Attach Me, Detach Me, Assemble Me Like You Work

Donatien Grolaux[1,2], Jean Vanderdonckt[1], and Peter Van Roy[2]

[1] School of Management, Information Systems Unit, Place des Doyens,
[2] Dept. of Computing Science and Engineering, Place Sainte Barbe,
Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
ned@info.ucl.ac.be, vanderdonckt@isys.ucl.ac.be,
pvr@info.ucl.ac.be

Abstract. Detachable user interfaces consist of graphical user interfaces whose parts or whole can be detached at run-time from their host, migrated onto another computing platform while carrying out the task, possibly adapted to the new platform and attached to the target platform in a peer-to-peer fashion. Detaching is the property of splitting a part of a UI for transferring it onto another platform. AttAaching is the reciprocal property: a part of an existing interface can be attached to the currently being used interface so as to recompose another one on-demand, according to user's needs, task requirements. Assembling interface parts by detaching and attaching allows dynamically composing, decomposing and re-composing new interfaces on demand. To support this interaction paradigm, a development infrastructure has been developed based on a series of primitives such as display, undisplay, copy, expose, return, transfer, delegate, and switch. We exemplify it with QTkDraw, a painting application with attaching and detaching based on the development infrastructure.

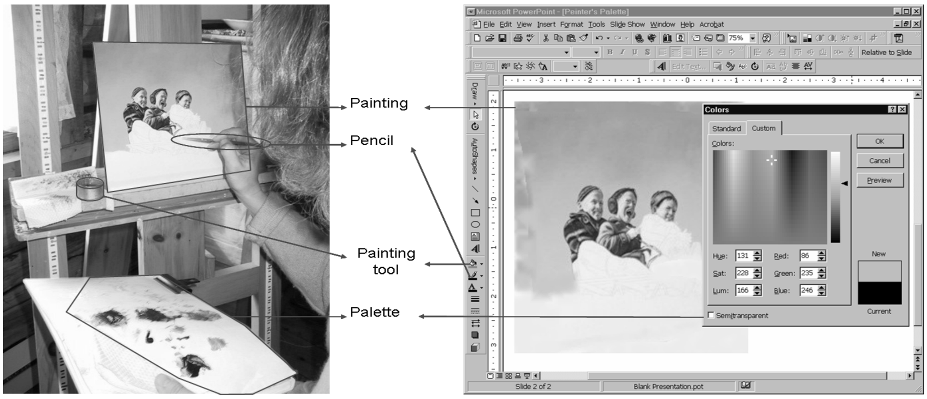
1 Introduction

With the advent of ubiquitous computing and the ever increasing amount of computing platforms, the user is encouraged to work in more varying conditions that were not expected before. From a user's perspective, various scenarios may occur:

1. *Users may move between different computing platforms whilst involved in a task*: when buying a movie on DVD a user might initially search for it from her desktop computer, read the reviews of the DVD on a PDA on the train on the way home from work, and then order it using a WAP-enabled mobile phone.
2. *The context of use may change whilst the user is interacting*: the train may go into a dark tunnel so the screen of the PDA dims, the noise level will rise so the volume of audio feedback increases so it can still be heard.
3. *Users may want to collaborate on a task using heterogeneous computing platforms*: the user decides to phone up a friend who has seen the movie and look at the reviews with her, one person using WebTV and the other using a laptop, so the same information is presented radically differently.

There are many other similar situations where these types of interactions may occur, for example, graphic expert teams doing collaborative drawing tasks using information shared across multiple computing platforms, or a stock market trader who wants to access the same market data on his desktop computer and his mobile phone when she is away from her desk. We can easily extend these scenarios for multi-user communication where users interact from different contexts of use and even the type of coordination and communication that can occur among them depends on a number of aspects related to the context of use. Although more mobile computing platforms exist, they are not always compatible (they do not share the same operating system), communicant (the communication protocols are different), and composable (once together, computing platforms cannot take advantage of the newly available resources to return to another situation when some platform is leaving). Since the User Interfaces (UIs) that are running on these heterogeneous platforms cannot be composed, they are rather inflexible for reconfiguring at run-time and they may impose configurations that are not natural to the user.

For example, when a painter is painting a scene, the painting *is* the main focus of attention, while all tools (e.g., the color palette, the pencil, and the painting tools) remain secondary, available at hand when needed. Unfortunately, this is not the case with most painting/drawing software where the real world is reproduced by a working area representing the painting and a series of menu bars and tool bars containing families of related tools. When many of these bars are displayed, the UI rapidly becomes cluttered so as to reduce the working area to its minimum (Fig. 1). This UI is not considered natural [9] in the sense that tools contained in such bars are not required all the time during interaction, but solely at certain specific moments (e.g., changing the color, increasing the size of the pencil, choosing a painting effect). Of course, the end user can customize the display of tool bars, but this operation remains manual, tedious, repetitive and not related to the main task. Some UIs tend to improve this by displaying toolbars only when they are related to any object manipulated (e.g., an image, a rectangle) and undisplaying them afterwards. For example, PaintShopPro™ includes a 'Tool Options' dialog box that is displayed according the tool currently being selected. Although this partially reduces the screen density, it provokes fast visual change of the UI that may confuse the user [9].



Hue: 131 Sat: 228 Lum: 166 Red: 86 Green: 235 Blue: 246

Fig. 1. Natural world vs. user interface world

The availability of today's computing platforms ranging from the traditional PC and the laptop to handheld PC and pocket PC invites us to address this problem by exploiting interaction between multiple surfaces of interaction [5] at the same time. In the painter example, a more natural UI, i.e. a UI that would mimic more the real world depending on availability of platforms, would be the largest screen used as the main painting area and a Pocket PC used only for displaying tool bars and picking there the right tools on demand.

To support this scenario and any similar situation where the user may want to compose, decompose and re-compose the components of a UI on-demand, depending on users' needs, task requirements and platforms availability, we introduce a new interaction paradigm, called *Detachable User Interfaces* that are characterized by the 'Demi-Plat' set of properties (Fig. 2):

– ***De****tachability*: any UI component of the interactive application of interest can be detached from its host UI, provided it is authorized to do so, while continuing to carrying out the corresponding interactive task.
– ***Mi****gratability*: the detached UI component is migrated from the source computing platform running the interactive application to another target platform, possible equipped with totally different operating systems, protocols, screen resolution.
– ***Pla****stifiability*: the migrated UI component is adapted according to the new constraints posed by the new target computing platform, if needed [3].
– ***At****tachability*: the plastified UI component is attached to any UI running on the target computing platform, if needed.

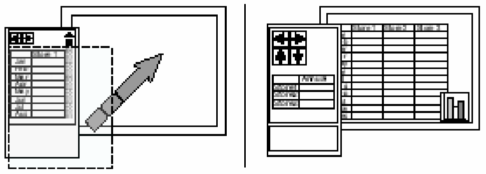


Fig. 2. The basic principle of detachable user interface

The remainder of this paper is structured as follows. The next section 2 summarizes the related work in the domain of dynamically changing UIs on different platforms. Then, the definitions, motivations, the design choices and a definition of the four 'Demi-Plat' properties are provided in Section 3, along with the primitive operations required to support them. Section 4 explains the development infrastructure that we developed to support the interaction paradigm of detachable UIs. Then, a complete implementation is described in Section 5, based on the above scenario of the painter: QTkDraw is an interactive painting software supporting the four properties. In particular, any toolbar can be detached from the initial application to any other computing platform, even running a different operating system (e.g., from a PC to Mac and back), can be automatically adapted to it, and can continue interaction with the main screen. Finally, a conclusion reports on the original points of detachable UIs, some open questions and future work in Section 6.

2 Related Work

In order to uniformly compare existing work we will take the common scenario of the Painter's palette as represented in Fig. 1 and as described in the introduction.

The first steps that have been made towards moving UIs between screens were achieved by virtual window managers capable of remotely accessing an application over the network, such as X-Windows X11 remote displays (http://www.x.org/), Virtual Network Computing (http://www.uk.research.att.com/vnc/), and Windows Terminal Server (http://www.microsoft.com/windows2000/technologies/terminal/ default.asp). It is possible to launch an interactive application locally, but to transfer the UI input/output to another workstation. These solutions are controlled by the underlying operating system with a service that is independent of the interactive application. These solutions suffer from the following drawbacks: the UI cannot control its own transfer since it is independent from the service, the UI can only be moved among workstations of the same operating system (e.g., Unix or Windows), there is no adaptation to the target platform, it cannot be dissociated, and it is a client/server solution (a server that has nothing to do with the interactive application is required to run the solution ; if the server disappears, the interactive application also disappears).

Pioneering work in migration has been done by Bharat & Cardelli [2]: their migratory applications are able to move from one platform to another one at run-time, provided that the operating system remains the same. While this is probably the first truly migrating application, the main restriction is that the whole application is migrated. The situation is similar for multi-user applications when an application should be transferred to another user as in [7]. In *The Migration Project* [1], only the UI is migrated, in part or in whole, from one computing platform to another. At run-time, the user can choose the platform where to migrate. But only web pages are migrated between platforms (thus the example toolbar can be run), a migration server is required and all the various UIs for the different platforms are pre-computed.

Remote Commander [11] is an application that supports all keyboard and mouse functions and displays screen images on the handheld PC, so it can serve as a host for our example's toolbars, but the handheld PC is the only platform capable of welcoming the controls. It is not possible to decompose or recompose UI parts, the portion that is migrated needs to be predefined.

The *Pick & Drop* interaction paradigm [12] supports migration of information between platforms, like other interaction techniques and migration environments such as *i-Land* [15], *Stanford Interactive Mural* [8], *Aura* [14], *ConnecTables* [16]. But these solutions do not support the properties of detachability, attachability and plasticity when migrating a UI across platforms. In addition, all the platforms should belong to the same family, which is rarely the case when people meet or for a single person. For instance, the Stanford Interactive Mural enables user to freely move windows from one screen to another, the screens being displayed on walls, side by side or not, but the whole configuration is predefined and described in a topology model that does not accommodate entries and leavings of different platforms. Only I-AM [4,5] today exhibits the capabilities of platform discovery and UI plasticity at the same time. A meta-UI [4] is defined to control the migration process [10] across various platforms and in varying circumstances, thus releasing the user from having a predefined configuration. In contrast, detachable UIs allow people to migrate parts or whole of the UI by direct manipulation of the parts that can be effectively migrated.

3 Definitions, Motivations, and Design Choices

A UI *migration* is hereby defined as the action of transferring a UI from one source computing platform to a target one, such as from a desktop computer to a handheld device. A UI is said to be *migratable* if it holds the migration ability. A migration is said to be *total*, respectively *partial*, when the whole interactive application, respectively the UI, are migrated [1,4]. If we decompose a UI into the control which is responsible for the UI behavior and the presentation which is responsible for presenting information to the user, *control migration* [1] migrates only the control component while the presentation remains. In *presentation migration* [1], the situation is the inverse: the presentation component is migrated while the control remains on the source platform. When the migration is *mixed* [1], different parts of both the control and the presentation are migrated. To support all these different cases of migration, a special UI is required that will perform the required steps to conduct the migration, such as identification of migration possibility, proposal for migration, selection of migration alternative, and execution of the migration itself. Since these types of migrations and underlying steps require complex handling of UI events and procedures, the UI responsible for migration is even more complex and not always visible to the eyes of the end user. This UI is referred to as the *meta-user interface* in [4], i.e. the UI for controlling the run-time migration of the UI of the interactive systems. A meta-UI could be *system initiated* (the system initiates the migration), *user-initiated* (the user initiates the migration), or *mixed-initiated* (the user and the system collaborate to perform the migration).

A UI *component* is hereby defined as any part or whole of a UI of interest. It can be an individual widget (e.g., a control), a composed widget (e.g., a tool bar or a group box with contained widgets), a container (e.g., an area displaying an activity chart), a child or an application window, or any combination of these. The computing platform is referred to as the complete hardware/software environment that is considered as a whole, including the operating system and the input/output capabilities.

A *detachable UI* is a UI from which any allowed portion can be detached at run-time from one platform, migrated and adapted to another one. We now detail the four main properties of detachable UIs, as referred to the 'Demi-Plat' properties:

3.1 Detachability

Any UI component with its current status of interaction can be detached at any time. Detaching a UI is achieved by dragging a portion of the UI and dropping it outside the UI: the migration could be partial or total, presentation-, control-oriented or mixed, and user-initiated. Different types of detachability exist:

1. *Full screen* when the entire UIs of all applications running on the current platform are detached.

2. *Window* when an entire user/system-selected window or any portion of it is detached. For instance, a whole window within the border, along with its title bar, its menu bar, the scroll bar or captions lines.
3. *Active window* when the windows that has the focus of interaction on the desktop is detached when the detach operation is invoked.
4. *Region* when any user-defined rectangular region of the UI is detached. For instance, a user may select by direct manipulation a rectangle surrounding components subject to detachment.
5. *Fixed region* when a user-defined rectangular fixed region of the platform desktop defined by absolute pixel coordinates.
6. *Widget* when any individual widget is detached.

For example, to detach a palette from a drawing application, a region will be selected. When only a particular tool is required to detach, the widget part will be used instead. The fixed region can be used for instance for the menu bar of an application provided that it has been maximized full screen. In addition to the detachability property, any UI component can be declared detachable or not, splittable or not. *Detachability* decides whether a UI component can be detached to another platform or should remain fixed with the main UI. *Splittability* specifies whenever a composed UI component can be detached in itself, but that none of its sub-components can be detached individually. For example, a color palette can be declared *unsplittable* to avoid widespreading of color schemes on different surfaces. Any component that is contained in an upper-level component that is unsplitttable cannot be detached. The detach mode is invoked by triggering a special function which can be tailored on any supported platform, e.g. a function key (F12) on PC and workstation, a menu item on handheld and pocket PCs. Then, by direct manipulation, the user can visually determine the UI component subject to detachment depending on the cursor position: the component subject to detachment is highlighted. When the cursor is inside an undetachable area, respectively a detachable area, it is transformed into a forbidden sign (🛇) (Fig. 3a), respectively a hand (Fig. 3b) before migration.

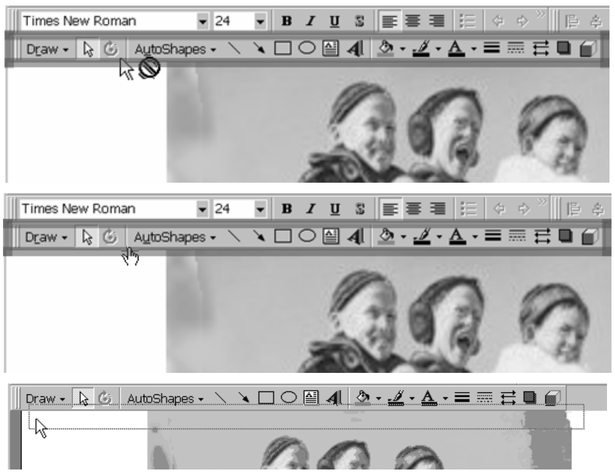


Fig. 3. Detaching a UI component before migration (forbidden area, allowed area, migration)

3.2 Migration

Migration consists of transferring any UI component (presentation and dialogue states) from one platform to another, which can be characterized along four axes:

- Amount of platforms: the migration can be *one-to-one* (from one platform to another one) or *one-to-many* (from one platform to many platforms).
- Amount of users: the migration is said to be *single-user*, respectively *multi-user*, when it occurs across platforms owned by one user, respectively by many users.
- Amount of platform types: the migration is said to be *one-threaded*, respectively *multi-threaded*, when it occurs between platforms of the same type (e.g. between two PCs), respectively of different types (e.g., from a PC to a PDA that does not necessarily run the same operating system).
- Amount of interaction surfaces: the migration can be *mono-surface*, respectively *multi-surface*, when it occurs from one interaction surface to another (e.g., from screen to screen), respectively from one surface to multiple surfaces [5,6] at the same time (e.g., from one screen to several different screens of various sizes).

For example, the QTkDraw is one-to-one (e.g., the tool bars are transferred from the PC to the Pocket PC), single-user (it is expected to be for the usability of the same user), *multi-threaded* (because of different platforms involved), and *mono-surface* (only the tool bars are migrated to a Pocket PC, although separate tool bars can migrate to different Pocket PCs). To support these configurations, a set of primitives is now defined that will be further supported in the implementation.

Display (UI, platform). Any component of the currently being used UI is displayed on a given platform. In the multi-user case, the display is remote on the other one.

Undisplay (UI, platform). Any component of the currently being used UI being on display on a given platform is erased.

Copy (UI, source, target). Any component of the currently being used UI with its current status of presentation (e.g., activated and deactivated parts) and dialogue (e.g., values already entered) is copied from the source platform to a target platform. This primitive results in having two copies of the same UI component with the status preserved, but which can now work independently of each other. The source and target UIs live their life independently. For example, a first drawing is realized and at a certain timestamp, there is a need to continue with two separate versions of the drawing to expand it with different alternatives.

Expose (UI, source, target). Any component of the currently being used UI with its current status is copied from the source platform to the target platform and frozen. Only the source UI can continue to live, the other being merely exposed to the target platform for viewing purpose and being closed afterwards. For example, one user wants to a show to a colleague the current version of a drawing to get her advice, but does not want to allow her to apply any modification.

Return (UI, target, source). Any component of the currently being used UI with its current status that has been copied previously, after living on its own, can be returned to the platform which initiated it. For example, a drawing that has been separately modified at a certain stage by a colleague can be returned to its originator. Then, the UI of concern disappears from the current platform and appears again in its new state of the platform from where it has been copied.

Transfer (UI, source, target). Any UI component with its status is copied from the source to the target and deleted from the source platform to live its life on the target.

Delegate (UI, source, target). A delegation is defined by a sequence of transfer and return. For example, a user wants to completely delegate the realization of a drawing and recuperate the results when done.

Switch (Source UI, source, Target UI, target). Two UI components of two different UIs with their status are exchanged between a source and a target. The source UI is transferred to the target and the target UI is transferred to the source. For example, when two persons working in a collaborative environment need to swap their work and to continue on each others' work.

The Copy, Expose, and Transfer primitives can be made multi-user, multi-platform by repeating the same process for multiple platforms at the same time.

3.3 Plasticity

The property of plasticity [3] is defined as the property of adapting a user interface depending on the change of the context of use, while preserving predefined usability conditions. In our case, the UI that is immigrated in the new target computing platform can be submitted to the process of plastification, if it holds the plastifiability. For instance, if a toolbar is moved from a desktop PC to a handheld PC, and only this component, then the toolbar can be magnified by increasing the size of each button belonging to the toolbar. Or the initial size of the toolbar can be preserved. If the size of the UI element that emigrated from the source platform is larger that the screen resolution of the target platform where it should immigrate, then it can be submitted to a series of plasticity rules, such as widget replacement, size reduction, text summarization techniques, repositioning of widgets, and reshuffling of components. For this purpose, we used the PlaceHolder technique (http://www.mozart-oz.org) to contain any part of the UI that can be submitted to plasticity. Thanks to this system, a container is generated at run-time that only knows its components after firing the appropriate plasticity rules. Once these subcomponents are known, their size and locations can be computed so as to determine the final size of the PlaceHolder.

3.4 Attachability

The *attachability* is defined by analogy with detachability since it is the inverse of detachability. Any UI component of interest can be attached back to its previously detached UI or to any other UI. Thanks to the attachability property, it is possible to support a UI development process by copy/paste. In traditional visual programming, any UI is drawn by composition of widgets dragged from a tool palette onto a working area. This process does not support per se composition of new UI from previously defined UIs. Of course, it is possible to copy/paste parts of the widgets, but there is a need to redraw everything. In Programming by demonstration, a UI that will be implemented is demonstrated and then derived. Here, when a UI component is attached to another UI component, they are automatically merged so as to create an entirely new UI. There is no need to redraw the UI and this operation can be done at run-time rather than at design-time. Or any selected component from one UI can be copied, dragged and dropped into another UI to compose a new UI merging functions which are the sum of functions provided by the individual components.

4 Development Infrastructure for Detachable User Interfaces

To support the above properties, we have developed techniques for making UI detachable by relying on the Mozart-Oz environment (www.mozart-oz.org) that intrinsically supports distributed computing. This environment is multi-platform: a freely downloadable version exists for Linux, Windows, and Macintosh operating systems, thus providing us the advantage that any UI that will be made detachable thanks to this infrastructure will be able to migrate between *any* operating system in a *peer-to-peer* fashion as there is no need to run a server. Each interactive application can manage its own detachability and attachability. We now describe the indirection mechanism that supports at the application level the properties of detachability, attachability, and migration. The toolkit creates a window out of a declarative data structure, called an Oz record, similar in expressiveness to XML. This data structure describes many (if not all) aspects of the window that are specifiable declaratively: the widgets that compose the window, their initial states, their geometry inside the window, their behavior upon window resizing, etcetera. Also, using the handle parameter of the widgets in the description record, controller objects are created that allows a dynamic interaction between the UI and the application once the window has been created. In summary, this toolkit uses first a record `DR` to create the window in its initial state; during the creation of the window, `Oi` objects are created to further control individual widgets in an object-oriented imperative way.

Let us build a migratable window from a description record `DR` (Fig. 4). The handle parameters of `DR` are bound to $\mathtt{P_i}$ proxy objects instead of the usual `Oi` objects, and a `CM` communication manager object is created. The original `DR` record is also stored by `CM`. At this stage, there is no display D site yet.

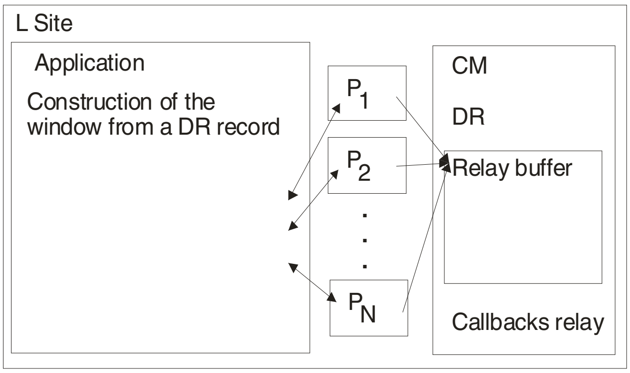


Fig. 4. Definition of a migratable window

The proxy objects will act as the local representatives of the actual `Oi` widget objects. There are at least two ways to implement Pi objects:

1. `Pi` objects reflect the whole semantics of their corresponding `Oi` objects. They don't rely on any `Oi` object to serve their purpose. This requires a huge amount of development and maintenance work: each widget must exist in an actual and proxy flavor.
2. `Pi` objects are generic objects that relay application messages to their currently connected `Oi`. This is the solution used by our toolkit. As a side effect, a `Pi` object cannot work correctly unless it is connected to an actual `Oi` object. When not connected, method invocation messages are buffered; only when connected these messages are processed by the display site.

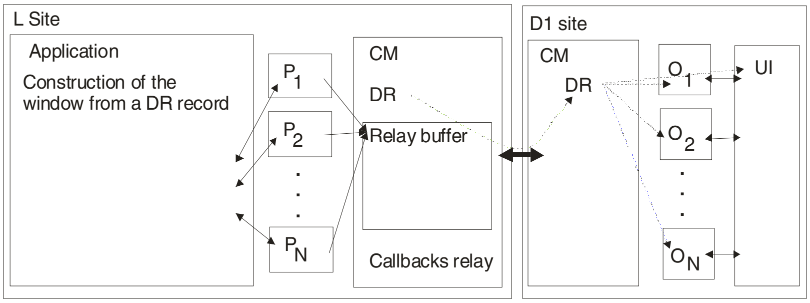


Fig. 5. Configuration of a migrated window

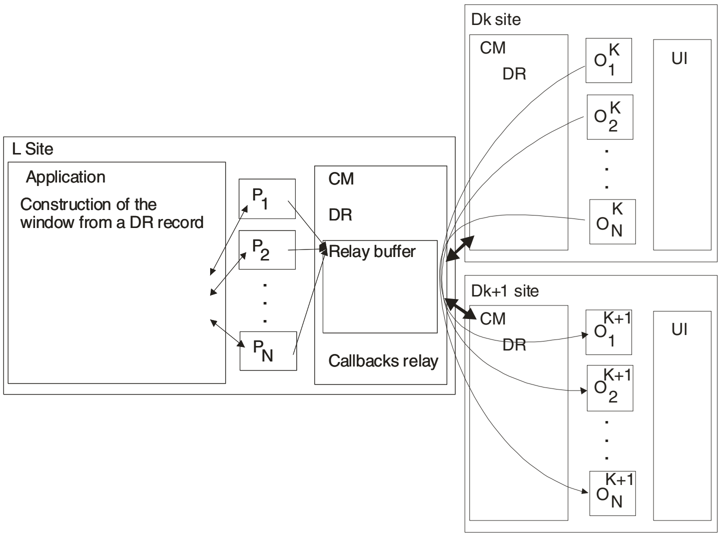


Fig. 6. Migration to another site

When the first remote display site $\mathtt{D_1}$ connects to the $\mathtt{CM}$ of $\mathtt{L}$ (Fig. 5), the $\mathtt{DR}$ record is sent. $\mathtt{D_1}$ creates the effective UI and the $\mathtt{O_i}$'s from this $\mathtt{DR}$. At this moment, the application can start working with the migrated UI; the buffered messages are sent first. When migrating to a $\mathtt{D_{k+1}}$ site (Fig. 6), the actual user interface and $\mathtt{O^{k+1}{}_i}$'s are still created from the $\mathtt{DR}$ record. However the visual aspects of the widgets might have changed since their creation time, and the $\mathtt{D_{k+1}}$ site should reflect that. Let's define:

- $\mathtt{VA(O)=\{v}$ | $\mathtt{v}$ is a visually observable aspect of the widget controlled by $\mathtt{O\}}$
- $\mathtt{get(O,v)}$: returns the current value of the visual aspect $\mathtt{v}$ of $\mathtt{O}$.
- $\mathtt{set(O,v,s)}$: sets the visual aspect $\mathtt{v}$ of $\mathtt{O}$ to $\mathtt{s}$.

After the user interface and $\mathtt{O^{(k+1)}{}_i}$'s are created at $\mathtt{D_{k+1}}$, $\forall$ $\mathtt{i}$ in $\mathtt{1..N}$, $\forall$ $\mathtt{v}$ in $\mathtt{VA(O^k{}_i)}$: $\mathtt{set(O^{k+1}{}_i,v,get(O^k{}_i,v))}$. In practice, $\mathtt{P_i}$'s are used to store the visual parameters: $\mathtt{P_i}$'s contain a dictionary that supports the operations: $\mathtt{get(P,v)}$: returns the value of the key $\mathtt{v}$ of the dictionary of $\mathtt{P}$ and $\mathtt{set(P,v,s)}$: sets the key $\mathtt{v}$ of the dictionary of $\mathtt{P}$ to $\mathtt{s}$. When disconnecting from a display site $\mathtt{D_k}$, $\forall$ $\mathtt{i}$ in $\mathtt{1..N}$, $\forall$ $\mathtt{v}$ in $\mathtt{VA(O^k{}_i)}$, $\mathtt{set(P_i,v,get(O^k{}_i,v))}$. When connecting to a display site $\mathtt{D_{k+1}}$, $\forall$ $\mathtt{i}$ in $\mathtt{1..N}$, $\forall$ $\mathtt{v}$ in $\mathtt{VA(O^{k+1}{}_i)}$, $\mathtt{set(O^{k+1}{}_i,v,get(P_i,v))}$.

5 The QTkDraw Demonstration Application

In this section we demonstrate the results of using the development infrastructure explained in Section 4 for the QTkDraw application that serves as a demonstration. We then applied the development infrastructure to obtain the detachable UI reproduced in Fig. 7, where two UI components were declared detachable, splittable. Fig. 7 shows a screenshot of the application before detaching the toolbar (left arrow) and the color bar (right arrow). This demonstration is available at http://www.isys.ucl.ac.be/bchi/members/dgr/palette.html.

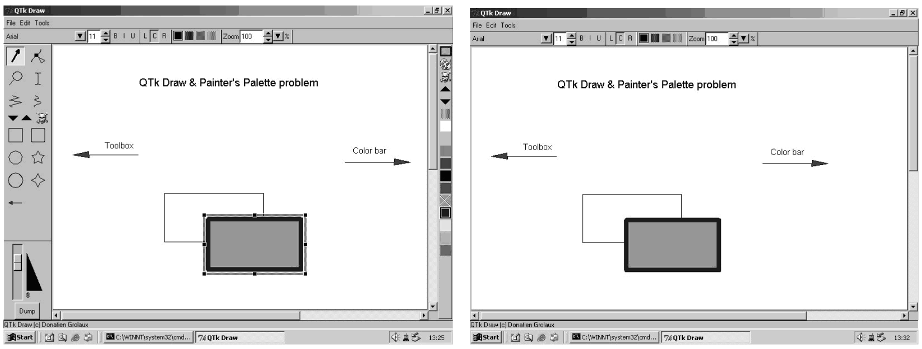


Fig. 7. Detaching the toolbar and the color bar from a desktop to PocketPCs

Since the application is developed on top of Tcl/Tk which is itself running on several computing platforms (i.e., Linux, Windows, and Macintosh) the native Look &

Feel of the platform is preserved. Therefore, the same application can run on all these platforms without changing one line of code: they are simply re-interpreted on top of the development infrastructure.

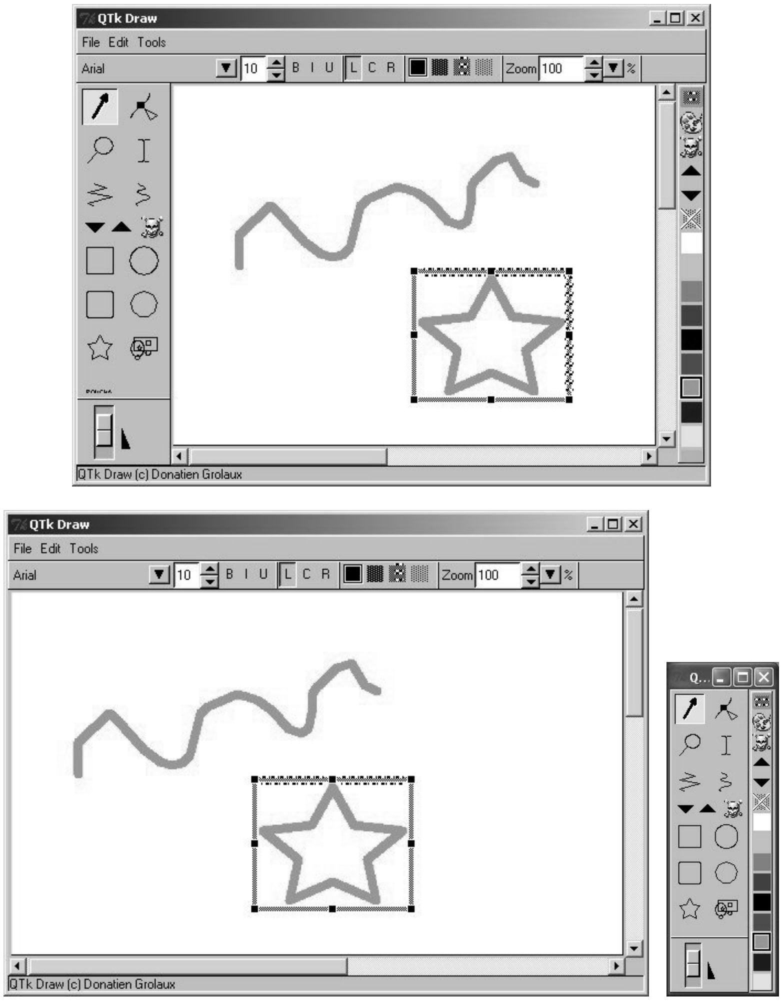


Fig. 8. Detaching the toolbar and the color palette from a TabletPC and attaching them together on a desktop

In Fig. 7 and Fig. 8, the screenshots have been taken in the Windows environment, but they could have been taken in any other environment equally. Even among different platforms running the same operating system and window manager (here, Windows), the application can be run on different devices such as desktop (Fig. 7), TabletPC (Fig. 8), and PocketPC (Fig. 9) by accommodating the different resolutions, with or without plasticity depending on underlying plasticity rules embedded and called. Thanks to the availability of the Mozart software platform for different operating systems (e.g., Microsoft Windows, Apple OS X, and Linux), it is possible to run the same UIs with the same support for Demi-plat properties without changing any line of code. The same UI transparently runs on all these platforms. This facility also allows us to think about migrating a UI across computing platforms running different operating systems since the code of the application can be run indifferently on any of these platforms.

In Fig. 7, 8, the toolbar and the color bar have been detached from the initial windows, thus freeing some real estate and provoking a resizing of the window. The two bars have been merged to be displayed on the monitor of a desktop PC, as pictured in Fig. 9. They could have been maintained separated as well.

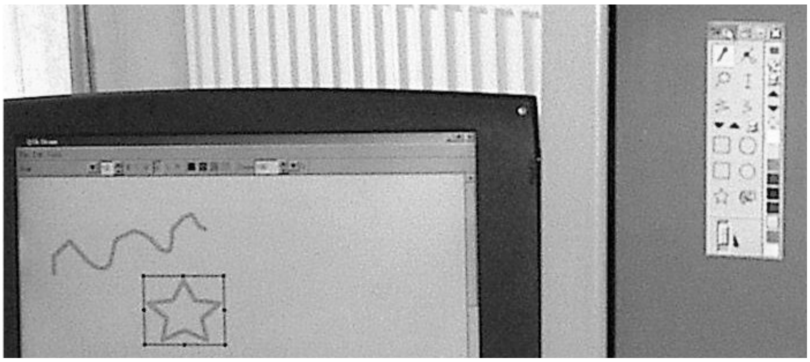


Fig. 9. Detaching the toolbar and the color palette from a TabletPC and attaching them together on a desktop (picture of situation in Fig. 8)

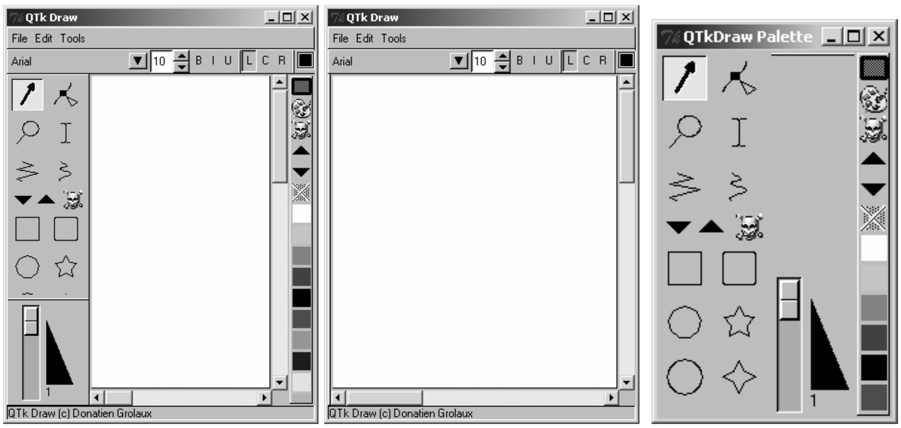


Fig. 10. Detaching the toolbar and the color palette from a PC and attaching them together on another PocketPC

Fig. 10 depicts another configuration in which QTkDraw is executed: first, the complete application is running on a PC, then the toolbar and the color bar are in turn detached from the initial PC and migrated onto a PocketPC. The migration of the second color bar onto the same target PocketPC provokes an attaching of the second bar to the first one, thus leading to repositioning and resizing the bars to fit in a general PlaceHolder. Note that in this case the rule is not detachable, therefore it cannot be migrated onto any other platform. From Fig. 7 to Fig. 10, there is no problem of detaching, attaching the two bars at any time from one platform to another. There is no need of migration server since the application satisfies the 'Demi-Plat' properties itself. The user does not loose the control after detaching and attaching: the UI state is preserved. Actually, there is even no true need to save and restore the UI state since it is simply redirected to another platform wirelessly.

6 Conclusion

This paper presented a development infrastructure supporting detachable UIs. From the application point of view, this is a transparent process: there is no difference between using a stationary UI and a migratable one. A painting application has been changed to behave like the painter's palette (Fig. 2): the tool bar and the color bar can be taken away from the main window, and migrated to any other computer. The difference between the stationary version of the application and the migratable one is around 30 lines of code out of more than 8000. The application that receives the migrated UI is also around 30 lines of code. Note that the core of the application can be extended as if the whole application was purely stationary. As a window can contain an arbitrary amount of migrated UIs at the same time, it is also possible to dynamically compose a UI from different migrated UI components. One could imagine several different applications managing different aspects of a unique problem: their UIs are conveniently migrated to a single place. The system administrator migrates the UIs from all these applications into a single window. This window is migrated between his desktop when he is in front of his desk, and his laptop computer when he is away. Also the development cost of this application is almost the same as the development cost of a stationary version, very little change is required to make the information migrating. This toolkit provides low cost migration mechanism that enables us to have more freedom with multi-platform ubiquitous UIs.

Acknowledgements

We acknowledge the support of the Pirates and Salamandre research projects and the SIMILAR network of excellence, the European research task force creating human-machine interfaces similar to human-human communication (http://www.similar.cc).

References

1. Bandelloni, R., Paternò, F.: Flexible Interface Migration. In: Proceedings of ACM Con. on Intelligent User Interfaces IUI'04 (Funchal). ACM Press, New York (2004) 148–155
2. Bharat, K.A., Cardelli, L.: Migratory Applications Distributed User Interfaces. In: Proc. of ACM Conf. on User Interface Software Technology UIST'95. ACM Press (1995) 133–142

3. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target UI. Interacting with Computers 15,3
4. Coutaz, J., Balme, L., Lachenal, Ch., Barralon, N.: Software Infrastructure for Distributed Migratable User Interfaces. In: Proc. of UbiHCISys Workshop on UbiComp 2003 (2003).
5. Coutaz, J., Lachenal, C., Calvary, G., Thevenin, D.: Software Architecture Adaptivity for Multisurface Interaction and Plasticity. In: Proc. of IFIP Workshop on Software Architecture Requirements for CSCW–CSCW'2000 Workshop.
6. Coutaz, J., Lachenal, Ch., Dupuy-Chessa, S., Ontology for Multi-Surface Interaction. In: Proc. of IFIP Conf. on Human-Computer Interaction Interact'2003. IOS Press (2003).
7. Dewan, P., Choudhary, R.: Coupling the User Interfaces of a Multiuser Program. ACM Transactions on Computer-Human Interaction 2,1 (2000) 1–39
8. Guimbretière, F., Stone, M., Winograd, T.: Fluid Interaction with High-resolution Wall-size Displays. In: Proc. of ACM Conf. on User Interface Software Technology UIST'2001
9. Jacobson, J.: Configuring Multiscreen Displays with Existing Computer Equipment. In: Proc. of Conf. on Human Factors HFES'2002.
10. Milojicic, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process Migration. ACM Computing Surveys 32, 3 (September 2000) 241–299
11. Myers, B.A., Nichols, J., Wobbrock, J.O., Miller, R.C.: Taking Handheld Devices to the Next Level. IEEE Computer 37, 12 (December 2004) 36–43
12. Rekimoto, J.: Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments. In: Proc. of UIST'97. ACM Press, New York (1997) 31–39
13. Rekimoto, J., Masanori, S.: Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments. In: Proc. of CHI'99. ACM Press, NY (1999) 378–385
14. Sousa, J., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: Proc. of IEEE-IFIP Conf. on Software Architecture (2002)
15. Streitz, N., *et al.*: i-LAND: An interactive Landscape for Creativity and Innovation. In: Proc. of ACM Conf. on Human Factors in Computing Systems CHI'99. 120–127
16. Tandler, P., *et al.*: ConnecTables: Dynamic coupling of displays for the flexible creation of shared workspaces. In: Proc. of UIST'01. ACM Press, New York (2001) 11–20