An Architecture for Supporting Multi – Device, Client – Adaptive Services

Guido Menkhaus
Software Research Lab
Constance University
D-78457 Constance, Germany
guido.menkhaus@uni-konstanz.de

Abstract

The original idea of the World Wide Web was to create a universe of network-accessible information. Since its inception, the World Wide Web has evolved from a means to help people access and use information into an interactive medium. This has caused a dramatic increase in the development effort for interactive services, which now have to support multiple clients with widely varying computing and user interface capabilities. Personalization features tend to render this task even more complex. The paper presents the MUSA (Multiple User Interfaces, Single Application) system which addresses both issues with the introduction of an event graph that abstracts user interface and personalization issues from the implementation of the service on different client-devices.

1 INTRODUCTION

The current trend of Web access and computing is drifting away from the desktop PC as the principal device to access services and information on the Internet to consumer devices such as mobile phones, handheld computers and a wide spectrum of Personal Digital Assistants (PDAs). Technological advances allow the manufacturing of smaller and more portable but increasingly powerful computing devices [Compaq 2001; Palm 2000]. As a consequence, devices that originally have been designed for a special purpose - mobile phones were originally designed for audio communication, PDAs provided organization facilities - nowadays embrace Internet technology. This tendency has led to the emergence of the ubiquitous computing paradigm [Weiser 1991; Weiser 1993; Ubinet 1999].

Currently, the impact of ubiquitous computing on software engineering manifests itself mostly at the Middleware level [Roman *et al.* 2000]. Until recently Middleware platforms were targeted at vertical coverage of specific scenarios, which included the support of just a few powerful computing devices with high-resolution monitor and sophisticated graphical capabilities [Garlan 2000]. Vertical coverage of a few devices and configurations does not scale up well to environments that include a wide variety of different devices. Consumer devices with integrated Internet-access are becoming more popular and their diversity grows with their market penetration and with the extension of the mobile communication infrastructure. This requires software architectures that are capable of supporting horizontal coverage of a wide range of devices and scenarios and that can promote the ubiquitous computing paradigm. Ubiquitous computing demands a shift from Middleware platforms that assume computational environments based on conventional desktop user interfaces to architectures providing support for a wide variety of computing devices with low or non-visual user interface (UI) capabilities [Abowd 1999]. These UIs range from graphical UIs on displays with varying quality and size, Web-based interfaces using applets, to automatic speech recognition and natural language understanding.

Standard software architectures following the Model-View-Controller or similar architectures have already considered the means offered by abstract UIs, which separate the abstraction from its implementation, to address the issue of multiple UIs [Bass *et al.* 1998; Gamma *et al.* 1995]. However, the solutions they propose need to be revisited to assess and improve their compatibility with the ubiquitous computing paradigm [McCann and Roman 1998]. They advocate the separation of the system functionality to isolate UI concerns from the service logic. This eases the integration of multiple UI, which share the same service logic. The concept of abstraction and separation is used to ensure that changes to a component do not affect other components, as long as the interface of the component remains unchanged. General software architectures based on the ideas of abstraction and separation address issues regarding multiple UIs only along one dimension. They have the potential to support multiple UIs only under the restrictive assumption

that the UI guarantees the implementation of the service's presentation logic in a complete and satisfactory way. However, nothing ensures that the existing service presentation logic's demands can be mapped onto a new UI. This unique vertical coverage of UI issues is only sufficient for services, which are designed and intended to run on dedicated hardware architecture. Designers of services, which aim to run on a wide range of different computational devices with fundamental differences in human machine interaction, have to consider the aspect of horizontal coverage as an architectural issue. Middleware targeting at horizontal coverage requires an architecture that is flexible to adapt to the growing variety of devices with different UI requirements.

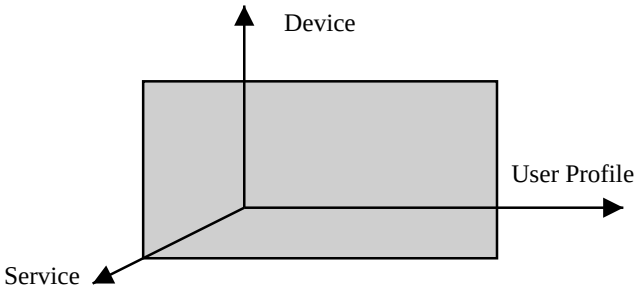


Figure 1: 3-dimensional service space: An architecture is desirable that supports multiple user profiles and access from multiple devices for a single service.

UI research has put much effort into personalizing services in order to enhance human-machine interaction. The possibility to directly manipulate interfaces to access information and invoke services, i.e. to personalize and organize the working space, is important for a user using services on a variety of computing devices because the UI on these devices may have only limited UI capacities. Focusing on the aspects of a service that the user actually needs on a specific device enhances the use of the service, is of significant advantage to the user, and thus of crucial importance for the acceptance of a service. Only services that give users an obvious and immediate benefit find market acceptance.

The paper introduces the architecture of the MUSA system that employs an event driven dialog architecture to decouple services from their UI and to allow the personalization of the latter (c.f. Figure 1). It presents an event graph for designing and implementing dynamic interactive services. The event graph is a result of an analysis of the requirements of interactive services, which support a variety of different UIs.

The remaining sections of the article are organized as follows. Section 2 discusses the design goals of the MUSA architecture. A detailed description of the MUSA architecture is given in Section 3, introducing the application controller of the MUSA Core system and describing the event graph as well as the set of employed events. How to personalize a mobile service designed with the MUSA system is discussed in Section 4. In Section 5 some other related work in the domain of mobile service architecture is presented. A

summary with concluding remarks closes the paper.

2 DESIGN GOALS

Modern interactive services need to be more and more flexible in order to adapt to the growing variety of computing environments. The development of such systems is subject to well-known problems including cost, maintainability, and sensitivity to changes [Taylor *et al.* 1995]. In this section, we elaborate design properties and discuss how the MUSA architecture has achieved these goals.

Flexibility. In system architecture two opposing principal concerns are the minimization of collaboration between components and maximization of cohesion within a component. Collaboration must be minimized in order to avoid interference between different concerns. The decomposition of concerns eases the integration and change of new components. At the same time, it is desirable to maximize the cohesion of a system to effectively localize concerns and to limit the number of components, which have to change in the presence of modifications of a specific concern [Szyperski 1998]. The identification of the proper balance between these opposing forces is the search for an effective separation of concerns [Garzotto *et al.* 1993; Rossi *et al.* 1999]. The MUSA architecture provides the separation of concerns at three levels (Figure 2):

1. Between the service and the navigational structure. The objective is the reduction of modification of the navigational structure in case of a modification of the service and vice-versa.

2. Between the service and the abstract UI. The separation aims at the mutual reduction of modifications in the case of a change of the service or UI.

3. Between the abstract UI and the concrete UI. The goal is to decouple the abstraction from its implementation, so that the two can vary independently. It facilitates the removal, addition and modification of UI concerns. The separation of concerns directly impacts the flexibility of the service and promotes and eases the modification, extension and maintenance activity [Ball *et al.* 2000]. It also allows the independent variation of the UI, the navigational structure and the service.

Adaptability. The MUSA system allows direct manipulation of the user interface to give the user the possibility to organize his or her working space and thus to enhance human-machine interaction. This is made possible through the introduction of an event graph, which incorporates the navigational structure and the abstract UI.

Reuse of software components. Software reuse, to be truly effective, needs to go beyond the mere reuse of code fragments [Fontoura *et al.* 2001]. A fundamental principle of software engineering is the creation

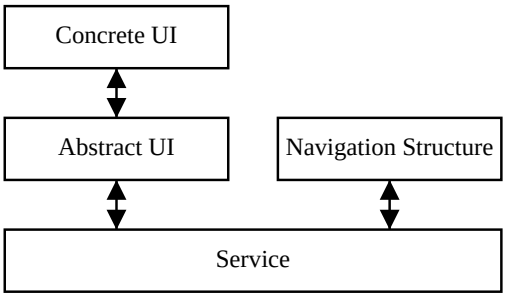


Figure 2: Separation of concerns.

of structures with well-defined and well-focused concerns. The MUSA architecture introduces the notion of Specialized Processing Units (SPU), which materializes a specific system feature. The introduction of SPUs promotes reuse and rapid prototyping. The MUSA system provides specific SPUs implementations, which puts the programmer in the situation to concentrate on the development of domain-specific services.

3 THE MUSA ARCHITECTURE – CONCEPTS AND CASE STUDY

The MUSA architecture is conceptually split into four tiers and employs an event-driven design of the kind commonly used for interactive services [Wang 1998] (Figure 3). The client environment represents the first tier. No service data is installed on the client side and the client communicates via Internet or wireless Internet with the service. Typically the client is represented by a browser but could also be a device with no visualization capacity like a telephone. The second tier is the request processor, which deals with the client's request. The communication between the client's UI and the service, carried out by the MUSA architecture, passes through the request processor. It forwards the communication stream to the MUSA Core System. The request processor is the link between the MUSA core system and the client UI and converts the client requests into events, which are used throughout the MUSA architecture. This component allows for centralizing all client-specific request handling. The request processor receives a set of events from the MUSA core system. This set of events is enabled within the current dialog communication. The request processor transfers the set of events to the UI. The UI prompts the current events and collects requests resulting from the user interaction. The request processor converts client requests into events and dispatches them to the MUSA core system. The MUSA Core System represents the third tier. It consists of specialized processing units (SPU), which perform the multi UI and the personalization feature. It contains the service controller, which mediates between the user interaction and the delivery of service data to the user UI through the service adapter. The service controller handles the processing of events of the event graph. The event graph implements the navigational design, the content organization and the interaction of the service logic. The event graph abstracts these aspects of the service logic. The actual implementation of

the service's functionality is separated from the design and implementation of the event graph. The service logic represents the fourth tier. The service is the body of code for which the MUSA system provides the UI and personalization features. It processes the set of events and its associated computation. The service controller transfers in response to the event processing the next set of enabled events to the request processor. It assigns to the dialog semantic objects that encapsulate the knowledge of the event handling. The service communicates through the service adapter with the MUSA Core System and has no knowledge about the navigational structure or the presentation issues of the service data. It is designed independently from these issues.

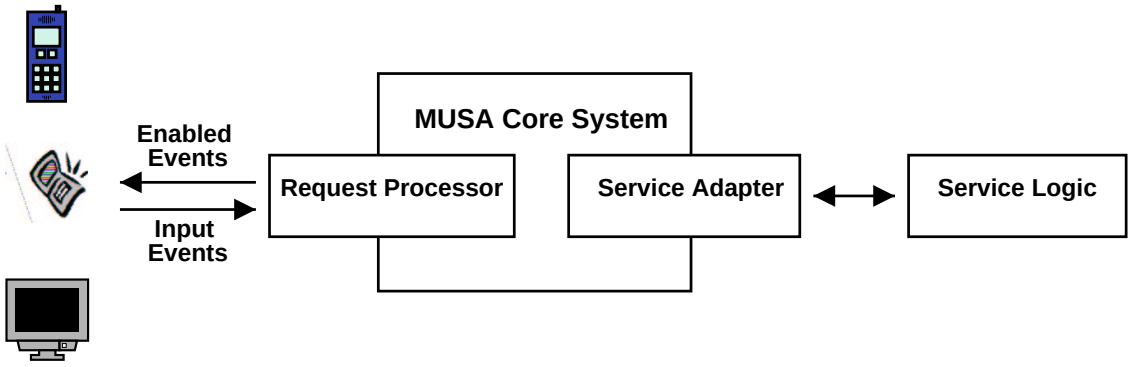
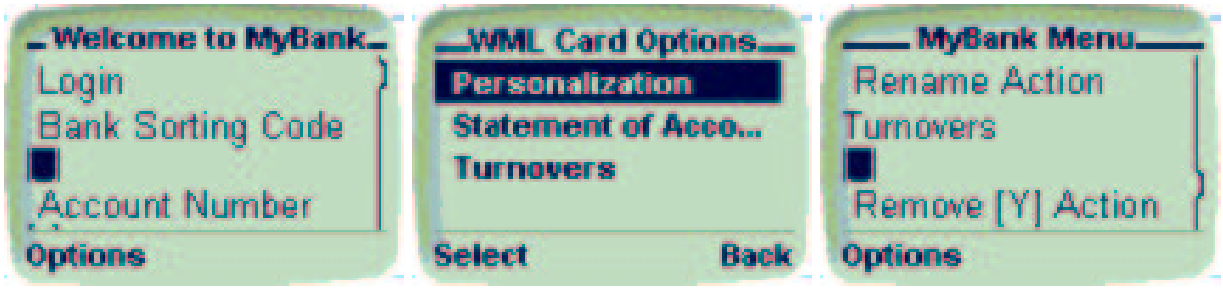


Figure 3: The MUSA architecture is a four-tier architecture consisting of the client UI, the request processor, the MUSA Core System, and the service logic.

3.1 A Sample Application Based on MUSA

Section 3 has roughly sketched the MUSA architecture. This section illustrates the MUSA system by means of an example. We have developed a prototype of a mobile banking application. In this mobile banking example, the user has to provide login data for user authentication. After authentication, the user can retrieve information about his accounts. Figure 4(a) illustrates the login screen of the application on a mobile telephone. Figure 5 shows the same screens as in Figure 4, but on a HTML browser. Figure 4(b) shows the menu of the banking application. The first menu item is the personalization request. If the user demands to personalize the bank menu dialog, the personalization dialog in Figure 4(c) is presented. Since the application was implemented with the MUSA architecture, UI issues and the navigational structure of the application were developed independently from the service logic.



(a) (b) (c)

Figure 4: (a) Illustration of the login screen display of the banking application. (b) The banking menu, with the option to personalize the dialog. (c) The personalization screen of dialog shown in (b). The option "Statement of Account" and "Turnover" can be renamed and removed from the dialog

(a) (b) (c)

Figure 5: HTML version of the WML screens in Figure 4.

3.2 MUSA Core System

In this section we elaborate on the architectural issues of the MUSA Core System. The main architectural concepts are layering, self-standing SPUs, independent components, and a set of interfaces. Theses concepts are explained in the following in more detail:

Layering. The MUSA core system is decomposed into a number of layers. The main layers of the system are the communication layer, the infrastructure layer, the specialized processing units (SPU) layer and the specialized processing component (SPC) layer, as shown in Figure 6. The infrastructure layer provides basic services to the other layers and is responsible for the configuration and dynamic instantiation of the SPC layer.

Each SPC has to register with the infrastructure layer to make its service available. The communication layer provides the communication means and determines the structure of the SPC layer. The SPC layer contains the processing knowledge of the MUSA core system. Each SPC has a well-defined interface and well-focused responsibility. A SPC offers a particular service. The SPU layer is basically identical to the SPC layer, but logically groups a number of SPCs into a SPU. A SPU performs a specific system functionality.

Self-standing SPUs. Adequate separation of system functionalities is one of the most common means of software design [Bass *et al.* 1998]. In the MUSA core system a main functionality is the personalization aspect, or the service controller, which handles the event-graph processing. The MUSA architecture localizes these concerns within a SPU. A SPU may be removed without affecting the rest of the system. The transformer functionality for example, is independent of the functionality of other software components and could be removed without affecting the rest of the system. In the special situation where two SPUs are not independent, the combination of the two introduces more than the sum of the two functionalities, i.e. the two develop a combined behavior [Winjstra 2000]. The combination of the involved SPUs forms a SPU of a higher level. The result is a hierarchy of SPUs (c.f. Figure 7). A SPU participating in a hierarchy of SPU cannot be removed without breaking the hierarchy and the implementation of the combined behavior. The service controller SPU and the personalization SPU are two SPUs, which are not independent of each other referring to a personalized service. The service controller SPU delivers a service, implemented as an event graph. The personalization SPU initiates modifications and transformations of the event graph during the process of personalization. I.e., the personalization SPU requires the service controller SPU. The two SPUs build on a higher level a personalized service SPU. Their combined behavior is the delivery of a service that can be personalized. The removal of the personalization SPU breaks the hierarchy and the combined behavior of the two. However, removing the personalization SPU does not affect the service, supervised by the service controller SPU.

Four SPUs, shown in Figure 6, are plugged in the MUSA system. The transformer SPU transforms the abstract UI of the event graph into a concrete UI applying a transformation on the event-graph dependent on the user's device. The service controller SPU manages the service and mediates between user interaction and delivery of service data to the user UI. The event graph manager SPU controls the access and transformation of the event graph. The personalization SPU implements the personalization feature of the MUSA enabled services.

Independent SPUs. The independence of a SPU is closely related to the previous point. In order to satisfy the design requirements of frequently changing interactive services, SPUs can be plugged in and out without compromising the functionality of the system.

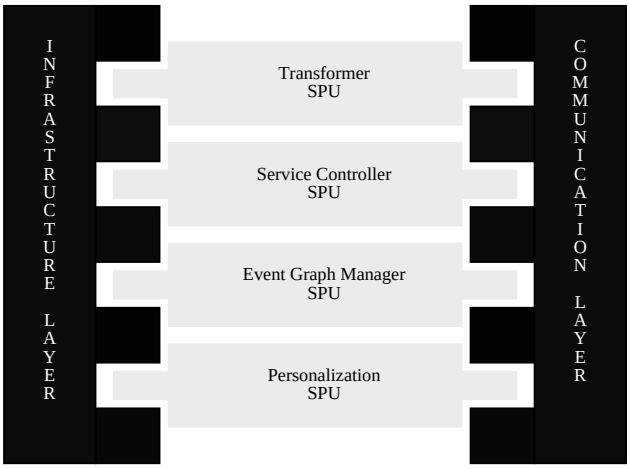


Figure 6: The MUSA core system architecture, with the Infrastructure, Communication and SPU Layer. Four SPUs are shown: Transformer, Service Controller, Event Graph Manager, and Personalization SPU.

Services are subject to frequent modifications. The design of UIs is permanently adapted to user requests and content changes with high frequency. The need to offer, optimize and modify services to stay competitive is high. Services are added, obsolete ones removed, and existing services adapted. The MUSA architecture with its possibility to plug in SPUs avoids a monolithic system structure and promotes separation of concerns. SPUs have no immediate knowledge of their mutual existence, unless it is required.

SPCs are grouped logically into a SPU. A SPC is a well-defined and localized service. SPCs belonging to a specific SPU perform in coordinated combination system features. The concept of logically grouped SPCs into SPUs follows the idea of framelets. Framelets are small sized, self-standing "frameworks". They represent logically related services. They have been introduced as a mean to decompose a framework into smaller entities, to partition the system in smaller units to master the quantitative complexity and to reduce design complexity [Pree and Koskimies 1999; Fontoura *et al.* 2001]. A SPC is the smallest reusable unit in the MUSA architecture and represents the building block of a SPU. The SPC approach follows the idea of a bottom-up, compositional view on system architecture. The focus is on construction and composition of higher-level building blocks out of simple units. A SPU consists of a number of SPCs and a specific communication structure. It is essential to integrate the SPCs correctly into the specific SPU structure.

Until now we have assumed a 1-to-m relationship between a SPU and its SPCs. However, a SPC can belong to n distinct SPUs. This means that there is an n-to-m relationship between SPUs and SPCs, as demonstrated in Figure 7. There are specific situations in that a SPC provides functionality to more than one SPU. A SPC that serves more than one SPU is called a Bridge SPC. SPUs containing Bridge SPCs can only be removed, while the Bridge SPCs remain in the system. SPUs, sharing a Bridge SPC form SPUs of a higher-level. The event graph manager SPU for example, controls the access and processing of the event

graphs. Every SPU that wants to access an event graph sends a request to the event graph manager SPU. The request passes through a reception SPC, which centralizes the request handling. The reception SPC is a Bridge SPC relative to the event graph manager SPU and each SPU that communicates with the event graph SPU.

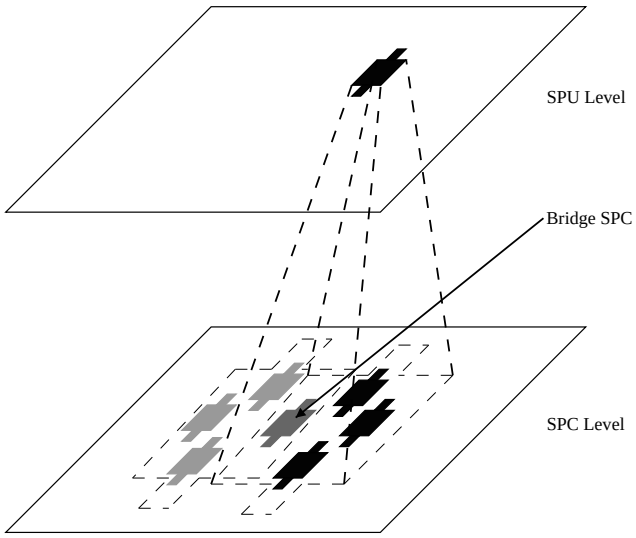


Figure 7: SPC and SPU level. SPUs share SPC on the SPC level.

Set of Interfaces. Each SPC registers with the infrastructure layer to make its service available. The infrastructure starts during the initialization phase, configures the SPUs and manages their life cycle. Each SPC is integrated into the communication structure and obtains services from this layer. SPCs and SPUs use each other's services and contribute to the overall system behavior. The plug-in approach of the MUSA architecture needs clearly defined interfaces between the different layers of the MUSA system and the SPCs themselves. There are three sets of interfaces. The first set is related to the interfaces of the system. Each SPC handles system aspects through this set of interfaces. Each SPC implements a service related interface, which form the second set of interfaces. A SPC that offers a service to other SPCs or SPUs has an interface available via which the service is obtainable. The third set of interfaces establishes connections between SPUs. This set of interfaces is a subset of the service-related interfaces of the SPCs, which constitute a SPU. Each SPC or SPU must integrate the set of architecture-related and service-related interfaces, to be able to participate in the MUSA system.

3.3 Service Controller SPU

The key idea of the MUSA architecture is that all devices share the same service logic. The service controller SPU controls the service. It mediates between the domain specific service and the presentation

specific concerns of the client UI. It operates on an event graph, which offers a high-level abstraction of the service/user interaction, completely independent of the service semantic. The fact that the service controller operates on an event graph makes it theoretically feasible to support completely different services at the same time within the same service controller. However, typically there is only one. The introduction of the event graph follows the idea of the reactive constraint graph introduced in [Ball *et al.* 2000] and the abstract depiction hierarchy presented in [Taylor *et al.* 1995]. The MUSA system is designed in an event-driven architecture, which is commonly used in UI environment [Wang 1998]. Services provided with the MUSA system support multiple UIs. This is why the event-graph suffers from the "least common denominator" problem. The difficulty consists of the support of a wide variety of possible UIs to interactive services. For example the graphical UI of a service intended for a desktop computer may be quite different to an UI that is appropriate for a mobile telephone with a very small display. The employed set of events is a subset of the events a wide range of UIs can implement. We found that a set of four types of events, namely navigation, notification, execution, and request event is sufficient for a wide range of interactive services. This set of design elements allows the development of services having a UI, which is rich enough to guarantee a seamless communication between the UI, and the service logic and to employ a wide range of devices with different UI. The introduction of the event-graph as a high-level abstraction of the service logic/user interaction allows rapid development of services independent of the UI. It was designed to ease the implementation and design of services. The traversal of the graph is driven by the reception of events from the UI. The graph traverses its nodes and processes the events. The graph consists of

1. Dialogs
2. Set of events. Each dialog contains a set of events.
3. Dialog segments. Each set of event logically consists of the dialog segments
   (a) Request,
   (b) Action
   (c) Response.

3.3.1 Dialog

The principal element of an interactive service is a dialog. A dialog is designed to represent a task or a subtask of a service. The user navigates from dialog to dialog while communicating with the service. Each dialog consists of a set of events, which is further subdivided into three distinct dialog segments. These three dialog segments reflect the course of a dialog: request, action, and response. A dialog node is designed from the point of view of the service logic. This is why the dialog segment request is not the client request,

but the request of the service for client event submission. Analogous to the dialog segment request, is the interpretation of the dialog segment action, which is not the user action, but the action on the part of the service. The three dialog segments are:

- **Request.** The client connects to the service. The service receives the client request and demands itself information from the client. The client submits an event or a set of events.
- **Action.** Based on the set of events and their associated event context, the service processes the events and returns a response.
- **Response.** The response is the request of a new triple of request, action and response.

Each dialog segment constitutes an element of a virtual UI. The virtual UI is finally mapped to a real UI. A dialog contains a set of combinations of the four events: navigation event, execution event, navigation event and notification event.

3.3.2 Navigation Event.

A service is realized as a sequence of dialogs (Figure 8). The set of navigation events abstracts the navigational structure of the service logic. It indicates to the service that the user requested to go to a specific dialog in the navigational structure of the service logic. This event navigates the user to a dialog that is associated with a specific part of the service logic, indicated by the event context. The navigation event can be processed in a visible and a non-visible mode.

- **Visible Navigation.** The visible navigation event is user-driven and part of the events which the UI prompts for user interaction during the request dialog segment. The user has to explicitly select the event to navigate to the next dialog.
- **Non-visible Navigation.** The non-visible navigation event is service-driven and navigates the UI to the target dialog without user intervention.

3.3.3 Notification Event.

The notification event models the content level of the service. It indicates to an instance of a UI, what notification or message the service requests to communicate to the user. This event transfers a message to the user. The notification event can have an execution event (c.f. Section 3.3.5) as its child-event. The execution event is embedded in the notification event to retrieve information from the service.

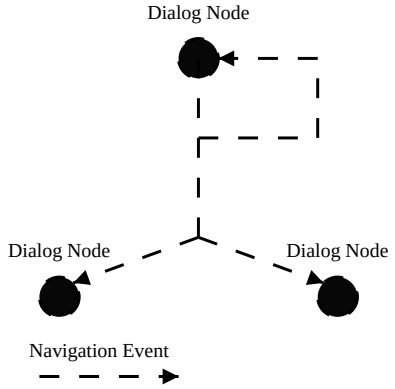


Figure 8: Outline of the concept of graph traversal. The navigation takes place as the user triggers a navigation event. The traversal occurs from a dialog node to another dialog node, or back to the same dialog node. The navigation can be automatic, i.e. non-visible. In this case, the user is not aware that the service has passed an intermediate dialog, since the navigation is immediate.

3.3.4 Request Event.

The request event represents the event that prompts the user for input, which the service requires for processing. The request event requests information from the user and verifies this data. The data is either accepted, or rejected. If the data is rejected the request event is evaluated to false and has evaluated unsuccessfully. Typically the request event is a child-event of the execution event (c.f. section 3.3.5). In general, information is requested in order to parameterize an execution event.

3.3.5 Execution Event.

The execution event comprises the behavioral aspects of the service logic. It links the UI with an action of the service logic. It communicates through the service adapter with the service logic and the service data. It changes the status of the service and manipulates the data. If the service's action needs to be parameterized, information is requested from the user by means of request events. The request events are child events and need to be evaluated in the scope of the execution event. The request events represent a constraint to their parent events and require to be successfully evaluated before a parent event is processed. If the service action returns data, this data is associated to the context of the event, just as the request event constitutes part of the context of the execution event. The execution event can be processed in two different modes.

- **Visible Execution Event.** In the visible mode, the execution event forms part of the UI and requests user interaction to be processed.

- **Non-visible Execution Event.** The non-visible execution event is automatically processed without user interaction. It is triggered and processed by the service. If the execution event requires parameterization and request events are part of the event sub-graph, the event cannot be processed in a non-visible mode.

3.3.6 Inter and Intra-Dialog Navigation

The navigational structure of the service logic is represented by the set of navigation events, which navigate between dialogs. The navigation between dialogs is the inter-dialog navigation. Inter-dialog navigation is either user-driven, in the case of a visible navigation event, or service-driven, if the navigation is non-visible. A dialog comprises request, user interaction and response and the intra-dialog navigation accounts for this part of the interactive communication between service and user. The intra-dialog communication within a specific dialog between request, action and response is done in response to event processing.

3.3.7 Event Evaluation.

An event consists of three sections, which are successively processed. Each section itself consists of a set of events.

- **Try-Section.** In the try-section the current event executes its associated action and its child-events, which constitute a constraint to the parent event. The try-section is successfully processed if every child event within this section evaluates successfully. If a child event evaluates to false, the try-section of the parent event evaluates unsuccessfully.

- **Satisfied-Section.** The satisfied-section is being executed, after the try-section of the current event has executed successfully.

- **Violated-Section.** If the try-section or the satisfied-section of the current event has executed unsuccessfully, the violated-section is processed.

After the typical cycle of intra-dialog navigation between request, user interaction and response has been accomplished, the inter-dialog navigation proceeds to the next dialog.

The concept of an event reflects the intra-dialog navigation structure. For example, the execution event contains a try, satisfied and violated-section. The try-section on the one hand and the satisfied and violated-section on the other hand correspond to the request and the response part of a dialog respectively. If the user navigates to a dialog node containing the execution event, the UI mapping of this event prompts for user interaction. If the try-section contains any request events, they prompt for information from the user as well. If the user submits the execution event to the service controller, the event is processed. If the

try-section evaluates successfully, the satisfied-section is processed, otherwise the violated-section. Figure 9 shows the structure of a dialog node and the set of events it contains. In this example, the dialog's set of events consists of a notification event, a navigation event and an execution event. The execution event is presented in more detail. It contains a try, satisfied and violated-section. The try section contains as child event a request event. The request event constrains the execution event and is prior evaluated.

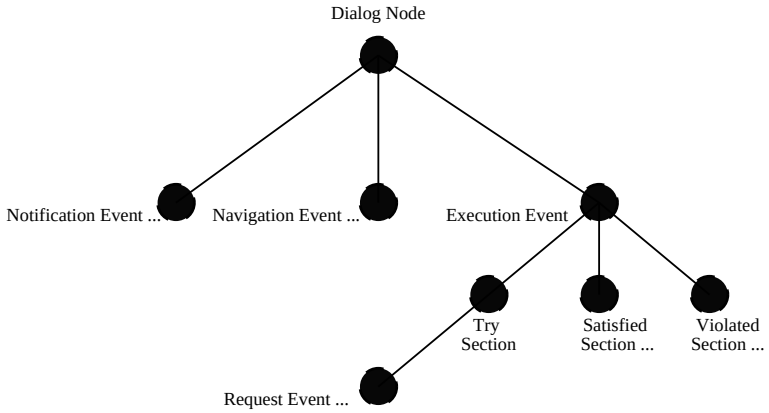


Figure 9: Dialog node with three events. The execution event is shown with try, satisfied and violated-section. The request event is a child event of the execution event.

3.4 MUSA enabled Application

Section 3.1 has illustrated the MUSA system by means of a mobile banking application. Figure 4(a) illustrates the login screen of the application on a mobile telephone. The corresponding dialog of the event-graph is shown in Figure 10. The dialog consists of a single visible execution event. The execution event's try-section contains several notification events and request events. The execution event validates the login and request events request authentication data from the user. The information, which the user has provided in each request event, is tested in the try-section of the request event. The content of this section has been omitted in Figure 10 to concentrate on the essential parts of the dialog design. The tests in the try-section have to be evaluated successfully to come to a positive authentication. Notification events precede request events to define what information is requested by means of the request events. The satisfied and violated-section are not shown in detail. The violated section contains an error message and a possibility to navigation back to the login screen. The satisfied-section contains an automatic navigation event, which navigates to the mobile banking menu screen.

The banking application can be accessed from different gadgets. The dialogs of the event-graph, like the dialog shown in Figure 10, are transformed in the transformer SPU into concrete UIs. The transformations are applied dependent on the user's device, through which the user accesses the service. A set

```
<Dialog name="welcomeDialog" alias="Welcome to MyBank" type="visible">
  <EventSet name="logIn" alias="Welcome to MyBank">
   <ExecutionEvent name="logInAction" alias="Anmelden" type="visible" declName="sBankAdapter">
    <TryEe>
     <NotificationEvent name="info1">
      Login
      Bank Sorting Code:
     </NotificationEvent>
     <RequestEvent name="blz" alias="Bank Sorting Code" value="" type="input">
      <TryRe> ...  </TryRe>
     </RequestEvent>
     <NotificationEvent name="info2">Account Number</NotificationEvent>
     <RequestEvent name="knr" alias="Account Number" value="" type="input">
      <TryRe> ...  </TryRe>
     </RequestEvent>
     <NotificationEvent name="info3">PIN</NotificationEvent>
     <RequestEvent name="pin" alias="PIN" value="" type="input">
      <TryRe> ...  </TryRe>
     </RequestEvent>
    <Satisfied> ...  </Satisfied>
    <Violated> ...  </Violated>
   </ExecutionEvent>
  </EventSet>
</Dialog>
```

Figure 10: The dialog that realizes the login UI. The dialog consists of an execution event, containing request events for user authentication. The execution event has attributes, indicating which application adapter to use and its visibility type. The satisfied and violated-sections for successful and unsuccessful processing of the execution event are not shown in detail.

of transformations is provided by the MUSA system. New transformations can easily be added by simply plugging in additional transformation SPCs into the transformer SPU. Existing transformer SPCs can be adapted through the modification of a set of configuration files.

4 PERSONALIZATION SPU: PERSONALIZING AN EVENT-GRAPH BASED SERVICE

The World Wide Web has established itself as a new type of information and service space. But there is a lack of systems, which can be adapted to individual user preferences. Web based services offer rarely the possibility for personalization and if they do, they typically offer personalization only for sophisticated graphical UI based services. The consequence is that most web-based services are completely impersonal. Every user sees the same service and has to adapt to the default service space. This causes a reduction of productivity for the user. Services rarely offer the user the possibility to

- Organize his/her service space,
- Increase productivity by submitting personal information, which the user is requested to fill in very often, like addresses, email addresses, names, etc.

The user needs the facilities to adapt the service to its personal needs for each device, as well as for each role in which he/she uses the device.

- The user accesses a service on different devices. According to the computing power and the visualization capacity of the device, the user will adjust how to use a specific service. The user will work with an email application differently on a desktop computer than on a PDA without keyboard. He/She will probably use the PDA to only read mail, but not to write lengthy messages. On the desktop computer he/she will use the complete functionality of the email application.
- The user might use the mail application for business and for private purpose. The menu or the principle address book will change subject to who is using the mail application, the user as the business person or as a private person.

A service should provide the potential to support different user profiles for different UIs and multiple profiles for the same UI. Figure 11 shows the three-dimensional personalization space. The same service can be used on different devices. For each device, there are different user profiles, accounting for the different usage of the service on different devices. For the same device there are different user profiles, accounting for the different role, in which a user accesses a service.

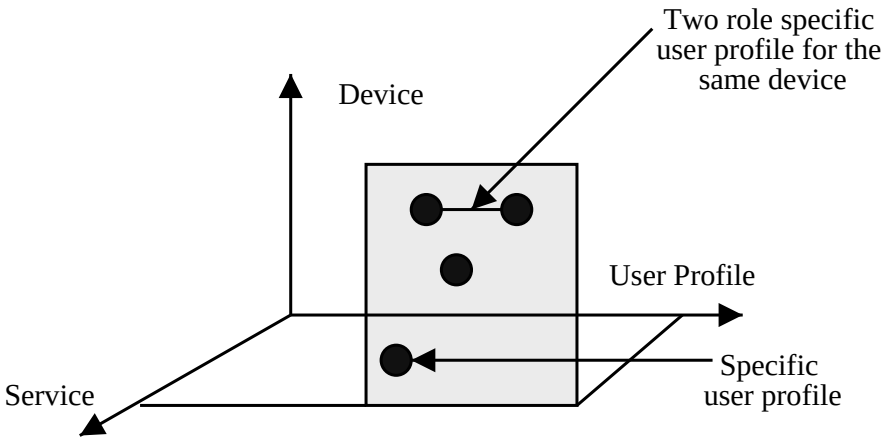


Figure 11: Three-dimensional personalization space. The same service can be used on different devices. For each device, there are different user profiles, accounting for the different usage of the service on different devices. For the same device there are different user profiles, accounting for the different role, in which a user accesses a service.

The integration of different user profiles for different devices forms a vital basis of the development of services, which use mixed channel communication. Mixed channel communication means the collaboration of different devices and/or users to accomplish an objective. With the MUSA system a part of a service could be adapted to run on a desktop computer, whereas another part would be optimized to run on a mobile telephone. For example a user could browse an online shop on a desktop computer. The payment would be conducted from a secure payment enabled mobile telephone. This would have the advantage that the user would not have to submit his/her credit card number.

One of the key features of the MUSA system is the fact that the event graph abstracts the implementation of the service. An additional feature is, that it is interpreted, instead of being compiled as in [Ball *et al.* 2000]. This allows the service, which is designed with the event graph, to be highly dynamic. The event graph can be modified without affecting the implementation of the service logic. The navigational structure and the content can be adapted to personal needs via a dedicated UI. The service logic does not need to undergo any modification and the user is able to personalize a dialog segment. Execution and navigation events can be removed to concentrate the layout of the client UI to the essential parts required on a specific device under a specific user role. These events can be added again under a different personalization configuration. In addition, events can be renamed, which is important on devices with small devices like mobile phones. Default values can be provided when the user is requested to provide information. The essential point is, that a service designed with the event graph automatically incorporates the personalization feature. A service designer does not have to consider this aspect, but can concentrate on developing the service.

Section 3.1 and 3.4 have illustrated the MUSA system by means of a mobile banking application example. Figure 4(b) shows the banking menu screen. It contains the menu items, "Personalization", "Statement of Account", and "Turnovers". The last two menu items correspond to navigation events in the event graph, which allow the navigation to their corresponding dialogs. The two navigation events are service specific, whereas the menu item "Personalization" has been added automatically by the MUSA system. It has been added, because the banking menu dialog has been tagged as a dialog on the event graph level, which can be personalized. A dialog can be personalized by simply adding a personalization tag to the dialog in the event graph, assuming the personalization SPU is plugged in the MUSA system. If the personalization SPU were missing in the MUSA system configuration the personalization feature would not be present. The personalization screen of the banking menu dialog in Figure 4(b) is shown in Figure 4(c). In this simple example the service specific navigation events can be renamed and removed from the screen.

5 COMPARISON TO OTHER WORK

Ongoing research projects offer different approaches to tackle the problems arising when computing moves from the desktop to a new pervasive computing devices. One straightforward approach is to adapt an existing service and its architecture in a way that it can be accessed from multiple devices. The advantage of this approach is that the service and its UI are optimized for each device. However, this approach requires the service to be rewritten several times to take the characteristics of each client device into account. This results in high redundancy and consistency checks of each version of the service. The administrative effort of this approach is prohibitive.

Most interactive Internet services were designed with a single UI in mind. If the service provider extends the service's range of client devices, the existing UI is adapted and the UI-elements are mapped to UI-elements of the new UI. In this approach the client uses the same service, regardless of the UI and device that accesses the service. However, the existing UI can be mapped satisfactorily to the new client's UI, only if the new client's set of UI elements is a subset of the original client's set. Kaasinen et al. pinpoint that the conversion of Web pages written in html to wml causes significant problems [Kaasinen *et al.* 2000].

Ball et al. introduce a reactive constraint graph to design and implement interactive services with multiple UI [Ball *et al.* 2000]. However, the reactive constraint graph, which incorporates the service logic and the content, is static and cannot therefore be modified without translating and recompiling the system. The MUSA system is inspired by this approach but radically differs from it in the realization of the service logic and UI implementation.

Oracle has developed an abstract device markup language, which is device independent [Oracle 2001]. The language is very simple and contains only 10 elements, due to the "least denominator problem". The

services developed through a specific tool can undergo personalization, using scripts to create dynamic services. However, the personalization of the services can only be done through a web browser, and not through the gadget, which is the target device for the personalization. Mobile modules are introduced to allow reuse of applications. Basically every application can be declared a mobile module, which can be reused through the use of a specific development tool.

IBM provides support to dynamically transcode HTML content to a variety of formats like WML, HDML, and iMode [IBM 2001]. This minimizes the need for multiple versions of the same Web site. However, the conversion from one format into another format cannot be done without loss of precision, if elements of the input format have no corresponding elements in the target format and cannot be mapped to other elements of that format or if they have no corresponding elements with exactly the same features. Elements offering no corresponding elements in the target data format are omitted and non-convertible elements are reduced to elements that can be transformed to equivalent functionality, i.e., their functional intent is preserved, although their functional presentation can be quite different. No personalization functionality is offered, since the design of a service is based on HTML, WML... i.e., languages dedicated exclusively to content presentation.

6 CONCLUDING REMARKS

This paper presents the key elements of the architecture of the MUSA system. Its design goals are flexibility, adaptability and reuse. The architecture is layered and its dynamics is based on an event driven approach. The MUSA system allows rapid development of services by decoupling the UI-related issues from the service logic. Additionally, it allows personalizing a service implemented as an event graph without incorporating this feature explicitly in the service implementation. This is especially important considering the growing popularity of service personalization especially on devices with reduced visualization capabilities.

REFERENCES

Abowd, G. (1999), "Software Engineering Issues for Ubiquitous Computing," *Proceedings of the 1999 International Conference on Software Engineering* , 75 – 84.

Ball, T., P. Danielson, L. Jagadeesan, R. Jagadeesan, K. Laeufer, P. Mataga, and K. Rehor (2000), "Sisl: Several Interfaces, Single Logic," *Internatinal Journal of Speech Technology 3*, 93–108.

Bass, L., P. Clemens, and R. Kazman (1998), *Sofware Architecture in Practice*, Addison Wesley, Reading, Mass.

Compaq (2001), "Project Mercury: Exploring the Future of Handheld Devices," http://www.crl.research.digital.com/projacts/mercury/.

Fontoura, M., W. Pree, and B. Rumpe (2001), *The UML Profile for Framework Architectures*, chapter Hints and Guidelines for the Framework Development and Adaptation Process, Addison Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of reusable object-oriented software*, Addison Wesley, Reading, Mass.

Garlan, D. (2000), "Software Architecture: A Roadmap," In *Conference on the Future of Software Engineering*, Ireland, pp. 91–101.

Garzotto, F., P. Paolini, and D. Schwabe (1993), "HDM - A Model-Based Approach to Hypertext Application Design," *ACM Transaction on Information Systems 11*, 1, 1–26.

IBM (2001), "IBM Websphere Transcoding Publisher, Version 3.5," http://www-4.ibm.com/software/webservers/transcoding/.

Kaasinen, E., M. Aaltonen, J. Kolari, S. Melakoski, and T. Laakko (2000), "Two Approaches to Bringing Internet Services to WAP Devices," http://www.www9.org/w9cdrom/228/228.html.

McCann, P. J. and G.-C. Roman (1998), "Compositional Programming Abstractions for Mobile Computing," *Software Engineering 24*, 2, 97–110.

Oracle (2001), "Oracle Mobile Online Studio, Developer's Guide," http://studio.oraclemobile.com/omp/site/Documentation/developers guide.htm.

Palm (2000), "Wireless Handhelds," http://www.palm.com/products/palmviix.

Pree, W. and K. Koskimies (1999), *Building Application Frameworks, Object Oriented Foundations of Framework Design*, chapter Framelets – Small is beautiful, Wiley Computer Publishing, p. 411 to 414, M. Fayad and D. Schmidt and R. Johnson (editors).

Roman, G., G. Picco, and A. Murphy (2000), "Software Engineering for Mobility: A Roadmap," .

Rossi, G., D. Schwabe, and A. Garrido (1999), "Designing Computational Hypermedia Applications," *Journal of Digital Information (JODI) 4*, 1.

Szyperski, C. (1998), *Component Software - Beyond Object-Oriented Programming*, Addison Wesley, Reading, Mass.

Taylor, R., K. A. Nies, G. A. Bolcer, C. A. Macfarlane, and K. M. Anderson (1995), "Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support," *ACM Transaction on Computer-Human Interaction 2*, 2, 105–144.

Ubinet (1999), "The Ubiquitous Internet Will be Wireless," *Computer 128*, 10, 126 – 127.

Wang, K. (1998), "An event driven model for dialogue systems," *Proceedings of the International Conference of Spoken Language Processing 2*, 393–396.

Weiser, M. (1991), "The computer of the 21st century," *Sci.Am 265*, 3, 94 – 104.

Weiser, M. (1993), "Some computer science issues in ubiquitous computing," *Commun. ACM 36*, 7, 75 –

84.

Winjstra, J. G. (2000), "Supporting Diversity with Component Frameworks as Architectural Elements," In *Proceedings of the 22nd International Conference of Software Engineering*, pp. 51 – 60.