A DESIGN SPACE FOR CONTEXT-SENSITIVE USER INTERFACES

Jean Vanderdonckt[1], Donatien Grolaux[1], Peter Van Roy[2], Quentin Limbourg[1], Benoît Macq[3], Benoît Michel[3]
Université catholique de Louvain, BCHI[1], INGI[2],TELE[3],
{vanderdonckt, grolaux, limbourg}@isys.ucl.ac.be, pvr@info.ucl.ac.be, macq@tele.ucl.ac.be, michel@similar.cc

Abstract

Context-sensitive user interfaces become a very important class of interfaces as they reconfigure their presentation and dialog according to various events generated in a dynamic context of use. Traditional procedural approaches for developing such use interfaces are very expressive, yet expensive to develop, difficult to use and verbose to write. Declarative model-based approaches have recently been considered as they allow user interface code generation from models, they are quick to manipulate, yet they lack expressiveness. It is argued that a mixed model-based approach obtained by mixing and integrating declarative and procedural elements combine advantages of both approaches without suffering of their shortcomings. An example shows how this new approach can effectively and efficiently produce context-sensitive user interfaces.

Keywords

Context-aware adaptation, Context of use, Interactive system, Multimodal interaction, User interface, USer Interface eXtensible Markup Language (UsiXML).

1 Introduction

Nowadays interactive applications users are more immersed in a constantly evolving environment where there is no longer an ability to predefine all possible configurations and conditions of the context of use. For instance, corporate environments, which are prompted to address the challenges of market internalization, have to create, introduce, and expand strategies to maintain or to improve their market position. For this purpose, they tend to switch from a business logic, where tasks are planned in a predefined way and their results are observed afterwards, to a dynamic and anticipative strategy that enables them to react to unpredicted contextual events as quick as possible. Moreover, users of such interactive applications supporting the activities of these organizations become more mobile. In order to react to those contextual events, they move with different computing platforms [3,6], ranging from a laptop or pocket computer to a Personal Digital Assistant (PDA); or they move from one computing platform to another [2,5,19], thus causing multiple opportunities for changing the conditions of the context of use. At runtime, the context of use may dynamically change: the computing platform may differ widely, the network bandwidth may decrease, the interaction and display capabilities may be reduced, the user may assume a new role in an ever-changing organization structure, the task may evolve, etc. Those changes have created a need for new user interfaces (UIs), that continue to support users in accomplishing their tasks while the context evolves in time, space, and resources. When the context of use changes, a particular UI may suggest a *reconfiguration*, that is an adaptation of its presentation and/or dialog to fit the new context of use. In this article, we characterize such *Context-Sensitive User Interfaces* (CSUIs) by first reporting on some challenges posed by this new UI type. We introduce a design space specifying the contextual changing parameters that need to be reflected in a CSUI in some way to continue to support users in their interactive tasks while the environment is changing. We provide some representative examples of CSUI based on the design space and we introduce a new method for developing a CSUI based on separability and correlability of models involved in UI design.

2 Challenges of Context-Sensitive User Interfaces

Developing CSUIs poses a series of challenges that still need to be solved due to several shortcomings:

♦ *Limited specification of context*: specifying the circumstances in which a wide range of varying contexts may occur and turning this information into precise design requirements of UI configurations (i.e. layout and dialog) constitutes a challenging problem. This problem has been addressed, but many representations of this context and many techniques on how to capture knowledge of the context variations have been introduced [2, 3,4,5,6,16,18]. Moreover, once such a context variation has been detected, there is no uniform way to reflect it in the UI. Sometimes, not all configurations of the context of use can be identified at the design time: rather, they are known at runtime. If these configurations are not supported, the user's task may be definitely interrupted.

♦ *Questionable usability:* a fixed UI may be considered usable in some expected contexts of use, where a given set of constraints is met [6]. These inflexible UIs tend to rapidly become inappropriate or unusable when the context of use changes, thus leading to a questionable usability. It this thus crucial to take the changing context into account while keeping a minimal usability.

♦ *Tremendous development effort:* CSUIs are traditionally developed through classical programming environments, such as Basic, C++, or Java [19]. In these environments, developing a CSUI typically involves designing the various configurations corresponding to the multiple contexts of use. Any change of this context is then reflected in a configuration change. Programming a dynamically reconfigurable UI remains a very complex task. A layout reconfiguration depending on a user change might be reasonably complex to specify, but may require hundreds lines of code to be supported. Not only may this activity increase the UI code portion, but also require a dedicated software architecture receiving contextual information thanks to context-aware widgets [4].

- *Increased testing and maintenance efforts:* as layout and dialog are often intertwined in a traditionally developed CSUI, the testing and the maintenance of configurations dealing with layout and/or dialog can become painful and unstructured. In particular, inserting a new configuration into an existing pool may undesirably affect several portions of code, thus lengthening the maintenance period. The development and maintenance efforts are easily duplicated for cross-platform UIs, while potentially reducing the consistency [22].

3 Design Space for Context-Sensitive User Interfaces

For years, there has been much interest in the adaptation of UIs as there is today a core of extensive research and development of the two facets of adaptation [8,16,19]:

- *Adaptivity*: when the system executes the adaptation for the user. For example, the system displays different levels of help depending on the type and frequency of errors made by a user.
- *Adaptability*: when the user executes the adaptation. For example, a user personalizes a UI according to selected preferences as in Figure 1.

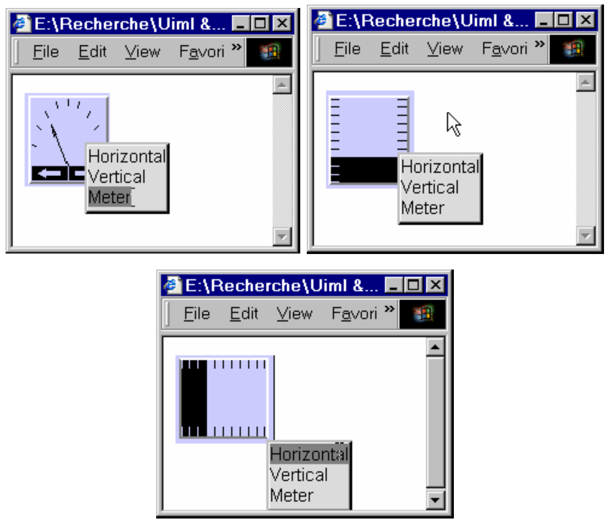


Figure 1. Adaptability of a widget for a bounded value.

Adaptation expresses some UI change according to possible types of variation, the most frequently used being, with respect to the user's characteristics, the user preferences, performance, the number of errors, the previous interaction history, and the possible disabilities in case of users with special needs. The ultimate goal of adaptation is to empower any user with a UI that is uniquely customized according to his or her particular needs so as to create a UI with maximal usability [5,20]. Since this usability highly depends on the context of use, any change of this context may no longer preserve the expected quality level of usability. Therefore, context-sensitivity is intended to constantly perform some adaptation to increase or at least to maintain this level of usability while the context of use is changing [4]. The availability or the lock of resources required for human-computer interaction should be taken into account when adapting a given UI.

Context-sensitivity subsumes many interesting forms that can be considered in isolation. One significant form of context-sensitivity is that of plasticity [2,20]: a *plastic* UI is a particular UI type sensitive to any variation of the computing platform and/or the environment. This environment encompasses physical aspects (e.g., noise and light conditions), software/hardware constraints (e.g., screen resolution, network bandwidth), and social positions (e.g., organization structure, task allocation and role definition). As a plastic UI is not necessarily intended to support user variation, it is assumed that the UI is manipulated by a predefined type of user who is supposed to be representative of the user population. A need appears for consolidating various experiences and approaches that have been undertaken under the umbrella of adaptive, adaptable, plastic, and reconfigurable UIs, that is a CSUI.

To represent the types of variation that can be theoretically considered in context-sensitivity, Figure 2 depicts a design space expanding a design space for adaptation [8] and another for plasticity [20]. This design space is presented like an action-reaction process: its upper part describes what type(s) of context variation may cause the reconfigurability (the action), while the bottom part describes what type of reconfigurability is undertaken (the reaction to the change of context).

Along the "*With respect to what?*" axis, context-sensitivity is concerned with the types of context variations raising the need for reconfiguring a UI [5]. These types of variations are located in one or any UI models. A UI model is a declarative, editable, and analyzable representation of some predefined aspects of a UI, according to relevant abstractions [15]. The main purpose of these model abstractions is to describe UI properties at a higher level than just programming code that frees developers from coping with too low level details and software/hardware dependencies. In such an approach, several models could be involved. Some models may contain sub-models: a computing-platform model contains models of the various interaction objects and devices to be manipulated on this platform; a help model is decomposed into guidance and tutorial models, along with separate help facilities. It is not required that all these models should be considered to develop a CSUI. Rather, this list is intended to locate where any event indicating a change of context occurs (action) and where the results of this change of context should be applied (reaction). Model-based approaches [6,9,13,14, 16,18,21,23] are particularly suitable for CSUIs for the following reasons:

- Models are convenient to slice the human-computer interaction complexity into expressive layers that can communicate through appropriate protocols (*principle of expressiveness*). For instance, the user (described in a user model) can interact with widgets (modeled in an interaction object model) that are part of the presentation (described in a presentation model) through a dialog (described in a dialog model) on a given screen (detailed in a computing platform model). Changing a screen for instance would only require a change in the computing platform model, and nowhere else [22].
- They tend to separate different facets of UI design and information into autonomous models (*principle of separability of concepts*), typically maintained as knowledge bases [10], while preserving coupling through design relationships between models' abstractions (*principle of correlability of concepts*).
- By abstracting requirements via concepts, they are able to identify the UI parts that are common or different, fixed or flexible, across a wide variety of different contexts of use. It is also argued that identifying these parts without a certain level of abstraction and independence would remain infeasible [2].
- They allow defining rules expressing ordered relationships between concepts hold in various models. New design knowledge can be incorporated by establishing mappings between models' concepts [16].
- They allow the adoption of a knowledge-based approach [10] on top of the models: once the models capture the relevant information, artificial intelligent techniques (e.g., production rules) can be deployed to reason on the models contents and produce a reaction.

Along the "*What?*" axis, context-sensitivity is concerned by the locus of reconfiguration: any model that is relevant to a running UI is considered. For example, any change of a computing platform characteristic (e.g., a screen resolution reduction declared in a computing-platform model) should trigger a presentation reconfiguration (e.g., a simpler UI with widgets consuming less screen real estate). In computer-based systems, any change of a user (e.g., the learning level of a student captured in a user model containing skills, experience, and cognitive profile) should reconfigure the tutorial (e.g., keeping advanced topics in a tutorial model). Moreover, a reconfiguration may occur in many models, e.g., the change of a user may also be mirrored in a change of presentation style.

Along with the "*For what?*" axis, context-sensitivity is concerned by the four steps considered in adaptation [8]. The initiative specifies the entity which initiates the need of reconfiguration. The proposal describes possible reconfigurations to be performed on the target models of the UI. The decision states the entity which decides to apply the reconfiguration when needed. The execution clarifies the entity which is responsible for effectively performing the reconfiguration that has been decided.

Along with the "*Who?*" axis, context-sensitivity is concerned by the responsibility of undertaking any adaptation step: it could be a user, a third party, the system or a mixed-initiative involving several actors. Typically, one entity (e.g., the system) is responsible for performing the four steps. But a system may prompt a user with possible reconfiguration mechanisms from which the user is able to pick up one, thus decreasing the negative disruptive effects induced by adaptivity [8]. A user may select one possible presentation style among a set of predefined ones (Figure 3), while keeping its functionality.

Along with the "*How many?*" axis, context-sensitivity is concerned by the number of reconfiguration occurrences required to achieve the context-sensitivity [5]. For example, one variation of screen resolution in the computing platform may result into several presentation and dialog reconfigurations. One task variation may lead to many presentation and dialog reconfiguration to reflect the fact that the task structure changed.

Along the "*When?*" axis, context-sensitivity is concerned by the moment during which the reconfiguration is effectively considered: at design-time, at runtime or both. For example, a web page is intrinsically designed to support various Web appliances (such as a classical web browser, a WAP-compatible cellular phone, a television set top box, and an Internet screen phone). Similarly, a web page may compute a frame rate of a video sequence at runtime, depending on the available bandwidth.

Along the "*With what?*" axis, context-sensitivity is concerned by the type of model needed to support the intended reconfiguration. A *passive model* holds static properties that are only read to perform a reconfiguration, while an *active model* holds dynamic properties that can be changed at runtime. A *shared model* can hold both kinds of properties. To accommodate multiple screen resolutions of a same platform, a UI needs to embark an active model to apply an appropriate reconfiguration [6]. When models are considered only at design time, they often remain passive.

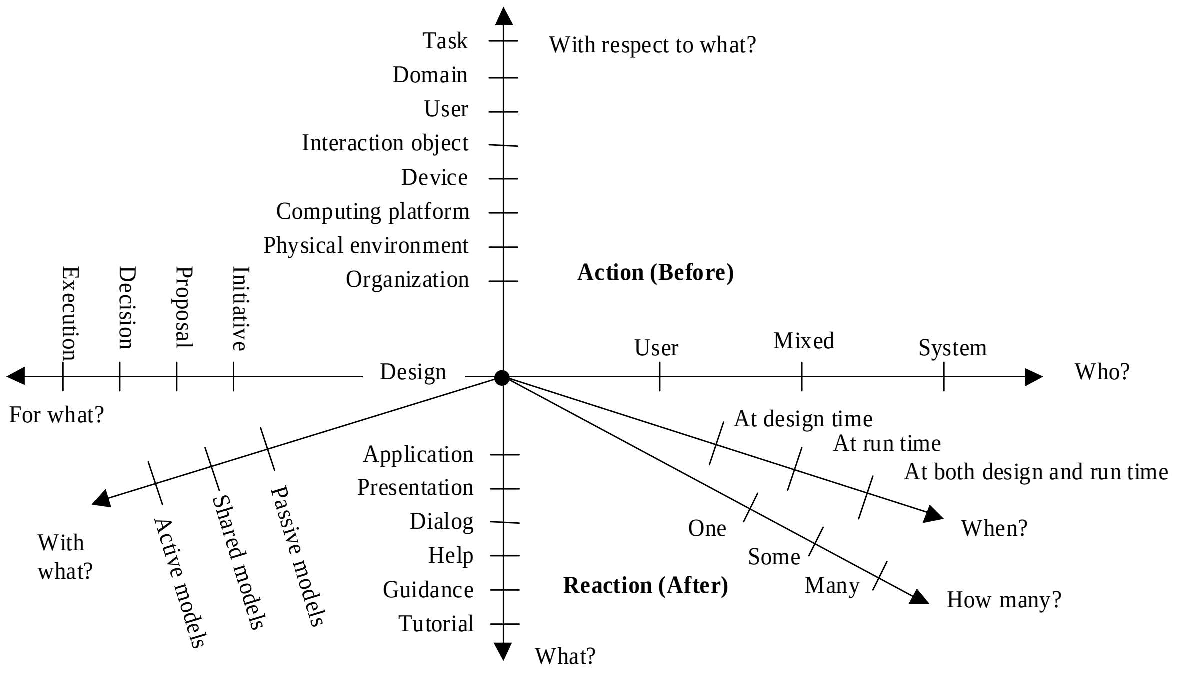


Figure 2. A design space for context sensitivity.

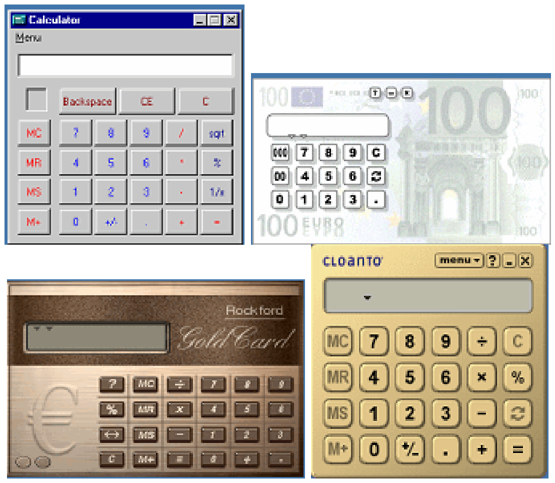
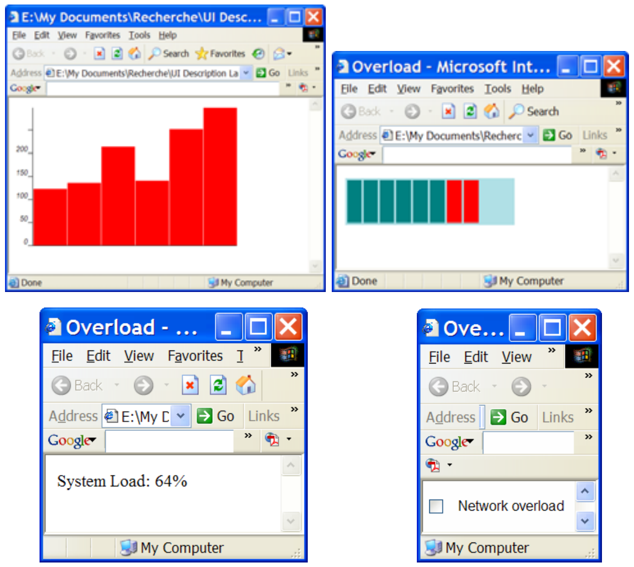


Figure 3. Multiple presentation styles for a single user interface.

4 Representative Examples of Context-Sensitivity

The design space for context-sensitivity is able to express several important UI categories:

1. *Multi-language UIs*: the UI is able to accommodate variation of the natural language according to the user's needs. For example, the user can switch from one language to another by selecting it from a UI menu, or the system can automatically set it according to a preference stated in a profile.
2. *Plastic UIs*: the UI is able to accommodate variations of both the computing platform (e.g., screen resolution, colors, operating systems) and the physical environment (e.g., the network bandwidth, the availability of interaction devices), while preserving usability, that is a set of properties specified during the requirements phase [2,20]. These properties are defined in [2,20]. For example, the same UI can display a network load in multiple forms according to varying screen constraints [6,20]. Figure 4 represents a plastic UI which accommodates different presentations of a network load while the window is being resized. This example is adapted from [20].


E:\My Documents\Recherche\UI Desc...
Overload - Microsoft Int...
Overload - ...
Ove...
Network overload
My Computer
Done

System Load: 64%

Figure 4. A plastic user interface for network load.

3. *Cross-platform UIs*: the UI is able to accommodate variations of the computing platform, while preserving a given usability level. For example, SUIT enables designers to design one UI that can run on different platforms, while preserving consistency [14].
4. *Migratory UIs*: the UI is able to accommodate relocation of the user terminal [1], while maintaining the same context for the application when the user switches between terminals of the same platform.
5. *Mobile or nomadic UIs*: the UI is able to accommodate variations of the context of use and the change of user position, while sustaining a given usability threshold. For example, the GUIDE system [3] is varying according to the user's location in the physical space and presents different information, accordingly. Figure 5 represents the virtual keyboard: when a numerical entry is performed, the keyboard may become available or unavailable depending on the user location. When the keyboard becomes (temporarily) unavailable, the edit field is replaced by a virtual keyboard.

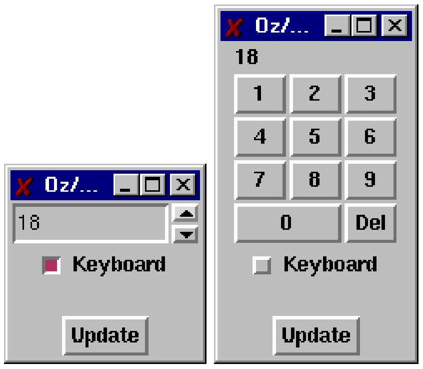


Figure 5. The virtual keyboard.

After highlighting some representative CSUI examples, we report on various methods used to develop such UIs and we discuss their advantages and shortcomings.

5 Development of a Context-Sensitive User Interface

What is the best way to develop a CSUI? The methods which are frequently used to develop them are either procedural or, more recently, declarative (model-based). Each approach has problems: the procedural approach is difficult to use whereas the declarative approach lacks expressiveness. We propose a *mixed model-based approach* that combines both declarative and procedural elements into a single environment. Declarative programming is often more appropriate for static, structural aspects, while procedural programming is more adapted to dynamic, behavior aspects. A representative example, a context-sensitive clock, is given to show the advantages of the mixed approach. Finally, we discuss what the mixed approach implies in terms of support from the underlying development platform.

5.1 Traditional approaches

The most frequently used approach to develop a CSUI consists in writing a code statically supporting all possible configurations in a selected programming environment, such as C++, Pascal, Basic or Java. These environments are typically procedural in the sense they allow developers to code various CSUI aspects in procedures and functions. Therefore, a single procedure may merge aspects which are relevant for different facets, e.g. the presentation, the dialog, and the user. Beyond procedural programming has been considered declarative programming in a model-based approach [13]. In this approach, information required in order to notify the changes of context and reconfiguration can be expressed into facts and rules.

5.2 Limits of procedural and declarative approaches in isolation

These approaches can be compared along several criteria:

- *Separability*: in a procedural approach, a single procedure may merge different aspects which are relevant to different facets, e.g. the presentation, the dialog, and the user, thus keeping separability low to moderate. Instead, in a declarative approach, each model contain only relevant aspects provided that a clean separation scheme is uniformly applied, thus raising separability from moderate to high.
- *Code size*: a purely declarative approach defines a set of possibilities for different attributes. The model chooses among this set of possibilities. This defines "what" without defining "how". A purely procedural approach gives a set of primitive operations and the ability to define other procedures by calling them. The model calls the primitive operations in order to "build" its content. Both approaches can potentially generate an important section of code, especially when models are considered verbose. For example, 4 functions and one hundred lines of JavaScript procedural code is needed for Figure 1, 12 functions and more than two hundreds lines of code for Figure 4.
- *Complexity*: however, lines of procedural code are more complex to write and to interpret than lines of specification in a model-based approach. Declarativity makes it easy to formally manipulate the model definitions (e.g., to translate into different representations) but reduces the scope of model manipulations.
- *Expressiveness*: a procedural is by nature more expressive than a declarative approach. In particular, a declarative approach is more convenient for describing static aspects (as the presentation) than for dynamic aspects (as the dialog) and vice versa. A declarative approach limits the expressiveness to what the designers initially put in the model, whereas procedurality has no limits on expressiveness, since it defines a full-fledged programming language.

5.3 Mixed declarative and procedural approach

We believe that each approach has fundamental disadvantages when used in isolation, and that a realistic approach should mix elements of both. To some degree, judging the relative expressiveness and ease of use of the approaches is subjective. For example, a model-based approach can be considered as "too declarative" as they try to present all possible configurations. Pure declarativity has only one way to compose parts, i.e., to tile them syntactically. This is such a strong limit on expressiveness that it requires to “enumerate” all the possibilities. Declarativity should consequently be used only in those UI parts where expressiveness is less important.

In a mixed approach, three areas are distinguished: the static structure of the widgets (nesting), their resize behavior (glue parameter), and their initial state. Other areas, in particular, event handling and widget updating, are handled by a procedural approach. The complete interface is defined by a record, with two kinds of components: other records (nested widgets, etc.) and procedure values (for event handlers and dynamic widget control). The procedure values are embedded in the record. A mixed approach using declarations for static definitions and procedures for dynamic ones is a promising idea, as we can keep the best of both worlds.

FlexClock illustrates this mixed approach: the view definitions are purely static definitions (description of the UI of the view, information for updating the clock according to the current time and information for determining which view to display according to the current window size). The description of a single view is thus only three lines of code! The dynamic aspects of FlexClock, e.g., the update of the views each second according to the new time and the selection of the best view upon window resize is left to a pure procedural approach. Mixing both approaches in the same language was possible only because of the existence of data structures in Oz [12,17] that could support it: records and lists

There exists an extensive support from the language to manipulate these data structures. As a direct result, we have the opportunity to dynamically create any type of record, i.e. description, we may need, while still remaining in the same language. The calendar of FlexClock illustrates this: the calendar is dynamically built from label widgets. A ‘ten lines of code’ loop creates the complete widget description, including the geometry management. Once done, the description is given to the QTk toolkit module [7] of the Mozart environment [11] to effectively build the UI. In practice, it is simpler to manipulate data structures of the language (which are supposed to be manipulated) instead of line of codes (which are intrinsically static). Moreover procedures in Oz are values [12]. Dynamic definitions can thus be embedded into procedures, i.e. values to create static definitions of dynamic behavior. This pushes a little more the possibility of dynamic definitions and was used for example to define the update of the views of FlexClock according to the time.
In summary, we can use a declarative approach whenever needed and keep a procedural approach elsewhere. We can even build declarations on-the-fly as needed, using a procedural approach! This is the key FlexClock is using for creating a highly dynamical interface in very few lines of code.

6 The FlexClock example

FlexClock is a simple context-sensitive clock application (Fig. 6). It was written using the mixed approach introduced above. We explain the design of FlexClock and show how declarative and procedural elements each have their place.

6.1 Architecture

The architecture is concurrent and event-driven. There are two event sources: (1) a clock that ticks once per second and (2) a 'resize' event for each resize of the window. Clock events are sent to all the views, and cause them to update their display. Resize events are input to a 'choice procedure' that picks the best view and displays it. There is no other interaction between these two event sources.

6.2 Implementation

The application is written in Oz [17] on the Mozart platform [11], using the QTk UI toolkit [7] on top of Tcl/Tk. The complete application is 370 lines of code, divided as follows:

- 270 lines for the clock-specific code, including definition of the analog clock widget (80 lines), calendar widget (50 lines), picking and drawing the Mozart icon (70 lines), and formatting utilities (70 lines).
- 60 lines for the complete definition of all 16 views. For each view, this includes the view declaration, the view update function, and the minimum view area, used to pick the best view.
- 40 lines for the event mechanisms (clock and resize) and definition of the choice function.

6.3 Definition of a view

Each view is defined by a record that combines declarative and procedural elements. Here is a typical view:

```
r(desc:label(handle:H3 glue:nswe
bg:white)
update:proc{$ T} {H3 set(text:{FmtTime
T}#'\n'#{FmtDate T})} end
area:60#30)<
```

The record has three fields and combines a declarative with a procedural aspect. We explain the meanings of the fields and why each chooses the approach it does. The 'desc' field is purely declarative. It defines a label widget with white background (`bg:white`) that is glued to its surroundings in all four directions (`glue:nswe`). The label widget is accessible by its handle `H3`, which is an object, used for the dynamic behavior. Note that the 'glue' field is a "declarative" specification of the resize behavior, i.e., of a "dynamic" aspect. That is, sometimes it is possible to "push" the declarative approach to specify part of the dynamic behavior. This is possible in areas that do not need a lot of expressiveness, e.g., resize behavior. It is advantageous to use the declarative approach if this is the case. The 'update' field is purely procedural. It defines a procedure that calls the handle `H3` to set a new text. The procedure's argument `T` is the data to be displayed; the procedure is free to display it as it sees fit. Since the refresh behavior can be quite complex, the procedural approach is appropriate. The 'area' field is declarative. It gives information that is used as input for the component that chooses the best view to be displayed.

6.4 What models are used

FlexClock applies the two quality criteria of concern that have been mentioned:

- *Separability*: each view is independently defined with respect to others. Modifying a view is a local modification without effect of the rest of the code.
- *Correlability*: all defined views are connected together throughout the view chooser, which is also separated from the rest of the code.

FlexClock uses only a few models; in particular a presentation and a dialog model. These models co-exist within the same language and programming environment. By analogy, consider type declarations in a programming language. They are part of the program, yet they play another role than the language operations.

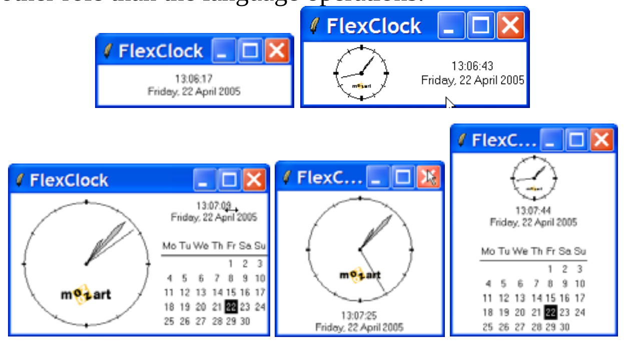


Figure 6. The FlexClock application.

6.5 Requirements for underlying platform

We see the following requirements for a platform to support the mixed approach:

- Strong data structure support, including construction and manipulation of dynamic record-like structures.
- Supports a functional language, for data structure manipulation.
- Supports embeddability: functions are values that can be embedded in data structures. This is important so that declarative data can contain procedural parts.
- Supports lightweight concurrency (threads, events).
- Has a declarative UI toolkit. The toolkit should allow specifying declaratively all parts of a UI that can take an advantage from it.

7 Conclusion

A design space and new method for developing context-sensitive UIs have been introduced. The design space is assumed to help designers to clearly locate, identify and separate the events that cause of change of context as well as the possible reconfigurations of the UI. The new development method highlights several benefits:

- ♦ It is *model-based*: this approach outreaches traditional procedural programming by its ability to record the knowledge required to manage context-sensitivity rather than programming it directly. This knowledge can be reused for a wide variety of contexts, including computing platforms and physical environments instead of redeveloping it for each target platform. In FlexClock for instance, a presentation model hold the different views, a dialog model considers the user interaction on these views, and a design model manages the transitions between views after a window resize.
- ♦ It is based on *separability of models*: as several models can be considered for an interactive application, they can be implemented in active models in various ways. Separating relevant aspects of each model in a separate model allows the developer to minimize the impact of change if a new context should be considered. In FlexClock for instance, defining a new view is simply a matter of inserting three lines of code, without modifying the rest of the code.
- ♦ It produces *readable, compact and effective code*: developing a CSUI in a typically procedural manner requires hundreds of lines of code while this method concentrates the code on the information needed to describe a change of context. In FlexClock, each response rule can be read by a human actor and interpreted by the system. The size of the program file is reduced thanks to the declarative approach for static aspects and procedural approach for dynamic aspects.

8 Acknowledgements

We gratefully acknowledge the support of the SIMILAR network of excellence, the European research task force creating human-machine interfaces similar to human-human communication (http://www.similar.cc), Sixth Framework Program (European Commission, FP6-IST1-2003-507609) and the Salamandre project, Région Wallonne, Initiatives III (Walloon Region, Convention n°001/4511). For more information on the specification language and tools, see http://www.usixml.org.

9 References

[1] K.A. Bharat and L. Cardelli, “Migratory Applications Distributed User Interfaces,” Proc. of ACM Conf. on User Interface Software Technology UIST’95, ACM Press, New York, pp. 133-142, 1995.

[2] G. Calvary, J. Coutaz, and D. Thévenin, “Embedding Plasticity in the Development Process of Interactive Systems,” Proc. of Workshop on User Interfaces for All UI4ALL’2000, ERCIM Press, 2000.

[3] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and Ch. Ef-Stratiou, “Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences,” Proc. of ACM Conf. on Human Factors in Computing Systems CHI’2000, ACM Press, New York, pp. 17-24, 2000.

[4] M. Crease, S. Brewster, and Ph. Gray, “Caring, Sharing Widgets: A Toolkit of Sensitive Widgets,” Proc. of BCS Conference on Human Computer HCI’2000, Springer-Verlag, Berlin, pp. 257-270, 2000.

[5] A.K. Dey and G.D. Abowd, “Support for adapting applications and interfaces to context,” In Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces, Seffah, A. and Javahery, H. (eds.), John Wiley & Sons, 2003.

[6] J. Eisenstein, J. Vanderdonckt, and A.R. Puerta, “Adapting to Mobile Contexts with User-Interface Modeling,” Proc. of IEEE Working Conference on Mobile Computer Applications WMCSA’2000, IEEE Press, Los Alamitos, 2000.

[7] D. Grolaux, QTk Module, March 13, 2000. Accessible at http://www.info.ucl.ac.be/people/ned/qtk/http-html/index.html

[8] D. Hartmut, U. Malinowski, T. Kuhme, and T. Schneider-Hufschmidt, “State of the Art in Adaptive User Interfaces,” in Adaptive User Interfaces, M. Schneider-Hufschmidt (ed.), North Holland, Amsterdam, pp. 1-48, 1993.

[9] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lopez, “UsiXML: a Language Supporting Multi-Path Development of User Interfaces,” Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS’2004 (Hamburg, July 11-13, 2004), Lecture Notes in Computer Science, vol. 3425, Springer-Verlag, Berlin, pp. 207-228, 2005.

[10] M.T. Maybury and W. Wahlster, “Readings in Intelligent User Interfaces,” Morgan Kaufmann, San Francisco, 1998.

[11] Mozart Consortium, “The Mozart Programming System (Oz 3),” January 1999. Accessible at http://www.mozart-oz.org

[12] J.K. Ousterhout, “Tcl and the Tk Toolkit,” Addison-Wesley, Reading, 1994.

[13] F. Paternò, “Model-based Design and Evaluation of Interactive Applications,” Springer Verlag, Berlin, November 1999.

[14] R. Pausch, M. Conway, and R. DeLine, “Lessons Learned from SUIT, the Simple User Interface Toolkit,” ACM Trans. on Office Inf. Systems, vol. 10, no. 4, pp. 320-344, 1992.

[15] A.R. Puerta, “A Model-Based Interface Development Environment,” IEEE Computer, vol. 14, no. 4, pp 40-48, July/August 1997.

[16] A.R. Puerta and J. Eisenstein, “Towards a General Computational Framework for Model-Based Interface Development Systems,” Proc. of ACM Conf. on Intelligent User Interfaces IUI’99, ACM Press, New York, pp. 171-178, 1999.

[17] G. Smolka, “The Oz Programming Model,” Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 1995.

[18] P. Szekely, P. *et al.*, “Declarative Interface Models For User Interface Construction Tools: The MASTERMIND Approach,” Proc. of IFIP Working Conference on Engineering the User Interfaces EHCI'95, L. Bass and C. Unger (eds.), Chapman & Hall, London, pp. 120-150, 1995

[19] P. Szekely, “Retrospective and Challenges for Model-Based Interface Environments,” Proc. of 2nd Int. Workshop on Computer-Aided Design of User Interfaces CADUI’96 (Namur, 4-6 June 1996), J. Vanderdonckt (ed.), Presses Universitaires de Namur, Namur, pp. xxi-xliv, 1996.

[20] D. Thevenin and J. Coutaz, “Plasticity of User Interfaces: Framework and Research Agenda,” Proc. of IFIP Conf. on Human-Computer Interaction Interact’99 (Edinburgh, Sept. 1999), IOS Press, Amsterdam, pp. 110-117, 1999.

[21] J. Vanderdonckt and F. Bodart, “Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection,” Proc. of ACM Conf. on Human Aspects in Computing Systems INTERCHI’93 (Amsterdam, 24-28 April 1993), ACM Press, New York, pp. 424-429, 1993.

[22] J. Vanderdonckt, “A MDA-Compliant Environment for Developing User Interfaces of Information Systems,” Proc. of 17th Conf. on Advanced Information Systems Engineering CAiSE'05 (Porto, 13-17 June 2005), O. Pastor & J. Falcão e Cunha (eds.), Lecture Notes in Computer Science, vol. 3520, Springer-Verlag, Berlin, pp. 16-31, 2005.

[23] Ch. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Green, “ITS: A Tool for Rapidly Developing Interactive Applications,” ACM Trans. on Information Systems, vol. 8, no. 3, pp. 204-236, July 1990.