Available online at www.sciencedirect.com
ScienceDirect
Computers & Graphics 30 (2006) 692–701
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Model-based user interface adaptation

Erik G. Nilsson*, Jacqueline Floch, Svein Hallsteinsen, Erlend Stav

SINTEF ICT, 7465 Trondheim, Norway

Abstract

Exploiting contextual information in user interfaces (UIs) on mobile equipment is an important means for enhancing the user experience on such devices. Implementing flexible and adaptive UIs that are multitasking and possibly exploiting multiple modalities requires expensive application-specific solutions, and reuse is difficult or impossible. To make it viable to implement adaptive UIs for a broader range of applications, there is both a need for new architecture and middleware, and ways of constructing applications. In this paper, we show how a combination of a patterns-based modelling language using compound UI components and mapping rules as building blocks, and a generic adaptive architecture based on components with ports and utility functions for finding the optimal configuration in a given situation facilitates implementation of applications with adaptive UIs. After an introduction discussing special needs for mobile UIs and related work, we briefly present our modelling approach, and the adaptive middleware architecture. With this as a background, we investigate how the combination of these facilitates model-based UI adaptation. Finally, we present some more general principles, illustrated by two examples, for how model-based approaches may be used for visual adaptation of UIs.

Keywords: Model-based interface design; Personalization and customization of interfaces; Patterns-based approaches; Adaptive architecture

1. Introduction

With the increasing mobility and pervasiveness of computing and communication technology, more and more software systems are used by people moving around. Limited screen size and interaction mechanisms on mobile equipment make it challenging to design user interfaces (UIs) on this type of equipment [1]. Compared to a stationary user, the context of a user exploiting a mobile solution changes more often [1,2]. The context changes are multidimensional, sometimes rapid, and comprise position, light, sound, network connectivity, and possibly biometrics [3]. For example, communication bandwidth changes dynamically in wireless communication networks and power is a scarce resource on battery-powered devices when outlet power is not available.

An important challenge when designing mobile UIs is to exploit knowledge about these changes in the user's context, and to use this knowledge to enhance the user experience [4,5]. In many cases, exploiting the context makes it possible to simplify the UI of a mobile application compared to a corresponding stationary one because much of the functionality needed to filter, search for and enter information may be omitted or simplified (because it is given by the user's context) [1]. To cater for this, the UIs on such solutions need to be adaptive in many cases. Furthermore, UI preferences change when on the move, because light and noise conditions change, or because hands and eyes are occupied elsewhere. Dynamic adaptation is required in order to retain usability, usefulness, and reliability of the application under such circumstances.

2. Related work

Within model-based UI research, there has been some interest in mobile UIs the last years [e.g., 4,6–16]. The focus in most of this work is on using models at design time for specifying either purely mobile UIs or having models that act as specifications across mobile and stationary UIs. If adaptation is present, it is often focusing on adapting the UI to a given platform as part of a code/UI generation process.

*Corresponding author.
E-mail addresses: Erik.G.Nilsson@sintef.no (E.G. Nilsson), Jacqueline.Floch@sintef.no (J. Floch), Svein.Hallsteinsen@sintef.no (S. Hallsteinsen), Erlend.Stav@sintef.no (E. Stav).

0097-8493/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2006.07.003

There are three main approaches for adaptive UIs, taking a model-based approach more or less into account. The first approach is to handle adaptation at design time [8,9,10,13,14,15,16,17]. This is the most common way of handling adaptation in model-based UI development environments. The UIs are adapted to different platforms as part of a UI generation process. This may also be done at run time [6], but still with one UI per platform. This differs from our approach in the way that we also provide adaptation on the individual platforms.

A second approach is to provide some kind of transformation mechanism at run time [18], i.e., a mechanism that transforms a UI designed for one platform to fit to a different one. The effect is thus similar to the first approach, only using different means.

A third approach is to provide adaptation mechanisms in the UI itself [2,4,19,20], i.e., it is considered part of the functionality of a UI and it is the responsibility of the UI itself both finding out what kind of changes to perform, and performing the changes when needed. This may cause the same effects as our approach, but requires more efforts for developers, as the adaptation is not handled by generic middleware components. In our work, we investigate solutions where the adaptation is managed and handled by independent, generic mechanisms, grouped together to an adaptation middleware. By using this model, the adaptation middleware is both responsible for finding out when to perform an adaptation and for doing the actual adaptation. Repo et al. [19,20] also use an independent middleware in their approach, but it is neither based on component frameworks nor on a model-based approach for describing the variants of a UI.

3. A UI modelling approach based on modelling patterns and compound UI components

In this section we give a brief presentation of the modelling approach presented in [12]. Below, we will show how this modelling approach can be used to make development of adaptive UIs easier. In this section, we focus on the motivation for the modelling approach and its main principles, concepts and features.

Most model-based languages and tools suffer from a combination of two connected characteristics: the languages offer concepts on a too low level of abstraction, and the building blocks are too simple. The building blocks available may be on a certain level of abstraction (like a *choice element* concept that is an abstraction of radio group, drop-down list box, list box, etc.), but are still fairly basic building blocks when a UI is to be specified. With this type of building blocks, a UI specification is an instance hierarchy of such modelling constructs on the given abstraction level. This works well as long as the same instance hierarchy is applicable on all the platforms. If the specification is to work across platforms with a certain level of differences, e.g., with large differences in screen size, there may be a need to have different instance hierarchies on each platform.

This is often handled by dividing the specification of a given UI in two parts, one describing the commonalities across the platforms and one describing the specialities on each platform. This division must usually be done at a quite early stage in a UI specification [21]. Furthermore, the amount of specification code constituting the platform-specific parts tends to be more voluminous than the common part. In such a situation, developing the UI on each platform from scratch may be just as efficient [22].

Unlike most other model-based languages, the modelling approach presented in [12] use a combination of compound components and modelling patterns [23,24] as building blocks. Compound (or composite) UI components are used to be able to have equal or similar model instances on platforms with significant differences (including traditional GUI, Web UIs and UIs on mobile equipment). Modelling patterns are used partly to obtain the necessary level of abstraction to facilitate common models across different platforms, and partly to render it possible to define generic mapping (or transformation) rules from the patterns-based, abstract compound components to concrete UIs on different platforms. A mapping rule is a generic and operational description of how a modelling pattern instance should be transformed to a running UI on a given platform, and mapping rules are an important part of the modelling framework. As a modelling pattern usually involves a number of objects, a UI supporting a modelling pattern must be a *composition* of different UI components (each being simple or composite). The transformation rules must be instantiated with the same concrete classes that the patterns are instantiated with. The modelling approach facilitates development of UIs that are "richer" and more dynamic than what is possible using HTML/XML technology today [22,25].

To utilize the potential of the modelling approach, it also includes a number of different mapping rules to concrete representations for each abstract compound UI component on each platform, based on preferences, desired UI style, modalities, etc. Fig. 1 shows how the different main parts of the modelling approach are connected, expressed using a Unified Modelling Language (UML) class model.

Using this modelling approach, a UI specification consists of a number of model pattern instances, a chosen number of mapping rule instances for each of the pattern instances and additional properties specified for all of these. By mapping rule instance we mean applying a mapping rule to a pattern instance, resulting in a set of UI components (instance hierarchy of concrete UI components) that together constitute the running UI. A specification may also include instances of patterns and/or mapping rules that are specified by the systems developer himself/herself. In addition, the modelling framework has features like extensibility (e.g., the possibility to add new building blocks and mapping rules easily) and recursive modelling

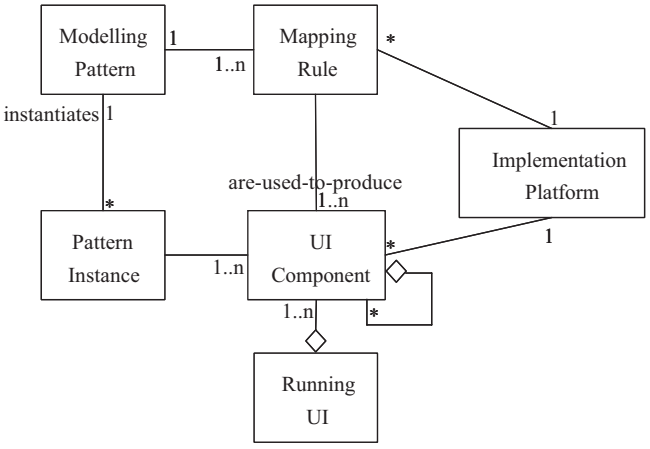

Fig. 1. Main concepts in the modelling approach.

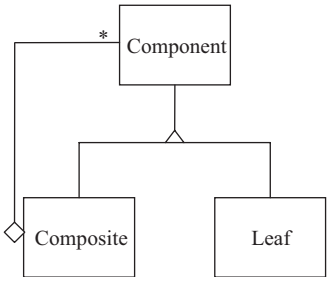

Fig. 2. The composite pattern.

(e.g., the possibility to construct new building blocks by combining existing ones).

The number of abstract components is limited, to make the modelling language comprehensible and to limit the amount of work needed to define all appropriate mappings. Yet, the set is sufficiently comprehensive to render it possible to use the modelling language to specify an arbitrary UI. Depending on the number of available mapping rules, it is also possible to use the modelling approach to realize *graceful degradation of user interfaces*, by having different mapping rules for each platform, exploiting the (lack of) possibilities on each one in an optimal way.

In [12] we presented an example focusing on how the modelling approach facilitates cross-platform development. In this example, we showed how two instantiations of the composite design pattern (file system and system for managing department structure and human resources in an organization) may be mapped to PC and PDA platforms using two different mapping rules on each platform. As we focus on adaptation in this paper, we present an example where the second instantiation of the composite pattern (human resource management) is mapped to PDA platform using two different mapping rules. In this section we show the pattern instance and the mapping rules. Figs. 2 and 3 show the composite pattern and the instantiation of it.

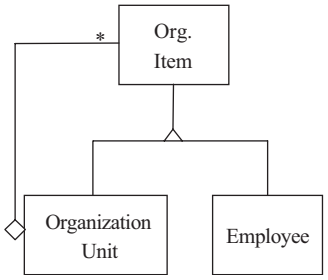

Fig. 3. Department structure/human resource mgmt. instantiation of the composite pattern.

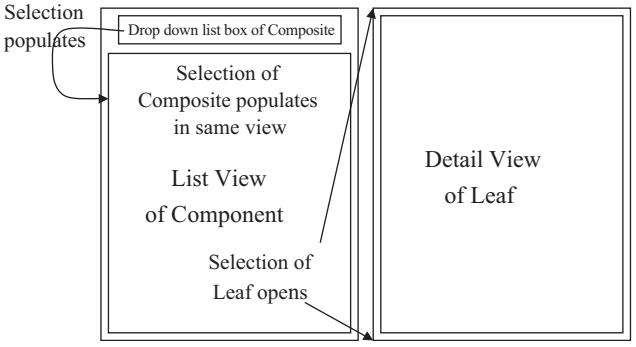

Fig. 4. Stylus usage mapping scheme for PDA presentation.

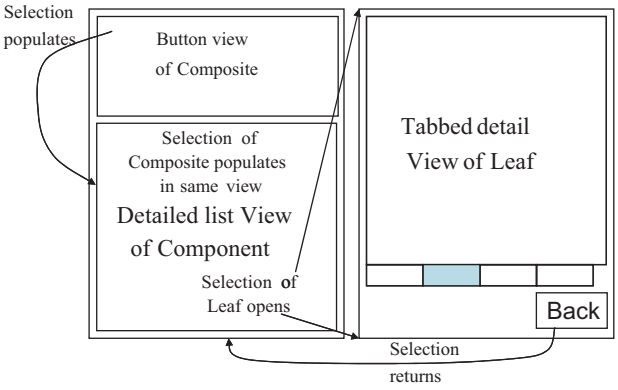

Fig. 5. Finger usage mapping scheme for PDA presentation.

The example shows two similar mapping rules for PDA platform, both using a GUI interaction style. The first mapping rule [12] (Fig. 4) is optimized for overview, i.e., it will present as many instances as possible on the screen at the same time, and also facilitates flexible navigation. The resulting UI requires that the user utilize a stylus when operating it. The second mapping (Fig. 5) is optimized for users not using stylus and focusing on details, i.e., all elements are larger, and the facilities for navigation are on the one side more accessible, but on the other side less flexible. The mapping rule is therefore designed so that the need for navigation is reduced, e.g., by providing more information in the list view. We show the concrete running UIs in Section 5 below where we explain how the modelling approach may facilitate adaptive UIs.

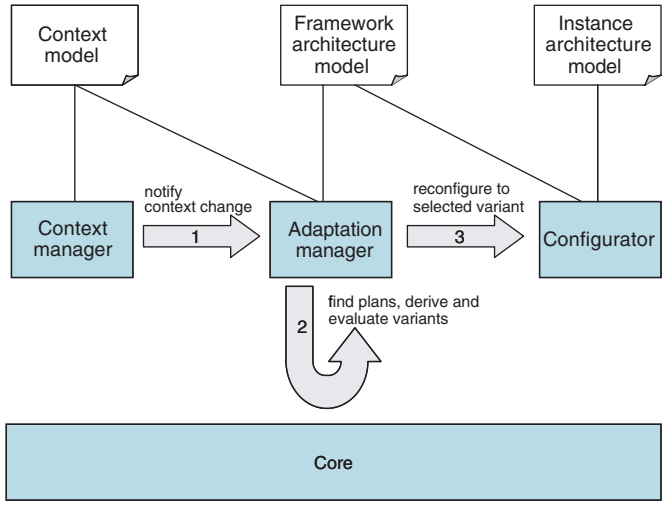

Fig. 6. Middleware architecture and the run-time models.

4. Adaptive architecture

In the FAMOUS[1] project we are developing support for handling adaptive applications in the context of mobile computing [26,27]. We use an architecture centric approach: exploiting architecture models to reason about and control adaptation at run time, and we use generic middleware components that realize the adaptation mechanisms such as reasoning about adaptation [27]. The middleware has three main functions:

1. Detect context changes.
2. Reason about the changes and make decisions about what adaptation to perform.
3. Perform the chosen adaptation.

Fig. 6 illustrates the middleware architecture (the numbers refer to the three functions just mentioned). The architecture is specified as a component framework [28], allowing middleware services to be composed in a flexible manner.

The central components of the adaptation middleware are the Context and Adaptation Managers, and the Configurator. In Sections 4.2–4.4 we describe the different parts of the middleware architecture through the available functionality (the three main functions mentioned above). Before doing this, we need to give some considerations to the models needed by the middleware to perform adaptation.

4.1. Architecture models

For the middleware to adapt an application, it requires a representation of the application architecture model that describes variability. The framework architecture model serves the needs of both the Adaptation Manager and the Configurator to understand the variability built into the framework and how to configure application variants with given properties.

We build applications as component frameworks. A framework consists of a model of the framework architecture and a set of components fitting into the architecture. We achieve variability by plugging different components that exhibit different properties into the framework. Application variants suited for different situations may be created from the framework by selecting a set of appropriate components. Application variants can differ in a number of ways, for e.g. UI, functional richness, quality properties provided to the user, how the components are deployed on a distributed computing infrastructure, and what resources and quality properties they need from the platform and network environment.

A component framework describes a composition of *component types*, these types being variation points at which various *component implementations* can be plugged in. Variability is thus achieved through the plug-in of component implementations whose externally observable behaviour conforms to the type. Each component plugged into a component framework may be an atomic component, or a composite component built as a component framework itself. In this way, an application can be assembled from a recursive structure of component frameworks.

Fig. 7 illustrates a simple component framework architecture. The application is implemented as a composition of three component types: UI, Ctrl (controller) and Db (database). Two atomic component implementations are available for the UI type. Normal UI implements a normal keyboard, pointer and display type of interface; handfree UI implements a hands-free interface based on audio input. Two implementations are available for the Db type. Basic Db implements a direct access to the database on the server side; caching Db replicates data on the client to be less vulnerable to variations in network bandwidth.

To discriminate between alternative component implementations, components are annotated with *properties*. Properties are used to qualify the services offered or needed by components. For example, properties can describe that a UI component implementation supports hands-free mode. Properties can also specify requirements to system resources such as memory or network needs. Properties are tightly related to context elements. In that way we are able to represent the dependencies between component implementations and context. It is the component type that defines the set of properties and their associations with ports. The properties of a composition or component are the aggregation of the properties of its ports.

Associated with implementations, *property predictor* functions are used to predict the properties of the implementations in a given context. These can be implemented in various ways including as constants, e.g., to express that a specific component implementation

[1]FAMOUS (Framework for Adaptive Mobile and Ubiquitous Services) is a strategic research programme at SINTEF ICT funded by the Norwegian Research Council.

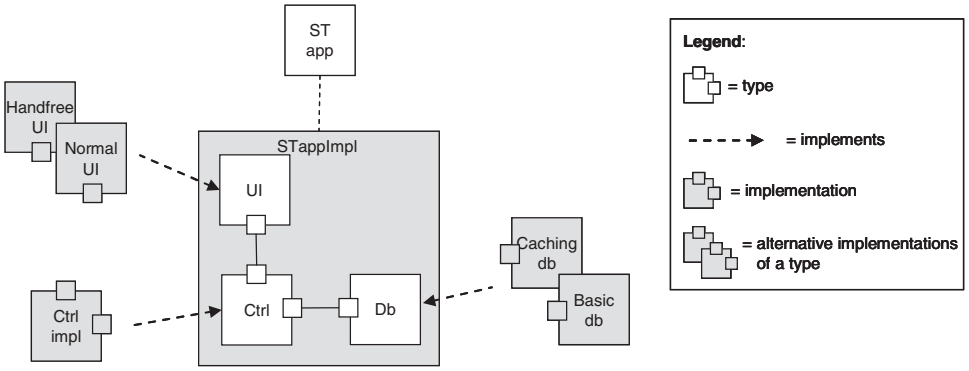

Fig. 7. Component framework.

requires up to 500 kB memory, or as functional expressions of other properties of the component itself, of collaborating components, or of inner components in the case of composite components. E.g., the response time offered by a component may depend on the network bandwidth offered to that component, or the memory needed by a component may depend on the memory needed by inner components.

To decide what adaptation to perform, we first need to decide which application variant best fits the current context. For this, architecture models describe *utility* functions that assign a scalar value to each possible application variant as a function of application properties in a given context. The goal of adaptation management is to determine the application variant with the highest utility in any given context (i.e., when context changes), and adapt the application accordingly. Utility functions are defined by the developer; they are typically weighted means of the differences between the offered and needed property constraints.

Fig. 8 illustrates properties and utility for the simple example described in Fig. 7.

4.2. Run-time models

When an application is launched, the middleware interprets the models specified by the architect to produce a run-time model: the *framework architecture model*. When building this run-time model, the middleware identifies all alternative components that can be plugged in the component frameworks described by the compile-time models. The process of populating the framework architecture model with alternative components is called *planning* [29]. Planning is a recursive process that for each variation point, i.e., component type, defined by the architecture searches for component implementations that fit the variation point.

In addition to the framework architecture model, the middleware also builds and maintains an *instance architecture model* which is a model of the running application variant. This model is important during the evaluation of the running application variant and in the reconfiguration process.

4.3. Context change detection and adaptation reasoning

When the framework architecture model has been built, the middleware is able to determine the properties of interest for the evaluation of the application variants. As these properties relate to context elements, assigning values to properties requires monitoring of the context. Context reasoning, such as aggregation, derivation, and prediction, may also be needed. The *Context Manager* is responsible for these functions. It is also responsible for providing the Adaptation Manager with relevant context information when appropriate, e.g., when context changes occur. As part of context changes, we may also consider the deployment of new components. Such changes could lead to the computation of a new framework architecture model.

When a context change occurs, the *Adaptation Manager* is responsible for reasoning about the impact of context changes on the application and for selecting an application variant that is the most suitable for the current context and user needs. In the current implementation, the Adaptation Manager evaluates the variants as described by the framework architecture model by calculating the utility of each variant in the current context. It also evaluates the current application instance as described by the instance architecture model. In the case that a variant is found to have a better utility than the current instance, the reconfiguration process takes place. In order to avoid unnecessary frequent system adaptations, the Adaptation Manager also considers whether the improvement following a reconfiguration is justified.

4.4. Application reconfiguration

The *Configurator* is responsible for the reconfiguration of an application. By reasoning on the application instance model and the model of the new variant, the Configurator is capable of deriving the reconfiguration steps. Reconfiguration may require bringing components in safe state, deleting or replacing component instances, instantiating components, transferring states, etc.

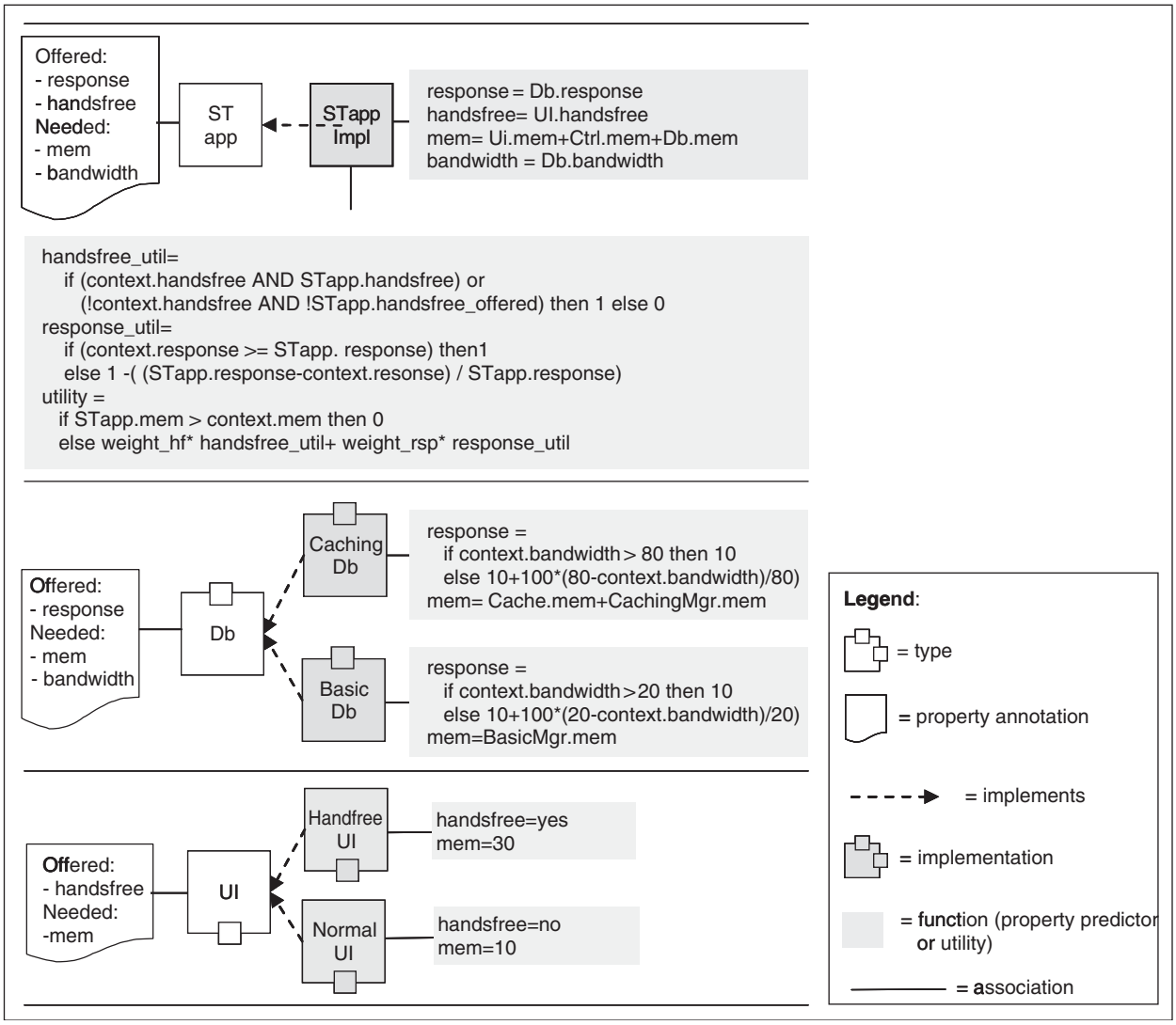

Fig. 8. Properties and utility.

4.5. The developer task

The specification of properties and utility functions is a specialized task that is performed by the developer (or the architect), not by the service user. Both experience and tool support may contribute to facilitate that task. We are currently developing pilot services in an industrial context. This will allow us to evaluate the difficulty of specifying variability in architecture models and provide us with some hints from which initial guidelines and patterns can be derived. We also plan to develop a simulation environment for tuning properties.

5. Combining the modelling approach and the architecture to facilitate adaptive UIs

The adaptive architecture uses models at run time to facilitate any type of adaptive behaviour in an application. In this section, we will show how the patterns-based modelling approach and the architecture may be *combined* to ease development of adaptive UIs, facilitating fairly advanced UI adaptation mechanisms at run time [30].

Normally, a model-based systems development tool does the mapping from the UI models to concrete UIs in the *design phase* (e.g., as a code generation process), i.e., before the system is deployed to the users. The adaptation mechanism may exploit that the modelling approach offers a number of different mapping rules for each modelling pattern. The adaptive functionality is obtained by making the choice of which mapping rule to use at *run time*, i.e., after the system is deployed to the users.

5.1. Using the adaptive architecture

As seen above, the adaptive architecture facilitates mechanisms for component-based systems to be adapted at run time. To utilize this architecture for the presented modelling approach, a number of mapping rules must be applied at design time, so that the adaptation mechanisms have a number of configurations to choose from. This may

be done automatically by a code generation facility at design time. A UI at run time will thus consist of a number of UI components arranged in a structure that the adaptation middleware may exploit.

If we consider the example presented in Figs. 2–5, the developer chooses that the two mapping rules should be available at run time. This causes the model transformation mechanism that normally would generate one running UI per platform to generate the necessary UI components for both the mapping rules. These UIs would look like as in Figs. 9 and 10.

Each of these UIs consists of an instance hierarchy of concrete UI components. These two instance hierarchies are registered as alternatives in the adaptive architecture. The description of when the middleware should choose each alternative is given through the utility function. Above, we stated that the utility function is specified by the developer/architect. This is normally the case, but in this special situation where the mapping rules are generic, knowledge about the utility of the individual mapping rules should be available, i.e., describing a mapping rule may include describing (as a utility function) the factors that make the mapping rule best suited. Thus, the developer of the application may be given (i.e., it is generated) a default utility function for each mapping that he may refine for the specific needs of this instantiation. Once the versions are registered in the middleware and the utility function is in place, the application may be started. Which version that will be used at a given time is depending on the information obtained by the Context Manager with regards to the utility function.

What the adaptation mechanism does at a conceptual level is choosing which mapping rule to apply while an application is running. Of course, the total number of mapping rules to apply (two in the example above) must be decided by the developer. What is appealing with the

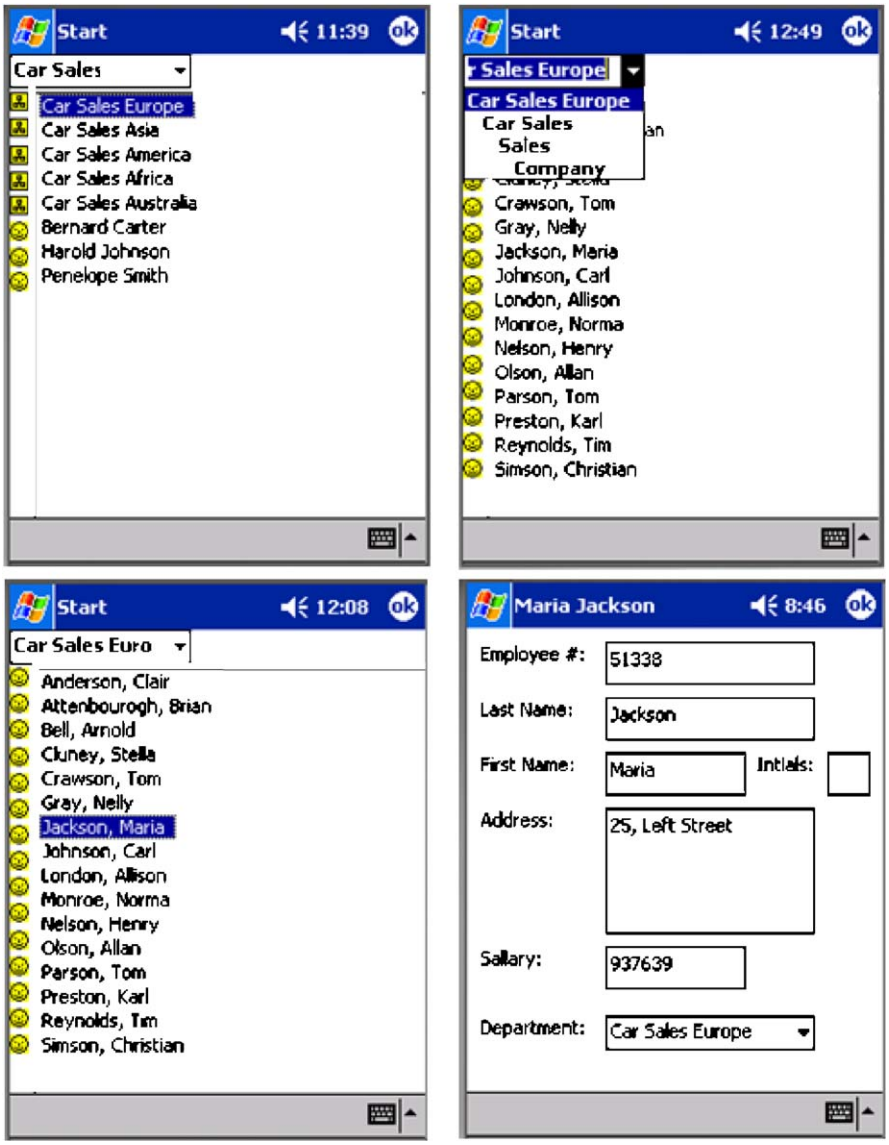

Fig. 9. Stylus usage optimized instantiation of the example application—second view shows navigation to ancestors.

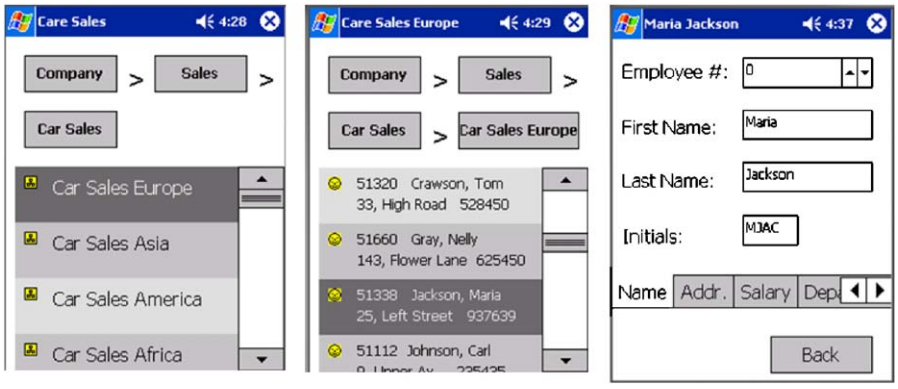

Fig. 10. Finger usage optimized instantiation of the example application.

combination is that it requires almost no additional effort from the developer to offer adaptive features compared to having just one alternative per platform. This is of course only true as long as appropriate mapping rules already exist. If this is not the case, and the developer must make his own mapping rules, the work involved will increase significantly.

The example above shows a moderate size adaptation, i.e., having two versions of a UI that are similar, but also have significant differences. It is easy to see that adding a third mapping rule changing the UI to a speech driven one, fits easily into the scheme, given that such a mapping rule exists. For the user, this type of adaptation, i.e., one involving a different modality, is more radical.

5.2. Types of adaptation

The most obvious types of adaptations that this approach facilitates are *radical changes*, i.e., changes regarding the main principles for how the UI behaves, like changes in the main modality to use or change in the UI style (e.g., from a forms-based to one using icons and drag-and-drop to a wizard-based one). Adaptation is performed by choosing between a set of available mapping rules representing different modalities of interface styles, exchanging the whole instance hierarchy, handled by the adaptation middleware.

As the example above shows, the approach also fits very well for *moderate changes*. Adaptation is performed in the same way as for radical changes, but the differences between the variants are smaller. Thus, the mapping rules become more similar, e.g., different versions of a mapping rule for one modality using a given style. This may cause the number of mapping rules to become larger, and if the degree of overlap between the different rules is large, changing one of them may cause the need for doing the same type of change on the parts of the mapping rule that are shared by other mapping rules. A way of handling this could be to have a sub-typing mechanism that lets different mapping rules inherit from a common ancestor.

To support *low level changes,* e.g., add or remove elements, change icon, colours, table headings, sorting of lists, menus, toolbars, the pattern instantiation mechanisms can be used instead of applying different mapping rules. This may be viewed as an adaptation of the mapping rule, i.e., the changes will not influence the instance hierarchy that the mapping rule used to generate. This type of adaptation should normally not involve the adaptation middleware, except possibly the Context Manager that may be used as a trigger mechanism.

6. Model-based visual adaptation of UIs

Although supported by the modelling approach used for UI adaptation, the type of low level changes just presented, denoted *visual adaptation* in this section, may also be facilitated using more simple model-based means. This will of course make it necessary to implement most of the concrete adaptation mechanisms as part of the system that should exploit it. Some of these types of changes (often referred to as the dialog part of an application) are usually handled by program code. This may be a satisfactory solution if the rules prescribing the changes are fairly simple and stable over time. If the rules are complicated, and/or they change over time, a better solution could be to use a model-based approach. By model based in this context we mean that the adaptation rules are described in models that are separated from the application logic.

6.1. Issues and choices

When adapting *which information* to present, the models should tell which parts of the domain model that should be presented in given situations. This may include which classes to present, but more commonly which attributes to present or base a visualization on, or use for interacting with the application. This may include choice of which attributes to use in different dialogues, which attribute to use as "label" in different dialogues and in "tool tips", etc.

When adapting *how the information is presented*, the part of the domain model to use is determined, but not how it is visualized. Often this is an issue of mapping attribute values to visual coding, like which icon to use, different visual aspects of an icon, value sets for attributes,

sequences of attributes in dialogues etc., but it may also include choosing which interface component to use in a given situation.

In most fairly simple cases of visual adaptation, *managing the adaptation* could be done by the application itself. This means that the adaptation functionality, including handling the necessary models, is part of the application. In more complicated cases, or for product families or configurable standard systems for a specific trade [25], using a generic run-time system to handle parts of the adaptation could be an option, e.g., for managing context information, monitoring it and notifying applications about context changes. Still, how the application responds to the context changes is handled by the application itself.

When considering model-based visual adaptation, it is also important to know *which models to exploit*. Usually, there exist some domain models that handle aspects that are relevant also for how the UI should be adapted. If the domain models are available at run time, an adaptation mechanism may exploit these directly. In most cases, an adaptation mechanism needs additional information to work, like which UI element type to use in forms-based presentations, maximum number of characters, whether an attribute should be suppressed in the UI, the sequence in which the attributes should be shown in the presentations. Thus, specific UI models are usually needed. If the domain model is not available at run time, the UI models may also have to manage relevant aspects from this, like names, data types, etc. for attributes, address formats, and enumeration values for enumerator type attributes.

When deciding adaptation rules, both *models and data may be used*. In most cases, the adaptation is triggered by data, i.e., some change in attribute value or contextual data is used as basis for the adaptation. When performing the adaptation, both information from other attributes, and rules and meta information from models may be needed.

6.2. Examples

In the EU project SUPREME [31], tools for supporting complicated maintenance work (e.g., in nuclear power plants) were developed. One of the prototypes developed in the project use a 3D bar chart to visualize the time aspects of the maintenance work [32,33]. An important feature in this tool is a filtering feature, i.e., instead of searching for information, all relevant information is shown by default, and the user applies a set of filtering mechanisms to reduce the amount of information to show (based on time, part of the plant, status, etc.). In this tool, explicit UI models are used to configure and adapt aspects like which attributes to use for filtering, which type of UI mechanisms to use for each filtering attribute, as well as which attribute(s) to map to the z-axis of the 3D visualization. In this example, focus is both on which information to present and how it is presented, and adaptation is managed solely by the application. Both an active domain model and a separate UI model are exploited, and primarily models are used when doing the adaptation.

In the EU project AmbieSense [1,34], various technologies were developed facilitating easy development of mobile, contextual services, e.g., for travellers. In one of the demonstrators in the project, different contextual parameters were used to adapt which and how information was presented in services for users at an airport. The main features of the service was adapted to whether the user was departing, arriving, or in transfer. Within each of these main modes, details were adapted. E.g., information about check-in counters was only shown before the user passed the security control. Different toilets were shown on the map of the airport for handicapped users than for others. Information about tax-free shopping was only shown for international travellers. In a demonstrator service for travellers in a city, different aspects of the icons representing restaurants changed depending on properties of the restaurant and available information, like number of stars, type of restaurant, price category, whether the menu was available electronically through the service, etc. In this example, focus is both on which information to present and how it is presented. Adaptation is managed partly by the application and partly by a generic context middleware that was developed to handle the context changes to adapt to, this also being the main model that is exploited. Primarily data are used when doing the adaptation.

7. Conclusions and future work

To design usable mobile applications, exploiting context changes is of vital importance. The rapid context changes in a mobile setting cause the need for flexible and adaptive user interfaces that are multitasking and possibly exploiting multiple modalities. We have briefly presented a patterns-based modelling approach based on abstract, compound components and mapping rules to various target platforms. As both the number of such components (i.e., supported modelling patterns), the number of mappings for each pattern, and the number of target platforms are limited, it is possible to optimize the mappings with regard to usability and exploiting special features on each platform.

We have presented a middleware centric approach supporting the building of applications capable of adapting to a dynamically varying context as is typical of mobile use. The approach builds on the idea of achieving adaptability by building applications as component frameworks from which variants with different properties can be built dynamically. As the middleware is generic, it is made once, and may be used by any application conforming to certain requirements, thus giving clear saving for developers. The "price" for obtaining this saving, is that the application must be built in a specific way, and that the middleware requires some additional specifications.

In the paper, we have shown how the modelling approach may be extended to cover adaptable UIs at run

time exploiting the adaptation middleware. We have also discussed how a model-based approach may be used to realize adaptive features in UIs independently of the presented modelling approach.

At the current stage, the adaptation middleware is more mature than the modelling approach (e.g., we have implemented the adaptation middleware). Still, there are a number of challenges for both. For the adaptation middleware, making the optimization process connected to the utility function more efficient, especially for applications with many components, is both challenging and important. Also the architecture needs further development and experimentation. The modelling approach needs further refinement and details, both regarding the modelling patterns and mapping rules and how they should be used at design time, and how the mapping rules should be used to exploit the adaptation middleware to facilitate adaptive UIs at run time.

Acknowledgements

The work on which this paper is based is supported by the projects FAMOUS and UMBRA funded by the Norwegian Research Council, and the EU IST project AmbieSense.

References

[1] Nilsson EG, Rahlff O-W. Mobile and stationary user interfaces—differences and similarities based on two examples. In: Proceedings of HCI International 2003, 2003.
[2] Schmidt A, et al. Sensor-based adaptive mobile user interfaces. In: Proceedings of HCI International '99, 1999.
[3] Rahlff O-W, et al. Using personal traces in context space: towards context trace technology. Springer's Personal and Ubiquitous Computing (Special Issue on Situated Interaction and Context-Aware Computing) 2001;5(1).
[4] Calvary G, et al. Plasticity of user interfaces: a revised reference framework. In: Proceedings of TAMODIA'2002, 2002.
[5] Coutaz J, Rey G. Foundation for a theory of contextors. In: Proceedings of CADUI'02, 2002.
[6] Clerckx T, et al. Generating context-sensitive multiple device interfaces from design. In: Proceedings of CADUI'2004, 2004.
[7] Eisenstein J, et al. Applying model-based techniques to the development of UIs for mobile computers. In: Proceedings of IUI'2001, 2001.
[8] López Jaquero V, et al. Model-based design of adaptive user interfaces through connectors. In: Proceedings of DSV-IS 2003, 2003.
[9] Luyten K, et al. Migratable user interface descriptions in component-based development. In: Proceedings DSV-IS 2002, 2002.
[10] Mitrovic N, Mena E. Adaptive user interface for mobile devices. In: Proceedings of DSV-IS 2002, 2002.
[11] Muller A, et al. Model based user interface design using markup concepts. In: Proceedings of DSV-IS 2001, 2001.
[12] Nilsson EG. Combining compound conceptual user interface components with modelling patterns—a promising direction for model-based cross-platform user interface development. In: Proceedings of DSV-IS 2002, 2002.
[13] Paternò F, Santoro C. One model, many interfaces. In: Proceedings of CADUI'02, 2002.
[14] Pribeanu C, et al. Task modelling for context-sensitive user interfaces. In: Proceedings of DSV-IS 2001, 2001.
[15] Seffah A, Forbrig P. Multiple user interfaces: towards a task-driven and patterns-oriented design model. In: Proceedings of DSV-IS 2002, 2002.
[16] Souchon N, et al. Task modelling in multiple contexts of use. In: Proceedings of DSV-IS 2002, 2002.
[17] Furtado E. KnowiXML: a knowledge-based system generating multiple abstract user interfaces in USIXML. In: Proceedings of TAMODIA'2004, 2004.
[18] Nylander S, et al. Ubiquitous service access through adapted user interfaces on multiple devices. Personal and Ubiquitous Computing 2005;9(3):123–33.
[19] Repo P, Riekki J. Middleware support for implementing context-aware multimodal user interfaces. In: Proceedings of the third international conference on mobile and ubiquitous multimedia, 2004.
[20] Repo P. Facilitating user interface adaptation to mobile devices. In: Proceedings of NordiCHI 2004, 2004.
[21] Nilsson EG. Modelling user interfaces—challenges, requirements and solutions. In: Proceedings of Yggdrasil, 2001.
[22] Nilsson EG. User interface modelling and mobile applications—are we solving real world problems? Proceedings of TAMODIA'2002, 2002.
[23] Gamma E, et al. Design patterns—elements of reusable object-oriented software. Reading, MA: Addison-Wesley; 1995.
[24] Trætteberg H. Dialog modelling with interactors and UML State-charts—a hybrid approach. In: Proceedings of DSV-IS 2003, 2003.
[25] Nilsson EG. Using application domain specific run-time systems and lightweight user interface models—a novel approach for CADUI. In: Proceedings of CADUI'99, 1999.
[26] Floch J, et al. Using architecture models for runtime adaptability. IEEE Software (Special Issues on SW Architecture) 2006;23(2):62–70.
[27] Hallsteinsen S, et al. A middleware centric approach to building self-adapting systems, revised selected paper from software engineering and middleware. In: Fourth international workshop, SEM 2004, 2004.
[28] Szyperski C. Component software: beyond object-oriented programming. 2nd ed. Reading, MA: Addison-Wesley; 2002.
[29] Staehli R, et al. Designing adaptive middleware for reuse. Middleware 2004 Companion, third workshop on reflective and adaptive middleware, 2004.
[30] Nilsson EG, et al. Using a patterns-based modelling language and a model-based adaptation architecture to facilitate adaptive user interfaces. In: Proceedings of DSV-IS 2006, 2006.
[31] Löwgren J, Howard MV. SUPREME visualization concept. Recommendations and rationale. Sweden: University of Linköping; 1996.
[32] Jørgensen HD, Nilsson EG. SUPREME visualization tool, Whitepaper. Oslo, Norway: SINTEF; 1998.
[33] Jørgensen HD. Model-driven work management services. In: Proceedings of Concurrent engineering conference, CE 2003, 2003.
[34] Myrhaug HI, Göker A. AmbieSense—interactive information channels in the surroundings of the mobile user. In: Proceedings of HCI International 2003, 2003.