# Tool-Supported Interpreter-Based User Interface Architecture for Ubiquitous Computing

Lars Braubach, Alexander Pokahr, Daniel Moldt, Andreas Bartelt, and Winfried Lamersdorf

Distributed Systems Group, Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{braubach, pokahr, moldt, bartelt, lamersd}@informatik.uni-hamburg.de
http://vsis-www.informatik.uni-hamburg.de/

**Abstract.** With the upcoming era of Ubiquitous Computing (UbiComp) new demands on software engineering will arise. Fundamental needs for constructing user interfaces (UIs) in the context of UbiComp were identified and the subsumed results of a survey with special focus on model based user interface development environments (MB-UIDEs) are presented in this paper. It can be stated, that none of the examined systems is suitable for all the needs. Therefore a new architecture based on the Arch model is proposed, that supports the special UbiComp requirements. This layered architecture provides the desired flexibility with respect to different implementation techniques and UI modalities. It was implemented in a user interface development environment called Vesuf. Its usability was approved within the Global Info project [20], where heterogeneous services had to be integrated in a web portal.

## 1 Introduction

As covered below, UbiComp applications differ inherently from conventional applications. In [8] were identified several special UI needs for this kind of systems, which will be presented in the following. First of all it can be stated that UbiComp applications are somewhat more complex than comparable conventional programs, because they have to cope with dynamical changes during runtime. Therefore it should be an important objective for a UI construction system to hide some of this complexity (simplification).

In [2] Banavar et al. explain that the application development for UbiComp has to be device independent, because a single application should be usable from distinct entry points, e.g. from a laptop, handheld or even phone. This implicates some advantages for the user such as synchronous data and familiar handling across different devices. To achieve this vision within a UI construction system it is necessary to provide mechanisms for cross-platform user interfaces. Therefore a clean separation between user interface and functional core is needed, as well as mechanisms for connecting the separated parts. Moreover it should be possible to create different interface modalities for an application (flexibility), and the

P. Forbrig et al. (Eds.): DSV-IS 2002, LNCS 2545, pp. 89–103, 2002.
© Springer-Verlag Berlin Heidelberg 2002

**Table 1.** Surveyed systems

| Frameworks | Suggested literature |
|---|---|
| MVC-Client | [30] |
| SanFrancisco | [34] |
| JWAM | [6] |
| MVP | [27] |
| Amulet | [23] |

| MB-UIDEs | Suggested literature |
|---|---|
| Janus/Jade | [1] |
| Mobi-D | [28] |
| FUSE | [22,5] |
| TRIDENT | [7] |
| TADEUS | [14] |
| Teallach | [15,17] |
| MASTERMIND | [9,31] |
| BC-Prototyper | [35] |

system itself should be open for the integration of new interface modalities and implementation techniques (extensibility).

Banavar et al. further point out that the UbiComp paradigm will lead to substantial changes on how users perceive applications. They will understand the application as a composition of services, which takes into account the current context of use, e.g. the location, time or weather. To be able to support interfaces for this new kind of applications it is necessary to address the dynamic adaptation and composition of UIs.

The next section takes a look into what existing tools offer for the special UbiComp needs. In Sect. 3 a new model-based architecture and a concrete system for better accomplishing these goals are presented. Thereafter system details are introduced in Sect. 4, which help to achieve to some degree the UbiComp demands mentioned above. In Sect. 5 as an example a metadictionary service is described. Finally in Sect. 6 follows a summary of the results and an outlook.

## 2 Survey Subsumption

In search of a suitable system for constructing UIs of UbiComp applications several different categories of tools have been researched [8]. The two most promising approaches are frameworks and model-based systems, which were further investigated by typical representatives (see Table 1). The consolidated results of [8] as shown in Table 2 are presented next.

The goal of a *simplified* UI construction process is currently achieved by frameworks only. Because of the low abstraction level framework implementations offer a flat learning curve for the developers and additionally whitebox frameworks allow easy programmatic extension. MB-UIDEs suffer from the lack of well established standards in the field of some partial models for UIs, especially for presentation and dialogue control facets.

It is stated that the clean *separation* of UI and functional core is achieved by frameworks as well as by model-based systems. While most frameworks utilize agent-based architectures like MVC [10] and PAC [11] the MB-UIDEs accomplish *separation* in a natural way through their models. The connection between these two components, relevant for an executable interface, is established by program-

**Table 2.** Subsumed UbiComp-characteristics of the system categories [8]

| | Simplification | Separation Connection | Extensibility UI / Connection | UI Flexibility | Adaptation | Composition |
|---|---|---|---|---|---|---|
| Frameworks | x | x/x | o/(x) | o | o | o |
| Model–Based Systems | o | x/(x) | (x)/(x) | (x) | (x) | o |

x : Nearly all systems (x): Few systems o: None of the tested systems have the property

ming within the framework context, whereas model-based systems use descriptive techniques to link the interface to the functional core. Some of the tested MB-UIDEs like TRIDENT or FUSE do not establish the *connection* between UI and functional core, leading to substantial additional effort for manually linking the parts.

Frameworks do not address the *extensibility* with regard to various UI modalities, as there are different frameworks for different purposes, e.g. for web-services or for interactive systems. Regarding the extensibility with respect to further implementation techniques, some frameworks address concepts for integrating database systems. Model-based systems are conceptually qualified for both extensibility issues. In practice only few of the tested systems exploit the potential of the model-based approach for supporting different interface modalities, or regard extension mechanisms to integrate more than one implementation technique (like Janus and Teallach).

When considering the *flexibility* of frameworks it is obvious that these types of systems are not able to support easy mechanisms for changing interface modalities, because the glue between UI and functional core has to be programmed. Also few MB-UIDEs utilize their declarativeness for flexibility (except TADEUS). The same is true for *adaptation* and *composition* aspects, which should be considered when building constructing systems for UbiComp. MASTERMIND is the only system, which addresses adaptation with respect to display properties. No tested framework even tries to cope with one of these aspects.

From this comparison it can be concluded, that the non-declarative approaches (frameworks) are not well suited for UbiComp UI construction systems, because they do not offer enough flexibility and are settled on a too low abstraction level. Model-based systems overcome these conceptual problems and possess all features which make them appear fully qualified for UbiComp needs. Within this group the interpreter-based systems (see [12]) seem to be the most promising, because they are able to handle the UbiComp needs with respect to dynamic changes of the UI during runtime. Before utilizing model-based systems for UbiComp some problems have to be solved. One important aspect is the lack of established UI standards. Further on the support of various interface modalities must be improved, and the research in the field of adaptation and composition must be carried on.

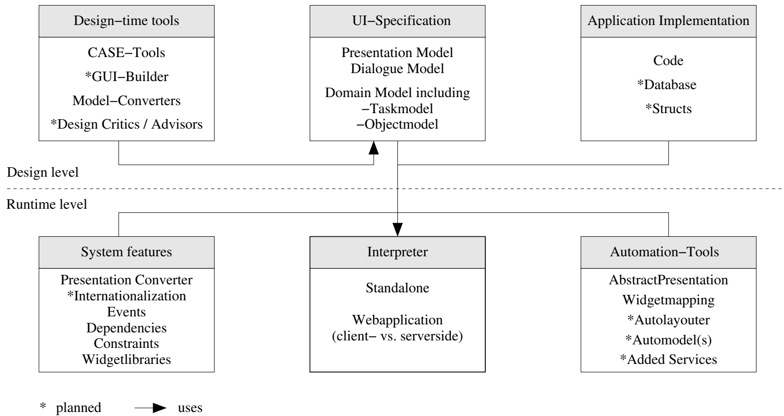


**Fig. 1.** Vesuf system components

## 3 Vesuf System

As a starting point for the design of a new system, a wide research in the field of tool-categories, techniques and architectures and their applicability with regard to UbiComp has been carried out [8], and a vision of a UbiComp development environment has been conceived [26]. This section will present the Vesuf system, a first step towards realizing the vision.

### 3.1 Vesuf Overview

As pointed out in the last section, interpreter-based MB-UIDEs are potentially well suited for all UbiComp requirements. Thus the Vesuf system consists of the components shown in Fig. 1. To realize an application within the Vesuf system the functional core can be implemented system independent (*application implementation*). This means that no system specific code-intrusions are necessary. In addition the different submodels for the *UI-specification* have to be created. To assist the developers in this process several tools can be used (*design-time tools*). At runtime the model-information will be evaluated by the *interpreter* which utilizes different *automation tools.* It constructs an executable UI by using further *system features* which are presented in detail in Sect. 4.

There is no fixed methodology to follow when developing applications with Vesuf and the models can be defined in an independent manner. The Vesuf system is designed to support what in this paper is called “slinky automation”. This allows to start the development process with minimal specification effort, and utilize the automation tools to a high degree, in order to have executable interfaces in early design stages. During the further development the model specifications will become more elaborated, and therefore the need for automation will decrease, while the UI quality increases (see [32]). With this approach inspired by HUMANOID [33], rapid prototyping is possible.

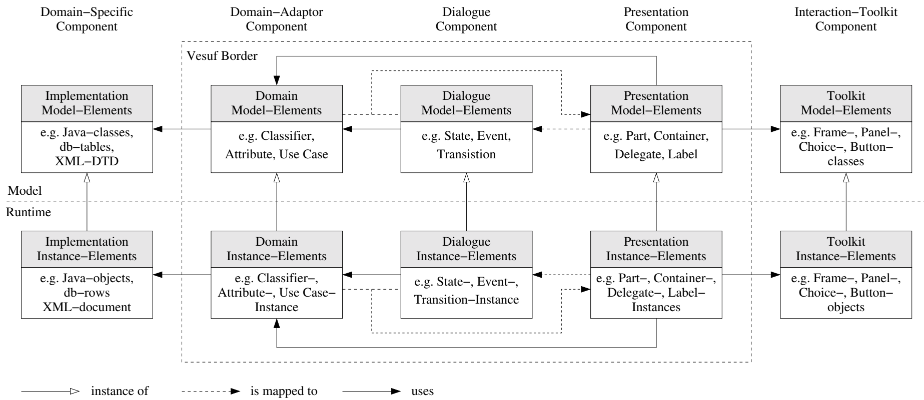


**Fig. 2.** Vesuf architecture

To be of use in a broad UbiComp context the system offers two different interpreter-modes. Active interpreters such as the GUIRunner and the VesufApplet are suitable for interactive applications and manage the transition between views by opening and closing application windows. Passive interpreters like the ApplicationServlet are mediator components which pass on all collected changes from the UI to the functional core and backwards.

### 3.2 Runtime Architecture

The layered architecture of the Vesuf system is pictured in Fig. 2. Primarily it must be stressed that a strict separation in so called model and instance elements is established in all layers. In [21] Kent et al. propose to introduce this separation as a foundation of the UML metamodel. Model elements are used to describe the various models, whereas instance elements represent concrete occurrences of the model elements at runtime.

Furthermore the horizontal partitioning in five separate components is conceived after the paragon of the Arch model [4]. The corresponding names of the Arch model components are denoted above each Vesuf layer. The Arch concept assures the independence from the implementation- and toolkit-specific side through the introduction of two adaptor components. In the domain-adaptor component all information about tasks and domain entities is encapsulated, while in the presentation component abstract and concrete display information is held. A central dialogue component is responsible for managing the global state of the UI.

In the following the models and their connections will be sketched. The domain-adaptor component is specified by a composite domain model, which consists of tasks and domain objects. This domain model is adopted from the UML metamodel and use cases are used for describing task behaviour.[1]

[1] It is planned to use the ConcurTaskTrees [25] as foundation for the task model in a subsequent release.

The dialogue layer contains a dialogue model, which currently is built upon the UML statechart semantics. Each application view is represented by one state. For concurrency purposes the concept of control flows is established. Each control flow is an autonomous state machine, which controls one part of the UI. To determine what will be displayed to the user, states are connected to domain entities (e.g. tasks or objects) via paths.

To map tuples of domain element and dialogue state to presentation elements a flexible mapping is used. The presentation metamodel extends the UML and is based upon functional roles. A presentation element is linked to a toolkit element which is responsible for its appearance, and to a domain element for realizing the information flow between implementation and user.

## 4 System Details

An overview of the proposed architecture has been presented, but it is yet to show, how it fulfills the identified requirements. Ubiquitous Computing demands the clean separation of device-specific and device-independent parts of an application. The Arch model provides five components, that can be abstractly developed and reused across a range of different environments for any application, or be specialized, e.g., to enable the use of device-specific features, according to the suitability in the application context. The border between context-, device- and user-independent and -specific parts can be flexibly shifted across the components of the Arch model. For easy integration of the components, a combination of several concepts is proposed.

These concepts heavily rely on the interpreter-style of the architecture, and therefore have not been found in recent generator-style model based systems (like Mobi-D, TADEUS, Janus, etc.). They have been inspired by other interpreter-style model-based systems such as ITS [36], HUMANOID and framework approaches like Amulet. In the following it is stated how these concepts aid in fulfilling the requirements presented in Sect. 1.

### 4.1 Domain-Adaptor Layer

Most model based systems try to solve the problem, how to connect the UI to the application implementation. In Vesuf the domain-adaptor layer conceals the implementation layer, and represents the application functionality from the viewpoint of the user interface.

With a flexible mechanism called "implementation accessor", any type of implementation technology (i.e. legacy system) can be integrated into the Vesuf environment. Currently the system has full support for Java implementations and prototypical support for implementations in relational databases.

The use of the UML metamodel as foundation of the domain-adaptor layer establishes a unified view on top of implementation details, and allows the specification of user interface related meta information (e.g. constraints) as first class objects together with the domain entities.

### 4.2 Paths

Puerta and Eisenstein [29] describe the mapping problem between what they call abstract (task, object) and concrete (dialogue, presentation) elements. They regard the solution to this problem as "essential for the construction of model-based systems". In Vesuf the connection between abstract and concrete elements is established using paths.

The Vesuf path language allows to specify navigational paths across all elements of the metamodel (e.g. classifiers, attributes, constraints). It is comparable to XPath [37] which provides navigational access to different node types of XML-documents, such as element, attribute and text. XPath introduces the concept of axes which specify DOM associations to follow (e.g. child, attribute, descendant).

The associations in the UML metamodel (e.g. attributes of classifiers) provide the axes of the Vesuf path language. For example to refer to the value of an attribute (starting from a point object), one would write Point.<attribute>X.<value>, or refer to the constraints of the attribute by writing Point.<attribute>X.<constraints>. Note that the identifiers in angle brackets are the axes that denote references of the UML metamodel, while the other identifiers (e.g. "Point") denote elements in the domain model of the application.

To be evaluated, a path is instantiated with an instance of the starting element (e.g. point). As the references in the object graph may change, the endpoint element of a path instance may also change. Paths hide the problems of dynamical changes, because they provide a static way to refer to dynamically changing elements. For example a presentation element uses a path as reference to the domain element it displays. While the displayed element may change over time (as described in Sect. 4.4), the path will always be the same.

Although paths are evaluated at runtime they are statically typed, and their correctness can be checked against the domain model at design time. Since paths can be used to navigate across all different models (object, task, dialogue and presentation model), they provide the glue, to stick together the different components of an application. Therefore most of the higher level concepts rely on paths.

### 4.3 Extended Constraint Semantics

In UML a constraint is a semantic condition or restriction expressed in text, represented in the metamodel by a boolean expression on an associated model element [24]. In Vesuf constraints are used to stipulate possible user interactions and valid user input. As in the Seeheim model [16] this places the validation of user input into the application interface layer and not into the presentation layer. This facilitates the reuse of input validation for all interface modalities.

The semantics of constraints are extended to allow the specification of supplementary information, that is utilized by presentation elements. For this purpose constraints include additional properties with special meanings. The additional

**Fig. 3.** Use of constraints for presentation elements

properties are specified as constant literals, or as paths referring to domain attributes or operations, which are evaluated dynamically at runtime. Since constraints can be realized as operations in the implementation layer, tools can be used, to generate constraint implementations, e.g., from OCL-specifications. Currently Vesuf defines five different types of constraint properties which may be used independently or together. These are described next:

**valid** specifies the boolean expression to validate user interaction (e.g. for operations or navigational events) or input (for attribute and parameter values). This represents the standard UML semantics of constraints.

**check** specifies via a path an operation, that may throw an exception, when the constraint is not valid. The exception object can include additional information, why the constraint is invalid, e.g. a text message as in Fig. 3b.

**values** specifies the set of possible values, e.g., for an attribute or parameter. The set of possible values can for example be used by presentation elements to create radio buttons or to fill in lists or drop-down boxes (see Fig. 3a).

**range** alternatively to a set of possible values a range can be specified. All Vesuf built-in data types (e.g. integer, float, date) support this. Range handling for application specific data types can be added (e.g. for IP-address). The range constraint can be used to handle interaction with scrollbars (see Fig. 3c).

**active** For dynamically prohibiting access to certain elements (e.g. attributes and operations) of individual objects in the domain-adaptor layer, the active constraint is used. It causes enabling and disabling of interaction elements and therefore provides a way for realizing intra-dialogue behaviour in the domain-adaptor layer, that is "inherited" (i.e. mirrored) by all specialized interface modalities.

When specified with paths, constraint properties are evaluated dynamically and internal changes are propagated by events as described in Sect. 4.4 and exemplified in Sect. 5. Besides the predefined types, custom constraint properties can be specified and then be used by the presentation layer.

The use of constraints enforces a certain level of usability, because they provide meaningful error messages. When used in conjunction with widget-mapping

techniques, appropriate interaction elements are automatically selected based on the type of constraints specified for domain-adaptor layer elements.

### 4.4 Events and Dependencies

In Vesuf an event dispatcher component manages the collection, generation and multicasting of events. Events are initiated by instance elements in the domain layer (e.g. value of an attribute changed) or dialogue layer (e.g. a state change in a dialogue state machine). Event handlers can be registered on any type of instance element, and are used to couple loosely elements from different models. Besides propagating events to the appropriate handlers, the dispatcher manages the generation of dependent events. With dependencies the need for explicit event handlers in the presentation layer, reacting to changes in the other layers, is reduced. The system manages two types of dependencies, as described next.

Dependencies can be specified explicitly in the domain model using the UML dependency element. It is augmented by a tagged value, that specifies a path from the client to the supplier element of the dependency. The path enables the system to determine at runtime the supplier instance elements, that participate in any dependencies, and take the appropriate actions, when these supplier elements initiate events. An example for a dependency that has to be specified explicitly is the area attribute of a rectangle object, that depends on the values of the width and height attributes. When the width or the height value is changed, the dependency will cause an update of any presentation elements displaying the area attribute.

The second type of dependency arises from the use of paths for specifying properties of elements. When a section of a path changes, the endpoint of the path, and therefore the element that specifies the path (e.g. a view) also changes. When for example in a circle object a new center point is set, presentation elements displaying the x and y attributes of the old center point object are automatically adjusted to refer to the new center point. This is handled by Vesuf internally, and events for any dependent elements are automatically generated and published in the next event multicast (together with the initial event, that triggered the dependency).

### 4.5 Presentation Metamodel Based on Functional Roles

Since the behavioural aspects of the UI are captured in the domain-adaptor (using constraints and dependencies) and the dialogue layer, the Vesuf system features a very lightweight presentation layer, thus supporting the flexibility and extensibility requirements. The UML metamodel is extended with new elements rooted in an element called *part*. Besides this generic interface element four different types of elements are introduced (see Fig. 4). The system utilizes UIML [18] for specifying presentation models, because of its genericity and tailorability to specific environments.

The motivation behind the lightweight approach is, that the presentation elements only provide the glue between the concrete interaction elements in the

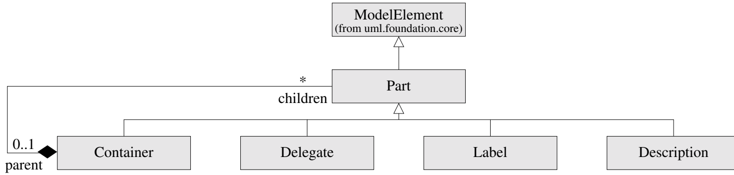


**Fig. 4.** Presentation metamodel

toolkit specific layer, and the application specific elements in the domain-layer. Therefore, in contrast to other proposals such as UML*i* [13], the elements of the presentation metamodel are not classified by their specific interaction capability (e.g. input, output, ...) but rather by the intention behind the element, i.e., the functional aspect of the connection between a toolkit element and a domain element. Three basic functional roles of atomic UI elements have been identified: *Delegates*, *labels* and *descriptions*. These elements are atomic in the sense, that they do not contain other elements.

For interaction elements, the concept of UI-delegates as self-contained representatives of domain elements, proposed by Holub in [19], is adopted. *Delegates* are the most important parts of the UI, as they enable the user to interact with elements in the other layers. In Vesuf, the actual domain-adaptor and dialogue layer elements, that are represented by a *delegate* are referenced via paths, which can be resolved at runtime to yield the corresponding instance elements. *Delegates* mediate, e.g., between attributes in the domain-adaptor layer and textfields in the toolkit layer.

The other two basic part-types (*label* and *description*) are not used for interaction, but to provide structural and usage information to the user. *Labels* are designations of elements in the other layers, usually placed near *delegates*, to designate which domain-adaptor or dialogue layer elements the *delegate* refers to. In the toolkit layer, *labels* are realized by texts, icon images or characteristic sounds. *Description* elements provide usage information related to elements in the domain-adaptor or dialogue layer, and can be used to provide context sensitive help (e.g. as tooltips). In addition to description texts these elements use information provided by constraints to inform the user about the current state of the interaction (e.g. invalid input).

To organize presentation elements in groups, *container* elements are used. They recursively aggregate the atomic presentation elements (delegates, labels and descriptions) that are to be displayed as a presentation unit. The container hierarchy is usually, but not necessarily, reflected by similar structures in the toolkit layer.

The lightweight presentation metamodel leads to simple, and easy to maintain interface descriptions. Furthermore, it allows the system to be easily extended by new interface modalities.

### 4.6 Interoperation of the Concepts

It has been shown, how the aforementioned concepts aid in fulfilling the requirements posed by UbiComp. The domain-adaptor component is the backbone of the architecture and features a runtime environment which manages model and instance elements with automatic handling of constraints, events and dependencies. Paths are defined on top of the modelled domain structure, to establish the connection with the other layers. The presentation elements use the potentially dynamic information of the domain-adaptor layer elements, to extract the properties of their widgets.

The dependency mechanism allows for intra dialogue control (e.g. en- and disabling of buttons) to be specified abstractly in the domain model (e.g. allowed parameter values for method invocations). The event mechanism automatically updates widgets that are dependent in this way. This mechanism of inheriting behaviour from the domain-adaptor facilitates robust and consistent behaviour in all interface modalities, and avoids redundancy in the different presentation models of an application.

## 5 Example

To prove the practical utilizability of the Vesuf system, it has been applied in the context of the GlobalInfo project [20]. It was used to integrate several services into the PublicationPORTAL [3]. One of these services is the metadictionary service that allows to query several online dictionary web sites (Fig. 5). The realization and integration of the services is described in detail in [8].

Using the example of the metadictionary Java application (see Fig. 5b), it is shown, how the concepts allow behaviour defined in the domain-adaptor layer to be mirrored in the presentation layer. The first example is the invocation of the *Translate* operation that uses the constraint / dependency mechanism. The operation is represented in the presentation layer using a button delegate. Choice delegates are used to represent the *To* and *From* parameters of the operation. These two delegates are self contained and not connected to each other. Nevertheless, when the user supplies parameter values, that are valid on their own but invalid in combination (i.e. a translation not supported by any dictionary), the button representing the operation will be disabled, since an appropriate constraint is specified in the domain model. This behaviour will appear in all presentations that support dynamic en- and disabling of operation delegates. In static presentations (e.g. web forms as in Fig. 5d) the user will be able to invoke operations with invalid parameter combinations, and will subsequently be presented an error page describing the constraint violation.

The second example utilizes the dynamic path concept. Consider the lower half of the metadictionary window in Fig. 5b, where a left hand side list box allows for selection of a result set, which is then presented in detail at the right hand side. This is realized in Vesuf by just using a path to the same attribute denoting the selected result set for both delegates. When the attribute value

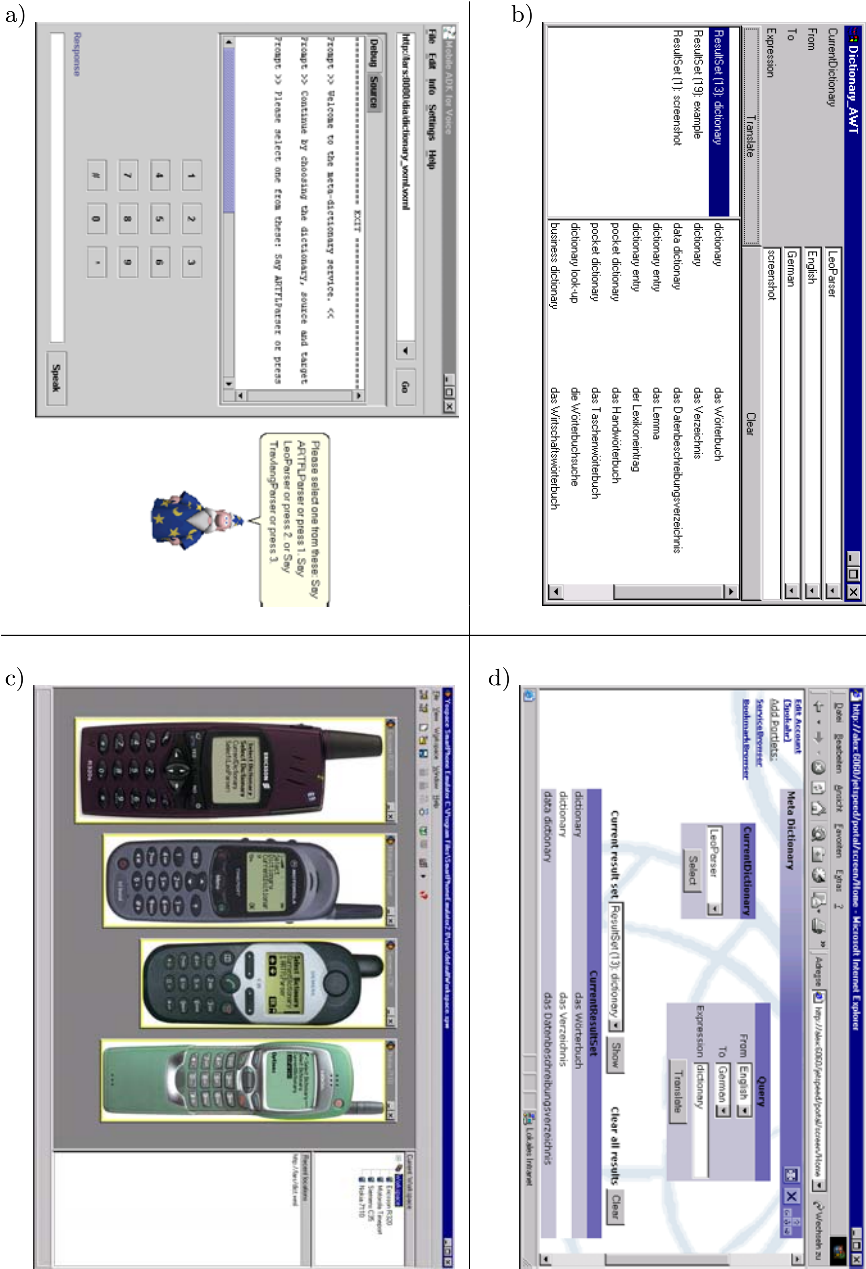


**Fig. 5.** Screenshots of the metadictionary service: Executed by the Motorola Mobile ADK for Voice (a), as Java application (b), displayed in the Yospace SmartPhone emulator (c), embedded in the PublicationPORTAL (d)

represented by the list box is changed, all interaction elements in the detail panel will be notified that a new result set has been selected and will therefore update their presentation appropriately. The same behaviour is exhibited in the web-portlet (Fig. 5d), where the current result set can be selected with the *Show* button and is subsequently presented at the bottom of the page.

## 6 Conclusion and Outlook

In this paper six major goals for UI construction tools in the context of UbiComp were presented. The proposed model-based architecture has been developed to address the arising needs. It is now shortly summarized how the architecture and the system characteristics help to fulfill these requirements.

*Simplification* of the development process is addressed by using standards to a high degree. The system is based on UML wherever applicable (domain-, task- and dialogue model) and uses UIML as a standard notation for the *simple* functional role based presentation model. Furthermore the development process is *simplified* by using automation tools and applying the slinky automation idea.

The architecture enforces the *separation* of the UI in five components according to the Arch model and introduces a generic mechanism (paths) to *connect* elements of these components at runtime. The *extensibility* with respect to new UI modalities is supported in a natural way by the underlying architecture and the usage of the delegate concept. To allow simple *extensions* with regard to different implementation techniques the implementation accessor concept is introduced. *Flexibility* with regard to UIs is achieved by applying constraints and dependencies. They relieve the presentation components from complex responsibilities such as input validation and intra dialogue behaviour. The resulting lightweightness of presentation components was one of the main goals with respect to UbiComp, featuring a minimum of redundancy between different UI specifications and a maximum of reusability of information rich domain models.

For *adaptation* and *composition* a sound foundation is set by the architectural separation of model and runtime layer and the interpreter-style is well suited to react to dynamical changes imposed by these demands.

Further research within the Vesuf project[2] will cover reusability aspects, especially the construction of UIs in a “LEGO” like manner. This becomes possible by the removal of all behaviour from the presentation layer and allowing UI components (LEGO bricks) to be placed on the domain-adaptor layer (LEGO ground plane). The connections between the bricks and the plane are established with paths and therefore allow to build up very complex interface elements from simpler ones in a declarative way.

## References

[1] H. Balzert. From OOA to GUIs – the JANUS System. In *Proceedings of IFIP INTERACT'95: Human-Computer Interaction*, pages 319–324, 1995.

[2] The Vesuf project is available at: `http://vesuf.sourceforge.net`

[2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. An Application Model for Pervasive Computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM-00)*, pages 266–274, N. Y., August 6–11 2000. ACM Press.
[3] A. Bartelt, D. Faensen, L. Faulstich, E. Schallehn, and C. Zirpins. Building Infrastructures for Digital Libraries. In *DELOS Workshop on Interoperability in Digital Libraries*, volume No. 01/W06. ERCIM Workshop Proceedings, 9 2001.
[4] L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szczur. A Metamodel for the Runtime Architecture of an Interactive System. *ACM SIGCHI Bulletin*, 24(1):32–37, 1992.
[5] B. Bauer. Generating User Interfaces from Formal Specifications of the Application. In F. Bodart and J. Vanderdonckt, editors, *Proceedings of DSV-IS'96*. Eurographics, June 1996.
[6] W.-G. Bleek, G. Gryczan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, and H. Züllighoven. Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen. *OBJEKTSpektrum*, February 1999.
[7] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, and J. Vanderdonckt. A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype. In F. Paterno, editor, *Proceedings of DSV-IS'94*, pages 25–39. Eurographics, June 1994.
[8] L. Braubach and A. Pokahr. Vesuf, eine modellbasierte User Interface Entwicklungsumgebung für das Ubiquitous Computing, vorgestellt anhand der Fallstudie PublicationPORTAL. Master's thesis, Universität Hamburg, 2001.
[9] T. Browne, D. Davila, S. Rugaber, and K. Stirewalt. Using Declarative Descriptions to Model User Interfaces with MASTERMIND. In F. Paterno and P. Palanque, editors, *Formal Methods in Human Computer Interaction*. Springer, 1997.
[10] S. Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller(MVC).
`http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`, 1992.
[11] J. Coutaz. PAC: An Object Oriented Model for Implementing User Interfaces. *ACM SIGCHI Bulletin*, 19(2):37–41, 1987.
[12] P. P. da Silva. User Interface Declarative Models and Development Environments: A Survey. In P. Palanque and F. Paterno, editors, *Proceedings of DSV-IS'2000*, pages 207–226. Springer, 2001.
[13] P. P. da Silva and N. W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 117–132. Springer, 2000.
[14] T. Elwert and E. Schlungbaum. Modelling and generation of graphical user interfaces in the TADEUS approach. In P. Palanque and R. Bastide, editors, *Proceedings of DSV-IS'95*, Eurographics, pages 193–208, Wien, 1995. Springer.
[15] P. Gray, R. Cooper, J. Kennedy, P. Barclay, and T. Griffiths. A Lightweight Presentation Model for Database User Interfaces. In *Proceedings of ERCIM'98*. ERCIM, 1998.
[16] M. Green. Report on Dialogue Specification Tools. In G. E. Pfaff, editor, *User Interface Management Systems: Proceedings of the Seeheim Workshop*, pages 9–20, Berlin, 1985. Springer.
[17] T. Griffiths, J. McKirdy, N. Paton, J. Kennedy, R. Cooper, B. Barclay, C. Goble, P. Gray, M. Smyth, A. West, and A. Dinn. An Open Model-Based Interface Development System: The Teallach Approach. In P. Markopoulos and P. Johnson, editors, *Proceedings of DSV-IS'98*, pages 32–49. Eurographics, June 1998.

[18] Harmonia Inc. *User Interface Markup Language Specification, version 2.0a*, 2000.
[19] A. Holub. Building user interfaces for object-oriented systems, Part 2: The visual-proxy architecture. *JavaWorld*, September 1999.
[20] Global Info. Globale Elektronische und Multimediale Informationssysteme für Naturwissenschaft und Technik des bmb+f. Bundesministerium für Bildung und Forschung (bmb+f), `http://www.global-info.org`, 2001.
[21] S. Kent, A. Evans, and B. Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *ECOOP'99 Workshop Reader*, pages 33–56. Springer, 1999.
[22] F. Lonczewski and S. Schreiber. Generating User Interfaces with the FUSE-System. Technical Report TUM-Info-9612, TU-München, 1996.
[23] B. Myers, R. McDaniel, and R. Miller. The Amulet Prototype-Instance Framework. In M. Fayad and D. Schmidt, editors, *Object-Oriented Application Frameworks*. Wiley & Sons, 1999.
[24] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, September 2001.
[25] F. Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer, 1999.
[26] A. Pokahr, L. Braubach, A. Bartelt, D. Moldt, and W. Lamersdorf. Vesuf, eine modellbasierte User Interface Entwicklungsumgebung für das Ubiquitous Computing. In H. Oberquelle, editor, *Mensch & Computer 2002*. Teubner, September 2002. To appear.
[27] M. Potel. Model-View-Presenter. The Taligent Programming Model for C++ and Java. `http://www-106.ibm.com/developerworks/library/mvp.html`, 1996.
[28] A. R. Puerta. A Model-Based Interface Development Environment. *IEEE Software*, 14(4):40–47, July/August 1997.
[29] A. R. Puerta and J. Eisenstein. Towards a General Computational Framework for Model-Based Interface Development Systems. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces*, pages 171–178, 1999.
[30] R. Sanderson. MVC-Client: Putting Model-View-Controller to work. `http://www.fourbit.com/resources/papers.shtml`, 1999.
[31] P. Szekely. Declarative interface models for user interface construction tools : The MASTERMIND approach. In L. Bass and C. Unger, editors, *Engineering for Human-Computer Interaction*. Chapman & Hall, 1996.
[32] P. Szekely. Retrospective and Challenges for Model-Based Interface Development. In F. Bodart and J. Vanderdonckt, editors, *Proceedings of DSV-IS'96*, Eurographics, pages 1–27, Wien, 1996. Springer.
[33] P. Szekely, P. Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The Humanoid Model of Interface Design. In *CHI*, pages 507–515, May 1992.
[34] P. Tamminga, D. Faidherbe, L. Misciagna, and F. Yuliani. SanFrancisco GUI Framework: A Primer. `http://www.ibm.com/Java/SanFrancisco/`, 1999.
[35] H. van Emde Boas-Lubsen. Business Component Prototyper for SanFrancisco: An experiment in architecture for application development tools. *IBM Systems Journal*, 39(2):248–266, February 2000.
[36] C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: A Tool for Rapidly Developing Interactive Applications. *ACM Transactions on Information Systems*, 8(3):204–236, July 1990.
[37] World Wide Web Consortium (W3C). *XML Path Language (XPath), version 1.0*, November 1999.