An efficient model-based methodology for developing device-independent mobile applications

Mario G.C.A. Cimino *, Francesco Marcelloni

Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, University of Pisa, Largo Lucio Lazzarino 1, 56122 Pisa, Italy

Keywords:
Mobile user-interface modeling
Delivery-context aware user interface
User interface markup language (UIML)
Composite capabilities/preference profiles (CC/PP)
Model driven architecture

A B S T R A C T

Current methodologies for developing mobile applications are mostly based on the application programming interfaces (APIs) offered by the native platform. Hence, most solutions are characterized by a low portability and/or reusability. In this paper, we propose a novel methodology based on a declarative and device-independent approach for developing event-driven mobile applications. The methodology relies on: (i) an abstract mobile device based on the user interface markup language; (ii) a content adaptation mechanism based on user preferences; (iii) a context adaptation mechanism based on a standardized context of delivery; (iv) a uniform set of client-side APIs based on an interface object model; (v) an efficient transformational model.

More specifically, in the design phase, the application is modeled as platform-independent on the abstract mobile device. In the execution phase, the application is automatically tailored to the specific platform on the basis of the content and context adaptation mechanisms. We describe the analysis, design and implementation of a framework, called MODIF, which supports the proposed methodology, and show its application to the development of both business and consumer real-world applications on Apple iPhone™ and Google Android™ mobile devices. Finally, we discuss how the experience of using MODIF highlights the quality of the methodology in terms of automation of the lifecycle, expressiveness and readability of the representation, efficiency of the compilation/interpretation, fast learning curve and predictability.

1. Introduction

The number and variety of handheld devices (HDs) connected to the Internet is rapidly growing. As a consequence, there is an emergent need for sustainable design methodologies able to generate services automatically tailored to the features of these different HDs [37]. In particular, the market of HDs, consisting of smart phones and personal digital assistants, raises a basic challenge in the design and implementation of user interfaces: the aim is to make a platform-neutral description of a service efficiently executed and consistently represented on the most popular mobile devices [3,16,22,29,41]. In the industry, current methodologies are based on the development of ad hoc solutions, which are characterized by a low portability and/or reusability [30]. In contrast, in the literature, several methodologies have been proposed for multi-platform design. The most promising paradigm is the model-based user interface (UI) development, which is aimed at providing a systematic approach for specifying the UI by means of models. These models are then translated into a final code executed on the target device. In this approach, a proper specification language and related transformation engines are required, in order to generate the final code for the UI, adapted to a supported target [18]. Indeed, the final code for a target device cannot be generally generated a priori during the design stage, but it should be dynamically provided by an adaptation engine, since target devices evolve over time, according to a number of non functional requirements.

In most of the industrial systems, the adaptation procedure is defined in ad-hoc manner [18]. Thus, most of these systems are using hard-coded adaptation techniques, in which the adaptation code is mixed with the rest of the code of the application, making the reuse of the designed adaptation procedures almost impossible. Hence, a more engineered process is possible only raising the level of abstraction in adaptation design, going towards a model-based paradigm [1,17,45]. For instance, in [36], the proposed model-based approach for UIs is aimed at defining a general abstract structure based on the notion of task. A task is a structured sets of activities that a user has to perform to attain goals, interacting with a system influenced by its contextual environment (e.g., to read reviews with a PDA, to order books via a phone). Here, the basic idea is to capture all the relevant requirements at the task level,

* Corresponding author. Tel.: +39 050 2217455; fax: +39 050 2217600.
E-mail addresses: m.cimino@iet.unipi.it (M.G.C.A. Cimino), f.marcelloni@iet.unipi.it (F. Marcelloni).

and then to use such information to generate effective UIs tailored to each type of platform considered, such as desktop, handheld, phone, and so on [23,35]. This design cycle requires a number of transformations in order to obtain final applications for supporting the users' activities, through various devices accessed in different contexts (nomadic applications) [24,38]. In [39,40], a task modeling process captures the business that the UI should guarantee, providing support across the entire lifecycle of a multi-platform interface: design, development, operation, management, organization and evaluation.

The mentioned approaches are mostly aimed at capturing general requirements of universal usability for a broad class of target devices. The large extent of the class, however, limits the degree of automation/efficiency of the entire process. Indeed, any adaptation pattern can be automated only under particular constraints imposed on the target devices. For instance, in terms of design, an abstract view of a mobile application can be entirely transformed into a concrete executable application by software agents only if a limited variety of these devices is considered. To this aim, in order to foster automation and efficiency, we focus only on the HD class of devices with the intent of generating mobile applications that are both responsive to the events generated by the users and characterized by a model-level specification. Thus, we do not consider, for instance, client devices such as notebook, tablet, desktop, projector, web tv, and wap which are excluded from the HD class.

HD features determine some constraints that force specific design choices. Such choices can concern the representation language, the communication and the synchronization between targets and controllers, the distribution of the controllers, and so on [44]. For instance, the set of delivery contexts includes remote controllers and therefore usability might be affected by response latency. A solution for reducing this problem is to allow some local computation on the controller. This choice leads to support the rich internet application (RIA) paradigm on the client-side. Currently, in the case of HD the only acceptable solution to this problem is based on the native GUI toolkit. Indeed, the document-centric interface of web browsers does not supply rich controls. Also, the extensive use of JavaScript, plug-ins, or java virtual machines brings serious compatibility and inefficiency issues.

We propose a novel methodology based on a declarative and device-independent approach to develop event-driven mobile applications for HDs. Thus, neither document-centric (i.e., browser-based) applications nor the directions of research and development conducted in the literature to address the problem of making a web site accessible in multiple environments [46] are considered in this paper. The methodology relies on: (i) an abstract mobile device based on the user interface markup language; (ii) a content adaptation mechanism based on user preferences; (iii) a context adaptation mechanism based on a standardized context of delivery; (iv) a uniform set of client-side APIs based on an interface object model; (v) an efficient transformational model. Further, in our methodology, the mobile applications are supposed to be developed from scratch, so as to take a complete life cycle into account, considering the large number of different HDs today available and the variety of purposes for which HDs can be used. More specifically, in the design phase of our methodology, the application is modeled as platform-independent on the abstract mobile device. In the execution phase, the application is automatically tailored to the specific HD on the basis of the content and context adaptation mechanisms. We describe in detail a framework to support the methodology, called mobile device independence framework (MODIF). Further, we show the use of the framework to develop two applications on Apple iPhone™ and Google Android™ (iPhone and Android for short) mobile devices. Finally, we discuss how the experience of using MODIF highlights the advantages of exploiting the methodology in the development of mobile applications.

The paper is organized as follows: the next section is devoted to related work. In Section 3, languages for multi-platform modeling are analyzed and compared. Section 4 discusses our model-based methodology to develop multi-platform UIs. Here, the lifecycle of an application is organized as a series of model-to-model transformations, structured into an architectural framework of layers. This architectural view is detailed in Section 5. Section 6 describes the content and context adaptation layers, providing a more detailed design of the server side. The client side view is studied in Section 7, pointing out how the adapted application description can be efficiently transformed into native code, keeping a uniform set of Application Programming Interfaces (APIs) between platforms. Section 8 provides some implementation details of the architectural framework. Finally, Section 9 comprises case studies over *iPhone* and *Android* OSs and for two different mobile applications, as well as a discussion on the experience of using MODIF.

2. Designing device-independent user interfaces: related work

In this section, we analyze the related work in the field of device-independent UI design, highlighting the most relevant differences with respect to our methodology. In the literature, a number of approaches have been introduced for designing browser-based applications. For instance, in [6] the authors propose a development framework for form-based web applications on heterogeneous devices. Here, a semiautomatic process allows the customization of a generic application for specific target devices as well as the transformation of existing HTML pages into generic application artifacts. The main reason for using HTML as a specification of UIs is to take advantage of the fact that there exist web browsers for virtually every platform. However, HTML is not adequate to describe a mobile application that is not document-centric. Further, web browsers are not appropriate for implementing many interaction patterns available on smart phones.

A growing number of papers are using the user interface markup language (UIML) [32], i.e., an XML-based language aimed at expressing UIs for multiple software platforms on different devices and for multiple applications. A UIML rendering engine capable of generating platform-specific UIs from a high-level UI description is presented in [19]. Here, a tree representation of the UIML document is built, inclusive of style and behavior. The engine is separated into a rendering core and multiple rendering backends, each for a corresponding specific vocabulary. The UIML document is first transformed into a general purpose document object model (DOM) representation by the rendering core. Then, the rendering backend loads dynamically the mapping to concrete widgets and builds the interface. Thanks to the reflection mechanisms, the rendering core is also able to realize a dynamic mapping to a specific widget. Although the approach is flexible both in terms of device types and interface adaptation, it does not focus on efficiency and HDs. In contrast, our methodology does not require a dynamic mapping to a specific widget. Indeed, the designer takes into account an abstract toolkit that is platform independent and functionally complete in terms of patterns. Such patterns are dynamically configured and adapted in terms of context of delivery (e.g., screen size of the HD), and the transformation of the abstract toolkit to a concrete one is based on an efficient static mapping on the client side. This allows the UIML rendering format to be purposely designed for an efficient rendering process.

In [21], a generic multi-platform UI builder based on UIML is presented: the builder exploits a design approach aimed at incrementally generating UIs. The designer works with the concrete graphical representations, and the tool maintains a synchronized

platform-independent UIML representation. The approach relies on a transformation engine, and multiple UIML rendering engines. The use of UIML is quite atypical. Indeed, the designer works in a concrete toolkit, and UIML is employed only as an abstract representation. Further, the transformation engine is used to generate initial designs for new platforms based on the previously designed interfaces. Thus, the work mainly focuses on proposing an intuitive multi-platform user interface design approach, which is flexible and usable. However, as declared by the authors themselves, the rule-based transformation engine, exploited in the approach, is affected by some limitations caused by the difficulty of manipulating the transformation rules. Indeed, this work does not consider efficiency aspects since it does not take the development of an efficient runtime engine into account. In contrast, by limiting the target devices to only HDs, we avoid using rules, thus increasing efficiency and adaptivity.

In [2], UIML is enriched by adding alternate vocabularies and transformation algorithms, thus allowing the developer to build a single specification for a family of devices. The approach focuses on adaptability, i.e., it can be customized by the designer creating a new family of devices. However, unlike our methodology, it does not take into account efficiency and adaptivity requirements. Further, it does not propose the design of an architectural framework supporting the entire life cycle of an application.

The use of a markup language designed for developing device-independent applications is crucial to describe the several facets of UIs at a higher abstraction level. If these descriptions are complete, it is possible to automate the construction of different platform dependent descriptions. There is a large body of research on abstract user interface descriptions and user interface generation. The approaches differ from each other in transformational model, workflow, automation degree, target applications, category of device, etc. Thus, to compare our methodology with these approaches is not significant, due to their heterogeneity. However, for the sake of completeness, in the following we will summarize the most relevant approaches proposed in the literature.

In [10] a method that supports adaptation in UI design is presented, automating particular features of the design process. In particular, here, interface design is viewed theoretically as a process of creating mappings between various formal elements at different levels of abstraction. A decision tree is used to perform such automatic mappings. In [13], interface adaptation is treated as an optimization problem, i.e., the rendering of an interface on a specific device is made so as to meet the constraints of the device and to minimize the estimated effort for the user's expected interface actions. In [14], the authors provide an evaluation of two different systems, which, given a model of the user, automatically generate personalized interfaces. More specifically, the systems use a model of the user's motor capabilities and a model of the user's preferences, respectively. Results show that the automatic generation of capability-based interfaces improves both performance and satisfaction of users. A system for automatically generating remote control interfaces is discussed in [26]. In particular, the system relies on a communication protocol, adaptors for translating from proprietary appliance protocols to the proposed protocol, and a specification language for describing the functions of an appliance, as well as generators that build interfaces from specifications. In [27], the interface generation is improved using a technique that uses parameterized templates in the appliance model, to specify when design conventions might be automatically applied in the UI. In [28], remote control interfaces are automatically generated by identifying similarities between different devices and users so as to optimize the generation process. In [31] a method for designing and developing services that are accessed by using many different devices is considered. This method is based on a separation of the user–service interaction and presentation. More specifically, the system is made of three main parts: the interaction specification language, customization forms, and the interaction engines.

3. A comparison of languages for multi-platform modeling of content and services

In this section, we first introduce the declarative and imperative languages to support multi-platform modeling, and then we focus on UIML as a powerful abstraction of a UI in terms of structure, content, style and behavior.

HDs can be thought of as miniaturized computers. They provide a small display for output, a small alphanumeric or touch keyboard, a stylus and voice capabilities for input, limited memory, processing power and bandwidth. Thus, metaphors used for desktop devices do not scale down and fail to provide a good foundation for building user interfaces (UIs) for HDs. For instance, the *windows* metaphor cannot be implemented, because small screens can only display one window at a time effectively. The *card* metaphor addresses this problem, considering the interface as a stack (deck) of cards with only one card being visible at any time.

Markup languages for HD have tried to overcome these limits. C-HTML [47], for instance, is a subset of HTML for HD. WML [34] is another XML language for narrow-band devices. V-XML [49] is a standard for conversational interfaces. These declarative languages hide implementation details, reducing the amount of expertise and time needed to develop the software. However, the implementation of an engine for these standards requires a corresponding mapping to platform-dependent programming languages. To map interface elements at platform level is very expensive since a deep knowledge of underlying APIs is required. Thus, the use of the above XML standards is not convenient, if it is not supported by a proper level of abstraction that helps to hide the differences between classes of HDs. Abstraction helps portability across different mobile devices and mobile operating systems, and various human–device interaction paradigms [11].

In the field of imperative languages, a considerable step towards portability has been carried out by the Java Micro Edition™ [42] framework, which provides an effective way to hide the native platform. Unfortunately, important mobile platforms such as *Microsoft Windows Mobile* and *iPhone* do not natively support Java ME. Moreover, the use of Java for developing user interfaces is for experts: novice programmers must invest valuable time to master it. Finally, Java is an imperative language and, if not natively supported by the mobile OS, slows down program execution without offering a higher abstraction with respect to other native languages, such as C# [9] or Objective-C [4]. UIs are today implemented on an increasing number of software and hardware technologies, thus demanding new abstractions [38].

UIML is based on an extension to the model/view/controller paradigm, called the meta interface model (MIM) [32]. MIM provides a six-way separation or factorization of the UI elements: UI *structure*, *style*, *content*, *behavior* rules, connection of the UI to external API (*business logic*), and mapping of the vocabulary used for class/property/event names to widget set (*presentation elements*). This canonical factorization pursues the evolution of the W3C specifications in the UI area, which has followed a path of gradually splitting a UI description into orthogonal parts [20]. In fact, up until HTML 3.2, there was no partition in a UI. In HTML 4, the style was separated (via CSS and XSL-FO) from the remaining part of the UI specification. Then, in XForms, the portion of a document that represents a form was extracted from the UI specification. Finally, in XML Events, events were managed separately from the other aspects of the UI. Thus, the factorization of the UI

elements made in UIML is compatible with the most important W3C standards.

UIML is today a standard of the OASIS, a global consortium that drives the development, convergence and adoption of e-business and web service standards. Since 1999, the initial work on UIML was conceived to create multi-platform UIs. Hence, UIML does not contain tags specific to a particular UI toolkit. UIML defines generic tags common to any toolkit, and language elements to map these tags to a particular toolkit. In a UIML document the vocabulary of a particular toolkit appears as the value of the tag attributes. An external toolkit-specific document enumerates for a particular toolkit a vocabulary of toolkit components and their property names. Although UIML allows a multi-platform description of UIs, there is a limited similarity between platform-specific descriptions, when platform-specific vocabularies are used. This means that the UI designer has to create separate user interfaces for each platform, using its own platform-specific vocabulary. A vocabulary is defined in terms of a set of concrete user interface elements with associated properties and behavior.

More specifically, in the MIM, a UI has an abstract *structure* made of a set of interface elements with which the end-user interacts. Each interface element, i.e., a *part*, may be organized differently for different categories of end-users and different families of services. Each interface part has *content* (e.g., text, sounds, images, etc.). The mapping between an interface part and the associated artifact is carried out by using special elements. The interface portion of a UIML document defines a virtual tree of parts with their associated *content*, *style* and *behavior*. Hence, in UIML there is a differentiation between *structure* and *content*. The *behavior* element of UIML describes the actions that occur when an end-user interacts with a UI, and is built on rule-based languages. Each rule contains a condition and a sequence of actions. Whenever a condition is true, the associated actions are executed. A condition becomes true when suitable *events* are triggered. Each action can change a property or variable of some part of the UI, invoke a function in a scripting language or from a software object, in order to specify the application logic that modifies dynamically the view or the model of an application. Furthermore, UIML provides reusable pieces of a UI as a *template*. This allows a canonical description of a user interface through the UI lifetime.

4. A new model-based methodology for developing mobile applications: the trade-off between abstraction and granularity

This section shows a comparative view of UI toolkits in terms of abstraction level and granularity, and introduces our model-based methodology starting from the model driven architecture of the OMG group.

An important feature of UIML is the level of abstraction used to describe a UI. UIML does not contain tags specifically related to a particular UI toolkit, e.g., <BUTTON>, <WINDOW> and <MENU>. It captures the elements that are common to any UI toolkit through a set of generic tags, defining special attributes that map these elements to any toolkit. Therefore, a UIML author needs to specify, in a separate document, the UI toolkit (e.g., Java Swing,[1] Microsoft Foundation Classes,[2] WML, etc.) to which he/she wishes to map the UIML document. This binding mechanism, from an abstract element of an interface to a concrete presentation component of a UI toolkit, can be implemented at different abstraction levels.

Indeed, UIML should only capture the human–computer interaction (HCI) designer's intent. This is possible by allowing descriptions of the UIs expressed using abstractions of the HCI designer's

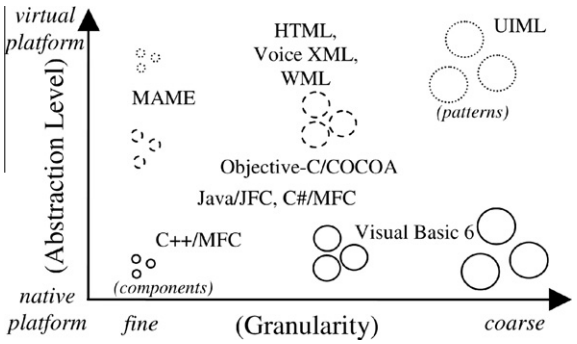

Fig. 1. Granularity and abstraction levels of UI toolkit components.

choices, rather than using UI designer languages and/or widgets. If a traditional UI development tool is used, e.g., Visual Basic,[3] Dreamweaver,[4] and NetBeans,[5] UI designers are forced to map their thinking to Visual Basic, HTML forms, and Java widgets, respectively, because the palette they use to design the UI contains widgets. In contrast, since UIML is a meta-language to which a vocabulary can be added, one can devise vocabularies specific to training courses, auto and industrial automation, thus separating the HCI designer's intent from the particular widgets used in a specific target language or device.

The aim of this paper is to propose a novel design methodology based on a purposely designed UIML abstract toolkit. To this concern, two key concepts are abstraction and granularity. In general, *abstraction* in a specification/programming language means to hide implementation details from the language, thus reducing the amount of expertise and time needed to develop the software. For example, the Java virtual machine abstracts the native architecture, providing APIs over it. *Granularity* pertains to the average quantity of functionality encapsulated by a single atomic element or method of the API, which can be natively implemented. For instance, in a service-oriented architecture, service granularity defines the capability of a specific service.

Indeed, an abstract toolkit based on *patterns* can enable a very efficient UIML description as well as a very efficient implementation, because each pattern can be natively implemented. A pattern captures the essence of a successful solution to a recurring usability problem in interactive systems [7]. For instance, Apple has identified several simple and intuitive patterns, not very dissimilar from those for traditional mouse use, specialized for a multi-touch interface. Such patterns are provided via the UITouch class, which is one of many related classes in the UIKit framework.[6] Patterns are both at a higher level of abstraction and a coarser granularity than normal toolkit components. There is, however, a tradeoff between providing designers with a high level of abstraction and giving them the control over the interface design. Thus, in designing an abstract UI toolkit, a crucial decision is the *granularity* of the toolkit elements. For instance, a toolkit that includes only simple controls such as line, rectangle, text, is a fine-grained toolkit, which allows skilled developers to control each pixel of the GUI. A toolkit that includes higher-level controls such as windows, buttons, menus, and text fields, is a coarse-grained toolkit. Fig. 1 represents some language for UI and their relative position in the abstraction/granularity space. Here, elements that are more complex are represented with large circles, whereas elements that are more abstract are represented with dotted circles.

[1] http://java.sun.com/docs/books/tutorial/uiswing.
[2] http://msdn.microsoft.com/en-us/library/d06h2x6e%28VS.71%29.aspx.
[3] http://msdn.microsoft.com/en-us/vbasic.
[4] http://www.adobe.com/it/products/dreamweaver.
[5] http://netbeans.org/features.
[6] http://developer.apple.com/library/ios.

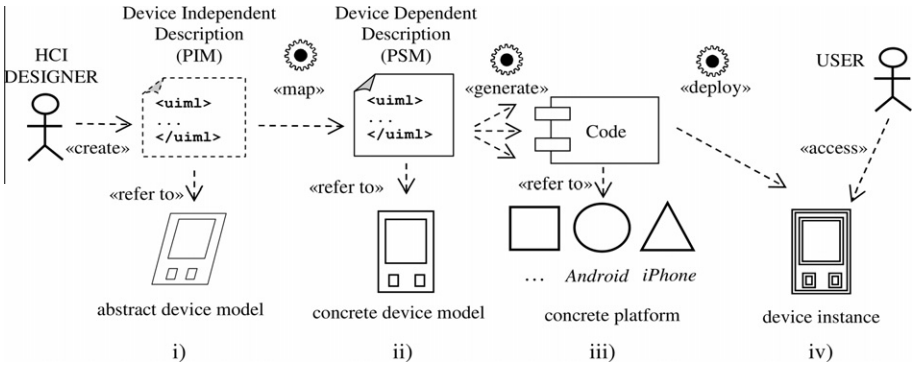

Fig. 2. A model-based methodology for developing mobile applications.

In general, a declarative language is more abstract than an imperative language, because the latter specifies in detail how to perform a task, while the former only specifies what the task is [38]. Since Markup languages are by their nature declarative, HTML, V-XML, WML, and UIML are at the top side of the figure. In general, the abstraction can be realized also with an imperative language that hides architectural aspects. For instance, multiple arcade machine emulator (MAME) is an emulator designed to recreate the hardware of vintage game systems in software on modern personal computers and other platforms, thus using the original program code. MAME provides abstraction, but there is no granularity at all, as it interprets each line of code. Hence, it is at the top left side of the figure. Imperative platform-dependent languages such as C++ have a low abstraction and a low granularity, because they do not natively support powerful APIs to create GUI, as, for instance, Visual Basic 6 does. Finally, modern imperative languages (Java, C#, Objective-C) based on virtual machines are at the middle of the figure, because they support a certain class of devices and the construction of a GUI requires more details than in Visual Basic 6.

Our proposal comes from the role that a toolkit can play considering different levels of abstraction of its components. To allow the development of a mobile UI according to an abstract UI toolkit, we have to design first an abstract mobile device, in a platform independent way. This has to be done considering a coarse granularity in order to be efficiently implemented on HDs. Then model adaptation/transformation techniques can be used to refine the design by adding details specific to the chosen platform.

In order to produce an architectural framework that depends on a formal, declarative, implementation-neutral description of the UI [12], we follow a model-based methodology which is inspired to the model driven architecture (MDA) of the OMG group [5]. In the MDA, building models are defined by using the unified modeling language (UML) notation. At the highest level, the MDA has a platform-independent paradigm, which can be employed in the model-based UI realm. The purpose of this paradigm is to capture the requirements that are specific to a particular domain. At the next level, the platform independent model (PIM) has a lower degree of platform-independence, for a set of different platforms of similar types. It is equivalent to an abstract UI in the model-based UI realm. The platform specific model (PSM) combines the functionality of the PIM with the requirements of the target platform. Finally, a platform model provides the details of a particular platform and the provided services [1]. Hence, the key to enable an MDA methodology is model transformation, which will be discussed at several levels in this paper. Indeed, UIML in its current version lacks of a transformation model. An overview of the methodology proposed in this paper for HDs is shown in Fig. 2.

The initial UIML description (Fig. 2i) is created by an HCI designer, considering an abstract device model: it is independent of any technology, but it is completely detailed in terms of functionality, thanks to the use of a set of interaction patterns, discovered via the aforementioned process. This description is then automatically transformed into a device-dependent description (DDD) (Fig. 2ii), which considers a concrete device model. This mechanism is based on the variation or adaptation of patterns [5]. Patterns may need different features (attributes, operations, constraints) to be effective in different delivery contexts. For instance, some new features may be added and/or optional features may be omitted. This allows patterns to be configurable and adaptable. The DDD is a model that contains both business and platform information, and is the basis of source code and associated artifacts. Once the DDD has been generated, it is "compiled" so as to generate other artifacts, such as deployment descriptors, build files, and so on for a particular platform model (concrete model in Fig. 2iii). Finally, the code is deployed and rendered on a device instance (Fig. 2iv). It can be observed that this approach separates concerns: the business functionality is represented in a PIM (abstract device model), and the implementation aspects are represented in a PSM (concrete device model). This makes the PIM reusable over many different platforms, provided that there is a suitable PIM-to-PSM mapping agent, and a PSM-to-code compiler agent for the target platform.

5. Architectural view of the proposed model

In this section, we first introduce the delivery context and the related standards, and then outline our system architecture.

Delivery context information is typically used to provide an appropriate format that makes the service suitable for the capabilities of a delivery device [50]. The adaptation can be performed: (i) by the originating server; (ii) by an intermediary in the delivery path; (iii) by a user agent. In [50], the term delivery context (DC) refers to a set of attributes that characterize the capabilities of the access mechanism, the user preferences and other aspects of the context into which a resource has to be delivered. Potential characteristics that might be expressed in the DC are: (i) interaction (I/O modalities and parameters), (ii) user agent capabilities, (iii) connection, (iv) location, (v) locale,[7] (vi) environment, (vii) level of discourse and (viii) trust.

Some aspects of the DC, such as user preferences, normally require manual configuration. User preferences related to application personalization could be transmitted as part of the DC. It is impor-

[7] A locale represents a geographical region.

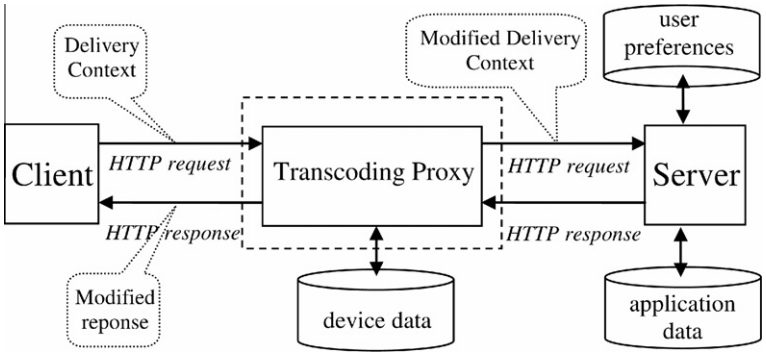

Fig. 3. Intermediate adaptation paradigm.

tant that the user is provided with sufficient flexibility to control those characteristics, especially where the needs of the user may differ from those provided in standard configurations. From the point of view of the designer of a system with DC information, three important components need to be defined to ensure interoperable implementations: (a) a data structure and a vocabulary for exchanging DC information; (b) a protocol for conveying the DC information; (c) a processing model for handling the DC information.

Considering a canonical client–server web paradigm, the client, originating a request for some resources, may also include some DC information, which can help in handling the request appropriately. In practice, the context information may be included as part of the request, or may be supplied indirectly as a reference to information that is stored separately. DC information may also be used locally. Fig. 3 represents a paradigm in which the adaptation is made by an intermediary in the delivery path.

Here, the intermediary may modify the request, providing new delivery context information, in such a way that the response can be adapted appropriately. In the most general situation, a sequence of intermediaries may provide additional DC information at different points in the request path from client to server, and may modify the response in the response path from the server to the client. The response may be modified based on any delivery context information available at that point in the response path. In some situations, an intermediary may block delivery context information from being passed further along the request path. The delivery context also has wider significance than its usage in developing adapted content. The application that runs on the user agent (typically the client device) can utilize the device and environment context information for providing contextual adaptation.

The W3C has specified a data structure and a sample vocabulary for profiles that can convey delivery context information, i.e., the composite capabilities/preferences profile (CC/PP) [48]. CC/PP is a vocabulary extension of the resource description framework standard: it allows different vocabularies to be designed and implemented by communities involved in developing applications, devices and browsers. Further, it also allows the dynamic composition of a delivery context profile from fragments of capability information that may be distributed among multiple repositories on the web. CC/PP is the preferred approach to communicate delivery context between clients, intermediaries and origin servers. It is the basis for UAProf [33], which is used to express the capabilities of the mobile devices in the proposed framework.

In order to implement the interoperable exchange of delivery context information, it is necessary to specify how the information is conveyed as part of a request protocol. No consensus has been reached on how more general delivery context information can be conveyed. When expressing device capabilities, the strength of CC/PP is that it has a certain flexibility which is lacking in other languages. Indeed, CC/PP allows not only defining a fixed set of preferences that would be used to build device profiles, but also creating whole vocabularies, making the modeling of device and agent capabilities, and user preferences infinitely extensible.

An example of transformation mechanisms based on the delivery context is the *resize* of images to fit the device resolution [12]. This can be done automatically by the intermediary proxies based on the resolution information given by the CC/PP profile. Another example is the *reduction* of the capabilities of a specific graphical component, according to the capabilities offered by the device [51].

The open mobile alliance,[8] a leading industry forum for developing interoperable mobile service enablers, already implements CC/PP in their UAProf technology for WAP devices. This technology is used to help proxies transform content for mobile use.

Guidelines and paradigms illustrated up to here have been implemented and tested in a system architecture called mobile device independence framework (MODIF). Fig. 4 shows the overall architecture of MODIF. Here, the application repository (APP) contains the UIML description which characterizes the application as independent of the content and the context of delivery. This description is given considering an abstract device: the HCI designer can, therefore, focus on her/his activity without paying attention to mobile device features (e.g., precise screen dimension) and user preferences (e.g., theme). User preferences are stored in the content server (SERVER), and can be managed by the user via web interface. User preferences concern application parameters and content, and then are independent of the device model and type. For instance, different user preferences can be set for the same device model. Most of the user preferences have a default value. Hence, users do not need to set parameters for the execution of the application. Once the user launches a specific application, after the client request, this description goes through a content adaptation process that can be made by the content server with the user parameters. This process is performed by the content adaptation layer (in figure, dashed boxes represent logical layers). For instance, the specific user theme, which determines background color, font type, etc., can be resolved at this stage in the UIML description.

The UIML page is then sent to the Proxy to be processed by the context adaptation layer. Here, the Proxy can access the CC/PP descriptor of the user device to determine a set of interface features, such as screen size, device sensors, etc. [15]. The UIML description produced by the Proxy is tailored for a concrete device (e.g., *iPhone*, *Android*, *RIM Blackberry*, *Microsoft Windows Mobile*, *Symbian*, etc.). Once this description is loaded on the client, it can be "compiled" and rendered in an efficient manner, thanks to a series of Interface Object Model APIs. In the next sections, each layer will be described in detail.

[8] http://www.openmobilealliance.org/.

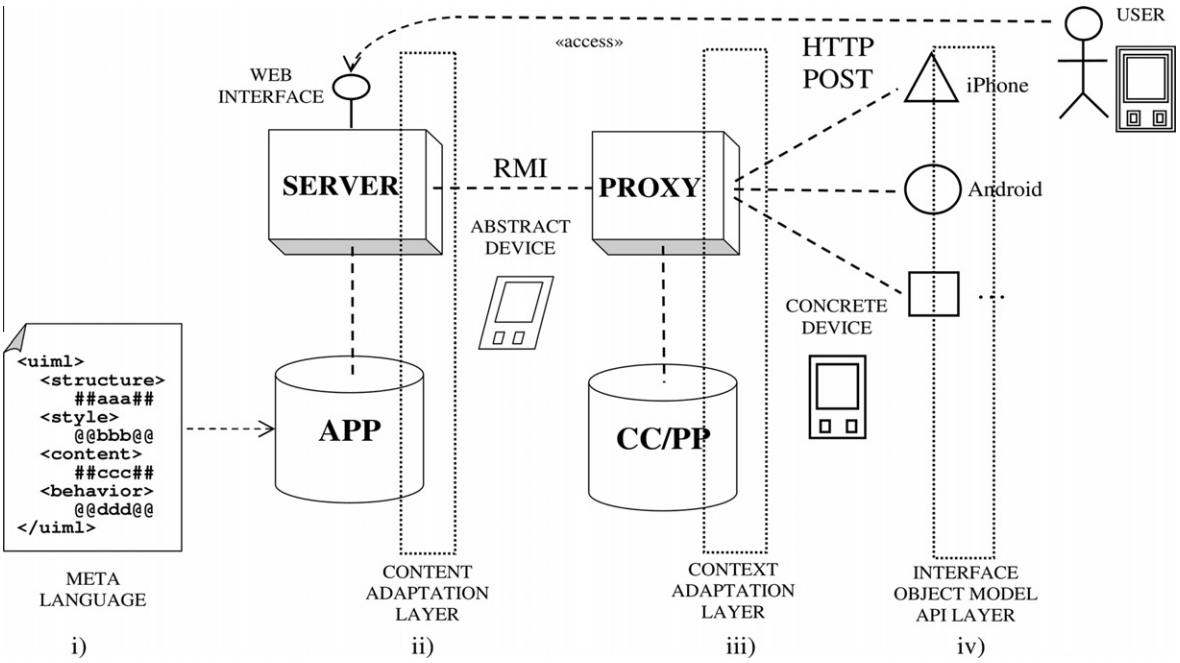

Fig. 4. The MODIF overall system architecture.

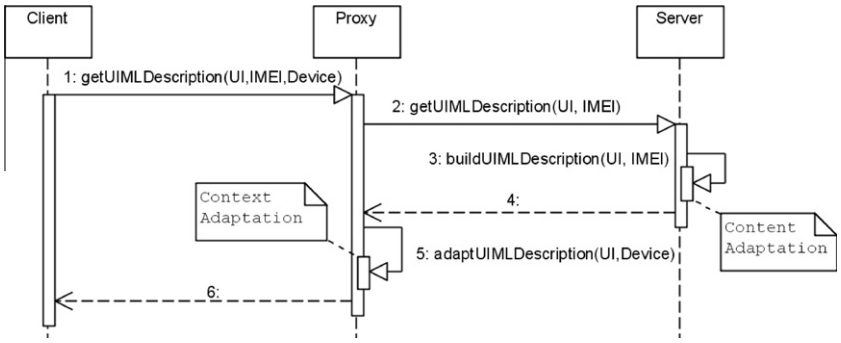

Fig. 5. Interaction between subsystems for building a UIML description.

6. Content and context adaptation layers: a behavioral model

In this section, we describe how the data on user preferences, context delivery and interface design are processed by MODIF, providing a more detailed design of each subsystem. In agreement with the intermediate adaptation paradigm of Fig. 3, MODIF is made of three subsequent processing nodes. Fig. 5 gives a sequence diagram with the interaction between client, proxy and server during the first request of an interface description.

More specifically, when a mobile application starts, the client sends a request to a specific application proxy for an interface description (1). The request, received by the proxy, contains an identifier of the interface description (valid within the specific application), a universal identifier of the device (e.g., the IMEI code), and the device name. The first two parameters are forwarded to the server (2). The server is then responsible for locating the specific (abstract) interface description and building the (abstract) description with a content adaptation process (3), which is based on the user preferences. In particular, the initial UIML description contains a series of *formal user preferences*, which need to be resolved by the server considering the IMEI code and the preferences of the owner of the device. For example, the *user theme* is the preference that encapsulates a set of graphical options, such as a specific background image or color, a foreground color, font family and size, etc. There can be either a default theme or a theme specified by the user via web interface. In the UIML abstract document, each formal user preference is marked with a special text pattern, and coded using a simplified XPATH expression. An XPATH expression allows identifying concrete values in the database of user preferences. Thus, the parser responsible for content adaptation can be very generic and completely independent of the user database and the user preferences. Fig. 6 shows a UIML fragment with some formal user preference. After the content resolution process, the text in bold is replaced by actual values (e.g., “arial”, 12, “green”).

The UIML description returned to the Proxy, in the form of a Java DOM object, is personalized for the specific user, but is still abstract and device-independent. It is the Proxy itself that produces a concrete UIML description, for the specific device model. Starting from the device name, the proxy is able to access a series of device capabilities represented in terms of UAProf description. In the UIML description, a special marker denotes the formal elements, attributes and values that need to be made concrete after the device adaptation. The adaptation process is made of a series of rules. For instance, let us consider two scenarios: (a) the adaptive scaling of an image and (b) the adaptive resizing of a component. In the

```
<part id="menuElementDescription" class="Label">
  <style id="menuElementDescriptionStyle">
    <property part-name="menuElementDescription" name="fontFamily">##user/theme/font##</property>
    <property part-name="menuElementDescription" name="fontSize">##user/theme/fontSize##</property>
    <property part-name="menuElementDescription" name="fontColor">##user/theme/fontColor##</property>
    ...
  </style>
</part>
```

Fig. 6. An example of formal user preferences for content adaptation.

first case, let us suppose that a specific device has a 320 × 480-resolution screen. On small screens, the stretching/resizing of an image results in a diminished image quality. Hence, the server side should be able to provide the same image content scaled for different resolution screens, dynamically or statically prepared, and located at different URIs. Let us consider an *abstract URI*, that is, a URI that identifies partially a resource content, e.g., at least a piece of path or name. Fig. 7a shows an example of URI transformation. Here the left side represents an abstract URI, whereas the right side is the same URI after the delivery-context adaptation process, once the screen resolution has been determined.

It is worth noting that the context adaptation process can adapt content chosen by the user, such as an image. For this reason it comes after the content adaptation process.

The second illustrative example concerns the adaptive resizing of a component. Let us consider an abstract element (pattern) called *graphical menu*, i.e., a menu with each item made of a text-label and an image-logo. In the graphical menu, the number of visible elements can be adaptively set, according to the available screen size. Fig. 7b shows an example, in which the context-adaptation process is able to resolve the concrete value for the abstract parameter *numElemVisible*. The context adaptation process is more powerful than the content adaptation one. It is possible, for instance, to introduce *abstract elements*, i.e., elements that can be included only in the structure of specific device classes. For instance, for some device with a small screen size, the screen footer and header have to be removed.

Once the context adaptation process is performed, the UIML concrete description is sent to the client, in the HTTP response. Indeed, both the content and context adaptation processes can be accomplished in a short time, considering that the UIML descriptions are usually short for a mobile interface, and are based on a series of declarative patterns which do not need to be specified in detail. Hence, the basic design rule of a coarse-grained UI modeling is that only very essential information should be detailed in a UIML description. In order to allow this process to be scaled up to thousands of users, a caching mechanism has been also implemented on the client side.

The web application cache is similar to the browser cache, i.e., a mechanism for the temporary storage of interfaces, such as UIML descriptions and multimedia content, and is employed to reduce bandwidth usage, server load, and perceived lag. In the MODIF paradigm, an application is made of a cluster of interfaces connected by events, and hence an interface could be requested many times, depending on the frequency of each event. Thus, a caching mechanism is very important to speed-up the transition between interfaces: only when the concrete UIML description is *out-of-date*, it should be removed from the cache. A concrete UIML description is out-of-date if the corresponding abstract UIML description and/or the user preferences have been updated. In order to capture when a concrete UIML description is out-of-date, we have exploited a mechanism of *automatic version synchronization*. The mechanism is based on two different timestamps: the last-modified date of the abstract UIML description and the last-received user preferences date. These two temporal instants are included in the header of the UIML file, and constitute the *version* of the interface description. Fig. 8 shows the *automatic version synchronization* process in a sequence diagram which extends the one shown in Fig. 5 to cope with caching. Here, the client sends a request of a UI by sending also the timestamps. If the description is up-to-date, the client receives a confirmation signal only. If the description is out-of-date, the entire process of building the UIML description is performed.

Another client-side caching mechanism has been implemented for the downloading of multimedia content (e.g., sound and image files). Such a mechanism starts during the parsing of the UIML document, at each URI. Here, the name of the file (within a specific application namespace) is the criterion to check whether the image has already been downloaded, and then if it is locally available.

7. The interface object model API layer

This section describes how on the client side a UIML description can be efficiently transformed into native code, keeping a uniform set of APIs for the different platforms.

In the MODIF paradigm, the client application receives a concrete UIML description, which is parsed and then rendered. UIML-based design fosters reusability, reducing redundancy, thanks to a six-way factorization of the UI elements, and a flexible reuse of fragments based on templates. In order to be efficient at runtime, an interface renderer module should be provided with a reorganized interface representation where all the information needed for each component is locally grouped. For example, to create an instance of the *Button* component, all the related information, such as style, content, behavior and structure are needed. If, for instance, the style of this button is shared with other components, it will be located in a different fragment of the UIML file. Hence, using the UIML file for rendering purposes is not efficient, because the information is often spread over the file. In theory, it

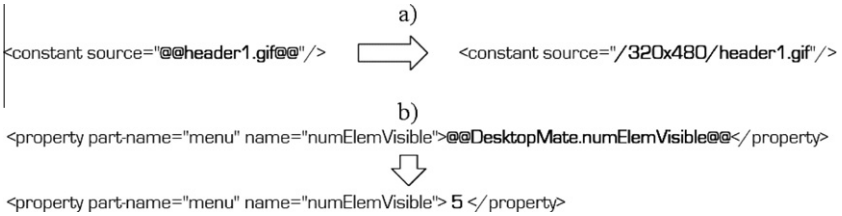

Fig. 7. An example of abstract URI (a) and parameter (b) for delivery context adaptation.

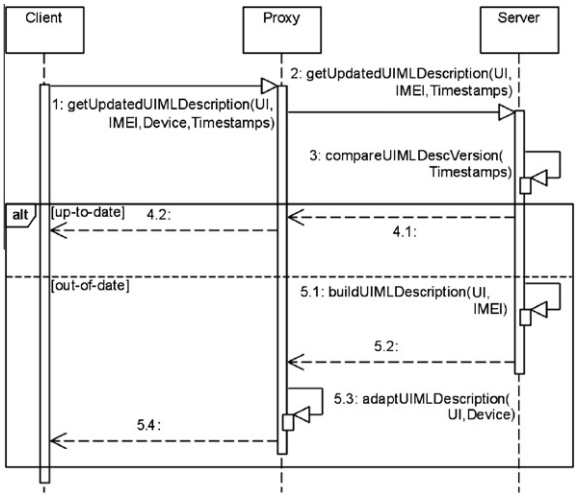

Fig. 8. The automatic version synchronization process.

might be possible to have locally the information for each component, by replicating the shared fragments. This would not be a good solution, since the file dimension would grow, thus increasing the memory request and time cost. Hence, an important step is to design a "compiled" view of the interface, suitable for the rendering engine, thus avoiding redundancy and keeping the efficiency. This means that the parser should produce an intermediate format for each interface description. According to MDA principles, this process should be based on a transformational model.

Fig. 9 shows the views of an event-driven application interface in the MODIF methodology. Here, an application in UIML is viewed as a graph of interfaces (Fig. 9i). From a given interface, an *event* typically produces the transition to another interface. Each interface can be described in terms of reusable segments of UIML elements by four basic aspects, namely, *Structure* (S), *Style* (Σ), *Content* (C), and *Behavior* (B), which can be shared and reused among different interfaces. In Fig. 9i, the Server and Proxy views of an application are shown. An event causes the restructuring of an interface, and this transition is represented by the "restructure" relation. For instance, the selection of an item in a main menu might cause the interface to become a sublevel menu, via restructuring of content and behavior, or to become a completely new service interface. It is also possible to use *Templates* (denoted as *T* in figure) as reusable pieces of interfaces. At the Client view (Fig. 9ii), UIML descriptions are transformed into an object-oriented structure, made of instances of classes derived from the abstract class *Block*, as shown in Fig. 10. Each Style, Content and Behavior segment is transformed into an object one time only, and referenced by some Structure block, via object references. Structure blocks are referenced by the Application block. The transition from one interface to another is made via a name reference, stored as a value in a behavior parameter. A name reference produces a dynamic loading of a new interface. This model allows the reuse of the interface elements, and a very efficient information access by the rendering engine.

Fig. 10 shows a class diagram representing the view of Fig. 9ii. Here, the fundamental brick is the abstract class *Block*, specialized by *ValueBlock* and *ReferenceBlock*. A *ValueBlock* is a set of pairs (*name*, *value*) that acts as a container for a generic UIML property. For instance, the pair ("align", "left"). A *ReferenceBlock* is a set of pairs (*name*, *reference*). Depending on the reference type, i.e., a reference to a *ValueBlock* or to a *ReferenceBlock*, this class is specialized by *MainReferenceBlock* and *SecondaryReferenceBlock*, respectively.

To make the interface representation clear, Fig. 11 shows a corresponding object diagram for a sample application. Here the object *Application* refers to a *MainReferenceBlock*, containing the names of all UIML files. In particular, the primary interface file is called as the application itself (*DesktopMate*), and the secondary one is a template describing the structure of a graphical menu element. Hence, for each UIML file, a corresponding *SecondaryReferenceBlock* is referred to. This object in its turn refers to a *ValueBlock*. In a *ValueBlock*, properties are grouped by component, and then efficiently accessed by the renderer. Furthermore, a *ValueBlock* can be shared among files in order to realize the reuse of pieces of interfaces.

Let us consider more specifically the dynamic view of the main engines that manage this structure, i.e., the parser and the rendering engine. At the beginning of the application, there is a prefetch process. In this process, the block structure is built considering a certain number of UIML files that can be immediately loaded, following some possible event on the main interface. It is worth noting that the process of construction of the block structure is based simply on a SAX parser, and therefore is very efficient. Once the parser has generated the internal interface representation, the rendering engine is responsible for binding each UIML component to an element of the toolkit, and hence to instantiate each component of the interface.

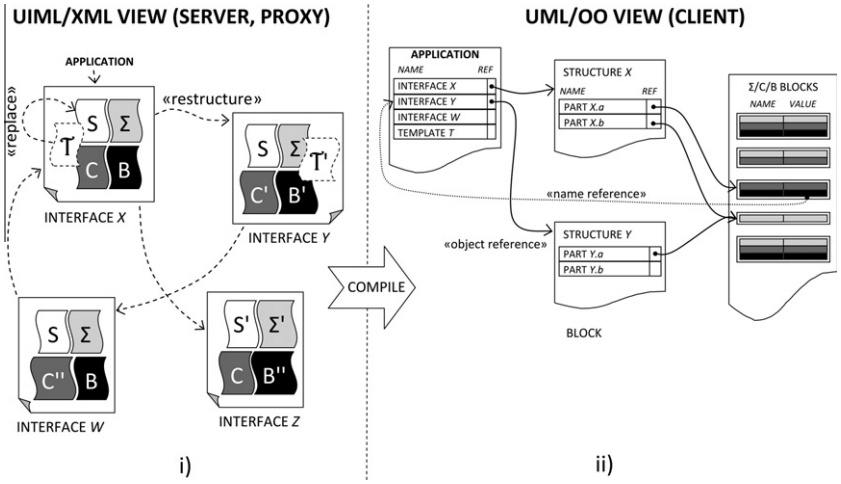

Fig. 9. Views of an event-driven application interface: server, proxy (i) and client (ii).

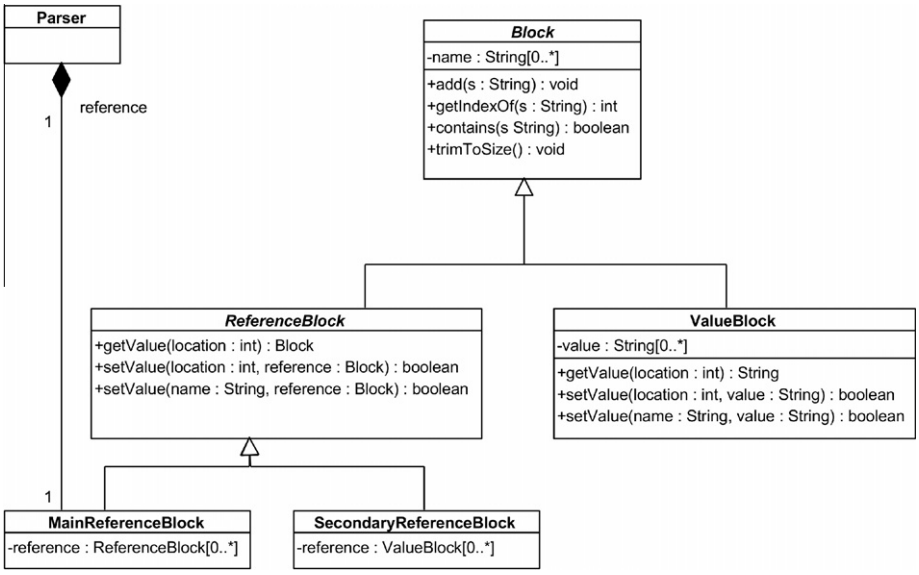

Fig. 10. The interface representation in the MODIF client (class diagram).

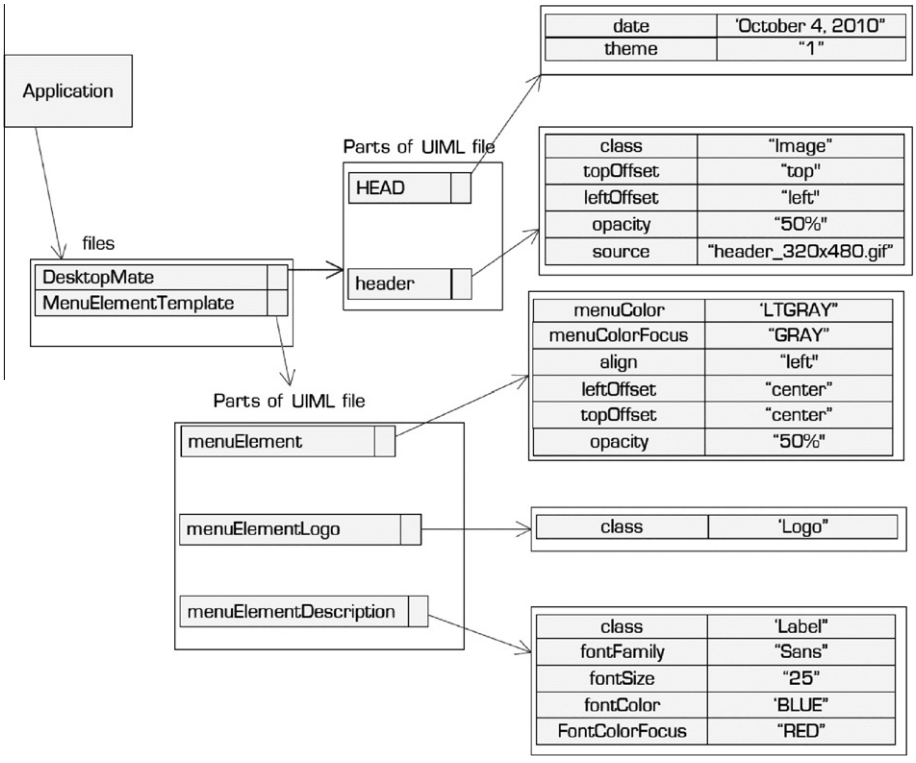

Fig. 11. An instance of interface representation in the MODIF client.

In our methodology, the generic toolkit that has been designed provides a common application-programming interface valid for all HDs, which has been inspired to the W3C DOM.[9] The proposed interface object model (IOM) is a cross-platform and language-independent convention for representing parts of a UI and interacting with them. The UI is supposed to be represented using the generic MODIF UIML toolkit, made of abstract patterns after the process of content and context adaptation. All aspects of the IOM (such as its parts) are implemented within the syntax of the native programming language, but the public interface of an IOM is specified in a corresponding API. With this generic interface, the UIML elements are mapped using a universal binding for any platform. Fig. 12 shows a top-level class diagram of the IOM API.

At the top of the structure, there is the *Interface* class, with generic factory methods to create and refer to parts. An interface is made of *Parts*. A *Part* is provided with a generic method *setAttribute*, which allows setting any property of the element. For each possible attribute, generic properties have been defined. For instance: *topOffset*, *width*, *height*, *content*, *opacity*, etc., according to

[9] http://www.w3.org/DOM/.

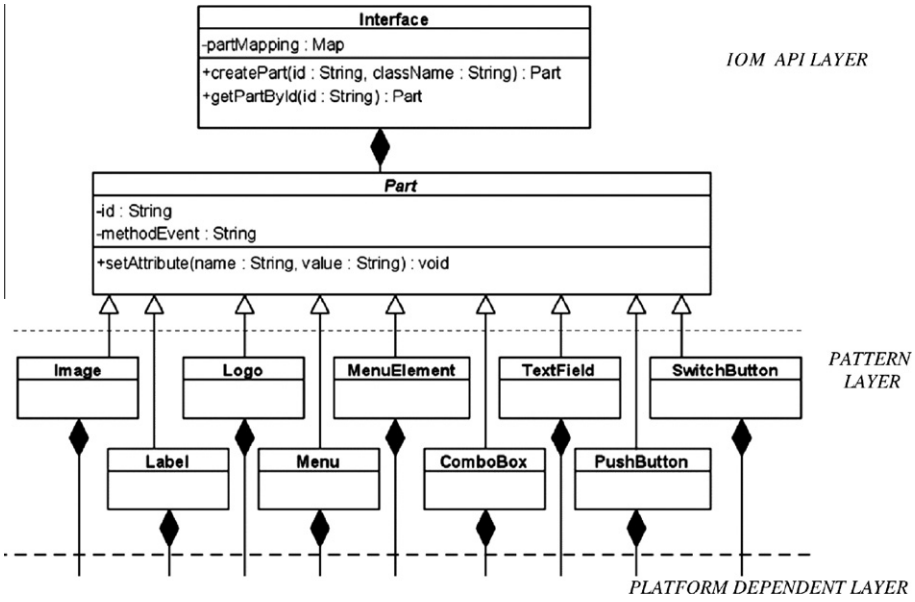

Fig. 12. Interface object model API.

the specific element. A *Part* can be specialized as a series of UI generic components, such as *Image* and *Label*. These are provided in the *Pattern Layer*. Each component is implemented using the native platform API, in the Platform Dependent Layer. For example, on Android OS, in the platform dependent layer there are *JImage*, *JLabel*, etc.; whereas on iPhone OS, there are *UIImage*, *UILabel*, etc. Some patterns are not natively provided in some platform, e.g., the *GraphicalMenu* on Blackbererry OS. In this case, the entire pattern is natively implemented, providing the same API in the Pattern Layer, in order to guarantee a universal mapping model-to-code for every platform.

8. Some implementation details of the MODIF architecture

In this section, we discuss some implementation details of the MODIF architecture. In particular, we discuss the structure of the MODIF subsystems *Server*, *Proxy* and *Client*, and how the main components of these subsystems interact with each other for carrying out the most relevant MODIF services.

8.1. The MODIF Server subsystem

Fig. 13 shows the package diagram of the *Server* subsystem, which is entirely based on the Sun GlassFish Enterprise Server [43]. Here, different notations have been used to denote modules with different levels of reuse. More specifically, a *service*-oriented module is denoted by a gear symbol and contains only application independent elements. A *program*-oriented module, denoted by a scroll icon, contains elements that a modeler/implementer may need to adapt to each different application with the assistance of an environment. A *human*-oriented module, denoted by a user icon, contains elements that are expected to be defined in an application-dependent manner.

In particular, according to the J2EE platform, packages *EntityBean* and *SessionBean* contain classes used to map the database and to implement the business logic, respectively.

The *UIMLVersionSynchronization* package implements the automatic version synchronization process. The *ImageProvider* package provides image content. The *UIMLDescription* package realizes the

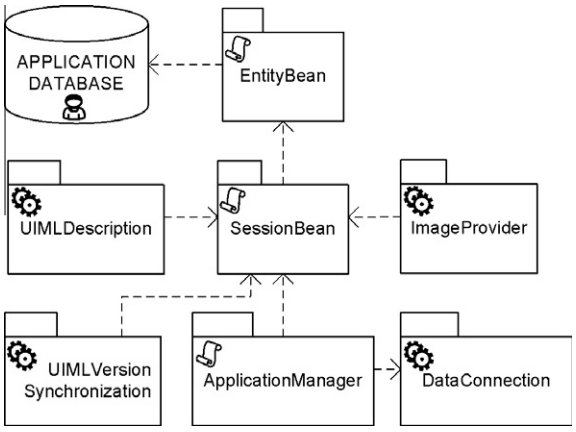

Fig. 13. Package diagram of the MODIF *Server* subsystem.

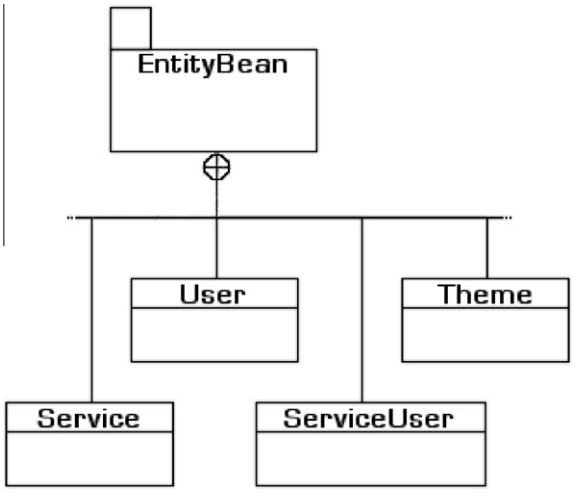

Fig. 14. An excerpt of the *EntityBean* package of the MODIF *Server* subsystem.

content adaptation process. The *DataConnection* and the *ApplicationManager* packages are responsible for the application I/O, and for processing and storing input data, respectively. It is worth noting that only the *ApplicationManager*, *EntityBean* and *SessionBean* packages are application-dependent.

The *EntityBean* package contains all the EJB components that access the information stored in the *Application Database*. Fig. 14 shows an excerpt of this package, with some important general purpose class such as *User*, *Theme*, and *Service*. Each class contains a set of management methods (e.g., *set* and *get*).

The *SessionBean* package contains the Java components to operate, via *EntityBean*, on the *Application Database*. Each component of this package is made by an interface and a corresponding concrete class, which implements the interface itself (e.g., the *ThemeSessionBean* class).

The *UIMLVersionSyncronization* package contains the EJB component which compares the versions of the server and client UIML descriptions, in the mechanism of automatic version synchronization. To this aim, the package contains: (i) a remote managing interface and the corresponding implementation; (ii) a method to access the UIML document on the server; (iii) a method to receive the XML client document. The last two methods are implemented by a SAX parser, via corresponding event handler.

The *UIMLDescription* package manages the content adaptation mechanism and the transmission of the UIML documents. More specifically, as shown in Fig. 15, the *UIMLDescription* component processes the UIML document via a DOM parser, whereas the *UIMLSequenceContent* component reads the content-related information from the *Application Database*. Finally, the *ImageProvider* package implements a service of image delivery. Images are stored on the file system.

Fig. 16 shows the interaction between the subsystem *Proxy* and the *UIMLDescription* package. For the sake of simplicity, the local interaction involving other components of the *Server* subsystem has been omitted. More specifically: a UIML description is requested by the *Proxy* (1), via the *UIMLDescriptionRemote* interface,

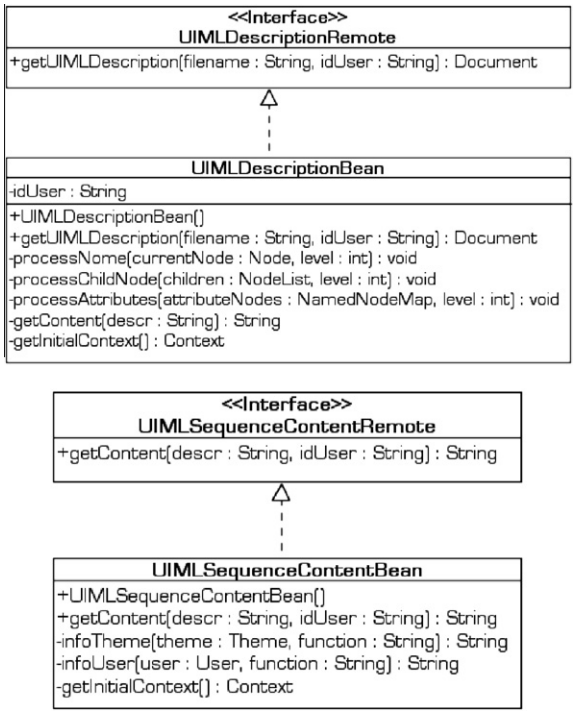

Fig. 15. Components of the *UIMLDescription* package of the MODIF *Server* subsystem.

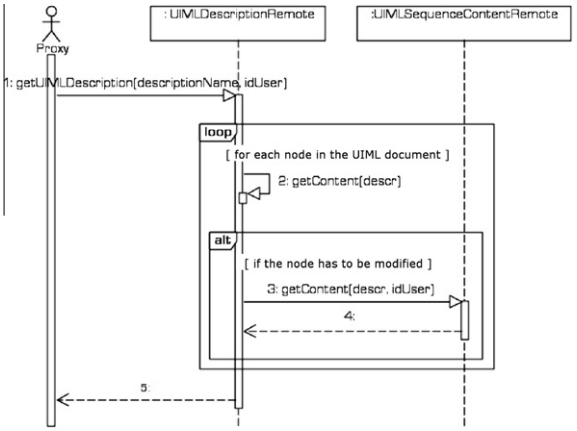

Fig. 16. Interaction between the subsystem *Proxy* and the *UIMLDescription* package.

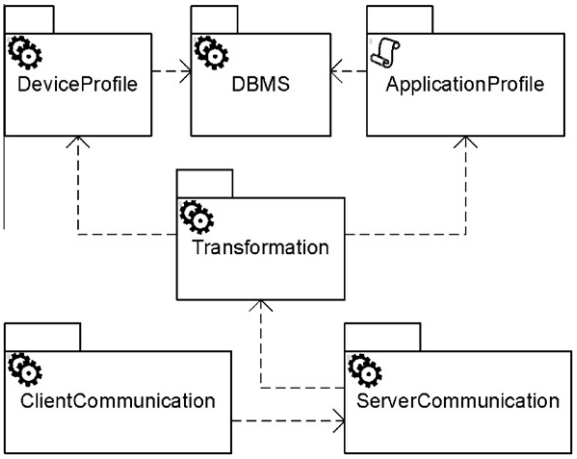

Fig. 17. Package diagram of the MODIF *Proxy* subsystem.

and then loaded from the *Server* file system; each content node is analyzed (2) and potentially modified (3 and 4) according to the content adaptation mechanism; finally, an adapted description is sent back to the *Proxy* (5).

8.2. The MODIF Proxy subsystem

Fig. 17 shows the package diagram of the MODIF *Proxy* subsystem. Here, the *DBMS* package manages the communication with

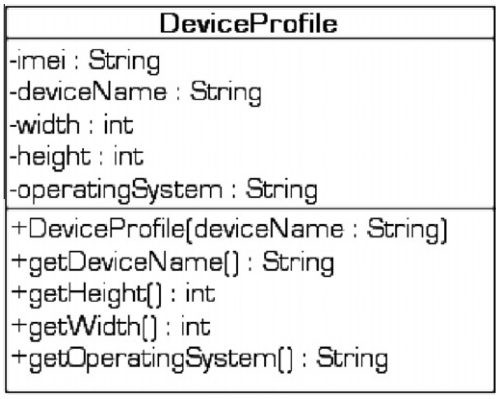

Fig. 18. An excerpt of the *DeviceProfile* class of the MODIF *Proxy* subsystem.

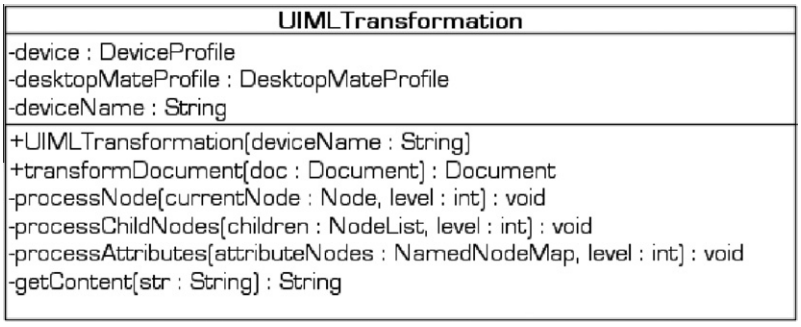

Fig. 19. The class *Transformation.UIMLTransformation* in the MODIF *Proxy* subsystem.

the database. The *DeviceProfile* package represents an abstraction of the CC/PP data and services. The *ApplicationProfile* package is strictly related to the specific mobile application. The *Transformation* package implements the context-adaptation process. Finally, the *ServerCommunication* and *ClientCommunication* packages manage the interaction with the Server and Client subsystems, respectively.

Fig. 18 shows an excerpt of the *DeviceProfile* class, in the *DeviceProfile* package, which represents the features of a HD.

The *Transformation* package is based on the *UIMLTransformation* (Fig. 19) and the *UIMLDescription* (Fig. 20) classes. The latter receives from the MODIF *Server* the DOM version of the UIML document, on which the context adaptation has been already made by the former according to the mobile device type. The *ServerCommunication* package contains classes to interact with remote objects provided by the MODIF *Server*, for example a class to retrieve the UIML descriptions. Finally, the *ClientCommunication* package includes the *ServletDescription* class, which interacts with the *Client* subsystem.

Fig. 21 shows the sequence diagram of the adaptation and delivery of the UIML description performed by the MODIF *Proxy* subsystem. For the sake of simplicity, the interaction involving other components of the *Proxy* and *Server* subsystems has been omitted. In particular: (1) the *Client* sends a request to *ServletDescription* for a contextualized UIML document; (2) a *DeviceProfile* object provides the client device properties such as the device name (3–4); a *UIMLDescription* object provides the result (5–11) after a context adaptation made for each element by a *UIMLTransformation* object (7–10). It is worth noting that the *Server* provides the *Proxy* with a DOM version of the UIML document.

8.3. The MODIF client

Fig. 22 shows the package diagram of the MODIF *Client* subsystem. Mostly, packages that are application-independent are provided, with the most important dependencies. In particular, the *ApplicationController* package manages threads for the application logic. The *Communicator* package contains services to communicate with the *Proxy* subsystem. The *Serializer* package generates the XML files with business data to be sent to the proxy. This is done during the automatic version synchronization process. The *Storing* package holds the images and the UIML descriptions. The *Model* package contains a set of classes that model the interface state, i.e., classes for the intermediate block structure. The *Parser* package processes the concrete UIML description and generates the intermediate block structure. The *Renderer* package is responsible for the instantiation of graphical components, based on the intermediate block structure. The *UIPart* package contains classes that implement the IOM API. The *UIController* package manages the user–device interactions. Finally, the *ApplicationLogic* package contains the classes specific to the application.

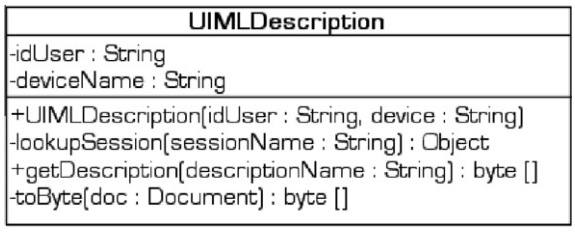

Fig. 20. The class *Transformation.UIMLDescription* in the MODIF *Proxy* subsystem.

For the sake of brevity, in the following, we only consider the most important classes of each package related to the transformation and rendering processes of an interface description. Fig. 23 shows the *ServerConnection* class in the *Communication* package. In particular the class is able to send a request to the *Proxy* for a UIML description, and to check the version of such description. Fig. 24 shows the *ParserUIML* class in the *Parser* package, which is responsible for processing the UIML document. Fig. 25 and Fig. 26 show the *InterfaceBuilder* and *InterfaceDisplay* classes, respectively. The former is responsible for the instantiation of the components of the user interface, according to the compiled description of the interface (Fig. 9.ii), whereas the latter is responsible for the instantiation of the layout of the user interface itself, including the above components. The *UIPart* package contains the *Interface* class, discussed in Fig. 12. Finally, the *DescriptionLoader* class in the *ApplicationController* package is shown in Fig. 27. This class is in charge of reading names of UIML descriptions from a buffer, sending a request to the *Server* only in case of out-of-date descriptions.

Fig. 28 shows the sequence diagram of the interface creation process. Here, *AplicationController* sends a request to *InterfaceBuilder* and to *InterfaceDisplay* to create (1) and render (6) the interface, respectively. The interface creation is made by creating each part (2–4).

Fig. 29 shows the sequence diagram of the UIML description loading process. For the sake of simplicity, the interaction involving other components of the *Client* and *Proxy* subsystems has been omitted. Here, *DescriptionNameBuffer*, *DescriptionBuffer*, and *ImageBuffer* are buffer objects used to allow an efficient communication between related threads. For instance, the thread which is in charge of loading images from the *Server* stops if the buffer of image names is empty; the thread which is in charge of loading UIML description stops if *DescriptionBuffer* is empty.

More specifically, the *DescriptionLoader* accesses the *DescriptionNameBuffer* (1–2); in case of out-of-date description, *it* sends a request to *ServerConnection* for an up-to-date description (3–4), which is put into the *DescriptionBuffer*, to be processed by the *ParserUIML*. Each image is loaded separately (9–10).

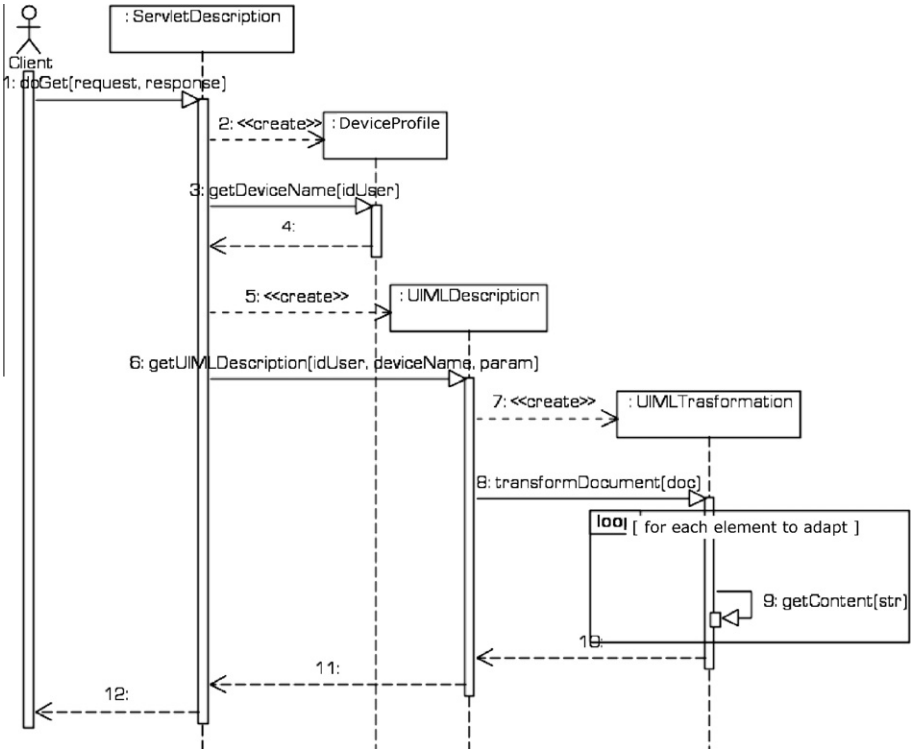

Fig. 21. Sequence diagram of the adaptation and delivery of the UIML description performed by the MODIF *Proxy* subsystem.

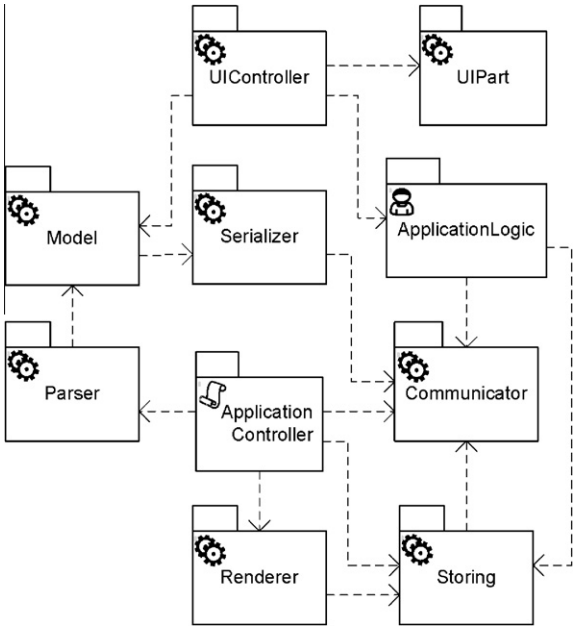

Fig. 22. Package diagram of the MODIF *Client* subsystem.

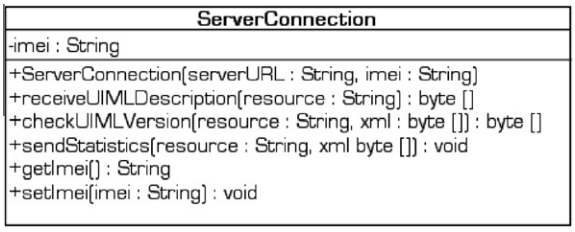

Fig. 23. The *Communication.ServerConnection* class in the MODIF *Client* subsystem.

The MODIF *Client* subsystem has been implemented for two different mobile OS: *Android* and *iPhone*. To test the overall framework, two real-world applications have been designed and implemented. In the following, we first illustrate both applications, and then we show some assessment of the framework properties.

9. Case studies

The first application is a customer application, called *DesktopMate*. It is a resource launcher for HDs, i.e., an advanced bookmark manager with the possibility of parameterized launch, and the capacity of tracking the launches history. At the interface level, the UI of *DesktopMate* can be designed by using three different kinds of UIML files. The first file describes the main interface, whose structure is represented in the mock-up of Fig. 30-a. More specifically, it consists of header, footer, graphical menu, and background. The second UIML file is a template that describes an item of the graphical menu in terms of structure and style. In particular, an item is made of a *Logo* and a *Label*. This template is referred to by the main interface so as to build the main menu, as well as by the third file, which describes the sublevel interface. The sublevel interface expresses a sublevel menu. A sublevel menu can be derived from the main menu via a change of the content and behavior segments, leaving the same structure and style segments. It is worth noting that the description of a UIML interface by four separate segments allows a high level of reusability of the interfaces. Fig. 30-b and Fig. 30-c show the main interfaces on *Android* and *iPhone*, respectively.

The second example is a business application, called *Tracking*. It is a service that tracks time and position of GPS-enabled HDs in real time. The configuration interface allows setting: (i) the frequency of track upload, by means of a *ComboBox*; (ii) the sampling period, by means of a *ComboBox*; (iii) track name and description,

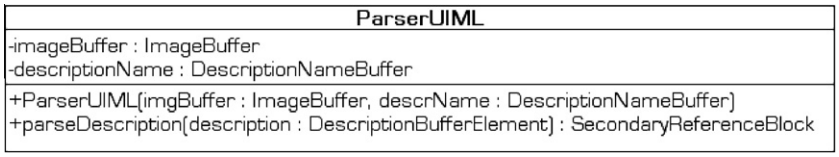

Fig. 24. The *Parser.ParserUIML* class in the MODIF *Client* subsystem.

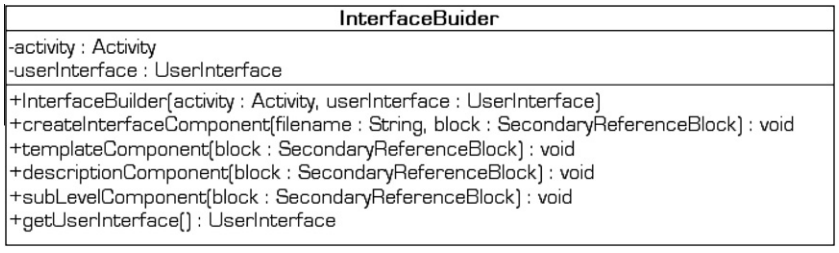

Fig. 25. The *Renderer.InterfaceBuilder* class in the MODIF *Client* subsystem.

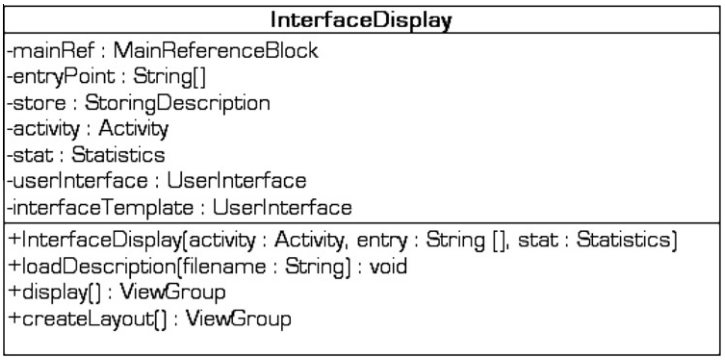

Fig. 26. The *Renderer.InterfaceDisplay* class in the MODIF *Client* subsystem.

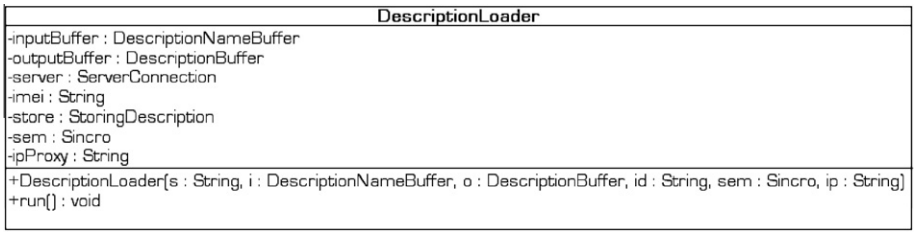

Fig. 27. The *ApplicationController.DescriptionLoader* class in the MODIF *Client* subsystem.

with two *TextField*; (iv) an alternative storage unit (if available on the device), by means of a *SwitchButton*; and (v) the start/stop of the tracking, by means of two *static buttons*, or a single *dynamic button*.

The main interface comprises also a background and a header. Fig. 31 shows the mock-up (a), and the *Android* (b) and *iPhone* (c) versions.

It is important to note that, although the abstract interface (a) is the same for all OSs, the concrete interfaces (b, c) can be very different, depending on the implementation of the abstract pattern on the native platform.

In the *Tracking* application, we also experienced a form of adaptation that is totally controlled on the client-side. More specifically, a *SwitchButton* allows choosing the SD Card as an alternative storage unit, if the card is plugged in. This availability depends on neither the device model nor the user preferences. It depends on the physical availability of a SD Card on a specific device instance. The Proxy server cannot manage this information. Hence, in the UIML model, the behavior of the *SwitchButton* has been set in order to allow the dynamic detection of the SD card. In case of unavailability, the button will not appear on the interface.

9.1. Experimental use of MODIF

During the development of the applications, we have also performed a number of interviews with two different groups of developers for evaluating the effectiveness of the proposed framework. A group (*developer group*) composed of three medium-skilled developers, who carried out the applicative task of implementing *DesktopMate* and *Tracking*, for both *iPhone* and *Android* OSs, using the proposed methodology. A group (*observer group*) composed of three high-skilled developers belonging to two different

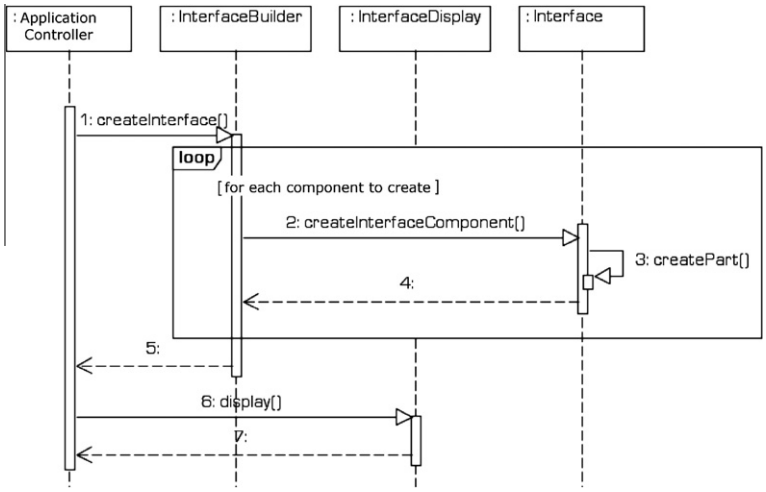

Fig. 28. Sequence diagram of the interface creation process in the MODIF *Client* subsystem.

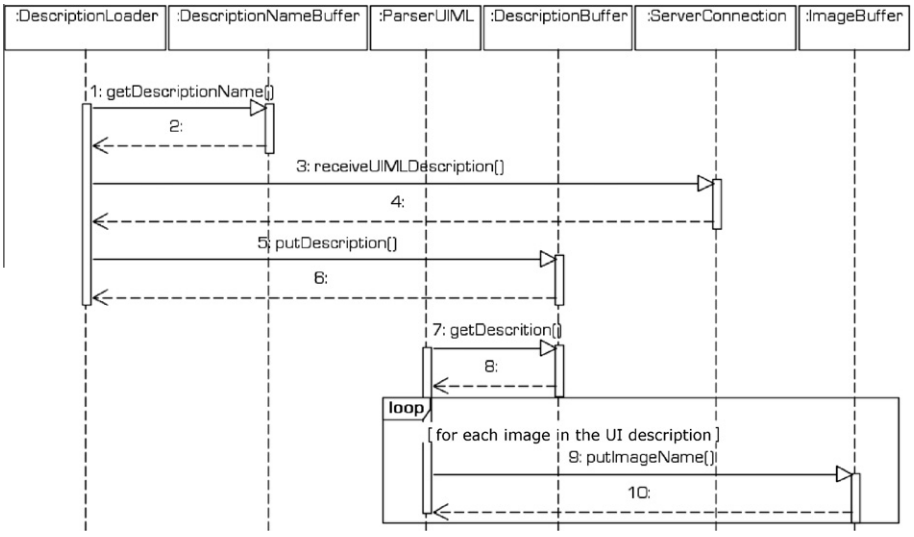

Fig. 29. Sequence diagram of the UIML description loading process in the MODIF *Client* subsystem.

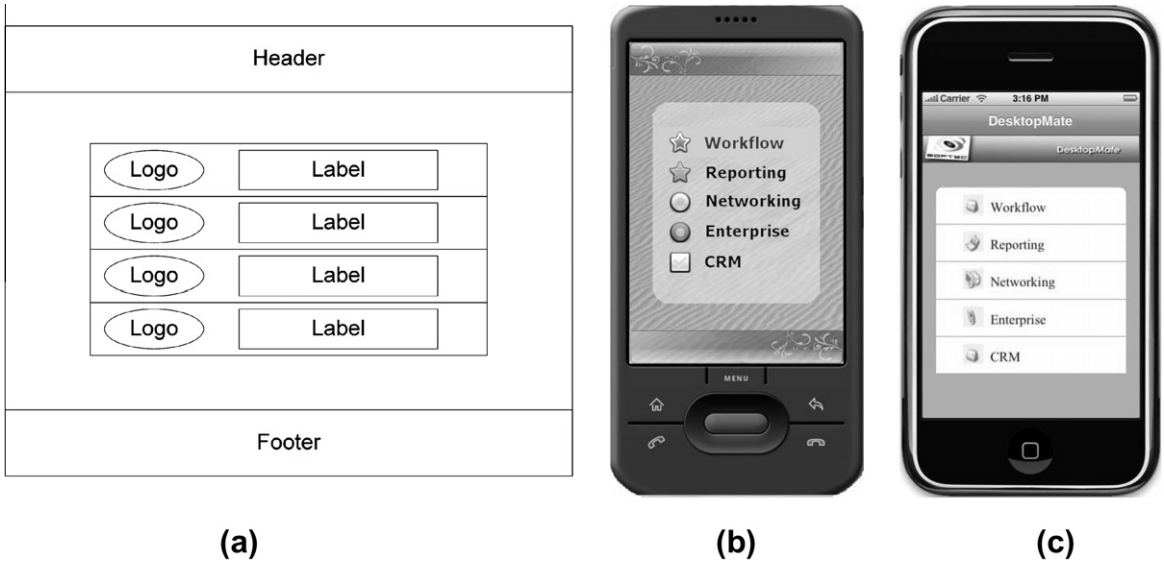

Fig. 30. The *DesktopMate* application: mock-up (a), *Android* (b) and *iPhone* (c) versions.

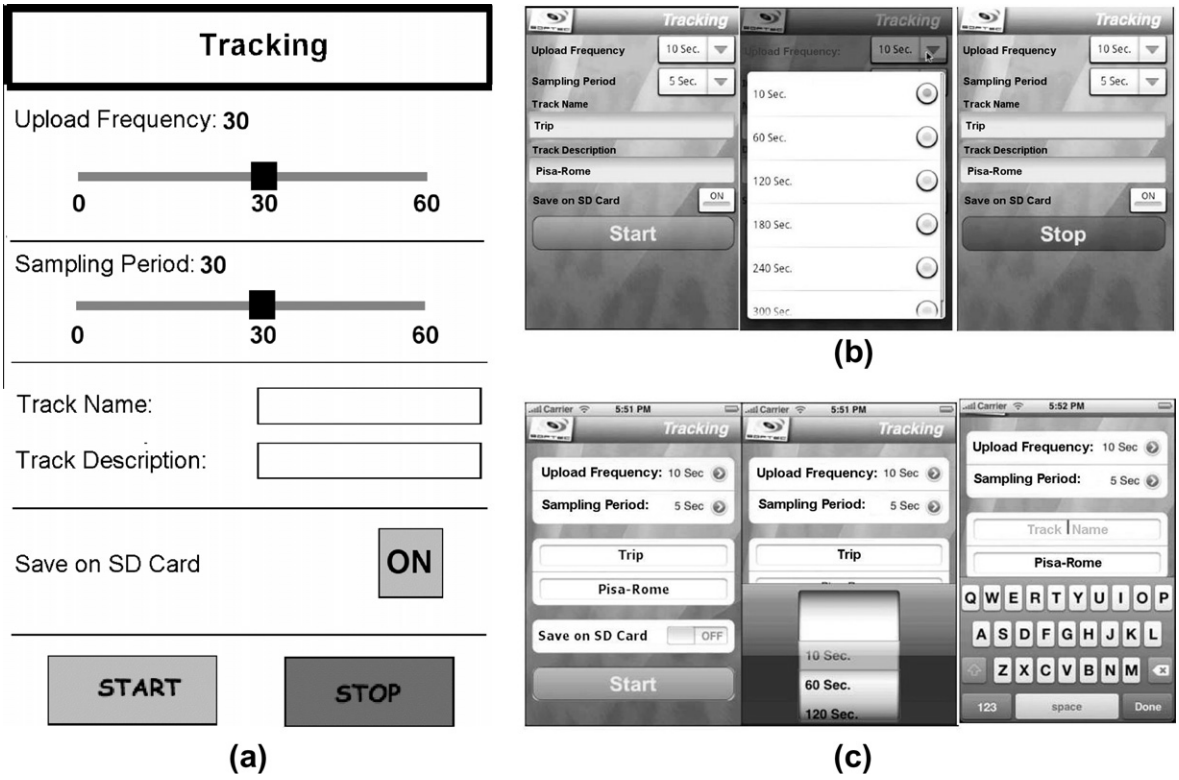

Fig. 31. The *Tracking* application: mock-up (a), *Android* (b) and *iPhone* (c) versions.

companies, who performed an accurate analysis of the activities performed by the first group.

The overall experimental activity has been split into two phases. In the first phase, two developers implemented independently of each other the *DesktopMate* on *iPhone* and *Android* OS, respectively. At the end of the first phase, both the two developers and the observers provided their first assessments. In the second phase, the third developer implemented *Tracking* on both *iPhone* and *Android* OS. At the end of the second phase, both the developer and the observers provided their final assessments. In particular, the assessment focused on the important properties for UI languages and tools discussed in [15]. Fig. 32 shows the general assessment for each considered property. The rating (*bad*, *medium*, *good*, and *very good*) is intended to be comparative with respect to the development of ad-hoc solutions, and averaged with respect to the opinion of the two groups.

More specifically, the *Effort for native platform* is the effort that the developer performed to cope with problem solving related to native platforms. It can be noticed that in the first phase this effort was higher than expected, because the development of a general framework requires a deeper knowledge of the platform, with respect to an ad-hoc solution. However, in the second phase this effort has been lower than expected, because in the second phase the developers have taken advantage of the reuse of API components developed in the first phase. The *Effort for application task* is the effort spent for developing the specific application rather than for developing the framework components. This effort has been usual in the first phase, because each component has been natively implemented in order to add some component to the framework. In the second phase, this effort has been lower than expected, because some reusable framework components had been already developed in the first phase. This has allowed the developer to focus only on the specific application requirements.

Expressiveness is the ability to express via an abstraction any real world aspect of interest. This property has been increasingly better than expected, mainly for the expressiveness of the UIML standard notation, with respect to an ad-hoc XML description. *Readability* has been evaluated very good in both the phases, thanks to the readability of UIML. Due to the engineered transformation engines (content and context adaptation, parsing and

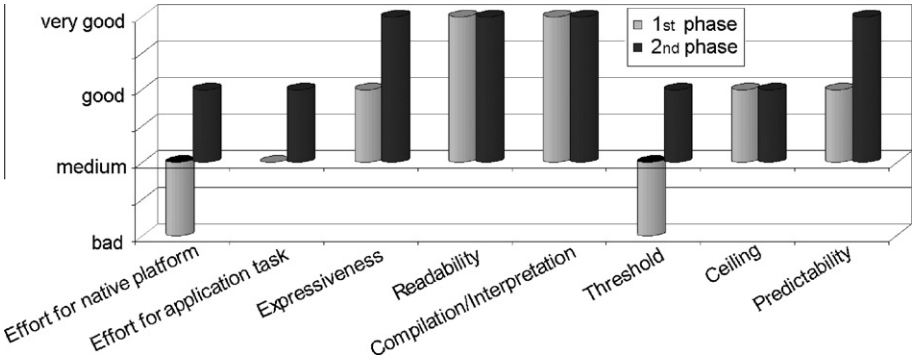

Fig. 32. Qualitative assessment of the proposed methodology.

rendering), the memory and time cost has been lower than in ad-hoc solutions. Thus, *Compilation/Interpretation* has been evaluated very good. *Threshold* and *ceiling* are two connected metrics in evaluating tools. *Threshold* refers to the difficulty of learning how the tool should be used. *Ceiling* expresses the potentialities of the tool, that is, how much can be done using the tool [8,25]. An important challenge is to develop tools characterized by a low difficulty of learning (i.e., a good threshold) and a high potentiality (i.e., a good ceiling) [25]. In general, some of the most successful tools are characterized by either low threshold and low ceiling, or high threshold and high ceiling [25]. In our case, the learning of a new language in the first phase has required more time than expected. However, in the second phase the learning has required less time than expected, thanks to the example developed in the first phase, made available to the third developer. Thus, the threshold has been assessed bad and good in the first and second phases, respectively. The ceiling is considered good in both the phases, i.e., higher than expected, because the proposed methodology does not limit the kinds of interfaces that can be produced. Indeed, the abstract toolkit is richer than the native toolkit for most mobile platforms, thus providing a wider range of possibilities to the designer. Finally, automatic techniques that are sometimes unpredictable are poorly received by designers. Thus, *Predictability* is a desirable feature: a high level of predictability reduces the probability of re-iteration, and improves early design planning and exploration. In our methodology, the designer has the control of what should be modified in terms of high-level notations for producing a desired change at the low-level.

10. Conclusions and future work

In this paper we have proposed a novel model-based methodology for developing event-driven applications for handheld devices. The methodology relies on an abstract mobile device, content and context adaptation mechanisms based on user preferences and standardized context of delivery, respectively, a uniform set of client-side APIs and an efficient transformational model. Further, the methodology employs two important standard specifications. The first specification is UIML, which allows a canonical description of an event-driven application, providing a proper abstraction level. The second specification is CC/PP, which offers a data structure and a sample vocabulary for describing the delivery context.

The paper also includes the analysis of transformational models and the design of MODIF, an architectural framework for adaptation, based on the device capabilities and the user preferences. Finally, some case studies have been discussed so as to show a practical use of MODIF. The experimental activity has highlighted important properties of the framework, such as automation of the lifecycle, expressiveness and readability of the representation, efficiency of the compilation/interpretation, fast learning curve, and predictability. The client side architecture has been implemented on *iPhone* and *Android* OS. In the near future, the framework will be adopted to develop applications on other important mobile OSs such as *BlackBerry*, *WindowsMobile* and *Nokia*.

Acknowledgements

This work was supported by the MOVAS Lab, a joint project at the University of Pisa between academy and industry. The authors would like to thank the company Softec S.p.a. Prato (Italy) for financial and technical support.

References

[1] M.F. Ali, A Transformation-Based Approach to Building Multi-Platform User Interfaces Using a Task Model and the User Interface Markup Language. Doctoral Thesis, UMI Order Number: AAI3172932, Virginia Polytechnic Institute & State University, Virginia, USA, 2005.
[2] M.F. Ali, M.A. Pérez-Quiñones, M. Abrams, E. Shell, Building Multi-Platform User Interfaces with UIML, in: Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces (CADUI'02), Valenciennes, France, May 2002, pp. 255–266.
[3] Apple, Inc., App Store, http://www.apple.com/iphone/appstore (accessed April 2011).
[4] Apple, Inc., The Objective-C Programming Language, http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf (accessed April 2011).
[5] J. Arlow, I. Neustadt, Enterprise Patterns and MDA, Addison-Wesley, Canada, 2003.
[6] G. Banavar, L. Bergman, R. Cardone, V. Chevalier, Y. Gaeremynck, F. Giraud, C. Halverson, S. Hirose, M. Hori, F. Kitayama, G. Kondoh, A. Kundu, K. Ono, A. Schade, D. Soroker, K. Winz, An authoring technology for multidevice Web applications, IEEE Pervasive Computing 3 (3) (2004) 83–93, http://dx.doi.org/10.1109/MPRV.2004.1321033.
[7] J.O. Borchers, A Pattern Approach to Interaction Design, John Wiley & Sons, Inc., New York, NY, 2001.
[8] P.F. Campos, N.J. Nunes, CanonSketch: A User-Centered Tool for Canonical Abstract Prototyping, in: R. Bastide, P. Palanque, J. Roth (Eds.), Proceedings of the 11th IFIP International Workshop on Design, Specification and Verification of Interactive System (EHCI-DSVIS 2004), LNCS 3425, 2005, pp. 146–163.
[9] ECMA, International, ECMA-334, C# Language Specification, fourth ed., June 2006, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.
[10] J. Eisenstein, A. Puerta, Adaptation in automated user-interface design, in: Proceedings of the 5th International Conference on intelligent User interfaces (IUI '00), New Orleans, Louisiana (USA), January 2000, pp. 74–81, http://dx.doi.org/10.1145/325737.325787.
[11] J. Eisenstein, J. Vanderdoncki, A. Puerta, Adapting to Mobile Contexts with User-Interface Modeling. Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000), Monterey, CA, USA, December 2000, pp. 83–92.
[12] J. Eisenstein, J. Vanderdonckt, A. Puerta, Applying model-based techniques to the development of UIs for mobile computers, in: Proceedings of the 6th ACM international Conference on intelligent User Interfaces (IUI'01), Santa Fe, NM, USA, January 2001, pp. 69–76, http://dx.doi.org/10.1145/359784.360122.
[13] K. Gajos, D.S. Weld, SUPPLE: automatically generating user interfaces, in: Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI'04), Funchal, Madeira, Portugal, January 2004, pp. 93–100.
[14] K.Z. Gajos, J.O. Wobbrock, D.S. Weld, Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces, in: Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems (CHI'08), Florence, Italy, April 2008, pp. 1257–1266, http://dx.doi.org/10.1145/1357054.1357250.
[15] E. Georgieva, T. Georgiev, Methodology for mobile devices characteristics recognition, in: Rachev, A. Smrikarov, D. Dimov (Eds.), Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07), vol. 285, Bulgaria, June 2007, pp. 1–6, http://dx.doi.org/10.1145/1330598.1330674.
[16] Google, Inc., Android Market, http://www.android.com/market (accessed April 2011).
[17] R.G. Hanumansetty, Model Based Approach for Context Aware and Adaptive UI Generation, Master Thesis, VT-masters-2004-08-26, Virginia Polytechnic Institute & State University, Virginia, USA, 2004.
[18] V. López-Jaquero, F. Montero, F. Real, Designing user interface adaptation rules with T: XML, in: Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '09), Sanibel Island, FL, USA, 2009, pp. 383–388, http://dx.doi.org/10.1145/1502650.1502705.
[19] K. Luyten, K. Thys, J. Vermeulen, K. Coninx, A generic approach for multi-device user interface rendering with UIML, in: Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces (CADUI'06), Bucharest, Romania, June 2006, pp. 175–182.
[20] M. Mcrae, OASIS User Interface Markup Language (UIML) TC, The Relationship of the UIML 3.1 Spec. to Other Standards/Working Groups, White paper. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml, July 2004.
[21] J. Meskens, J. Vermeulen, K. Luyten, K. Coninx, Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me, in: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'08), Napoli, Italy, May 2008, pp. 233–240.
[22] Microsoft, Corp., Windows Phone, http://marketplace.windowsphone.com (accessed April 2011).
[23] G. Mori, F. Paternò, C. Santoro, Tool support for designing nomadic applications, in: Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI '03), Miami, FL, USA, 2003, pp. 141–148, http://dx.doi.org/10.1145/604045.604069.
[24] G. Mori, F. Paternò, C. Santoro, Design and development of multidevice user interfaces through multiple logical descriptions, IEEE Transaction on Software Engineering 30 (8) (2004) 507–520, http://dx.doi.org/10.1109/TSE.2004.40.
[25] B. Myers, S.E. Hudson, R. Pausch, Past present and future of user interface software tools, ACM Transactions on Computer Human Interaction 7 (1) (2000) 3–28, http://dx.doi.org/10.1145/344949.344959.
[26] J. Nichols, B.A. Myers, M. Higgins, J. Hughes, T.K. Harris, R. Rosenfeld, M. Pignol, Generating remote control interfaces for complex appliances, in: Proceedings

of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST'02), Paris, France, October 2002, pp. 161–170, http://dx.doi.org/10.1145/571985.572008.
[27] J. Nichols, B.A. Myers, K. Litwack, Improving automatic interface generation with smart templates, in: Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI'04), Funchal, Madeira, Portugal, January 2004, pp. 286–288, http://dx.doi.org/10.1145/964442.964507.
[28] J. Nichols, B.A. Myers, B. Rothrock, UNIFORM: automatically generating consistent remote control user interfaces, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'06), Montréal, Québec, Canada, April 2006, pp. 611–620, http://dx.doi.org/10.1145/1124772.1124865.
[29] Nokia, Corp., Ovi Store, http://store.ovi.com, 2011 (accessed April 2011).
[30] S. Nylander, M. Bylund, Providing device independence to mobile services, in: Universal Access: Theoretical Perspectives, Practice, and Experience, Lecture Notes in Computer Science, vol. 2615, Springer, Berlin/Heidelberg, 2003, pp. 465–473.
[31] S. Nylander, M. Bylund, A. Waern, Ubiquitous service access through adapted user interfaces on multiple devices, Personal and Ubiquitous Computing 9 (3) (2005) 123–133, http://dx.doi.org/10.1007/s00779-004-0317-4.
[32] OASIS, User Interface Markup Language (UIML) Version 4.0, Committee Specification, http://docs.oasis-open.org/uiml/ns/uiml4.0, 2009 (1 May 2009).
[33] OMA (Open Mobile Alliance), User Agent Profile Specification, http://www.openmobilealliance.org/tech/affiliates/wap/wap-248-uaprof-20011020-a.pdf, 20 October 2001.
[34] OMA (Open Mobile Alliance), Wireless Markup Language (WML) 2.0, http://www.openmobilealliance.org/tech/affiliates/wap/wap-238-wml-20010911-a.pdf, 11 September 2001.
[35] F. Paternò, C. Santoro, One model, many interfaces, in: Proceedings of the Fourth International Conference on Computer-aided Design of User Interfaces (CADUI 2002), Valenciennes, France, May 2002, pp. 143–154.
[36] F. Paternò, C. Santoro, A unified method for designing interactive systems adaptable to mobile and stationary platforms, Interacting with Computers 15 (3) (2003) 349–366.
[37] F. Paternò, C. Santoro, J. Mantyjarvi, G. Mori, S. Sansone, Authoring pervasive multimodal user interfaces, International Journal of Web Engineering and Technology 4 (2) (2008) 235–261, http://dx.doi.org/10.1504/IJWET.2008.018099.
[38] C. Phanouriou, UIML: A Device-Independent User Interface Markup Language, Ph.D. Thesis, Virginia Polytechnic Institute, Blackburg, VA, USA, 2000.
[39] A. Puerta, J. Eisenstein, XIML: A Universal Language for User Interfaces, *RedWhale Software*, Technical Report, 2001, Available at: http://www.ximl.org/pages/docs.asp (accessed April 2011).
[40] A. Puerta, J. Eisenstein, XIML: a common representation for interaction data, in: Proceedings of the Seventh International Conference on Intelligent User Interfaces (IUI '02), San Francisco, CA, USA, January 2002, pp. 214–215, http://dx.doi.org/10.1145/502716.502763.
[41] RIM (Research In Motion), Lim, BlackBerry App World, 2011, http://www.blackberry.com/appworld (accessed April 2011).
[42] SUN, Microsystems, Inc., Java Platform, Micro Edition, 2010, http://java.sun.com/javame (accessed April 2010).
[43] SUN, Microsystems, Inc., Sun GlassFish Enterprise Server, 2011, http://developers.sun.com/appserver (accessed April 2011).
[44] S. Trewin, G. Zimmermann, G. Vanderheiden, Abstract user interface representations: how well do they support universal access, in: Proceedings of the 2003 Conference on Universal Usability (CUU '03), Vancouver, British Columbia, Canada, November 2003, pp. 77–84, http://dx.doi.org/10.1145/957205.957219.
[45] J. Vanderdonckt, Model-driven engineering of user interfaces: promises, successes, failures, and challenges, in: Proceedings of Romanian National Conference of Human–Computer Interaction (ROCHI'08), Iasi, Romania, 2008, pp. 1–10.
[46] J. Vanderdonckt, L. Bouillon, N. Souchon, Flexible reverse engineering of web pages with VAQUISTA, in: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE '01), Washington, DC, USA, October 2001, pp. 241.
[47] W3C, Compact HTML (C-HTML), http://www.w3.org/TR/1998/NOTE-compactHTML-19980209, 9 February 1998.
[48] W3C, Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0, http://www.w3.org/TR/CCPP-struct-vocab, 15 January 2004.
[49] W3C, Voice eXtensible Markup Language (V-XML) 2.0, http://www.w3.org/TR/voicexml20, 16 March 2004.
[50] W3C, Delivery Context Overview for Device Independence, http://www.w3.org/TR/di-dco, 20 March 2006.
[51] W3C, Device Description Repository Core Vocabulary, http://www.w3.org/TR/2008/NOTE-ddr-core-vocabulary-20080414, 14 April 2008.

Mario G.C.A. Cimino received the Ph.D. degree in Information Engineering from the University of Pisa (Italy) in 2007. In 2006, he spent six months as a visiting Ph.D. student in the Electrical and Computer Engineering Research Facility of the University of Alberta, Edmonton (Canada). He co-organized three editions of the Workshop on Computational Intelligence for Personalization in Web Content and Service Delivery. He is reviewer of research projects funded by the Czech Science Foundation. He is a member of the Editorial Board of the International Journal of Information Science and Computer Application. Since January 2007, he is with the Department of Information Engineering of the University of Pisa, as a member of the Computational Intelligence Group, and a research fellow in the Competence Centre on MObile Value Added Services (MOVAS). He is currently involved in the fields of Mobile Information Systems, Business Process Analysis, and Computational Intelligence. He is (co-) author of more than 20 publications.

Francesco Marcelloni received the Laurea degree in Electronics Engineering and the Ph.D. degree in Computer Engineering from the University of Pisa in 1991 and 1996, respectively. He is currently an associate professor at the Faculty of Engineering of the University of Pisa. He has co-founded the Computational Intelligence Group at the Department of Information Engineering of the University of Pisa in 2002. Further, he is the founder and head of the Competence Centre on MObile Value Added Services (MOVAS). His main research interests include mobile information systems, energy-efficient data compression and aggregation in wireless sensor nodes, multi-objective evolutionary fuzzy systems, clustering algorithms, web user profiling, system modeling, pattern recognition and signal analysis. He has co-edited two volumes, two journal special issues, and is (co-)author of a book and of more than 150 papers in international journals, books and conference proceedings. He has served as TPC co-chair of the ninth International Conference on Intelligent Systems Design and Applications (ISDA'09), and as general co-chair of ISDA'10. Currently, he serves as associate editor of three international journals and as TPC co-chair of the ISDA'11, Cordoba, Spain. Further, he is one of the co-organizers of the third Workshop on Computational Intelligence for Personalization in Web Content and Service Delivery (Cordoba, Spain). In 2011, he has been keynote speaker at the 5th IEEE International Workshop on Genetic and Evolutionary Fuzzy Systems, April 15, 2011, Paris, and will hold a plenary talk at the third International Conference of Soft Computing and Pattern Recognition (SoCPaR 2011), Dalian, China, October 14–16. He has been the main investigator of several projects supported by the European Commission, the Italian Ministry, the Tuscany region, and many private companies.