

SEFAGI: Simple Environment For Adaptable Graphical Interfaces: Generating user interfaces for different kinds of terminals

Frédérique Laforest, Tarak Chaari

LIRIS, Bat B. Pascal INSA 7 av Capelle 69621 Villeurbanne Cedex
federique.laforest@liris.cnrs.fr, tarak.chaari@liris.cnrs.fr

Abstract. The SEFAGI project takes place in domains where many different user interfaces are needed in the same application. Instead of manually developing all the required windows, we propose a platform that automatically generates the needed code from high level descriptions of these windows. Code generation is done for standard screens and for small screens on mobile terminals. New windows are automatically taken in charge by an execution layer on the terminal. Data adaptation to the different terminals is also provided. A platform-independent window description language has been defined.

Introduction

Nowadays, applications require more and more user interfaces. This can be due to different reasons. A large number of user profiles is one of them. The accentuation of distributed applications implies a larger number of end-users, and thus we have to take into account more and more user profiles. Medical applications are a good example to illustrate this evolution. The first computer based medical applications were dedicated to secretaries. Doctors on technical platforms (in radiology or biology) have then used the computer. Other doctors, nurses, paramedical practitioners and even patients are now contributing to the electronic medical record management. Today, home care projects aim to provide access to a common patient record. Practitioners coming at home or receiving the patient in office have access to the common electronic patient record. According to their job and specialty, they are interested in a specific data set to be shown or captured in a specific manner. They also have specific access rights on data.

The literature often explains that user interface development requires a high percentage of the entire application development time [1] and thus it constitutes a bottleneck for application's development life cycle. This awkward problem tends to grow as the user interfaces number grows in applications.

In SEFAGI, we tend to solve this problem by providing a platform that allows:

1. The construction of windows without any human coding,
2. The integration of new windows into applications,
3. The adaptation of windows to terminal capabilities.

For the first point, SEFAGI provides a library of predefined composite visual widgets (called panels) that guarantees the graphical construction of windows. The user associates Web services to panels to define an abstract window description stored in an XML file. SEFAGI automatically computes the executable code from this description. Then the window can be downloaded into the terminal. A library of predefined panels has been developed in our prototype. We have also included in our architecture a hierarchical services repository that stores the description of the available services.

For the second point, new windows can be added into applications on end-user terminals at any time. No need for the application to be entirely built before providing it to end-users. This can be compared to incremental products delivery. This way of working is interesting in such distributed and multi-user systems. The application is built incrementally, adding new windows at runtime or even new users profiles when necessary. Modifications on existing windows can also be done by modifying the abstract window description, re-generating the corresponding code and replacing the window code in each terminal when they connect to the system. This ensures an incremental design of windows, starting with a basic interface and refining needs on the run.

For the third point, code generation ensures adaptation to the terminal characteristics. For the moment, we have defined three main profiles: classical PC screens, very limited mobile terminals (phones) and mobile terminals (PDAs). Windows code is generated from the abstract window description taking in consideration hardware and software characteristics of the terminal. To ensure this adaptation, we have defined transformation rules to adapt the code and the data of the different windows.

After a rapid tour of related works on user interface generation, we present our abstract window description language. The next section explains the SEFAGI architecture. In section 5, we present the executable code generation and the adaptation rules that we have defined. The last section presents an example of generated windows. A conclusion makes a synthesis of this proposition and shows the new directions that we have taken to improve our work.

This work has been funded by the Rhône-Alpes French Region, in the SICOM telemedicine project that aims to define a generic information system for patient's home care.

Related works

User interface automatic generation is not a new domain of research. A lot of architectures have already been proposed and are called User Interface Design Environments (UIDE) [2]. We can cite Humanoid [3], Genius [4] or Mecano [5]. But all these approaches require providing a lot of information in complex models and descriptions [6]. These environments are so heavy and can not be used by non specialists. Their complexity explains why they are not widely used.

Simpler user interface development environments exist. For example, Jbuilder [7] and Visual Age for Java [8] help the developer by writing the code skeleton. But the

major part of the code remains to be hand-coded. That's why we are not satisfied by these products: we need a product ensuring no hand-coding at all.

The Abstract Window Description language that we have written is based on XML. Other projects have used XML to describe user interfaces. UIML [9] is a complete platform independent language. Harmonia is a web-based user interface generator using UIML descriptions [10]. An UIML description requires so many details for each widget. This makes the description so heavy and cannot be useful for non specialists. The same problem exists also in the XwingML language [11] which is a description language of Java Swing user interfaces. In this description, we find code fragments which reduces its utility. These considerations have encouraged us to develop new user interface description language with a higher granularity level.

Adaptation is another research topic close to our work. In [18] we find a definition to adaptation as the automatic and semi-automatic means to make resources usable on different kinds of terminals. In [19], Satyanarayanan says that adaptation is necessary when there is a considerable disparity between the supplied resource and the requested one. Many other works showed the importance of the adaptability or plasticity [20] of user interfaces to the contexts and the environments of their usage [21].

Two main directions are defined in adaptation:

Static adaptation consists in writing as many code versions as terminal types. Most of time, static adaptation has concerned existing applications for standard PCs and adapted them to mobile terminals [12][13][14]. In this case, written code is highly optimized for each terminal. But the drawback of such methods is the explosion of code versions due to the growth of terminal types. Manual coding of all the code versions is heavy and time consuming for the developers.

Dynamic adaptation makes adaptations on the fly: the interface is built and sent to the client on his demand. [15][16][17] are projects of web based user interface dynamic adaptations. The inconvenient of such approaches is the latency due to page generation at each query. Another disadvantage of the dynamic adaptation is the efficiency of the adaptation. In fact, hardware and software characteristics of the terminals grow fast and adaptation rules must follow this evolution.

For a perfect adaptation to the terminal characteristics we consider mixing the two methods with the maximum automation of the adaptation process.

The literature [22] differentiates two types of interfaces: input interfaces (physical devices and software tools for inputting data) and output interfaces (physical devices or software tools for data display). In this work we do not make this difference. Indeed, the selection criteria of a component - that it is for output or for input - are the same ones: the ergonomics and the comfort of use.

In user interface domain we distinguish two levels of interaction between the human and the machine: the physical level (material devices) and the logical level (software components). In this work we are only interested to the logical level. For example we are not interested in how a text field is filled (with the keyboard or through a software of voice recognition) but rather with the corresponding software component itself. The same consideration is taken with output devices. For example we are not interested in the physical size of the screen but in the size of the graphic

container that displays the user interface on the physical monitor. Thus, in the continuation of this article, the term "screen" indicates a displayed window on the physical screen of the terminal.

Panels and windows

Definition

User interface descriptions are often very heavy because of the huge quantity of details required to specify all parameters of each widget (size, position, color, events...). This is due to the low level description of reusable components (also called widgets). In this work, we have defined a higher level of reusable components: panels. A panel contains a set of grouped widgets that assures a typical functionality. For example, we have defined a panel to scroll image lists. Panels have been defined on the basis of user interfaces studies and we think they can answer many domain needs. Nevertheless, new panel descriptions can be added for specific needs. We have defined and implemented six panel types (see **Fig. 1.**):

- The table panel type allows to capture and to present data in a grid
- The text panel type allows to capture and to present a multi-line text
- The graphic panel type presents graphs or histograms for 2D values
- The image panel type presents a list of images and shows the selected image
- The video panel type presents a list of videos and plays the selected video
- The command panel type groups buttons that allow to execute actions on panels.

Each panel type contains a set of widgets, a priori geographically organized. For example, the image panel type is divided into 3 spaces: on the left, the list of image names, on the right, the displayed image and above the image, its textual description if provided. The command panel type has an horizontal disposition of buttons. Such a priori layouts can be seen restrictive, but they are fixed to reduce significantly the work of window construction, first for the person who makes new windows descriptions, and also for our tool which generates the corresponding code.

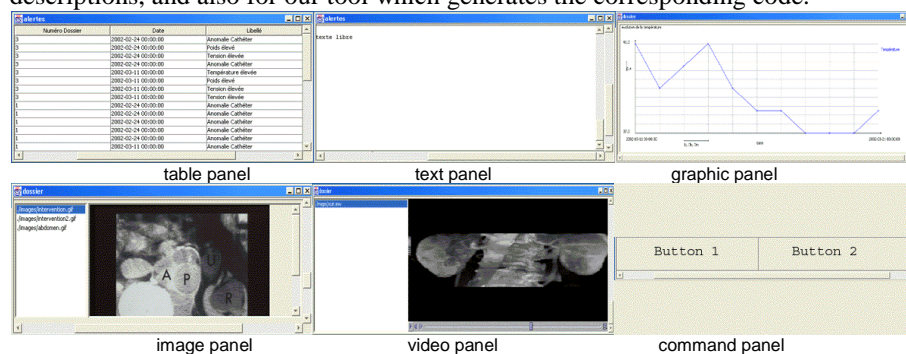


Fig. 1.: Implemented panel types, as seen on a standard PC

Each panel is associated to Web services that allow to:

- Fill in the panel widgets (get services),
- Update data from end-user captures (set services).

A panel automatically adapts to the services that are associated to it. Some adaptations are made at generation time, others at execution time. In a table panel, the number of columns corresponds to the number of data the related Web service provides. For example, the “getUserBasicInfo” service that returns name, first name and login will automatically make the table panel have 3 columns. This adaptation is made at generation time. Image formats are adapted to terminal capabilities. This is done at execution time. More information on adaptation is provided in another section of this article.

A window is thus defined as a list of panels associated to Web services. More precisely a window is defined as a list of panels. A panel is composed of a list of widgets which are associated to Web services input/output data. This constitutes an abstract window description. When a Web service is associated to a panel type, each input/output data is automatically associated to a panel widget. In a table panel, each data column returned by the service is associated to a column of the table. In a graphic panel, the first returned data column is used at the horizontal axis (X axis). Each other data column is used at the vertical axis (Y axis) and builds a different graphic: a service with 3 columns provides 2 graphics: one is $column2=f(column1)$ and the second is $column3=g(column1)$.

The concrete window code is calculated from this abstract description according to service specifications but not only. Windows and panels also have to adapt to the used terminal. For example, on a large screen, panels are organized vertically in the window. On mobile terminals, the lack of graphical space requires to divide a window on few screens: panels of the same window are accessed using a contextual menu; some panels are also divided into some screens (tables, images and video). We have defined a set of adaptation rules that ensures the adaptation of the code and the data for each terminal type. These rules are explained in another section of this article. Before exposing adaptation rules, we present in the next section how we structure and store abstract descriptions of windows.

XML Abstract Window Description (AWD)

We have chosen to use an intermediate internal format to store Abstract Window Descriptions. This AWD is platform-independent: an AWD may provide different codes for the different types of terminals. This intermediate format is based on XML: each window is described by an XML document. This intermediate AWD can be reused to create new windows or to update an existing window that requires modifications. **Fig. 2.** provides the DTD that we have defined for AWD documents.

A window has a name attribute, which serves as an identifier and as class file name. It contains panels and has a title. Each panel has an id, a type referencing its panel type in our panel types library, a default service (executed on the window instantiation). A panel has an optional title, a presentation specification of its

graphical components and a set of Web services descriptions. A service has an identifier and a set of parameters. Each parameter has a type, a name and a value description. A value is characterized by its localization (in which component), its initial instance and constraints (interval or enumeration of correct values). The localization of a parameter specifies the panel or the exchange structure (group id) from which we will read or display the value. A panel presentation provides the list of its graphical components. Each component has an id, a type, a title, editable and visible options. Button components have actions that assure the call of services in specified panels.

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT window      (title?, panel+)>
<!--ATTNLIST window name  CDATA #REQUIRED-->
<!ELEMENT title      (#PCDATA)>
<!--ELEMENT panel      (title?, service*, presentation)-->
<!--ATTNLIST panel id    ID #REQUIRED
      type              CDATA #IMPLIED
      defaultService IDREF #REQUIRED-->
<!--ELEMENT service    (parameter+)-->
<!--ATTNLIST service id  CDATA #REQUIRED-->
<!--ELEMENT parameter  name, value-->
<!--ATTNLIST parameter type  CDATA #REQUIRED-->
<!--ELEMENT name      (#PCDATA)-->
<!--ELEMENT value     (localization, initialValue, constraints?)-->
<!--ELEMENT localization (#PCDATA)-->
<!--ATTNLIST localization groupReference      CDATA #REQUIRED
      specificReference CDATA #REQUIRED-->
<!--ELEMENT constraints (constraint, constraint+)-->
<!--ATTNLIST constraints type (interval|enumeration) #REQUIRED-->
<!--ELEMENT constraint (#PCDATA)-->
<!--ELEMENT initialValue (#PCDATA)-->
<!--ELEMENT presentation (component+)-->
<!--ELEMENT component  (title, action?)-->
<!--ATTNLIST component  id ID #REQUIRED
      type(text|freeText|choiceList|checkBox|button|numeric)#REQUIRED
      editable (yes|no) #IMPLIED
      visible (yes|no) #IMPLIED-->
<!--ELEMENT action    (#PCDATA)-->
<!--ATTNLIST action targetPanelIDREF #REQUIRED
      targetService IDREF #REQUIRED >

```

Fig. 2.: DTD for AWD documents

The Abstract Window Description used in our project generates documents that are much smaller than other description languages one can find in the literature. Many implementation details are not provided in AWDs, as it remains at an abstract level. The interface look-out is not specified. This is an advantage for new windows construction as it guarantees efficiency (descriptions are rapid and the designer does not get lost in details) and consistency (no risk for widgets to overlap for example). But it can also be seen as a drawback: generated panels are “pre-formatted” and cannot be highly personalized. But the advantages it provides has balanced this inconvenient in case studies we have done.

Logical architecture of the SEFAGI platform

Our SEFAGI platform is based on three main entities, presented in **Fig. 3.**:

1. The **application server** contains patient records and application tasks in the form of business Web services.
2. The **terminals** are the tools for end-users to interact with the application server. They may be PCs, mobile phones, personal digital assistants, pocket PCs... connected to the Internet. They contain software layers to execute the application client part.
3. The **interface server** realizes the description of new user interfaces, the generation of the corresponding code and integrates them in the end users' terminals.

Each entity is presented in more details in the following subsections.

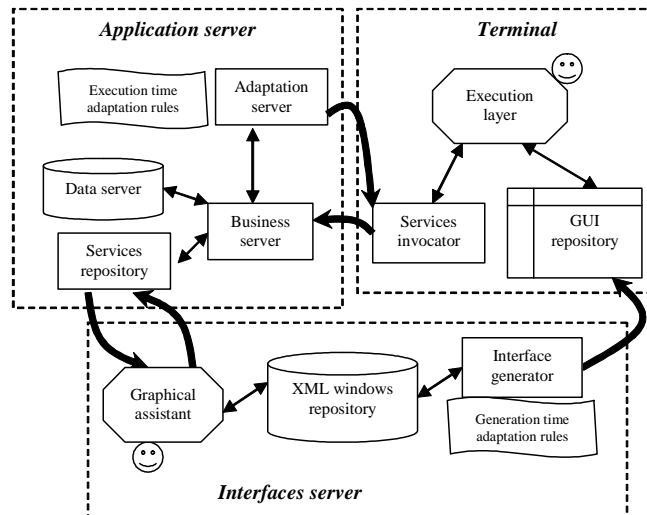


Fig. 3.: SEFAGI general architecture

Application server

The application server contains four main entities:

- **Data servers** include database management systems and file management systems,
- **Business servers** include procedures called by terminals, i.e. Web services,
- All available services are described in a **service repository**,
- **Adaptation servers** adapt data according to adaptation rules.

We have chosen to develop services with the W3C Web services standard. It has the advantage to provide a uniform interaction technique and does not compel to write all services with the same language. Existing procedures can be nested in a Web

service and thus save design, development and testing time. Most services are domain-oriented services, defined a priori at the same time as the records data structure are defined. All services gather the business knowledge (e.g. providing list of genes in a biological software...). We have also developed generic services allowing building a posteriori new services. The first generic service consists in a database SQL query service, receiving the query string, the database id and the connection information as parameters. We propose another generic service concerning image adaptation to the mobile terminals (size and format).

Service invocators and Web services communicate with the XMLRPC standard: service calls and return values are encapsulated in an XML exchange format. Parameters sent by the terminal include terminal description. The service transmits its returned values and the terminal description to the adaptation server. The adaptation server transforms (if necessary) returned values in a format and size supported by the terminal. For example, images are transformed into PNG format for mobile phones. Video data are not transmitted to terminals that cannot afford them: alternative textual descriptions are sent (if provided in the data server). Encryption services can also be computed on data before sending them to terminals.

The services repository is unique and provides a hierarchical description of all available services in the application. It provides this list to the interfaces server, so that the creation of a new window can use the proposed services in the window description.

Interface server

The interface server provides tools to:

- create new abstract windows descriptions,
- modify or reuse abstract windows descriptions,
- generate the code corresponding to abstract windows descriptions for each kind of terminal.

A **graphical assistant** helps making the first two points, the third point is the job of the **code generator**. The XML AWD is the output of the graphical assistant and serves as an input for the code generator.

The graphical assistant provides means to make new AWDs or to modify existing ones. Creating a new window consists in giving its title and then adding panels to it. Adding a panel requires two steps: choose the panel type from the provided list, and choose the Web services to attach to it (using a Web service repository interface). The AWD creation can stop at this step, or the user can change default settings of widgets in panels. For example, he can make a widget not editable, or change a list to a choice group... but this description remains at the logical/functional view level. No look-out parameter is provided. The designer has much less to do than in other existing proposals.

On **Fig. 4.**, the graphical representation of AWD is not a WYSIWYG presentation of final windows: the lookout of the executable window depends on the used terminal. Each panel of the AWD is presented as a panel containing tabs: one for the widgets, and one for each associated service. When the user validates the window description, the graphical assistant stores the new AWD in an XML file.

SEFAGI: Simple Environment For Adaptable Graphical Interfaces: Generating user interfaces for different kinds of terminals 9

A menu of the assistant allows to open an existing AWD, so that the user can update this description. This can be very useful to build user interfaces for different end-users groups and make modifications according to access rights to data for example.

The interface generator takes an XML AWD file as input and generates the executable code for the corresponding description. It automatically builds a code for each terminal type. This generation consists in two steps: first write the Java code corresponding to the window, and second compile it. It finally adds this code to the execution environment (in practice and for the moment, it consists in writing files in adequate repositories). The code is generated according to generation time rules and to the MVC model, as described in section 5 of this article.

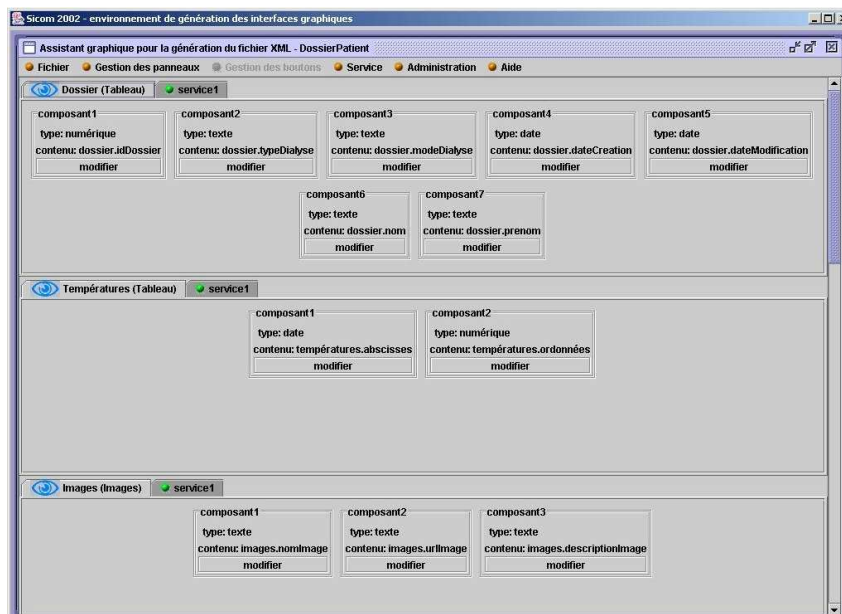


Fig. 4.: Graphical assistant : AWD graphical representation

Terminals

Each terminal must contain a Java Virtual Machine. This condition is not so restrictive: a large number of terminals nowadays provide a JVM. We have defined three main software components on terminals:

- A **repository** of generated windows coming from the interface server,
- An **execution layer** we have specifically designed to manage user interfaces,
- A **service invocator** which invokes distant application server procedures and gets their returned values.

The repository of generated windows contains all window code files that are available on the terminal. It consists in Java class files identified by their names.

The services invocator interacts with business servers by invoking Web Services on demand from the execution layer. It first encapsulates input data into an XML stream, and then sends them to the Web service. It gets the results of the Web service, decapsulates data and returns them to the execution layer. Each service call includes the terminal description, so that the adaptation server can adapt outputted data to the terminal capabilities.

The execution layer contains the code to run the whole application. It stores the panel types and widgets code library. It executes windows, manages data exchange between them... It also contains the main window of the application, a menu proposing the windows list, means to navigate from a window to another including inter-windows data exchange and a preliminary window for end-user authentication (login/password).

From XML Abstract Windows Descriptions to executable code

Windows Java code organization

Windows Java code is generated from abstract windows descriptions. To structure this task, we have chosen to write the code according to the MVC standard structure. Each window code is thus divided into three classes:

- the Model class manages user identification and the dialogue with Web services,
- the View class includes the definition of attributes (widgets for the most), attributes initializations in constructor, setter and getter methods on attributes,
- the Controller class includes events management and transmission of services calls to the Model class.

Some parts of the code are always the same for all generated windows. For example, user authentication is always the same. Moreover, we have written a widgets library that includes all the equivalent components of the AWD. Doing so, we have highly reduced the proportion of code to be written specifically for each new window. This library is included in each terminal. Specific parts of the code have to be generated for each window. This is done by our code generator (presented in another section).

Inter-windows data exchange

Navigation between windows during their execution is assured by navigation means, but also inter-windows data exchange. For example, selecting a line in a patient list provides the patient id to be used during the navigation in the next window (temperature graphics window and lab results window for example). To do so, we have defined a generic common data exchange structure accessible by all windows. This is implemented as a two-level structure: the first level defines groups that include

second level elements (attribute / value pairs). The view class of the window uses this structure to refresh its data.

Adaptation rules

Windows adaptation has to be done at different levels and at different times in the SEFAGI architecture: some adaptations concern display, others concern data. Some are made at generation time, others at execution time. In this section, we will present the set of rules we have defined, as well as examples.

Generation-time adaptation rules

Generation-time adaptation rules are executed while writing the executable code of a window from its AWD. For the moment, we have implemented only two criteria to decide which rule to apply: the screen size and image/video capability. We have defined three classes of rules for adaptation at generation time:

1. **Widget transformation rules** maps platform specific widgets to abstract widgets in AWD. These rules consist in a table providing for each abstract widget its corresponding platform specific class (e.g. an abstract button corresponds to a JButton on a large screen and to menu items on a small screen).
2. **Layout transformation rules** allow defining the position of the widgets inside a panel and of the panels inside a window. For example, the layout of the table panel is grid – like on a large screen; it is decomposed into two effective windows on a small screen: the first one serves as an index to access the second containing the selected line. A table line of the AWD becomes a table line on a large screen and a vertical list of items on a small screen. On a small screen, each panel of the AWD becomes an individual window. Layout transformations provide rules for each panel type and for each terminal type.
3. **Navigation transformation rules** are the third type of rules for code generation. They also depend on the screen size: when an AWD window is decomposed into many screens (for mobiles), navigation between the implemented windows should be included. Moreover, navigation means on a standard PC are not the same as on a mobile phone. Navigation transformations are a consequence of layout transformations: they are used to link implemented windows. Navigation transformations concern two elements: the widget to be selected to make the navigation, and the exchange of information between the implemented windows.

Execution-time adaptation rules

Other adaptations have to be made at execution time. Such adaptations concern data. Data adaptations depend on the terminal capabilities and on network transfer characteristics. We have defined three classes of execution-time adaptation rules:

1. **Content transformation rules** concern data format and terminals capabilities. For example, images are transmitted only if the terminal can show them: data formats are transformed into other ones supported by the target terminal (e.g. JPEG to PNG for images).

2. **Coherence transformation rules** concern Web services output and input. We gather the returned data from a Web service into a single block. We have standardized the data block structure so that any exchanged data is streamed in a standard format. We decorticate this structure at reception. We secondly take care of transactions during data exchanges. To avoid data loss, transactions have to be maintained between terminals and servers. To do so, we use checkpoints in data transmission.
3. **Transmission transformation rules** concern network regards. They transform exchanged data between Web services and terminals to adapt them to the XMLRPC standard stream.

Case study

In this case study, we present a window that contains three panels : the first is a table panel showing patient file id, dialysis type and mode, date of the patient's record creation and of its last modification, name and first name of patient. The second is another table panel which presents a temperature data list of the concerned patient. The third is an image panel.

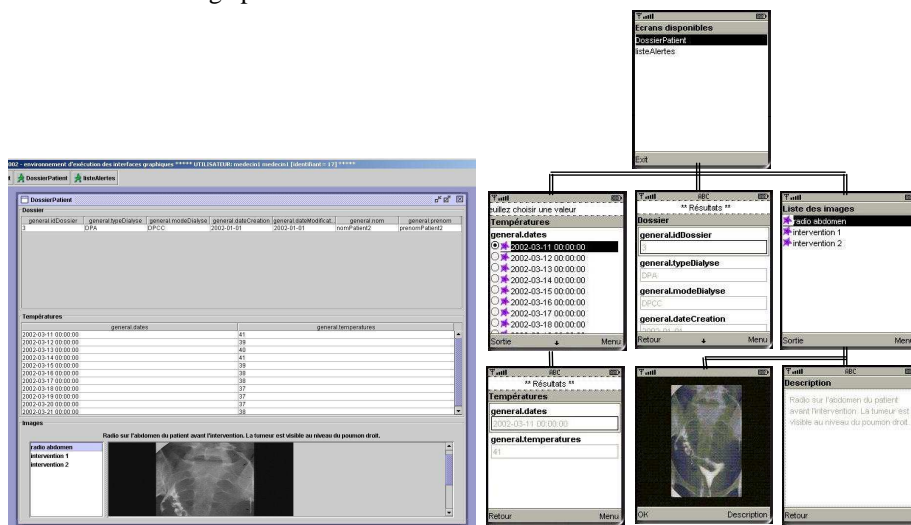


Fig. 5. Generated window for standard screens and its equivalent for mobile phones

Fig. 5. shows the generated window for standard PC screen and for mobile terminals. On the standard screen, we see the execution environment that manages the generated windows. A menu bar in the upper part of this window provides the list of available windows. This menu bar is translated as a root window on the mobile terminal. The first table panel is shown as a matrix on the standard screen containing only one line. As it contains only one line, it is shown as a vertical list of items on the small screen. The second panel is shown as a matrix on the large screen. It is decomposed into two windows on the small screens: the first provides the first

column values; the second provides the line corresponding to the selected value, with a vertical disposition of items. The third panel is an image panel. It is decomposed into three windows, one for each component. The small screen interface generates many windows and thus many navigation means. All navigation facilities are automatically computed at code generation.

Conclusion and perspectives

In this article, we have presented the SEFAGI project which aims to provide an information system that supports multi-platform user interfaces. The core application server contains Web services, adaptation services and a services repository. An interface server provides a graphical assistant to describe new windows and a code generator for standard terminals and mobile terminals. We have also developed the necessary software layer for windows management in the terminals.

In this project, user interfaces are adapted in three ways:

- First, end-users can easily create new screens on the available data through Web services. This can be seen as a static assisted user interface adaptation,
- Second, described user interfaces allow to automatically generate two kinds of codes: standard terminals interfaces and mobile terminals interfaces. This can be seen as a static automatic interface adaptation,
- Third, content data is adapted to terminal capabilities at runtime. This can be seen as a dynamic automatic interface adaptation.

We are now only at the beginning of our SEFAGI project and much more has to be done. For example, we have to think of a better organization of the AWD documents in the repository. This requires documents description and categorization. We also have to take into account much more adaptation parameters, like voice capability, color depth and others, but also more precise end-user profiles, and more adequate access rights management. Another point that could be studied concerns adaptation services. Adaptation rules are embedded in adaptation services. We should study a way to generalize adaptation services, so that they can use external rule repositories. This would provide much more flexibility to adaptation.

References

1. B. A. Myers, M. B. Rosson: Survey on User Interface Programming. In: P. Bauersfeld, J. Bennett, G. Lynch (eds.): Striking a Balance. Proceedings CHI'92 (Monterey, May 1992), New York: ACM Press, 1992, 195-202.
2. J. Foley, W. Kim, S. Kovacevic and K. Murray, UIDE - An Intelligent User Interface Design Environment. Architectures for Intelligent User Interfaces: Elements and Prototypes, Addison-Wesley, Reading MA, 1991, pp.339-384.

3. P. Szekely, P. Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In Proceedings SIGCHI'92. May 1992, pp. 507-515.
4. C. Janssen, A. Weisbecker, J. Ziegler: enerating User Interfaces from Data Models and Dialogue Net Specifications. In: S. Ashlund, et.al. (eds.): Bridges between Worlds. Proceedings InterCHI'93 (Amsterdam, April 1993). New York: ACM Press, 1993, 418-423.
5. A. Puerta: The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: J. Vanderdonckt (ed.): Computer-Aided Design of User Interfaces. Namur: Namur University Press, 1996, 19-36.
6. P. P. da Silva. User Interface Declarative Models and Development Environments: A Survey. In P. Palanque and F. Paterno, editors, Proceedings of DSV-IS'2000.
7. JBuilder Home Page, www.borland.com/jbuilder/, Last visited November 2003.
8. Visual Age for Java, <http://www-3.ibm.com/software/awdtools/vajava/>, Last visited November 2003.
9. Abrams et al., 1999 M. Abrams, C. Phanouriou, "UIML: an XML language for building device independent user interfaces", *XML'99*, Philadelphia, December 1999.
- 10.M. Boshernitsan, "Harmonia: A flexible framework for constructing interactive language-based programming tools", Technical Report, University of California, Berkeley, June 2001
- 11.S. St Laurent "Java, XML, and a New World of Open Components", New York Object Developers Group- <http://www.simonstl.com/articles/nycod/index.htm>, April 1999
- 12.Larson, 2002 Eric D. Larson, Wireless Java Application Saves Women's Cancer Center \$500,000 per Year, J2ME case studies, september 2002. <http://wireless.java.sun.com/midp/casestudies/wcc/>.
- 13.Massé, 2002 V. Massé, « La société MobilePlanet Europe fournit des terminaux mobiles à la Brigade des Sapeurs Pompiers de Paris (BSPP) pour l'équipement de ses véhicules d'intervention », Mobile Planet, juin 2002. http://www.mobileplanet.fr/m_includes/press_release/2002_brigade.asp
- 14.Benjaminsen *et al.*, 2002 S. Benjaminsen, A. Djora « JetRek: How organisational identities slowed down speedy requisitions », *BSA medical sociology conference*, september 2002, York. <http://www.sv.ntnu.no/iss/Aksel.Tjora/publications/Siri-york02-09.pdf>
- 15.TeraCom, 2002 TeraCom "Soluphone santé" 2002. <http://www.soluphone.com/SoluSante.pdf>
- 16.Lemlouma *et al.*, 2001 T. Lemlouma, N. Layaida, « A Framework for Media Resource Manipulation in an Adaptation and Negotiation Architecture », OPERA project, under submission, INRIA Rhône-Alpes, august 2001
- 17.Menkhaus, 2002 G. Menkhaus « Adaptive User Interface Generation in a Mobile Computing Environment », PhD Thesis, Salzburg University, 2002.
- 18.N. Layaida, Adaptabilité : pistes d'étude pour la définition d'une infrastructure d'accès au contenu multimédia pour des machines hétérogènes, Rapport Technique, INRIA Rhône-Alpes, octobre 1999.
- 19.M. Satyanarayanan, Pervasive Computing: Vision and challenge, IEEE Communications, 2001.

SEFAGI: Simple Environment For Adaptable Graphical Interfaces: Generating user interfaces for different kinds of terminals 15

- 20.G. Calvary et J. Coutaz "Plasticité des interfaces : une nécessité !", information-interaction-intelligence, *Actes des deuxièmes Assises nationales du GDR 13*, Cépaduès Editions, Nancy pp 247-261
- 21.G. Calvary, J. Coutaz and D. Thevenin "Supporting context changes for plastic user interfaces: a process and a mechanism", *Proceedings of HCI-IHM 2001, BCS conference series*, Springer Publ., pp. 349-363
- 22.L. Nigay "MATIS : un système multimodal d'information sur les transports aériens", *IHM'95, 7èmes journées de l'ingénierie de l'interaction Homme Machine*, CEPAD Publ., pp. 131-132.