Mashing Up Context-Aware Web Applications: A Component-Based Development Approach

Florian Daniel[1] and Maristella Matera[2]

[1] University of Trento
Via Sommarive 14, 38100 Povo (TN), Italy
daniel@disi.unitn.it
[2] DEI - Politecnico di Milano
Via Ponzio 34/5, 20133 Milano, Italy
matera@elet.polimi.it

Abstract. Context-awareness and adaptivity in web applications have been gaining momentum in web engineering over the last years, and it is nowadays recognized that, more than a mere technology aspect, they represent a first-class design concern. This acknowledgment has led to a revision of existing design methods and languages, finally resulting in runtime adaptation being considered a cross-cutting aspect throughout the whole development process. In this paper, we propose a radically new view on context-awareness and show how a well-done component-based development may allow the fast mashup of context-aware and adaptive web applications. The proposed approach comes with an intuitive graphical development environment, which will finally enable even end users themselves to mash up their adaptive applications.

1 Introduction

The current technological advances are shaping up various scenarios where people with different (dis)abilities interact with web applications through multiple types of mobile devices and in a variety of different contexts of use. That is, we are moving toward accessing at any time, from anywhere, and with any media customized services and contents. In such scenarios, the need for effective methodologies for the fast development of adaptive and context-aware web applications arises.

Adaptivity is intended as the autonomous capability of the application to react and change in response to specific events occurring during the execution of the application, so as to better suit dynamically changing user profile data. *Context-awareness* is intended as adaptivity based on generic context data, not limited to user profile data.

Some works already addressed adaptivity and context-awareness, spanning a number of perspectives: from the representation of context properties through formalized context models [1, 2], to the definition of high-level modeling abstractions for the conceptual design of adaptive behaviors [3–6] (in Section 2 we discuss the cited approaches in detail). Typically, all these approaches share the

same view over context-awareness and consider it an *explicit* design dimension, to be addressed with specific (sometimes complex) design artifacts.

In this paper we break with this interpretation and show how a *mashup approach* to the development of web applications may *implicitly* provide support for context-awareness. More specifically, we propose an innovative framework for the development of context-aware web applications, which tries to hide the complexity of adaptivity design, also fostering reuse. In line with the current trend supported by the recent Web 2.0 technologies [7], our approach indeed aims at empowering the users (both developers and end users) with an easy-to-use tool for mashing up applications by integrating ready (adaptive) services or application components.

The framework leverages our previous results in the design of context-aware web applications [6, 8] and in the component-based development of web applications [9, 10]. In particular, we exploit an event-driven paradigm, which enables the composition of self-contained, stand-alone applications (components) equipped with their own user interface [10, 9], and show how context-aware applications can be composed by integrating *context components* with generic components. Context components are in charge of monitoring the context and generating *context events* when the context changes [8]. Such context events are then mapped onto operations of the generic components, which in this way are enabled to change their internal state to adapt to context changes. Provided that context components and generic components are respectively able to generate context events and react to such events, the adaptivity logic simply resides in the composition logic, which defines the synchronization of components without requiring any "ad-hoc" extension of the composition framework for the specification and implementation of adaptive behaviors.

To further facilitate the application development, the proposed approach is also complemented with an intuitive visual development environment, which will eventually enable even end users to mash up their adaptive applications.

1.1 Motivating scenario

As a reference example throughout this paper, we have implemented a location-aware tourist guide through Trento, Italy. Trento is particularly suited to our application, as the city center, where all the interesting sights are located, is covered with free wireless Internet connection, providing for the necessary connectivity.[3] The application is a location-aware mashup of Google Maps and a tourist information system. Besides explicit user navigations and selections, the application also reacts to location changes tracked by means of a GPS device. Figure 1 shows a screen shot and explanations of the application.

[3] `http://www.wilmaproject.org/`

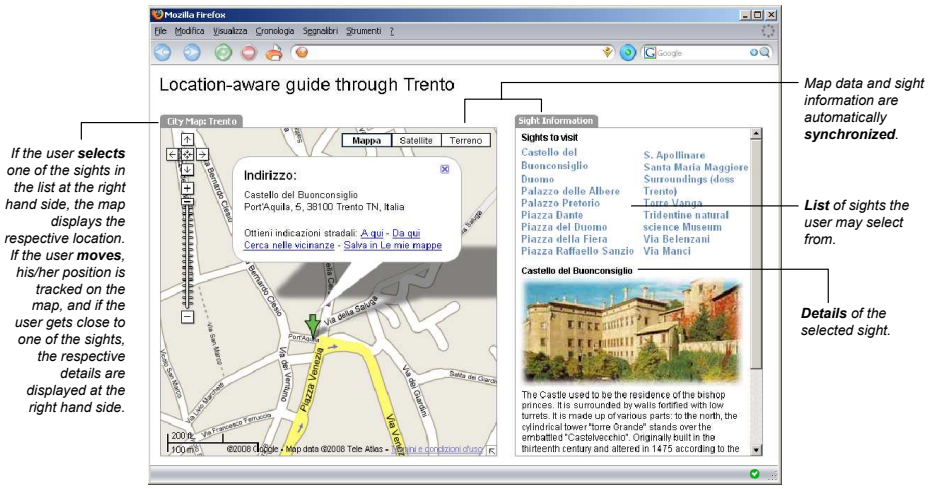

Fig. 1. Screen shot of the location-aware tourist guide based on GPS position data.

1.2 Contributions

Besides introducing a novel development paradigm, this paper provides an innovative vision on context-aware web applications. More specifically, we provide the following contributions:

- We show how the event-driven paradigm of our component-based web application development approach [9, 10] (Section 3) nicely suits the needs of context-aware applications. The resulting development approach (Section 4) hides most of the complexity of adaptivity design and fosters reuse.
- Consistently with the component-based approach, we show how also the context model, underlying all adaptive behaviors, can easily be mashed up, starting from so-called *context components*, generating context events (Section 4). Each context component is indeed in charge of monitoring and managing a separate context domain; therefore the "global" context model, needed to support adaptivity, is simply mashed up by composing context components.
- We equip the described approach with a Web 2.0 development and execution environment (Section 5), in order to enable also end users to mash up context-aware web applications. More precisely, we describe our easy-to-use, graphical editor and the light-weight, client-side execution environment.
- We finally summarize the main novelties of the proposed development method and provide an outlook over current and future works (Section 6).

2 Current approaches to context-awareness

Context-awareness has been mainly studied in the fields of ubiquitous, wearable, or mobile computing. Several applications have been developed [11, 12], and con-

text abstraction efforts have produced platforms or frameworks for rapid prototyping and implementation of context-aware software solutions [13]. However, recently some efforts have also been devoted to the Wed domain; they principally deal (i) with the gathering of context data and their representation as proper *context models*, and (ii) with the identification of modeling abstractions, able to support the design of adaptive behaviors.

Belotti et al. [14] address the problem of the fast and ease development of context-aware (web) applications and propose the use of a universal context engine in combination with a suitable content management system [15]. In such a framework, context affects the actual web application indirectly by altering the state of the database and is not able to trigger autonomously application functionalities. Also, developers have to deal with a centralized context model, possibly integrating the heterogeneous data coming from different context sources.

At a more conceptual level, some well known model-driven methodologies (such as Hera [16], UWE [5], and WebML [6]) aid developers in the design of adaptive web information systems, by extending design models with concepts and notations to specify adaptive behaviors. For example, Ceri et al. [6] propose an extension of the WebML model, in which adaptive pages are associated with a chain of operations that implement the page's adaptivity logic and are executed each time a context monitor [8] (which restricts the analysis of the context model to the only properties relevant to the currently viewed page) demands for adaptation. The modeling of adaptive actions leverages a set of adaptivity-specific units, especially related to the acquisition and management of context data and the enactment of adaptation actions. In [17] and [18, 19], the authors propose the use of event-condition-action rules for adaptivity specification and management.

The modeling approaches proposed so far allow developers to reason at a high level of abstraction. However, in all these approaches adaptivity design is strictly coupled with application design and requires the explicit and detailed specification of adaptivity rules and actions. The framework that we propose in this paper goes beyond these limitations and allows developers, or even end users, to concentrate on the global adaptive behavior of the application in an event-driven fashion, hiding the complexity of how single adaptation actions are executed – also fostering reuse, a typical feature of component-based development.

Finally, a family of approaches to personalization or adaptation is based on algebraic specifications and formal reasoning. For example, in [2] the authors extend SiteLang, a process algebra developed by the authors to express so-called application "stories". User preferences are specified by means of algebraic pre- and post-conditions that act as filters over a web information system's story space and tailor the algebraic expression of the story space to an individual user. Unfortunately, despite their effectiveness, such approaches make the implementation less intuitive, compared to both model-driven approaches and the component-based development proposed in this paper.

3 The Mixup approach to component-based web application development

In [10, 9] we have shown an approach to the fast development of web applications based on the composition of components that are equipped with own User Interfaces (UIs), i.e., we have introduced a systematic approach to mash up web applications. The distinguishing characteristic of the proposed approach is that it focuses on the integration of components at the presentation layer, leaving application and data management logic inside components. As illustrated in the rest of this section, component and composition models are inspired by research in the field of web services and the service-oriented architecture (SOA).

3.1 Component model

A so-called *UI component* is characterized by an abstract external interface that enables its integration into composite applications and the setup of inter-component communications. UI components are proper stand-alone applications, whose *state* is represented by the portion of UI published by the component and by the value of some component-specific *properties*. Components may generate *events*, which communicate to the outside world changes in the internal state; events are component-specific and typically follow a high-level semantics, e.g. a component that provides tourist information may publish an event `sightSelected` to notify other components of the selection performed by the user.[4] Components may have *operations*, which enable the outside world to modify the internal state of a component; operations are component-specific and follow the semantics of events, e.g. the tourist information component may have a `showSight` operation, which allows one to emulate a user selection from the outside.

Similarly to WSDL for web services, components are abstractly described via so-called UISDL (UI Service Description Language) descriptors. The `tinfo.uisdl` file in Figure 2 shows for example the UISDL descriptor of the tourist information component used in our case study (see Figure 1); for presentation purposes, the example is kept simple (e.g. we omit data types and technology bindings). After a mandatory header, the descriptor specifies the actual component: its identifier, its location, and its operations and events with possible parameters.

3.2 Composition model

Given the described abstract UI component model, the composition model can be kept simple. Indeed, we propose an *event-driven* model, where events from one component may be mapped to operations of one or more other components; mappings are expressed by means of so-called *listeners*. In addition to the direct mapping of events to operations, listeners also support *data transformations* in

[4] Low-level events such as mouse clicks or keystrokes do not represent meaningful events in the context of the proposed approach.

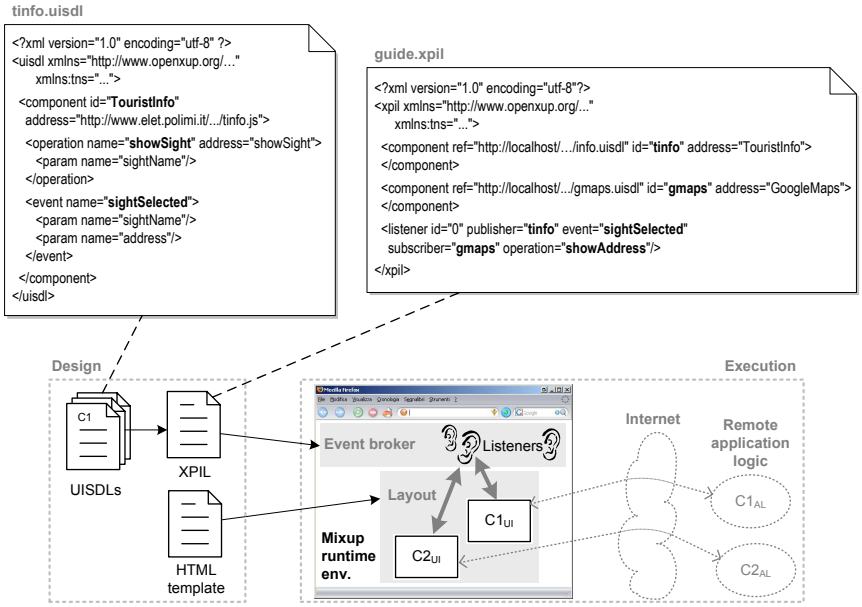

Fig. 2. The Mixup framework with UISDL descriptor and XPIL composition.

form of XSLT transformations, and the specification of more complex mapping logics via inline JavaScript. The definition of listeners represents the composition logic, while the *layout* of a composite application is specified by means of a suitable HTML template that contains placeholders, which can be used at runtime to embed and execute components, thereby re-using their UIs.[5]

The composition logic is expressed in XPIL (eXtensible Presentation Integration Language), an XML-based language specifying how UISDL-based components are integrated within single pages. The file `guide.xpil` in Figure 2 shows a simplified XPIL composition for our reference example (we omit complex data mappings or scripting). The composition refers to the two components in Figure 1: it references the UISDL descriptors of the tourist information component and of the Google Maps component and assigns unique identifiers (`tinfo` and `gmaps`). The identifiers are used in the specification of the listener that updates the map according to the user's selection of a sight: the `sightSelected` event of `tinfo` is mapped to the `showAddress` operation of `gmaps`. The HTML template corresponds to the HTML page shown in Figure 1 without the rendering of the two components.

[5] In our current implementation, we focus on web technologies, but conceptually our framework may also span other UI technologies.

3.3 The Mixup framework

The lower part of Figure 2 summarizes the overall framework. At design time, we mash components up starting from their UISDL documents; the mashup is stored as XPIL composition equipped with an HTML template for the layout. At runtime, a JavaScript/AJAX environment running in the client browser parses the XPIL file, instantiates the components, and places them into the layout for the rendering of the composite page. As shown in the figure, UI components may internally communicate with their own remote application logic necessary for the execution of the component; however, such communications are transparent to the designer who only focuses on the composition and layout logic.

4 Mashing up context-aware web applications

Leveraging our experience on context-aware applications [6, 18, 19], in this section we show how the described mashup approach naturally lends itself to the development of context-aware and adaptive web applications. The proposed approach is characterized by an adaptation specification logic that is very simple and consistent with the previously described composition logic, thus elevating the abstraction level at which developers deal with adaptivity compared to the approaches discussed in Section 2.

4.1 Adaptivity layers in Mixup

Context-awareness means runtime adaptation, i.e. *adaptivity*, in response to changes of context data, whose structure is expressed by means of some kind of context model. Let's first consider the typical adaptivity features of context-aware web applications, i.e. *what* we adapt. Considering the works discussed in Section 2, we can categorize the typical adaptivity actions supported in current adaptive/context-aware web applications into the following six features:

- *Adaptation of contents*: contents/data published in pages may be changed;
- *Enactment of operations*: external operations or services may be invoked;
- *Adaptation of style*: properties like colors and font sizes may be changed;
- *Adaptation of layout*: the arrangement of page contents may be re-organized;
- *Hiding/showing of links*: hyperlinks may be dynamically hidden or shown;
- *Automatic navigation actions*: hyperlinks may be automatically navigated on behalf of the user.

If we now consider our mashup approach, we can easily identify *two layers* at which adaptivity features may operate. The two layers are characterized by different adaptivity features:

1. *Component adaptations*: Components may internally support one, more, or all of the above features. Components are indeed small stand-alone applications, and as such they may be developed according to the approaches

discussed in Section 2 and based on own context-data. Alternatively, they may expose operations to be invoked from the outside to trigger the supported features; in this paper we will focus on such kind of operations. Well-developed components are key for the success of mashups in general; here we build atop of current best practices and experience.

2. *Composition adaptations*: also the composite application may support one, more, or all of the above features (a composite application may have its own business logic, in addition to the components it integrates). But the composite application may also support adaptivity features over its composition logic, which are new with respect to the above features:
   - *Hiding/showing of components*: components in the composition may be hidden or shown dynamically;
   - *Re-configuration of listeners*: new listeners may be added or existing listeners may be dropped at runtime;
   - *Selection of components*: new components may be dynamically selected (e.g. from a component registry) and added to the composition.

4.2 Enabling context-awareness through context components

But *to what* and *when* do we adapt our application? In short, we adapt to changes of context data, typically structured according to an application-specific context model, and we adapt as soon as such changes are registered by our application. It is exactly the triggering of adaptivity actions where the proposed composition approach shows its full power: instead of having a centralized context model that captures all context data and the respective changes, we use dedicated *context components*, which are in charge of observing a particular piece of context and of generating events in response to changes thereof. Such *context events* are then mapped to the operations of the components or of the composite application that are able to trigger changes to the internal state, i.e. to *adapt* the component or the whole application to changing context conditions.

Figure 3 shows how context components seamlessly extend the Mixup framework depicted in Figure 2, effectively enabling context-aware mashups. We distinguish between client-side and server-side context components. *Client-side* components fully run on the customer's device and enable the communication of *client context* data sensed via dedicated sensing modules. *Server-side* components run their business logic (i.e. the specific context management logic) on the server side, thus enabling the communication of context data representing centrally sensed or aggregated data. There may also be *remote* context components, which enable the access to context data that are outside of the control of the developer.[6] For instance, the following represent a few possible and reasonable context components:

[6] The described use of context components is fully in line with the idea of *context monitor* discussed in [8].

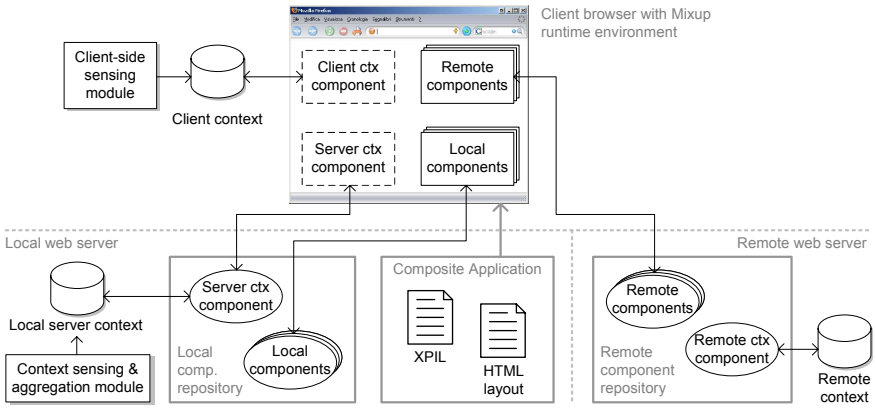

Fig. 3. The use of dedicated context components enables the development of context-aware web applications.

- *Location component*: a component, implemented at the client- or at the server-side[7], which may fire "low-level" position events, for example regarding longitude and latitude, or "high-level" events, for example regarding cities, streets, or countries;
- *User Model component*: a server-side component in charge of monitoring changes to a user model typically stored at the server side. The component could be used to provide advanced (e.g. adaptive) personalization features;
- *Time component*: a client-side component that may fire whatever event (specified at design time) at given time instants or intervals. The component could expose a set of configuration operations that allow the designer to set up the events to be emulated;
- *System component*: a server-side component that exposes runtime data about the health or performance of the running application;
- *Shared context component*: a server-side component based on context data aggregated at the server side, which e.g. may allow the generation of events that express the presence of other people in a same physical location;
- *Weather component*: a remote component that accesses weather forecasts provided by third-party service providers and supplies users with forecast data depending on their current GPS position.

As can be seen in the above examples, context components typically concentrate on one specific *context domain*. Therefore, we do not have any centralized

[7] The logic of this component depends on the mechanism adopted for capturing context data. For example, components implementing proprietary sensing mechanisms can be required on the client side, as it happens for GPS-equipped devices, or server-side components can be used to communicate to the client context data collected through centralized sensing infrastructures, such as those based on infrared signals or RFID.

reference *context model*. Just like the whole application itself, also the context model is simply "mashed up" by putting together the context components that are needed. We are hence in presence of a modular approach to construct the context model, which transparently integrates context data from disparate sources, possibly distributed over the Web.

It is finally worth noting that context components that generate context events are per definition *active* components that are able to communicate context data without an explicit user intervention, thus solely based on the dynamics of context. This therefore leads us to interpret context as an *independent actor*, working on the same application as users do [8].

4.3 Mashing up context-aware web applications

As explained above, context components generate events that can easily be mapped onto other components' operations (just like with conventional UI components) in order to trigger adaptivity features. When developing context-aware composite applications, the XPIL composition therefore also reflects the *adaptivity logic*, which is expressed by means of the listeners that specify how context events influence other components. The *layout* of context-aware applications is instead slightly different from non context-aware applications, as context components typically come without own UI, reason for which they are typically invisible in the composite application. Using context components in the Mixup framework does therefore not imply the need for any extension to the existing runtime environment nor to the composition model.

The adaptive features that can be used when developing a context-aware mashup very much depend on the capabilities of the components to be integrated. Well-designed components come with a rich set of operations, which may enable a wide spectrum of adaptive behaviors. As already discussed in earlier, *component adaptations* depend on the internal implementation of components (which typically is out of the control of the mashup developer), while *composition adaptations* can be freely defined by the developer.

It is worth noting that from a technical perspective the distinction between context components and conventional components is very blurred. Whether a component is a context component or not rather depends on the overall *semantics* of the application to be developed. As a matter of fact, each component that generates events may be considered a context component, as its events may be used to trigger adaptivity features in other components.

To equip our example application of Figure 1 with the necessary location-aware behavior, we use a location context component that tracks the user's position in terms of street name and number. Also, the operation `showSight` of the tourist information component accepts addresses in input and, if a corresponding sight can be found, publishes the respective details; otherwise, no changes are performed. In Figure 2 we introduced the XPIL logic for the integration of the UI components only. In order to achieve context-awareness, the following code lines need to be added to the XPIL composition of Figure 2 (`guide.xpil`):

```
<component ref="http://localhost/.../gps.uisdl" id="location"
   address="Location_Context">
</component>
<listener id="1" publisher="location" event="streetChanged"
                 subscriber="gmaps" operation="showAddress"/>
<listener id="2" publisher="location" event="streetChanged"
                 subscriber="tinfo" operation="showSight"/>
```

The first three lines import the location context component. The remaining lines define two listeners, one that couples the `streetChanged` event of the `location` component with the `showAddress` operation of the `gmaps` component, and one that couples the same event with the `showSight` operation of the `tinfo` component. The two listeners specify the adaptivity logic of the context-aware application.

4.4 Termination and confluence of adaptive behaviors

Although in the proposed approach we allow developers to specify cyclic dependencies among listeners, at runtime the execution environment does not allow for cycles. More precisely, at runtime possible events generated by a component in response to the invocation of one of its operations (the invocation is triggered by another event) are neglected, thus effectively preventing the cascaded or cyclic invocation of listeners. This implies that possible multiple dependencies from one and the same event must be explicitly modeled through suitable listeners (one for each dependency). Listeners reacting to the same event are activated concurrently, and the final result of their execution depends on the order of the invocation of the listeners. Therefore: *termination* of page computation is guaranteed, as there are no cyclic runtime dependencies among listeners; and *confluence* depends on the order in which events are generated and – for listeners activated on a same event – on the order in which listeners over a same event are specified in the XPIL composition. Consequently, there are no indeterministic behaviors in the evaluation of listeners, and the designer has full control over the runtime behavior of the composite context-aware application.

5 Implementing context-aware mashups

The development approach described in this paper is assisted by a visual development environment, which allows for the drag-and-drop composition of context-aware web applications [9]. Both composition logic and layout can be easily designed; for the layout, it is also possible to upload an own HTML template with placeholders. Figure 4 shows the graphical composition of the logic of our reference application: the three components (Google Maps, tourist information, and location context component) are represented by the icons in the modeling canvas; the listeners are represented by the connections among the components. A double click on components and listeners allows one to set the necessary parameters.

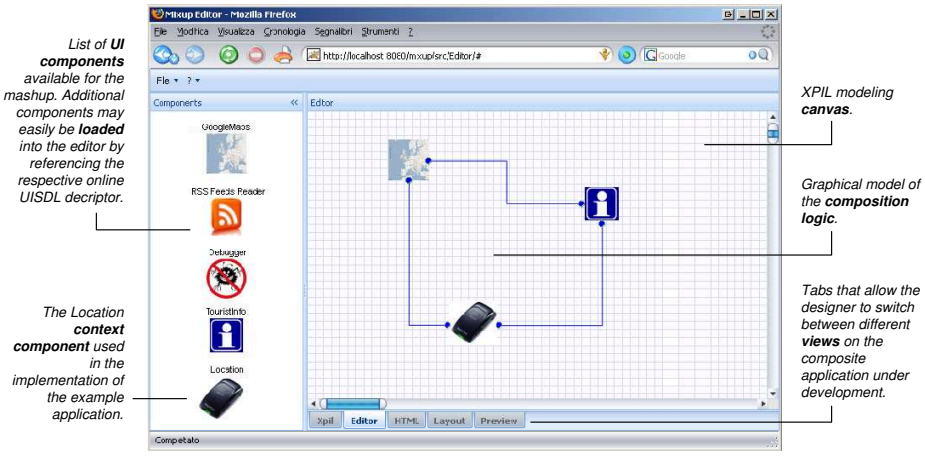

Fig. 4. The Mixup editor for the drag-and-drop mashup of XPIL and layout logic.

The editor is an AJAX application that runs in the web browser. Compositions can be stored on our web server and executed in a hosted fashion through the Mixup runtime environment, a JavaScript runtime library that parses the XPIL file and sets up the necessary communication among the components.

The location context component used in the case study is based on previous work [8]: it leverages a client-side Bluetooth GPS device, interfaced via the Chaeron GPS Library[8], and is wrapped by means of Flash (to exchange position data between the component and the GPS library) and JavaScript (to provide a UISDL-conform interface). The other two components are conventional UI components.

6 Discussion and outlook

Runtime adaptivity and context-awareness are relevant aspects in the design of modern web applications. Indeed, if we consider how they have been addressed so far by the most prominent conceptual design methods, we note that they have been treated like an explicit, first-class design concern. The fashion in which we develop context-aware or adaptive web applications in this paper allows us to make the most of such approaches, but it also goes *beyond* what has been done so far in this area. Provided a set of ready components, with the described mash up approach we indeed enable developers (or even end users) to focus on adaptive behaviors only, completely hiding the complexity of how adaptations are actually carried out.

Adaptivity is enabled and supported by self-contained stand-alone applications, i.e. UI components. The actual adaptivity logic is represented by means

[8] http://www.chaeron.com/gps.html

of a simple composition language, such as XPIL, interpreted during runtime by a light-weight runtime environment.[9] The simplicity of composing adaptive web applications, however, does not come for free: complexity resides *inside* the components, which provide for the necessary business logic to generate events and enact operations. In this setting, the design of the components (both UI and context components) becomes *crucial*, as the events they generate and the operations they support build up the expressive power of the adaptivity logic, which, hence, varies from application to application, depending on the components that are adopted. It is exactly the design of UISDL-compliant components that benefits most from existing approaches to the development of adaptive or context-aware web applications, since, as a matter of fact, components can be considered "traditional" web applications equipped with a UISDL API.

The described composition model based on XPIL does not only enable the mashing up of context-aware and adaptive applications, it also allows for the easy introduction of adaptivity features into already *existing* applications. It suffices to add a respective context component, generating suitable context events (e.g. changes in user characteristics, preferences, locations, etc.), to the composition and to map its events to the components of the composite application, in order to obtain adaptive behaviors. The graphical XPIL editor described in this paper also supports the modification of an application's adaptivity logic during *runtime*, thus paving the road (i) for *fast prototyping* and easy testing of otherwise complex application features during application design and (ii) for the efficient and consistent *evolution* of an application after its deployment.

When we first introduced our approach to the component-based development of web applications [10, 9] we were driven by the event-based paradigm by means of which typically user interfaces are developed in order to achieve what we called "presentation integration" (in analogy with data and application integration); we did actually not focus on adaptive application features. But a close look at the result of this effort allowed us to identify analogies with our previous research on adaptive and context-aware web applications [6, 18, 19]: mashing up context-aware web applications based on the described framework effectively means setting up *adaptation rules*, the listeners. The developed framework is thus intrinsically adaptive, a property that we heavily leverage in this paper.

In our future work we will focus on data transformations between events and operations, on complex navigation structures spanning multiple composite pages, on the integration of generic web services in a UISDL-compliant fashion, as well as on classical personalization features by means of dedicated user model components.

[9] Due to the highly UI- and event-based logic of our composition approach, we have opted for UISDL/XPIL, instead of for instance standard web service languages such as WSDL/BPEL, which do not natively support UIs. Although theoretically WSDL/BPEL could be used for similar purposes, this would however require extending the two languages with new features, which would only get the already complex languages more complex. We instead believe that our specific context demands for languages that are easily intelligible and easily executable (e.g via client-side code).

References

1. Henricksen, K., Indulska, J.: Modelling and using imperfect context information. In: PerCom Workshops, IEEE Computer Society (2004) 33–37
2. Schewe, K.D., Thalheim, B.: Reasoning about web information systems using story algebras. In Gottlob, G., Benczúr, A.A., Demetrovics, J., eds.: ADBIS. Volume 3255 of Lecture Notes in Computer Science., Springer (2004) 54–66
3. Fiala, Z., Hinz, M., Houben, G.J., Frasincar, F.: Design and implementation of component-based adaptive web presentations. In Haddad, H., Omicini, A., Wainwright, R.L., Liebrock, L.M., eds.: SAC, ACM (2004) 1698–1704
4. Frasincar, F., Barna, P., Houben, G.J., Fiala, Z.: Adaptation and reuse in designing web information systems. In: ITCC (1), IEEE Computer Society (2004) 387–291
5. Baumeister, H., Knapp, A., Koch, N., Zhang, G.: Modelling Adaptivity with Aspects. In: ICWE. (2005) 406–416
6. Ceri, S., Daniel, F., Matera, M., Facca, F.M.: Model-driven development of context-aware Web applications. ACM Transactions on Internet Technology **7** (2007)
7. Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R.: Understanding UI Integration: A survey of problems, technologies. Internet Computing **11** (2007) 59–66
8. Ceri, S., Daniel, F., Facca, F.M., Matera, M.: Model-Driven Engineering of Active Context-Awareness. World Wide Web Journal **10** (2007) 387–413
9. Yu, J., Benatallah, B., Casati, F., Daniel, F., Matera, M., Saint-Paul, R.: Mixup: a Development and Runtime Environment for Integration at the Presentation Layer. In: Proceedings ICWE'07. Volume 4607 of LNCS., Springer Verlag (2007) 479–484
10. Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M.: A Framework for Rapid Integration of Presentation Components. In: Proceedings of WWW'07, ACM Press (2007) 923 – 932
11. Want, R., Hopper, A., Falcao, V., Gibbons, J.: The Active Badge Location System. ACM Trans. Inf. Syst. **10** (1992) 91–102
12. Long, S., Kooper, R., Abowd, G.D., Atkeson, C.G.: Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In: MOBICOM. (1996) 97–107
13. Salber, D., Dey, A.K., Abowd, G.D.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: CHI. (1999) 434–441
14. Belotti, R., Decurtins, C., Grossniklaus, M., Norrie, M.C., Palinginis, A.: Interplay of Content and Context. J. Web Eng. **4** (2005) 57–78
15. Grossniklaus, M., Norrie, M.C.: Information Concepts for Content Management. In Huang, B., Ling, T.W., Mohania, M.K., Ng, W.K., Wen, J.R., Gupta, S.K., eds.: WISE Workshops, IEEE Computer Society (2002) 150–159
16. Vdovjak, R., Frasincar, F., Houben, G.J., Barna, P.: Engineering Semantic Web Information Systems in Hera. J. Web Eng. **2** (2003) 3–26
17. Garrigós, I., Gómez, J., Barna, P., Houben, G.J.: A Reusable Personalization Model in Web Application Design. In: WISM'05. (2005)
18. Daniel, F., Matera, M., Morandi, A., Mortari, M., Pozzi, G.: Active rules for runtime adaptivity management. In: Workshop Proceedings of ICWE'07. (2007) 28–42
19. Daniel, F., Matera, M., Pozzi, G.: Managing Runtime Adaptivity through Active Rules: the Bellerofonte Framework. Journal of Web Engineering **7** (2008) 179–199