. This definition, however, does not specify the granularity of a task in comparison with a *sub*task. A task, as we use it, must comply with the following requirements:

- It must not be too fine-grained such that a step in a subsequent task could be part of the current task without affecting the functionality of the application. For example, all text fields of a search mask that a user can fill in without requiring any additional system interaction must necessarily belong to the same task.
- It must not be too coarse-grained such that there exist two steps within a single task with one step requiring input data that is to be produced as the output of another step within the same task. For example, the search mask and the result list of a search engine query ought to belong to separate tasks.

In other words, a task must consist of steps which logically belong together from both the user's and the system's point of view. The purpose of this definition is that a task could well be displayed on a single view, i.e., on a single Web page.

3.1 Example

The database application discussed in this section is a typical example for Web-based intranet applications: a company wants to make its corporate employee directory (Fig. 1) accessible from various client platforms, e.g., a typical desktop browser and a PDA (with an HTML browser). On a desktop PC, due to its plentiful screen size, the application might consist of only three Web sites: the search mask, the result list, and an additional, more detailed, employee profile page. Although it might be only a rule of thumb, in this example, each of the three *desktop browser* Web sites should represent exactly one

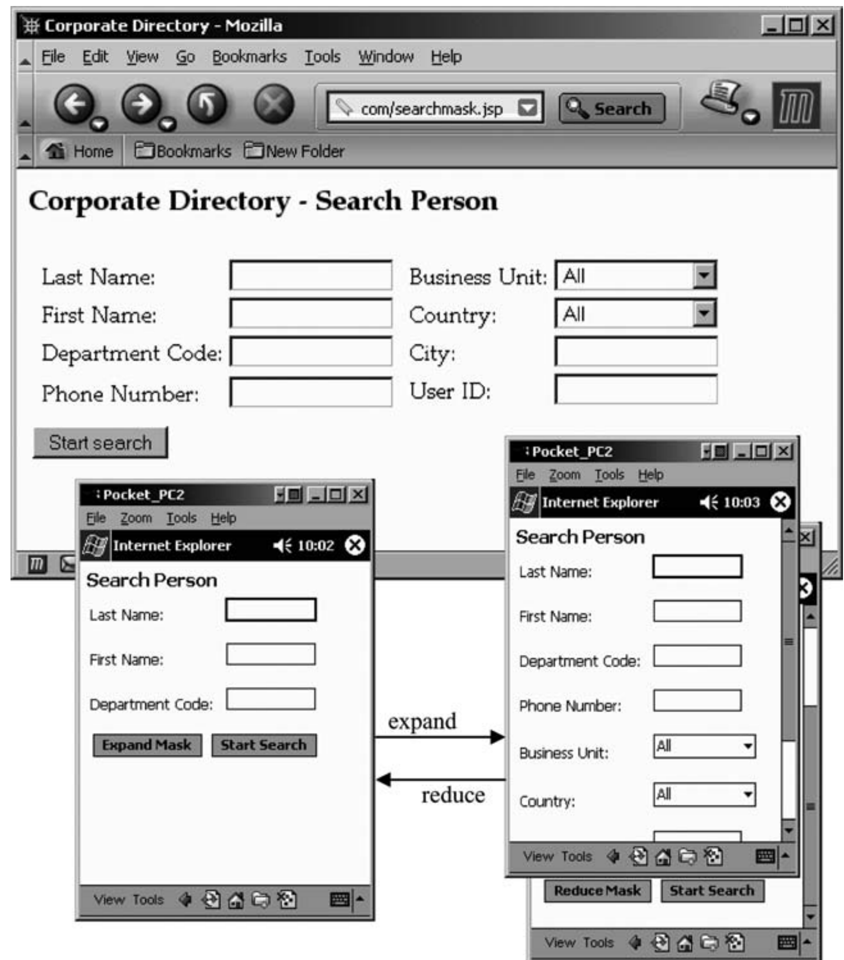

Fig. 1 How the first page of this example Web application might be rendered on a desktop and a PDA browser, where only the most frequently used input fields are displayed at first

task according to the preceding definition: the entire search mask as the entry point, the result list displaying the results of the directory query, and the detailed employee profile page displaying the results of a second query of the selected employee displaying a greater number of attributes from the directory. That is, task flow and screen flow are identical on the desktop browser.

On the PDA, however, task flow and screen flow might differ because of its limited screen size. For example, it might be sensible to display only a *reduced* search mask as an entry point with a subset of only the most frequently used input fields (e.g., only first and last name as well as the department code) and an additional " Expand Mask" button that would show all search fields. This click on the "Expand Mask" button would only alter the view state of the application, not its task state. In the expanded state, the user would be expected to scroll down in order to find all search fields. We shall refer to this example in subsequent sections.

3.2 A task-based design cycle for pervasive computing applications

The idea of a task-driven development methodology must now be put into a more concrete and practical development guideline. As an implication to our definition of the term "task" in combination with the task-based development approach, we suggest that the development of a pervasive Web application should proceed as follows:

1. Model the task flow of the application in order to express user requirements, e.g., using a UML activity diagram.
2. Since each of these tasks should comply with the above definition, each of these tasks is to be represented by a single view component. That is, create a view component (e.g., an empty JSP file) for each device and for each task.
3. For each target device and for each task, check if the capabilities of the device allow a one-to-one relationship between a task and a view. In other words, determine if it is feasible and sensible to display the task on the particular device as a single view component.
4. If this is the case for a particular view component, the view flow and the task flow are identical for this page and no changes have to be made. Go to step 7.
5. If this is not the case, the task has to be assigned several view states. That is, part of the task has to be grouped on separate screens, or maybe even left out completely, e.g., on a cellphone with an extremely restricted screen size and very little memory.

6. Use the additionally available view state information to enable the user to switch from one view of the same task to another. In a J2EE Web application environment, this could be accomplished by adding embedded Java code to a given JSP file (which represents the task state). This file retrieves the view state information and, thereby, decides which segment of the JSP file (e.g., either the "reduced" or the "expanded" part) to display. Furthermore, view state change buttons must be provided, e.g., by inserting an "Expand Mask" button.
7. Optimize the device-specific presentation, for example, by adjusting font sizes or defining CSS files.

The advantage of this approach is the clear-cut separation between global task state and view state, both at design-time (a Web design team could create complex view flow structures without affecting the team implementing task flow and business logic) and at run-time (as we will show in Sect. 5).

See Fig. 2 for how this algorithm might transform into application states and state transitions for the previously mentioned employee search example.

4 Design patterns for the implementation

After introducing our task-based design cycle for pervasive computing applications, we now want to discuss a design pattern for its realization within a J2EE environment. In order to add multi-device capabilities, we now take up the idea of separating task logic and user interaction and, therefore, *extend* a well known design pattern of the J2EE presentation tier [3].

4.1 The Service to Worker pattern

The Service to Worker pattern is a design pattern that primarily addresses the development of single-client Web applications. In particular, it decouples business logic from the view components and allows explicit state transition handling. We therefore consider it suitable for the type of Web applications discussed in this paper, i.e., typical thin-client intranet applications. Although many of the succeeding concepts can also be used in combination with other *Core J2EE Patterns* (http://java.sun.com/blueprints/corej2eepatterns/) or the more general MVC design pattern, the Service to Worker pattern provides the best basis for our extensions to facilitate pervasive computing application development.

The Service to Worker pattern is a combination of the Front Controller and the View Helper patterns in the J2EE presentation tier. For a more rigorous discussion on the Service to Worker J2EE design pattern, see [3] and http://java.sun.com/j2ee/. Its main feature added to the MVC architecture is the ability to comprise the state transition logic into a dedicated dispatcher class rather than the controller. The dispatcher accesses an XML file that defines all existing states and associates user actions with state transitions.

In contrast to existing MVC-based frameworks for pervasive computing Web applications (e.g., MVC-Portlets of IBM WebSphere Everyplace Access Portal Server 4.2.1), the concept described in this paper does not use multiple device-specific controller classes. In our approach, the controller has only two responsibilities: pass on every request to the dispatcher and, when required, call the appropriate business service. Yet, these business services are only called when the application's task state changes. And since all devices have the underlying task state transition model in common, a single device-independent controller class is sufficient.

4.2 Extension of the Service to Worker pattern

The Service to Worker pattern does not yet fully satisfy the needs of pervasive computing development. As mentioned in the previous subsection, Sun Microsystems' specification of the Service to Worker pattern introduces a dispatcher class to handle state transitions. However, it neither prescribes how to model the state transition logic (inside the XML file), nor does it include a description of the run-time component which realizes this model. Furthermore, since the Service to Worker pattern is designed for single device access, the issue of

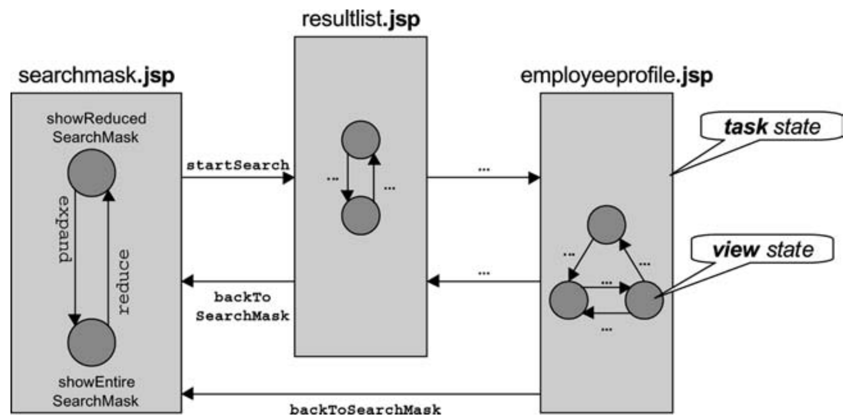

Fig. 2 A possible result of the algorithm for the previously mentioned corporate directory application example on a PDA device. The labels of the *arrows* correspond to action names. Each task state is represented by a single JSP file

device-specific views has to be addressed. Even the algorithm provided in Sect. 3 does not give any implementation details. We will take on these issues now by describing an implementation approach in greater technical detail.

For the purpose of modeling the device-independent state transition logic, we use an XML file named mappings.xml (as suggested by http://java.sun.com/blueprints/patterns/FrontController.html). For the device-specific view logic, we utilize an additional file named viewmappings.xml for each supported device. Both describe a finite state automaton, defining states, actions, and transitions.

We suggest implementing these automata as two dedicated JavaBeans classes: TaskStateBean and ViewStateBean (Fig. 3). Both are initialized with data of the XML files described above. At run-time, they provide a getState() and a doAction(String) method. The former returns the current state and the latter performs the assigned action and returns either the new state or throws an exception if the action is illegal. Additionally, the TaskStateBean can reference a device-specific ViewStateBean for each task state if the view state of a previously completed task has to be retained. For example, if the user clicks on "Back to Search Mask" which he left in the expanded state, it could again be rendered in the expanded state when the user returns.

Device-specific renderings are achieved by view components (JSP files) that are particular to exactly one device. There exists a single JSP file for every task/device combination that contains view-specific markup and, therefore, must also include code for accessing the ViewStateBean information. All JSP files may be placed together with the appropriate viewmappings.xml in a device-specific subfolder (e.g., /pda) in order to deliver straightforward naming conventions.

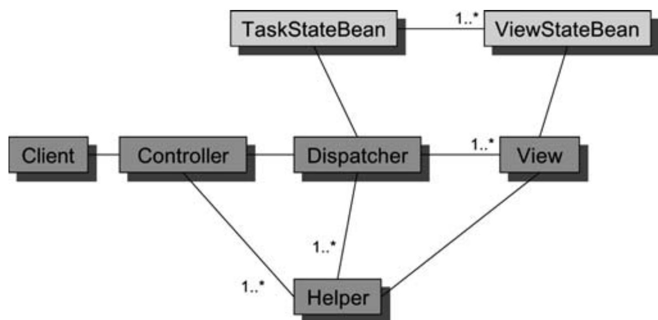

Fig. 3 The class diagram of the extended Service to Worker design pattern. The added elements are the two JavaBeans classes

After setting up the general structure, we now want to discuss the component interaction shown in Figs. 4 and 5, which refer to the example described above.

Consider the situation in which a PDA user fills in the search mask of the corporate employee directory example. In this scenario, the TaskStateBean is in the state searchmask.jsp and references the PDA version of the ViewStateBean, which initially is in the state showReducedSearchMask.

Next, the user clicks the "Start Search" button. This leads to the creation of an HTTP GET request with the action parameter startSearch, which is then sent to the controller (e.g., http://host/controller?action=startSearch). At this point, the request object will be enriched with information about the delivery context by the pervasive computing environment. For example, the IBM WebSphere Everyplace Access Server looks up the User-Agent string of the HTTP header in its device database and maps it to one of its pre-defined device classes (e.g., "Pocket Internet Explorer" is recognized as a "PDA" device). An attribute indicating this piece of information about the device class will then be added to the request object.

The controller reads the action value ("startSearch"), calls some helper class to handle the query (i.e., invoke an LDAP service), and then calls the dispatcher to display the results. Next, the dispatcher calls doAction("startSearch") of the TaskStateBean in order to determine what JSP file to display next. In this case, the action causes a task state transition, so the return value is resultlist.jsp. Additionally, the TaskSateBean initializes a ViewStateBean for the new task state and sets it to the default view state. According to the device class detected earlier, the dispatcher now displays resultlist.jsp of the /pda directory. Before generating the final markup, the embedded code of the JSP file queries the ViewStateBean to decide which view elements to display. Figure 4 gives an impression of how the internal

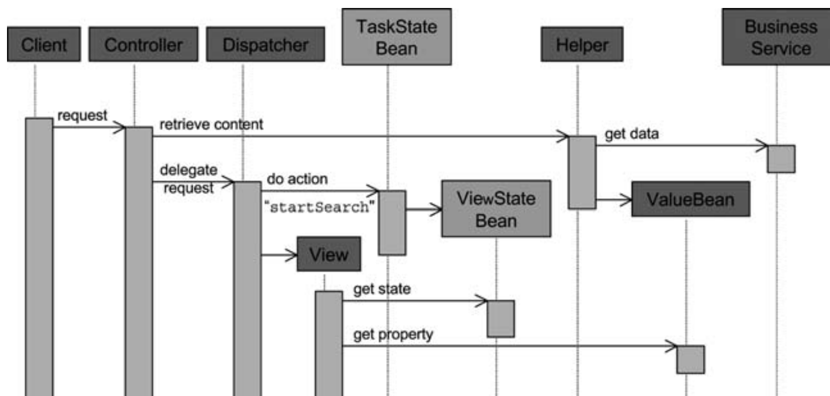

Fig. 4 The internal sequence for handling a click on the "Start Search" button, i.e., a typical task state changing action

handling sequence of the application for the "Start Search" scenario might look like.

The internal object interaction of a click on the "Expand Search Mask" button (instead of "Start Search") might, however, look fundamentally different. Therefore, Fig. 5 (view state change) differs from Fig. 4 (task state change) in so far as no business service has to be accessed.

In the situation of Fig. 5, the submitted action ("reduce") is not a task state changing action, so the TaskStateBean delegates the action handling to the ViewStateBean. After the ViewSateBean has changed its state, the TaskStateBean returns the old task state ("searchmask.jsp"). Consequently, the dispatcher calls the same JSP file (searchmask.jsp). But this time, the getState() method inside the embedded JSP code returns a different *view* state, which causes it to display different markup.

Figures 6 and 7 give an impression of how the previously mentioned search mask of the employee directory example might translate into XML and JSP code.

Notice that addressing actions is a mere implementation detail and depends on the set of available technologies of the Web application server. For a JSP-based portal environment, for example, the Java Specification Request (JSR) 168 (see http://www.jcp.org/en/jsr/detail?id = 168 and Fig. 8) defines a dedicated set of JSP tags for generating dynamic Uniform Resource Identifiers (URIs) which, when requested, enforce the invocation of an action handling method.

In general, this section points out that the separation of task state and view state effects the implementation of Web application in two ways: both the dynamic information about the *current* state (which, in this case, is represented by JavaBeans objects) and the static state *transition* information (which we suggested storing in XML files) have to be subdivided into their task state and view state components.

```
<viewmappings>
  <taskstate>  <name>searchmask.jsp</name>
    <defaultviewstate>
      showReducedSearchMask
    </defaultviewstate>
    <viewstate>  <name>showReducedSearchMask</name>
      <transition>
        <actionname>expand</actionname>
        <switchTo>showEntireSearchMask</switchTo>
      </transition>
      ... <!-- additional transitions -->
    </viewstate>
    <viewstate>  <name>showEntireSearchMask</name>
      <transition>
        <actionname>reduce</actionname>
        <switchTo>showReducedSearchMask</switchTo>
      </transition>
      ... <!-- additional transitions -->
    </viewstate>
    ... <!-- additional PDA-specific view states -->
  </taskstate>
  ... <!-- additional (global) task states -->
<viewmappings>
```

Fig. 6 How the /pda/viewmappings.xml file might look like; see also Fig. 2

```
<html>
...
  <% if (myViewStateBean.getState().
                    equals("showEntireSearchMask")) { %>
    ... <!-- the code of the entire search mask -->
    <a href="controller?action=reduce">Reduce Mask</a>
  <% } else if (myViewStateBean.getState().
                    equals("showReducedSearchMask")) { %>
    ... <!-- the code of the reduced search mask -->
    <a href="controller?action=expand">Expand Mask</a>
  <% } %>
...
</html>
```

Fig. 7 The corresponding JSP file /pda/searchmask.jsp

5 Additional benefits of the separation between task state and view state

The previous section describes a general solution for how to design task-based pervasive computing applications. We now discuss some additional benefits of our approach, which provides a solution to some typical pervasive computing issues.

5.1 Changing devices "on the fly"

The strict separation of task state and view state allows the implementation of a rather advanced feature that lets the behavior of the application move closer to the vision of pervasive and ubiquitous computing in which applications seamlessly move from one device to another, tracking the user [6, 7].

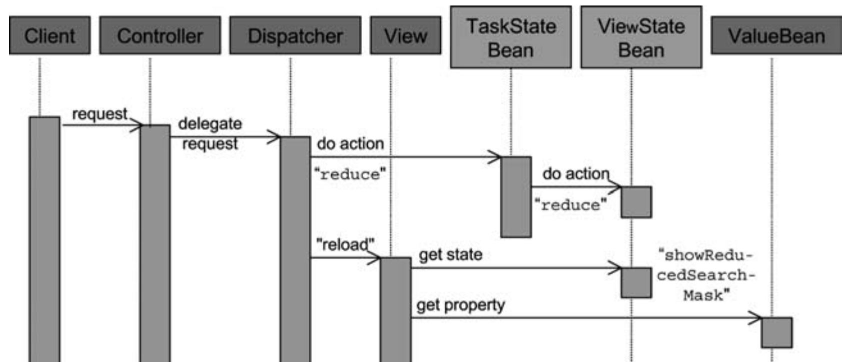

Fig. 5 A typical view state changing action such as a click on the "Reduce Search Mask" button on the PDA

```
<a href="<portlet:actionURL state="maximized">
        <portlet:param name="action" value="expand"/>
        </portlet:actionURL>" >Expand Search Mask</a>
```

Fig. 8 Sample JSP code according to the JSR 168 (version 1.0) specification

Consider the situation that a user of a Web application submits a query to the employee search engine of his company on his PDA and alters the view state of the result list that is particular to this gadget. As he enters his office, he doesn't want to enter the query again on his desktop PC, so he logs in and, in the case where the task state of the previous session and its value beans have been left unchanged, he is shown the *result list* in a desktop PC browser format, and not the search mask.

This behavior could obviously not be realized if the two devices had different task models, e.g., different JSP file names, because, even if the state of the application was handed over to the other device, it would not be guaranteed that this state (e.g., somePdaSpecificResultList.jsp) is still defined for the new device (e.g., if the desktop browser only features a single resultlist.jsp file). The separation of task state and view state is the key to support such behavior. When using the method described in this paper, it is assured that the two devices have their task state models in common. So, a dynamic device change could be implemented by simply handing over the "old" TaskStateBean to the "new" dispatcher class of the device. More precisely, the login procedure of the application would have to accomplish this task, and the "old" ViewStateBean could either be destroyed or retained as a separate object to possibly support the use of a single Web application with multiple devices in succession. That is, the implementation effort would merely be limited to the provision of a centralized TaskStateBean (and value beans) management, but still, the underlying application and business logic architecture would not have to be changed at all.

5.2 Using the XML task definition for code generation

Now, all the information required for both task state and view state transitions is completely accessible in the XML files. This allows partial automatization of the development process as defined in Sect. 3.

One way to utilize this information is to manually create one JSP file for each task state as specified in the mappings.xml file (e.g., including typical HTML or WML headers) and to create one subsection inside the appropriate JSP file for each view state based on the information in the viewmappings.xml file. (This is essentially the functionality of steps 2 and 6 in the algorithm of Sect. 3.) For small projects, manual execution of the algorithm might be sufficient. However, the task model included in the XML files described above is a good starting point for code generation, which could be helpful for larger projects with changing requirements.

Furthermore, since the task state transition information is also included in the mappings.xml file, the development environment could also offer the developer the appropriate set of actions that should be used in the respective JSP file. For example, the development environment might, based on the information given in the mappings.xml file, suggest that the searchmask.jsp file contains a link or a button with the action startSearch.

Also, the viewmappings.xml data could be used in a similar way. In the example of the finite view state automaton of the search mask (see Fig. 2), the development environment could generate navigation elements (e.g., a navigation bar) inside the search mask automatically. This navigation element would offer the user the functionality to switch from one view to another, e.g., from a reduced to an expanded state.

The main benefit of this approach is, on the one hand, less implementation effort and, on the other hand, assistance in maintaining a consistent user interface because changes made to the XML files will be cascaded down to the JSP files.

5.3 Fluid computing

The term "fluid computing" [10], which was introduced by the IBM Zurich Research Laboratory, represents a subarea of pervasive computing. It refers to the idea of using one application with multiple devices simultaneously. Therefore, it is necessary to propagate a state change on one device to all the other devices taking part in the current session.

By applying our approach to fluid computing, on the one hand, all the technical issues of pushing the notification of a state change to all devices still have to be tackled for each device. On the other hand, however, the effort of developing a fluid application with device-specific renderings (and, therefore, greater usability) could be decreased: assigning each device with its own view state model would ensure that the passive applications only have to be informed of state changes that do not merely affect a particular view state change of the active device. Also, in this case, task state changing actions and view state changing actions would have to be treated separately.

Eventually, the benefit of our approach for fluid computing is the possibility to fragment views and, thus, achieve increased usability on small devices.

6 Conclusions

This paper describes a development approach for Web applications that support multiple devices in a J2EE environment. The separation of task state and view state allows a more structured development approach and advanced run-time characteristics. In particular, device changes "on the fly" can be supported with little effort by retaining the device-independent task state informa-

tion. The cost of adding support of another device to a given Web application involves merely the presentation and view state layer. No changes have to be made to the task state transition model, while developers are still offered full control of design and usability issues (e.g., grouping and ordering).

However, this approach does not work for applications with device-specific *task* flow, e.g., if an Internet shop requires a cellphone customer to sign another agreement before an order can be legally accepted. Furthermore, the XML file schemas used in the example are only suggestions and have not yet been formally specified.

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, Massachusetts
2. Voegler G, Flor T (2003) Die integrierte Client-Entwicklung für PC und Mobile Endgeräte—am Beispiel der Portaltechnologie. In: Klaus D, Wolfgang K, Andreas O, Kai R, Wolfgang W (eds) Informatik 2003—Innovative Informatikanwendungen (Band 1). GI-Edition—Lecture notes in informatics (LNI), P-34. Köllen Verlag, Bonn, pp 207–211
3. Sun Microsystems (2002) Core J2EE patterns—Service to Worker. Available at http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceToWorker.html
4. Banavar G, Beck J, Gluzberg E, Munson J, Sussman J, Zukowski D (2000) Challenges: an application model for pervasive computing. In: Proceedings of the 6th annual ACM international conference on mobile computing and networking (MobiCom 2000), Boston, Massachusetts, August 2000
5. Burkhardt J, Henn H, Hepper S, Rindtorff K, Schäck T (2002) Pervasive computing: technology and architecture of mobile internet applications. Addison-Wesley, Reading, Massachusetts
6. Hansmann U, Merk L, Nicklous M, Stober T (2001) Pervasive computing handbook. Springer, Berlin Heidelberg New York
7. Banavar G, Bernstein A (2002) Software infrastructure and design challenges for ubiquitous computing applications. Commun ACM 45(12):92–96
8. Giannetti F (2002) Device independence web application framework (DIWAF). In: Proceedings of the HP Labs W3C workshop on device independent authoring techniques, SAP University, St. Leon-Rot, Germany, 25–26 September 2002
9. Paternò F, Santoro C (2002) One model, many interfaces. In: Proceedings of the 4th international conference on computer-aided design of user interfaces CADUI 2002), Valenciennes, France, May 2002, pp 143–154
10. Bourges-Waldegg D, Duponchel Y, Graf M, Moser M (1994, 2004) Fluid computing. IBM Zurich Research Laboratory, Switzerland. Available at http://www.zurich.ibm.com/fluid/index.html
11. Bergman L, Kichkaylo T, Banavar G, Sussman J (2001) Pervasive application development and the WYSIWYG pitfall. Lect Notes Comput Sci 2254:157–172