Task-Driven Plasticity: One Step Forward with UbiDraw

Jean Vanderdonckt and Juan Manuel Gonzalez Calleros

Belgian Laboratory of Computer-Human Interaction (BCHI)
Louvain School of Management (LSM), Université catholique de Louvain
Place des Doyens, 1 – B-1348 Louvain-la-Neuve, Belgium
jean.vanderdonckt@uclouvain.be,
juan.gonzalez@student.uclouvain.be

Abstract. Task-driven plasticity refers to as the capability of a user interface to exhibit plasticity driven by the user's task, i.e. the capability of a user interface to adapt itself to various contexts of use while preserving some predefined usability properties by performing adaptivity based on some task parameters such as complexity, frequency, and criticality. The predefined usability property considered in task-driven plasticity consists of maximizing the observability of user commands in a system-initiated way driven by the ranking of different tasks and sub-tasks. In order to illustrate this concept, we developed UbiDraw, a vectorial hand drawing application that adapts its user interface by displaying, undisplaying, resizing, and relocating tool bars and icons according to the current user's task, the task frequency, or the user's preference for some task. This application is built on top of a context watcher and a set of ubiquitous widgets. The context watchers probes the context of use by monitoring how the user is carrying out her current tasks (e.g., task preference, task frequency) whose definitions are given in a run-time task model. The context watcher sends this information to the ubiquitous widgets so as to support task-driven plasticity.

Keywords: adaptation of user interface, context-aware adaptation, plasticity of user interface, task-based design, task-driven plasticity, user interface description language.

1 Introduction and Motivations

The rise of ubiquitous computing [20] poses significant challenges for designing User Interfaces (UIs) that are adapted to new contexts of use [3,6,20,22]. In conventional interactive systems, the context of use is both limited (e.g., in terms of screen resolution, available input devices) and known (e.g., a person sitting in front of a PC). As computing platforms become more embedded in our daily environment or carried with us, the surrounding world essentially becomes an interface to virtually any type of interactive system. This implies some major changes in the design of these UIs. Porting the UI of specific systems (e.g., a route planning system) or of traditional, popular applications (e.g., a word processing system) to new computing platforms always faces the challenge of designing a UI that is compatible with the constraints imposed by the new computing platform. For instance, porting the UI of a vectorial drawing system from a PC to a PocketPC not only poses constraints of the screen resolution but also introduces alternative modalities of interaction for which the initial UI was not designed initially. For this purpose, many different strategies have been adopted that affect the initial UI design or not.

Techniques that do not affect the initial design include simple porting (when the initial UI is merely reproduced in contents and shapes to the new platform without any change) or zooming (when zoom in/out is applied to the initial UI to increase/decrease the size of a UI portion currently in use according to a focus of interest). While these techniques preserve the consistency between the different versions, the simple porting may dramatically reduce the available screen real estate while the zooming may induce many operations related to the zoom manipulation. Keeping a high number of menu options displayed continuously also maintains a high level of uncertainty on the UI and a high decision time.

The Hick-Hyman Law [16] specifies that this decision time is proportional to the logarithm of equally distributed options. This may suggest that a single screen with more options is more efficient for target selection than a series of screens with less options. But in this case, the screen density may increase, thus impacting the time for searching an item on the screen. For instance, Fig. 1 shows how a traditional UI for a PC-based drawing application is almost entirely reproduced for a PocketPC. Only the bottom left portion of the drawing UI displays some more options depending on the function selected. The rest of the UI remains constant over time.

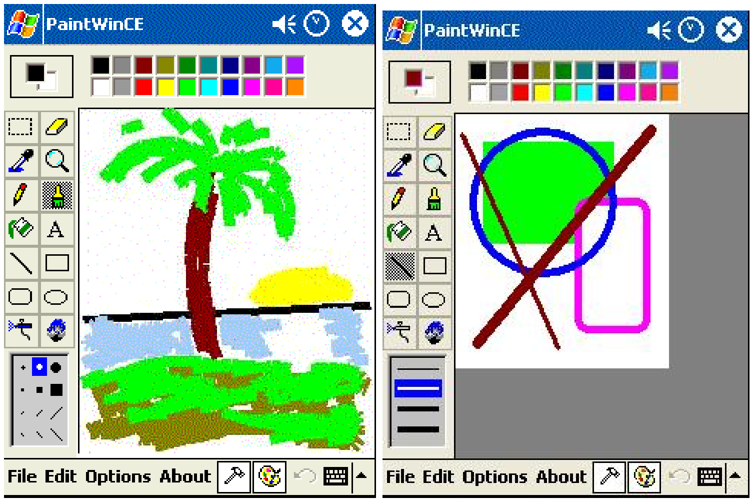

Fig. 1. Simple drawing application on a Pocket PC (from WinCEPaint - http:// www.abisoft.spb.ru/products/cepaint.html)

Techniques that affect the initial design statically may keep the same modality or not when adapting the initial UI. For instance, the UI components can be restructured into tabbed windows gathering functions that are related in principle to the same task. The quality of this gathering highly depends on the quality of the task analysis that has been conducted before. As another example, some Pocket PCs are equipped with physical buttons that can be reassigned to other functions depending on the system running. While this may reduce the functions presented on screen, the assignment may confuse the end user as it is neither systematic nor consistent throughout several interactive systems. In addition, some icons are drawn on these physical buttons, thus making them appropriate for one task (e.g., a particular view for a calendar), but irrelevant for another (e.g., what does a ”Month view” mean for a drawing system?). Similarly, information that was previously assigned to a graphical widget can be submitted to a more general change of modality: sound, voice, or gesture can advantageously replace a graphical widget, like in the sound widgets toolkit [2]. In an ultimate example, related functions can also presented in collapsible tool bars (Fig. 2), like the icons belt of MacOSX or like object toolbars in Corel PaintShop that change according to the object currently being drawn.

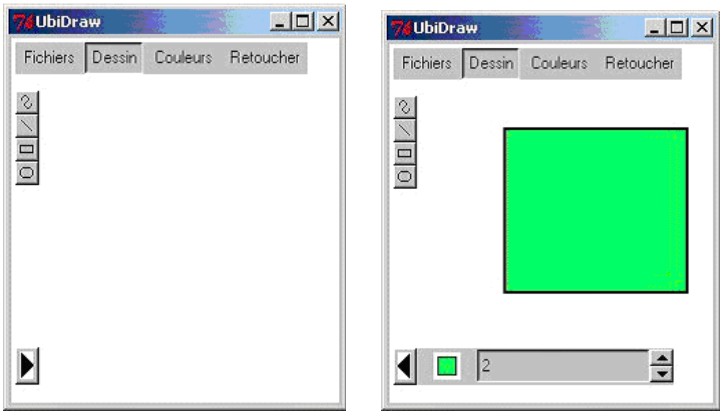

Fig. 2. Collapsible tool bars

Fig. 2 shows how the arrow at the bottom left corner can be expanded to display options related to the object being drawn (one property and its value at a time is displayed, all properties can be scrolled). When the object is finally drawn, the tool bar is collapsed. If another object is input, the arrow is expanded again with other similar properties. Techniques that affect the initial design dynamically open the door to yet unexplored or unexplored capabilities, including the notion of plastic UIs [3,4,7,24]. The plasticity of UIs concerns the capacity of a multi-context UI to preserve usability properties across the various contexts of use, the context of use being defined here as a triple (user, platform, environment) [3]. To exhibit such a capability, some reconfiguration of the UI is often needed. The reconfiguration of UI widgets such as dialog boxes, controls, menu bars and pull-down menus is an example of a physical adaptation. Another possibility is to adapt the very task that the system is to perform. Browne, Totterdell, and Normann [1] present a classification of adaptations in which they observe and regret that most adaptive systems embed hard wired mappings from the set of states to the set of possible adaptations, thus making the adaptivity mechanism rather inflexible. To go one step further than this type of adaptivity, while considering the plasticity, we would like to investigate to what extent UI can be ”plastified” at a higher level of concern than the physical one.

For this purpose, the remainder of this paper is structured as follows: Section 2 reports on some related work on the different levels of plasticity that have been explored so far. Section 3 describes UbiDraw, a vectorial drawing system whose UI supports task-driven plasticity based on a small toolkit of task-driven plastic widgets, called UbiWidgets. This application has been chosen because it is not a trivial UI: it is not a simple form web-based application, for which multiple adaptation mechanisms have been considered so far. Section 4 investigates the effect of using UbiWidgets on the user preference by conducting some usability testing. Finally, Section 5 concludes the paper by summarizing the advantages and shortcomings of this approach, mainly through properties of interest.

2 Related Work

Since the notion of plasticity has been introduced [24], many different works have been dedicated to experiencing how to implement an interactive system that satisfies this property. The notion of plasticity leaves open the usability or quality properties (e.g., [12]) with respect to which some level of usability should be maintained and leaves open the contextual characteristics with respect to which the UI should be made plastic. In the Cameleon Reference Framework [3], the context of use is defined as a triple (user, computing platform, environment), each of these dimensions being equipped with relevant contextual characteristics. In particular, the UsiXML User Interface Description Language (UIDL) [27] is compliant with this framework and deploys a series of attributes for each of these three dimensions. Consequently, any potential variation of one or many of these attributes may represent a change of context with respect to which the UI should be adapted. Of course, not all such variations should be supported, only those which are really significant.

The mechanism of the *software probe* for sensing the context of use has been explained in [4]: it allows deploying interactive systems that constantly probe the context of use for a significant change and that reflect such a change into a UI adaptation. As far as we know, this adaptation is performed at the level of the *Final UI* [3]. Jabarin demonstrated how to implement efficient software architecture for such a final-UI level plasticity [17]. Schneider *et al.* [21 introduced abstract user interfaces whose implementation is independent of the underlying computing platform and that offers multiple representations of concrete UIs for the same description. Therefore, the plasticity is located at the *Concrete UI* level as defined in the Cameleon Reference Framework [3]. All widgets, although called abstract, belong to a Graphical UI. They should not be confused with a AUI belonging to the Abstract UI level [3]. Crease *et al.* [5] introduced a toolkit of context-aware widgets that embed plasticity at the *Abstract UI* level [3]: in this toolkit, widgets have been abstracted with respect to the underlying physical environment so as to form platform-independent widgets. These widgets can also change their interaction modality.

Hence, the plasticity can be declined at any level of the Cameleon Reference Framework as noticed in [8,17], but so far only the lower levels of this framework have been successfully investigated. The only noticeable exception that we are aware of is the system of Comets [8], that propagates interaction needs from the final UI to the task and domain level through concrete and abstract UIs via a set of logical mappings. The support for plasticity is therefore distributed continuously from the final UI (lowest level) to the task and domain level (topmost level).

Our work differs from the aforementioned initiatives in that it drives the plasticity mechanism from a task model located at the task & domain level. It is then propagated downwards to dedicated widgets. A change of the context of use is firstly interpreted in terms of a task variation that is then reflected into the Concrete UI level and Final UI level, respectively. The difference between Comets [8] and UbiWidgets is that the task definition is embedded in a Comet that is developed fit-to-the purpose, while UbiWidgets is based on a mechanism exploiting a task model dynamically. This makes the system independent of any task. In addition, the concrete UI level is constantly modeled via a CUI as defined in the UsiXML (User Interface eXtensible Markup Language – http://www.usixml.org) [27] and the navigation is specified thanks to a system of screen transitions [26]. Not all attributes used in a UsiXML-compliant CUI are used here though, only a subset of them. On the other hand, the Comets maintain a perpetual correspondence between the Comet type (which is aware of the task it is supporting) and the FUI through AUI and CUI, thus making it more flexible than UbiWidgets supporting only the CUI level.

3 UbiDraw: A Task-Driven Plastic Drawing System

This section is structured as follows: first, a general overview of UbiDraw is provided that shows how the UI is adaptive with respect to the users' task; then, the underlying software architecture is explained, along with its context watcher; finally, UbiWidgets, the toolkit of widgets supporting plastic-driven plasticity, is described.

3.1 General Overview of UbiDraw

UbiDraw was developed using Mozart environment [28] and its graphical toolkit Qtk [13]. This environment is by definition multi-platform since it offers an implementation layer where a system is implemented once, and running similarly on Linux, Windows, and Mac platforms. Qtk has been itself implemented on top of the Mozart environment based on the Oz programming language, which is a multi-paradigm programming language. Qtk has been used similarly to implement FlexClock [14].

UbiDraw provides four set of drawing functionalities grouped by similarity in a toolbar attached to an item of the menu bar: File, Draw, Options, and Retouch. Every toolbar can be displayed at different locations of the main application window depending on the size and resolution of the application running on a particular platform. Each group may be displayed in three different ways according to its status (Fig. 3):

1. *Hidden*: all icons of the toolbar attached to the menu item are not visible.
2. *Vertically displayed*: all icons are arranged in a vertically-displayed tool bar.
3. *Horizontally displayed*: all icons are arranged in a horizontally-displayed tool bar.

Fig. 3 graphically depicts these three possible displays: Fig. 3a has the “File” and “Draw” toolbars displayed while the “Options” and “Retouch” toolbars are hidden so as to maximize the screen real estate (here, of a PocketPC running UbiDraw); Fig. 3b has the toolbar “Retouch” in vertical state since it is currently being displayed in a vertical way when activated; Fig. 3c has the “Options” and “Retouch” tool bars in horizontal state since they are displayed horizontally corresponding to the active menu items. Each toolbar does not necessarily displays all icons of the group: its size can range from none (when its status is hidden) to maximum (when all icons are displayed either in vertical or in horizontal status).

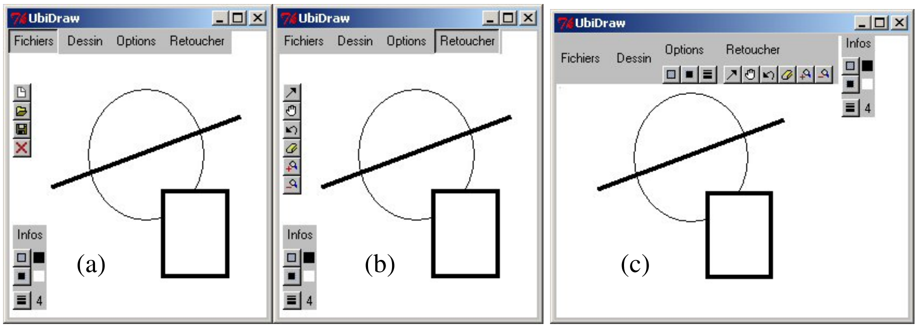

Fig. 3. The three different possible displays of tool bars

In order to determine the size of a non-hidden toolbar and how many icons should be displayed, UbiDraw is relying on a priority scale system where the icons being displayed are regulated by 3 priorities: the last icon being clicked, the rank representing the users’ preference/need for this icon, and the amount of clicks on this icon. Therefore, the higher the priority of an icon is, the more likely it will be displayed. In this way, UbiDraw can determine at run-time the UI configuration to be displayed. Fig. 4 reproduces a situation before and after run-time plasticity where the horizontal screen resolution has been increased.

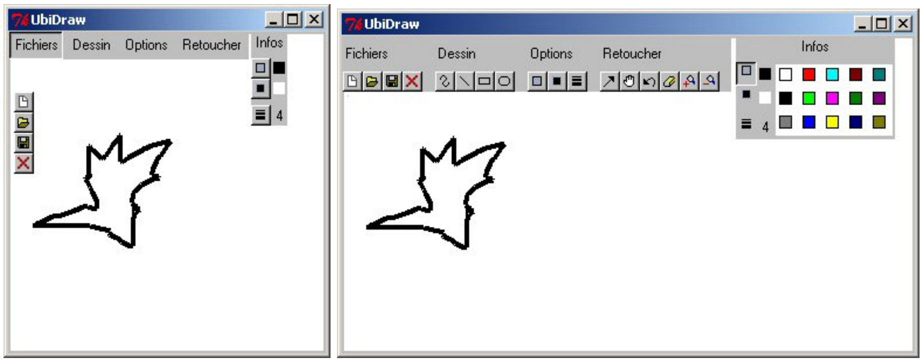

Fig. 4. UbiDraw before and after horizontal resizing of the main window

3.2 Software Architecture of UbiDraw

If we consider the process of plasticity with respect to a view of the software architecture, its processing can be located at different places [12]:

- *At the UI component*: the plasticity is then embedded in the widget level and becomes transparent for the developer;
- *At the UI adaptation component*: the plasticity is embodied in the component so that it can regulated more flexibly through appropriate techniques, such as production rules, inference mechanisms, decision trees, etc.
- *At the UI control component*: the plasticity is regulated at the highest possible level in the metamodel. In this case, only control rules govern the plasticity. We are not aware of ongoing work regarding this level of plasticity apart in Comets [8].

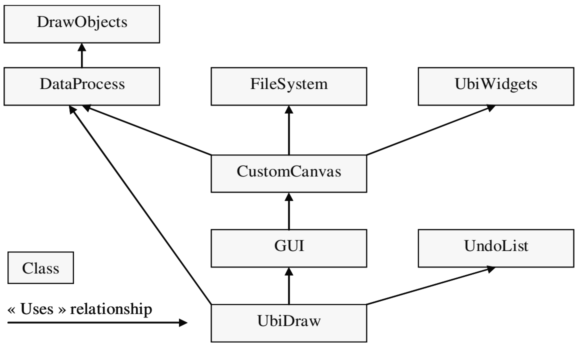

Fig. 5. Software architecture of UbiDraw

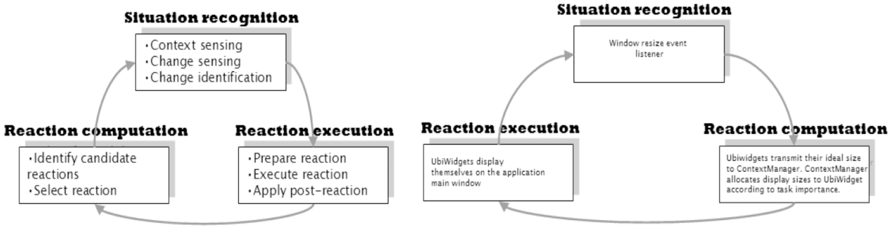
Situation recognition
•Context sensing
•Change sensing
•Change identification
Reaction computation
•Identify candidate reactions
•Select reaction
Reaction execution
•Prepare reaction
•Execute reaction
•Apply post-reaction
Situation recognition
Window resize event listener
Reaction execution
UbiWidgets display themselves on the application main window
Reaction computation
Ubiwidgets transmit their ideal size to ContextManager. ContextManager allocates display sizes to UbiWidget according to task importance.

Fig. 6. Steps of run-time plasticity in UbiDraw

For UbiDraw, we chose the last option. UbiDraw is implemented in several classes (Fig. 5): the main class uses respectively a GUI class (implemented as a concrete UI that will be further described later on), an undo list to keep track of action history, and a dataProcess class that uses the various drawing objects and facilities. The GUI mainly consists of a customCanvas that is in turn decomposed of UbiWidgets (the items of the menu bar and their associated tool bars with icons). The customCanvas selects one of the three states for each UbiWidget depending on the ContextWatcher (that is further described in the next sub-section) that is similar to the context probe [4]. The central component for the adaptation mechanism is the UbiWidget component. It contains a class called ContextWatcher, responsible for the placement of the widgets populating the application, and a class UbiWidget, whose instances are plastic widgets.

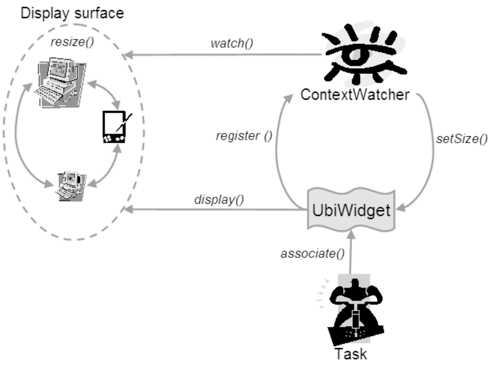

Fig. 7. The run-time mechanism of UbiWidget

Each drawing task of each group is assigned to a UbiWidget, which registers itself to the contextWatcher (Fig. 7) that assigns an initial size. Depending on that status, the UbiWidget displays itself or not. If the context changes, that is if the size of the main application window changes, the contxtWatcher, watching this display surface, is notified and, after calculation, sets a new status and size for each UbiWidget. Ubidraw is composed of a set of components, each assuming a set of functionalities of the application. Fig. 6 shows a general framework identifying several steps for run-time plasticity as it is implemented in UbiDraw. These steps are:

1. **Situation recognition** involves sensing the context, detecting context change and identify context change. In the case of UbiDraw window resize listener triggers the computation of a reaction;
2. **Computation of a reaction** consists in the following: identify candidate reaction, select candidate reactions. UbiDraw has one possible reaction i.e. recalculate layout, its calculation mechanism is explained below
3. **Execute reaction** consists of three steps: prepare the reaction, execute and close the reaction.

UbiDraw applies instantaneously a reaction result. Adaptation with UbiDraw always results from a user initiative (either he/she resizes a window or uses a different platform). Consequently, no particular precaution has to be taken to execute a reaction and there is no need to incorporate an initiative step since it the adaptive UI always triggers adaptivity after a significant change of context occurs. For this purpose, UbiDraw contains a watch method whose main algorithm is explained in pseudo-code below.

```
meth watch()
  Sx={QTk.wInfo width(@canvashandle)}
  Sy={QTk.wInfo height(@canvashandle)}

  % LeftSize provides information on space available
  LeftSizeX={NewCell Sx}
  LeftSizeY={NewCell Sy}

  % ScrollX determines where to locate UbiWidgets
  ScrollX={NewCell 0}

  % StatusList specifies if a UbiWidget should be displayed
  StatusList = {List.make {List.length @ubiwidgets $} $}

  in

  % UbiWidgets are sorted according to their rank of importance
  rankedubiwidgets <- {List.sort @ubiwidgets
                         fun{$ O1 O2}
                           if O1.rank > O2.rank
                           then
                              false
                           else
                              true
                           end
                         end}
    % First selection of UbiWidgets to be displayed
   {List.forAllInd @rankedubiwidgets
    proc{$ I UW}

    % If the available size is smaller than the minimal size of
    % the UbiWiget, the nit will be undisplayed. If not, it
    % will be displayed.
    if {Access LeftSizeX $}<{UW getMinSizeX($)}
       then
       % UbiWidget will be undisplayed (hidden)
         {UW hide()}
         {List.nth StatusList I $}='Hide'
       else
       % UbiWidget will be displayed and its minimal size will
       % be removed from pool of available space
         {Assign LeftSizeX {Access LeftSizeX $}
          -{UW getMinSizeX($)}}
         {List.nth StatusList I $}='Show'
       end
    end}

    % Now, we know which UbiWidgets will be displayed. The
    % remaining available space is then shared among them.
    % For this purpose, all UbiWidgets coordinates are computed
    % via the Scroll function and the allocated space is then
    % passed to them.
   {List.forAllInd @rankedubiwidgets
```

```
    proc{$ I UW}
     if {List.nth StatusList I $}=='Show'
         then
         % Only the maximum size should be allocated to UbiWidg.
           if {Access LeftSizeX $}<{UW getMaxSizeX($)}
               -{UW getMinSizeX($)}
           then
           % If the space allocated is less than the UbiWidget
           % maximum size, this means that it benefits from
           % remaining available space thanks to the priority
             {UW setCoords({Access ScrollX $} 0)}
             {Assign ScrollX {Access ScrollX $}
              +{UW getMinSizeX($)}+{Access LeftSizeX $}}
              {Assign LeftSizeX 0}
           else
             {UW setCoords({Access ScrollX $} 0)}
             {Assign LeftSizeX {Access LeftSizeX $}
              -({UW getMaxSizeX($)}-{UW getMinSizeX($)})}
             {Assign ScrollX {Access ScrollX $}
              +{UW getMaxSizeX($)}}
            end
       end
    end}
  end
```

3.3 The ContextWatcher

The ContextWatcher is equipped with a method called watch which observers any change in the drawing canvas size and applies the appropriate presentation. In order to compute the most appropriate trasformation the ContextWatcher needs three information from every UbiWidgets registered to it: its minimal size, its maximal size, the ranking of the task it supports. The ranking establishes a priority mechanism. The ContextWatcher sorts the UbiWidgets according to their ranking level and, consequently, the widget with the highest ranking will be rendered first. The placement algorithm will always try to place a maximal number of widgets onto the canvas. Consequently UbiWidget minimal sizes are firstly taken into account. If, considering all minimal sizes, all widgets can not be rendered, the space left by unrendered widgets is distributed, on a first rank first serve, among remaining widgets.

The ContextWatcher communicates to each UbiWidget its actual size, and location onto the canvas. UbiWidget can now draw itself. Some tasks are considered as indispensable to the application. In this case, their ranking can be set to 0. Consequently the widgets that support them will be rendered whatever the available size, even if this size in lower than the min size of the widget. Furthermore the registration mechanism allows widgets to register or unregister dynamically. That is to say that from the moment that a widget provides its minimal size, maximal size and the ranking of the task it supports, it can be integrated into the current UI at run-time. The ContextWatcher communicates their position and size constraints to UbiWidgets. Considering this, UbiWidgets have the faculty to choose between different states. The show method assumes the selection of the appropriate presentation.

Table 1 shows different UbiWidget size allocations over time: in the first three rows, 3 UbiWidgets are being allocated a minimum size, a maximum size, and a rank. If the screen resolution is increased to, say, 55 pixels, then the next three rows show the new mimimum size, the increment, and the final allocated size. The last three rows show the same when the screen resolution has been increased of 90 pixels.

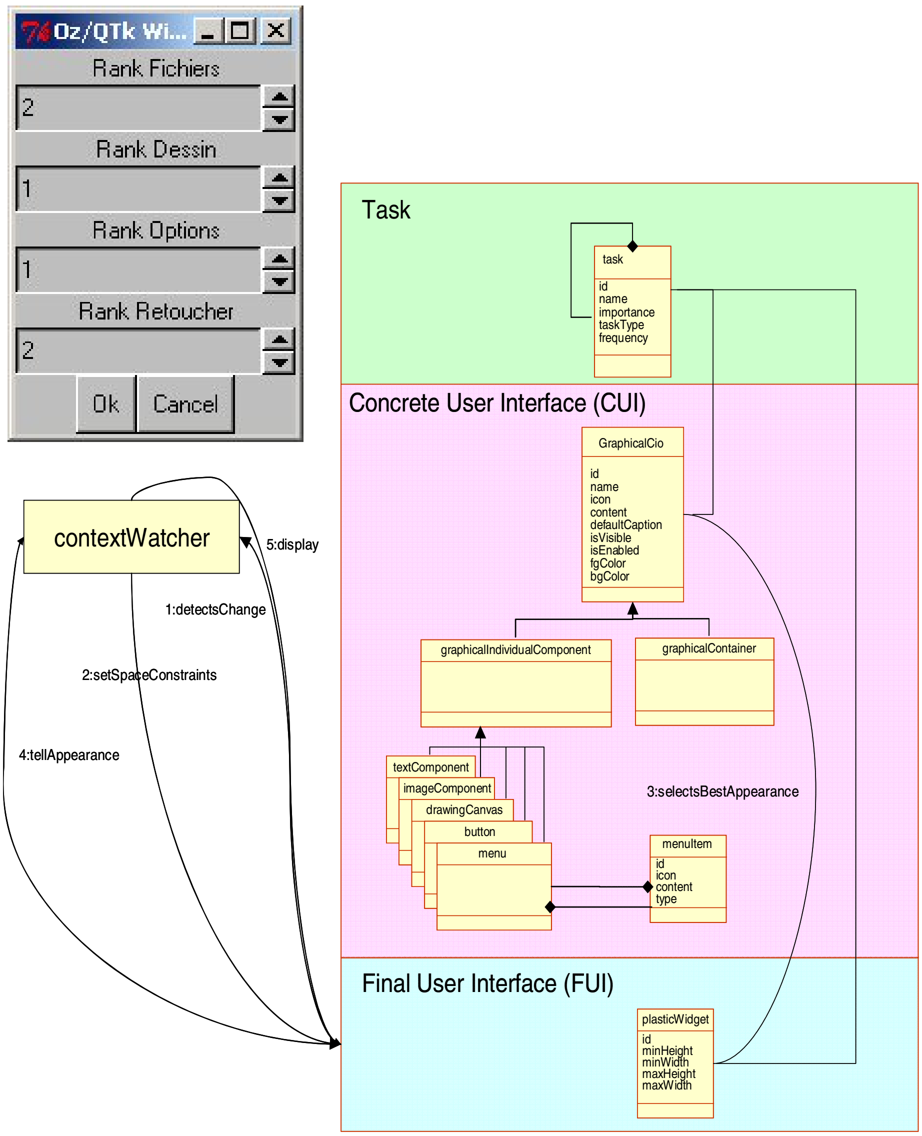
Oz/QTk Wi...
Rank Fichiers: 2
Rank Dessin: 1
Rank Options: 1
Rank Retoucher: 2
Ok Cancel

Fig. 8. Links between the context watcher and the underlying models

Table 1. Example of UbiWidget size allocations

	UbiWidget1	UbiWidget2	UbiWidget 3
Minimum size	20	30	10
Maximum size	40	60	20
Rank	1	2	2
Minimum size	20	30	10
Increment	5	0	0
Allocated size	20 + 5 = 25	30	/
Minimum size	20	30	10
Increment	40	60	20
Allocated size	20 + 20 = 40	30 + 10 = 40	10

Fig. 8 graphically depicts the links between the various UbiDraw components (in particular, the context watcher) and the underlying models: a minimal task model consists of decomposition of tasks into sub-tasks, each with its own parameters; each task is linked to appropriate graphicalCIO (according to UsiXML name) such as textComponent, drawingCanvas, etc. wich are then associated to a menu item in the menu bar. In this way, a simple concrete UI is maintained at run-time from which the context watcher can retrieve properties values (e.g., the rank of each task as represented in the top left corner of Fig. 8) and to which the context watcher can assign new values. The model of the CUI is then interpreted into a final UI thanks to the run-time mechanism of Qtk that stores a GUI in terms of records. Each time a plasticity operation occurs, these records maintaining the models are updated.

4 Usability Analysis by User Testing

Method. In order to test the UbiDraw usability, a questionnaire-based evaluation was performed on a sample of 9 users chosen for their heterogeneous level 1) of expertise in computer manipulation expertise, the fact that they already used an iPaq PocketPC was notably taken into account 2) familiarity with the task at hand that is to say computer supported drawing. Users were asked to perform four different tasks: load an existing drawing, draw a line, draw a rectangle with mid-sized lines and, finally, draw a house. The first three tasks had to be realized as quick as possible. The last task (a higher level task) was proposed to be realized on a desktop-based platform. For this last task, the user was explicitly invited to test the plasticity of the application, that is to say to resize the main window to fit his/her task. Furthermore, the user was asked to indicate which adaptation mechanism s/he favored. These choices refer to heuristics presented in section i.e., ranking click number, click number Ranking. The user was then invited to rank the available tasks according to his preferences. S/he was then invited to test the application with and without his customized ranking. The results were collected in a questionnaire with items represented according to 7-point Likert scale. Items were 7-point graphic scales, anchored at the end points with the terms "Strongly agree" for 1, "Strongly disagree" for 7, and a "Not applicable" (N/A) point outside the scale. Some space was left at the end of the questionnaires for positive and negative aspects, and for further comments.

Results and discussion. From the adaptation perspective it seems that most of the users preferred the ’task ranking’ heuristic to the ’number of clicks’ heuristic. This choice was mainly made by experienced users. This may be explained by the fact that experienced users knew a priori which tasks where more important for them in a drawing application whether inexperienced users wanted to feel the system adapt while using the software. It is also very interesting to note that there was no real consensus between users on the ranking of tasks. This provides us with an unexpected argument foe the need of adaptation mechanisms. Finally, most of the users found that the adaptation mechanism did not disturb them at all in the realization of tasks. Table 2 shows the results collected from this user testing: all participants were able to complete each task in a reasonable amount of time (the last task being of course the longest) and a moderate error rate. Table 3 reports on the final preference for the groups of items. Table 4 gives the average score for each item found in the questionnaire (UbiDraw is easy to use, UbiDraw is more handy than a piece of paper, UbiDraw benefits from a useful context-sensitive help, UbiDraw provides a clear feedback for available functions, UbiDraw enables me to draw what I want, UbiDraw is flexible to use and its adaptation does not disturb task completion, UbiDraw is pleasant to use).

Table 2. Results collected from the user testing

Task	Task completion rate	Speed	Error rate
1	100 %	12 s	0,1
2	100 %	19 s	0,7
3	100 %	18 s	0,7
4	100 %	232 s	1,4

Table 3. Participants’ preference for groups of icons

	File	Draw	Options	Retouch
Rank in first configuration	1	2	3	4
Rank in second configuration	2	1	1	2

Table 4. Results from the questionnaire

Item	1	2	3	4	5	6	7
Average	6	4	6	5	5	6	6

5 Conclusion

In this paper, a drawing application called UbiDraw has been presented that benefit from some original properties:

– *A unique form of plasticity*: a mechanism for UI plasticity of both the presentation and the dialogue levels was implemented in order to maximize the observability [12] of UI widgets throughout task completion.

- *A task-driven mechanism*: the display of the four tool boxes is influenced by the respective task frequencies or ranking of these tasks by the user, thus providing some support to plasticity at the task level rather than at the interface level.
- *An instantiation of the general software architecture for plasticity* as introduced in [4]: thanks to the UbiWidget, the UbiMenu, and the ContextWatcher, the plasticity mechanism is supported in a way that leaves room for further inclusion of other functions and tool boxes without affecting the whole architecture. Again, the general software architecture [4] has been proved applicable to an unreached level of flexibility.
- *A distribution of responsibilities*: it is interesting to notice that the control of screen real estate is not concentrated into one single place: rather than having each widget with total local control or totally governed by a higher level controller, the control of screen space in UbiDraw is distributed between the ContextWatcher level, which is responsible for assigning a location and a portion of the screen to a UbiWidget, and the UbiWidget itself, which is responsible for finding out the most usable presentation among the set of alternatives maintained at the widget level. The algorithm used for that has been briefly outlined.
- *A reasonable usability*: although a preliminary user testing conducted to assess the plasticity of UbiDraw revealed that UbiDraw was rather positively adopted by both novice and expert users, it is important to proceed with more empirical studies. Adaptive UIs are well known to induce some sort confusion in the behavior of the end user, whatever the type of adaptation. Indeed, as soon as there is some automatic change in the UI without the prior demand or consent of the end user, some sort of perturbation may arise. We are not aware of any empirical study that proves the positive impact of plasticity on usability, but there are several studies [10,25,29] that prove that for UI adaptivity. Therefore, we reasonable believe that, since plasticity could be considered as a particular case of UI adaptivity, the observation may apply as well to plasticity. Jameson *et al.* [18] argues for the need of empirical basis for adaptation in general and provides a framework for this purpose. Right now, different usability criteria may be considered in evaluating task-driven plastic UIs like the one implemented in UbiDraw to analyse the perturbation type that may be induced by plasticity. For instance, SUPPLE++ demonstrated that it is possible to automatically generate graphical UIs that positively affect predictability and accuracy [10] for general users or motor-impaired [11]. Since today there is no consensus on how to assess the adaptation in general [18,25], we do not know exactly what metric to use for assessing the plasticity, although it has been recognized that it should be a multi-criteria approach.
- *Consistency*: each UI change resulting from changing the context of use (here, the screen resolution changes) should be uniformly applied and perceived as such by the end user. This may turn out hard to achieve as small close changes of window sizes may be perceived as rather different adaptations of the UI.
- *Continuity*: more general than consistency, each UI change resulting from changing the context of use should preserve the three levels of continuity: perceptual, functional, and cognitive [3,9]. Continuity is also a property that can be significant for adaptation to the context of use, as observed in [9].

These criteria, and perhaps other ones, prove that further investigation is required to fully assess the usability properties of interest that are predefined in the plasticity notion. UbiDraw is on the other hand restricted to a simple context change: window resizing and change of platform. We did not investigate further how other changes of contextual properties may significantly or not affect the UI plasticity.

References

1. Brown, D., Totterdell, P., Norman, M.: Adaptive User Interfaces. Academic Press, London (1990)
2. Brewster, S.: The Design of Sonically-Enhanced Widgets. Interacting with Computers 11(2), 211–235 (1998)
3. Calvary, G., Coutaz, J., Thevenin, D.: A Unifying Reference Framework for the Development of Plastic User Interfaces. In: Nigay, L., Little, M.R. (eds.) EHCI 2001. LNCS, vol. 2254, pp. 173–192. Springer, Heidelberg (2001)
4. Calvary, G., Coutaz, J., Thevenin, D.: Supporting Context Changes for Plastic User Interfaces: A Process and a Mechanism. In: Proc. of Joint Conf. on Human-Computer Interaction IHM-HCI 2001, Lille, September 12-14, 2001, pp. 349–363. Springer, London (2001)
5. Crease, M., Brewster, S., Gray, Ph.: Caring, Sharing Widgets: A Toolkit of Sensitive Widgets. In: Proc. of BCS Conf. on Human-Computer Interaction HCI 2000 "People and computers XIV", Sunderland, September 5-8, 2000, pp. 257–270. Springer, London (2000)
6. Coutaz, J., Balme, L., Alvaro, X., Calvary, G., Demeure, A., Sottet, J.-S.: An MDE-SOA Approach to Support Plastic User Interfaces in Ambient Spaces. In: Stephanidis, C. (ed.) UAHCI 2007 (Part II). LNCS, vol. 4555, pp. 63–72. Springer, Heidelberg (2007)
7. Coutaz, J., Calvary, G.: HCI and Software Engineering: Designing for User Interface Plasticity. In: Sears, A., Jacko, J. (eds.) The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications. Human Factor and Ergonomics series, pp. 1107–1125. Taylor & Francis CRC Press (2008)
8. Demeure, A., Calvary, G., Coutaz, J., Vanderdonckt, J.: The Comets Inspector: Towards Run Time Plasticity Control based on a Semantic Network. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 324–338. Springer, Heidelberg (2007)
9. Florins, M., Trevisan, D., Vanderdonckt, J.: The Continuity Property in Mixed Reality and Multi-platform Systems: a Comparative Study. In: Proc. of 5th Int. Conf. on Computer-Aided Design of User Interfaces CADUI 2004, Funchal, January 14-16, 2004, pp. 321–332. Kluwer Academics Pub., Dordrecht (2004)
10. Gajos, K., Everitt, K., Tan, D.S., Czerwinsky, M., Weld, D.S.: Predictability and Accuracy in adaptive user interfaces. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI 2008, Florence, April 5-10, 2008, pp. 1271–1274. ACM Press, New York (2008)
11. Gajos, K., Wobbrock, J.O., Weld, D.: Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI 2008, Florence, April 5-10, 2008, pp. 1257–1266. ACM Press, New York (2008)
12. Gram, Ch., Cockton, G.: Design Principles for Interactive Software. Chapman & Hall Publishers, London (1996)

13. Grolaux, D., Van Roy, P., Vanderdonckt, J.: QTk: A Mixed Model-Based Approach to Designing Executable User Interfaces. In: Nigay, L., Little, M.R. (eds.) EHCI 2001. LNCS, vol. 2254, pp. 109–110. Springer, Heidelberg (2001)
14. Grolaux, D., Van Roy, P., Vanderdonckt, J.: FlexClock, a Plastic Clock Written in Oz with the QTk toolkit. In: Proc. of 1st Int. Workshop on Task Models and Diagrams for user interface design TAMODIA 2002, July 18-19, 2002, pp. 135–142. Academy of Economic Studies of Bucharest, INFOREC Printing House, Bucharest (2002)
15. Grolaux, D., Vanderdonckt, J., Van Roy, P.: Attach me, Detach me, Assemble me like You Work. In: Costabile, M.F., Paternó, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 198–212. Springer, Heidelberg (2005)
16. Hick, W.E.: On the rate of gain of information. Quarterly Journal of Experimental Psychology 4, 11–26 (1952)
17. Jabarin, B., Graham, N.T.C.: Architectures for Widget-Level Plasticity. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 124–138. Springer, Heidelberg (2003)
18. Jameson, A., Grossman-Hutter, B., March, L., Rummer, R.: Creating an empirical basis for adaptation techniques. In: Proc. of ACM Conf. on Intelligent User Interfaces IUI 2000, New Orleans, January 9-12, 2000, pp. 149–156. ACM Press, New York (2000)
19. Montero, F., López-Jaquero, V., Molina, J.P., González, P.: An Approach to Develop User Interfaces with Plasticity. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 420–423. Springer, Heidelberg (2003)
20. Rekimoto, J., Masanori, S.: Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI 1999, Pittsburgh, May 15-20, 1999, pp. 378–385. ACM Press, NY (1999)
21. Schneider, K.A., Cordy, J.R.: Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems. In: DSV-IS 2002. LNCS, vol. 2545, pp. 28–48. Springer, Heidelberg (2002)
22. Sendin, M., Lores, J., Montero, F., Lopez, V.: Towards a Framework to develop plastic user interfaces. In: Chittaro, L. (ed.) Mobile HCI 2003. LNCS, vol. 2795, pp. 428–433. Springer, Heidelberg (2003)
23. Sottet, J.-S., Calvary, G., Favre, J.-M., Coutaz, J., Demeure, A., Balme, L.: Towards Model-Driven Engineering of Plastic User Interfaces. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 191–200. Springer, Heidelberg (2006)
24. Thevenin, D., Coutaz, J.: Plasticity of User Interfaces: Framework and Research Agenda. In: Proc. of IFIP Int. Conf. on Human-Computer Interaction Interact 1999, Edinburgh, September 1999, pp. 110–117. IOS Press, Amsterdam (1999)
25. Tsandilas, T., Schraefel, M.C.: An empirical assessment of adaptation techniques. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI 2005, Portland, April 2-7, 2008, pp. 2009–2012. ACM Press, New York (2005)
26. Vanderdonckt, J., Limbourg, Q., Florins, M.: Deriving the Navigational Structure of a User Interface. In: Proc. of 9th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT 2003, Zurich, September 1-5, 2003, pp. 455–462. IOS Press, Amsterdam (2003)
27. Vanderdonckt, J.: A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 16–31. Springer, Heidelberg (2005)
28. Van Roy, P., Haridi, S.: Concepts. MIT Press, New York (2004)
29. Weibelzahl, S.: Evaluation of adaptive systems. In: Bauer, M., Gmytrasiewicz, P.J., Vassileva, J. (eds.) UM 2001. LNCS (LNAI), vol. 2109, pp. 292–294. Springer, Heidelberg (2001)