

Transparent Migration and Adaptation in a Graphical User Interface Toolkit

Donatien Grolaux

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

September 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Examination committee:

Peter **Van Roy** (Advisor)	UCL/INGI, Belgium
Jean **Vanderdonckt** (Advisor)	UCL/ISYS, Belgium
Marc **Lobelle**	UCL/INGI, Belgium
Joëlle **Coutaz**	Univ. Joseph Fourier, France
Seif **Haridi**	Royal Institute of Technology, Sweden
Yves **Deville** (Chair)	UCL/INGI, Belgium

Transparent Migration and Adaptation in a Graphical User Interface Toolkit

by Donatien Grolaux

Computing Science and Engineering Department
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

Acknowledgements

I would like to express my thanks to:

- My advisors, Jean Vanderdonckt and Professor Peter Van Roy, for their constant support and enthusiasm regarding my work.
- Professors Joëlle Coutaz, Marc Lobelle, and Seif Haridi for accepting to participate to the jury of this thesis.
- My colleagues from the Information Systems Unit (ISYS) from Louvain School of Management (LSM) and the Dept. of Computing Sciences (INGI) of Université catholique de Louvain (UCL).
- My family and friends.

This thesis was realized thanks to the support of:

- The PIRATES research project (Methods and Tools for Dependable Transparent Distributed Programming), Walloon Region of Belgium, Convention 9713540.
- The SELFMAN research project (Self Management for Large-Scale Distributed Systems based on Structured Overlay Networks and Components), European Sixth Framework Programme, Priority 2, Information Society Technologies.
- The DESTINE research project (Design & Evaluation Studio For Intent-Based Ergonomic Web Sites), «WIST» Wallonie Information Science & Technology research program (Walloon Region), under Convention n°315577.
- The SIMILAR network of excellence, the European research task force creating human-machine interfaces similar to human-human communication, supported by the 6th Framework Program of the European Commission, under contract FP6-IST1-2003-507609 (www.similar.cc).
- The UsiXML Consortium, for User Interface eXtensible Markup Language (www.usixml.org).

Table of Contents

Table of Figures

Chapter 1 Introduction & Contributions

1.1 Motivation: the challenge of developing user interfaces for ubiquitous computing

1.1.1 First dimension: variety of computing platforms

In Marc Weiser's vision for ubiquitous computing [Weis91,93], the paradigms of one computer for many users and of one personal computer per user will give way to the paradigm of one user exposed to multiple computing platforms simultaneously. In this paradigm, the trend is that the number of computing platforms is dramatically increasing (Figure 1-1). So is the variety of these computing platforms that we define as "any combination of hardware and software components on which the user interface will run": watch, mobile phone, smartphone, Personal Digital Assistant (PDA), PocketPC, Blackberry, handbag PC, Internet Screenphone, tablet PC, laptop, desktop, wall screen,...

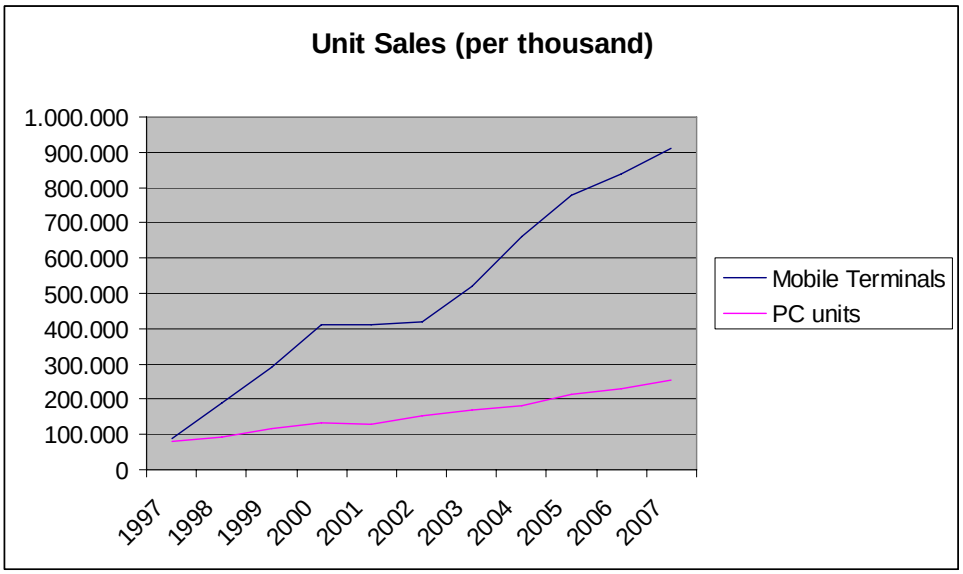


Figure 1-1 Evolution of amount of computers in Ubiquitous Computing (source: Gartner)

These computing platforms may differ in a large amount of factors such as, but not limited to [Chu04]: screen size (probably the most determining factor [Chae04]), screen resolution, color palette, network bandwidth, battery, Central Processing Unit (CPU) power, available memory, interaction devices (e.g., keypad, keyboard, mouse, stylus, trackball, laser pointer), and interaction styles (e.g., form fill-in, direct manipulation [Shn83]). At a

given time, the capabilities of such computing platforms could be determined and from there be taken into account in the User Interface (UI) development life cycle. Since future computing platforms are unknown, it is impossible to predict their capabilities and therefore to take them all into account. Even existing platforms continuously evolve over time with more expanded capabilities: today, highly portable platforms (e.g., mobile phones) suffer from more constrained capabilities as opposed to poorly portable platforms (e.g., stationary desktops) which benefit from high-end capabilities. The current trend [Pier04] reveals that the ratio between portability and capabilities is improving: capabilities of luggable platform will benefit from more powerful capabilities in the near future (Figure 1-2).

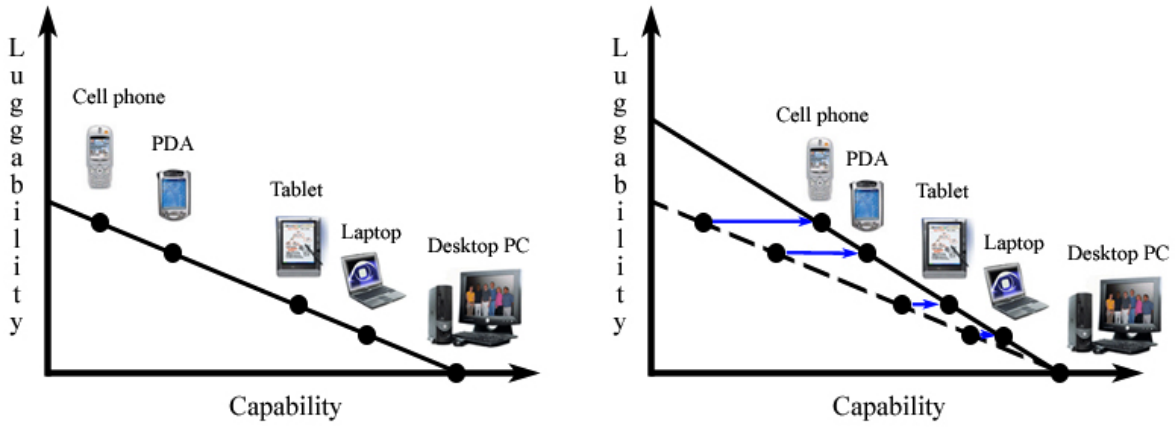


Figure 1-2 Constraints of some current computing platforms (from [Pier04])

1.1.2 Second dimension: variety of locations

As electronic devices with interactive capability are getting pervasive in all aspects of our modern life, we, the users, are getting more mobile. We still expect these devices to help us achieve complex tasks, but the relations between the devices and the tasks they help achieve are blurred. For example, email services used to be reserved to networked computers only, while nowadays one could use his computer at home, his mobile phone on the move, a handheld using a Wifi connection at an airport, or a WebTV at a friend's place. All these devices have different user interface capabilities: size of the display, presence of a mouse, a keyboard, voice synthesizes, voice recognition, and multi-modality in general. So it's quite natural to have different interaction mechanisms between these different devices. However as the task is the same, one would expect some help managing these aspects so that we can use a single application for all devices instead of a different one per device. The developer should focus on the functionality of the application, and the tools deal with the dynamic aspect of taking a user interface away from a device and put it onto another one: we call this the migration of the user interface. Also the tools should deal with changing the user interface to better fit the change of context (characterized by the device running the UI, the environment of the user, the user preferences...) while keeping some level of usability: we call this the adaptation of the user interface. Current technologies like Java and the Web offer very unsatisfactory solutions for this problem. The email example is well known, because it is already part of our life. But one could imagine that any computer application should also be able to follow the user when

he is moving from devices to devices, each time adapting itself to the underlying user interaction capabilities. **Basically we want to shift applications from a device centric view, to a mobile user view**. The application should be able to follow a user through his mobility, offering him an adapted user interface on each device he is accessing. This shift also introduces a new interesting way of collaboration where several users bring together their respective tasks into a single place (the screen of a single device), so as to make global decisions. Also if the application is able to run its user interface on several devices at the same time, which in turn are used by several users, then we also have a multi-user collaboration mechanism.

There exist several technologies for displaying user interfaces to remote locations. [Vnc] allows to remotely display and control the display of a computer. [X11] allows executing an application on a computer and the GUI on another one. These mechanisms are independent of the application which has no knowledge or effect on them. As a result, this shortcoming prevents the application from adapting the user interface to the device it is on. Also the application cannot react by itself to a change of environment, for example to benefit from a device providing a new screen to display more information, or to recover from the failure of the device currently displaying its GUI.

1.2 Thesis

1.2.1 Thesis goal

We argue that developing an application (in the sense of computer software that employs the capabilities of a computer directly to a task that the user wishes to perform) with a consistent, usable and adapted user interfaces for multiple platforms simultaneously is a task that would benefit from:

G1. *Application control.* The control of the distribution of the user interface should be given directly to the application in opposition to no control as is generally the case with an external tool.

G2. *Application feedback.* Along with the control, the application should also be notified of the distribution events, so as to take further action if necessary, like adaptation of the UI. This feedback should be implemented by an event driven interface, like all other UI events.

G3. *Platform independence.* The GUI should be implemented using an abstraction that is constant for all platforms/environments. It is not the role of the developer of the application to manage the specifics of each platforms, it is the role of the tools. For example the tools should be able to render the same GUI differently according to the interaction resources available, while maintaining a constant abstraction at the application level. Working at this level of abstraction is very close to working with stationary non adaptable single user interfaces.

G4. *Transparent migration.* The tools dynamically manage the distributed aspect of running the user interface on several devices. In particular, a runtime migration is

managed by the tools while letting the application run concurrently uninterrupted. We call this property transparent migration. Note that the application is stationary and stays at the place it was initially started; the migration is limited to the UI.

G5. *Transparent adaptation*. The tools allows dynamically switching between alternate renderings and/or interaction mechanisms for (parts of) the GUI. For example the GUI is adapted to use interaction techniques well suited to the available interaction resources of the device it is migrated into. This is also executed while letting the application run concurrently uninterrupted. We call this property transparent adaptation.

G6. *Adaptation repository*. The tools can be further configured to support more platforms/environments/adaptations. The tools act as repositories for new adaptations of already existing UI. This property allows the reusability of the adaptations.

Therefore, we will defend the following thesis:

> The design and development of distributed user interfaces benefit from an approach that abstracts the migration and adaptation aspects away from the development of the UI (1). This approach is supported by a tool that executes the distributed UI orthogonally to the concurrent execution of the application. In particular the UI is migratable and adaptable while the application is executed concurrently uninterrupted (2). We call the conjunction of (1) and (2) *transparent* migration and adaptation. The application pilots the tool as needed for a migration or an adaptation, and receives the feedback in an event driven fashion.

The contributions of this work are:

- Introducing the migration and adaptation at the graphical toolkit level as a binding between a programming language and an existing graphical toolkit.
- Creation of a dedicated construct that reduces the development cost of this binding.
- Using a capability based approach for the migration.
- Allowing the dynamic migration of the UI transparently from the execution of the application.
- Using a configuration approach for the adaptation.
- Allowing the dynamic adaptation of the UI transparently from the execution of the application.
- As a side effect, this work also introduces some multi-user aspects.

Section 2.4 details the published papers related to this work.

1.2.2 The Mozart Programming System

This thesis involves the development of a tool for supporting the distributed aspects of the UIs. We used the Mozart Programming System [Moza] [Vanr04] for several reasons:

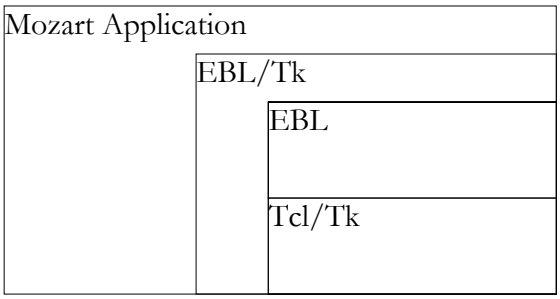
- For distribution, Mozart provides a true network transparent implementation with support for network awareness, openness, and fault tolerance.

- Mozart is based on the Oz language, which supports declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole.

This text is full of examples written for Mozart. Unfamiliar readers are advised to read the Tutorial of Oz available at http://www.mozart-oz.org/home/doc/tutorial/index.html. Note that examples are often directly executable inside the programming interface of Mozart.

1.2.3 EBL

The tool developed to support this thesis is called Enhanced Binding Layer (EBL). EBL is a middleware that must be interfaced to one (or more) actual graphical toolkit(s), and results in a graphical toolkit for Mozart with support for migration and adaptation. In particular a Tcl/Tk [Oust94] version has been created, named EBL/Tk (also known as ETk), and is provided for Mozart [ETk]. This module is available for download at http://gforge.info.ucl.ac.be/frs/?group_id=24.



- EBL/Tk presents itself as a Mozart binding for the Tcl/Tk toolkit. EBL/Tk relies on the concept of widgets (an interface element that a computer user interacts with) and supports those of Tcl/Tk. As Mozart and Tcl/Tk support many different platforms (Windows, Linux, Max OS X, UNIX ...), their combination is also multi-platform. This point implements G3.
- All widgets of EBL/Tk can be dynamically migrated from one place to another, which can be on an entirely different device. Applications use the widgets independently of their migration status, be they local, migrated, or currently migrating. In particular a runtime migration is managed by EBL/Tk independently of the concurrent execution of the application. Also the distributed architecture of the widgets is directly implemented by the site offering the widget and the site displaying it, there is no dependency on an external infrastructure like a server. This point implements G1 and G4.
- All widgets of EBL/Tk support alternate representations (or renderings). The application can switch at any time between these representations. Applications use the widgets independently of their representation state, even during a switch

between states. In particular a runtime adaptation is managed by EBL/Tk independently of the concurrent execution of the application. This point implements G1, G5, and G6.
- EBL/Tk is open to new widget definitions and/or alternate representations. This point implements G6.
- EBL/Tk provides feedback using an event based mechanism. The feedback includes typical graphical toolkit events (mouse clicks, key typing) as well as events related to the migration/adaptation of the UI. This point implements G2.

As a bonus, EBL/Tk also provides further interesting functionality not directly related to the thesis statement above:

- EBL/Tk can be used in a purely imperative way using object-orientation, or by a mixed declarative/imperative approach that simplifies the creation of windows. The declarative approach relies heavily on the record data structure of Oz, which has a tree-like symbolic structure.
- And lastly, EBL/Tk allows the migration of a component into more than one device simultaneously, providing what we call a *simple multi-user* functionality.

The support for migration, adaptation, extensibility of widget definitions and/or representations, mixed declarative/imperative approach, and multi-user functionality is provided by the EBL middleware and not by the external toolkit. EBL could be used with a different toolkit than Tcl/Tk, and the resulting toolkit would also support this functionality:

- EBL is toolkit agnostic, there is no reference to an external toolkit like Tcl/Tk.
- EBL assumes very basic functionality of the low level toolkit it will be bound to: creation and removal of widgets.
- Further support is provided by EBL if the low level toolkit also supports a grid geometry manager.

Although we have not a proper implementation to support this claim, an EBL/Ajax implementation has been envisioned by mixing the techniques El-Ansary and I developed for QHTML [Elan04] along with EBL.

1.2.4 EBL/Tk example

To further fix the ideas, this section presents a simple example of a Mozart application that uses EBL/Tk to create an application that is migratable and adaptable. **This example is a first illustration of what is possible thanks to the work developed in this thesis**.

Figure 1-3 Example of an adaptable application

The UI is composed of a widget that allows selecting between a radiobutton representation, a listbox representation or a dropdown menu representation. Each time the user selects one of these items, the widget is adapted correspondingly.

```
UI={Build
    window(name:window
           selector(name:selector
                    items:["Radiobuttons" "Listbox" "Menu"]
                    curselection:1
                    text:"Selector"
                    action:proc{$}
                              R={UI.selector get(curselection:$)}
                           in
                              {UI.selector
setContext((default#listbox#menu).R)}
                           end))}
{UI.window show}
```

The code of this example is trivial: an Oz record defines the UI, and is given to the build function of ETk. This function creates the whole UI, and returns an object for further controlling its components. In this example, the Oz record is composed of a window in which there is a single selector widget that allows selecting between three items. The features of the window and the selector define different aspects of these widgets:

- Their name for future referencing as features of the object returned by the `Build` function.
- The initial state of their parameters. For the selector widget, the `items` feature defines the list of items to choose from, the `curselection` feature defines the item currently selected, and the `text` feature defines the text displayed on top of the widget.
- The code to execute in reaction to a user interaction. The `action` feature of the selector widget is executed each time the user selects an item. This code gets the index of the item currently selected, and adapts the widget correspondingly. This is achieved by calling the `setContext` method with a value that is accepted by the widget. In the case of a selector widget, the acceptable values are `default` for the default radiobutton representation, `listbox` for the listbox representation, and `menu` for the menu representation.

Finally windows are created hidden by default; a call to the `show` method of the window is required to make it visible.

Independently of this adaptation functionality, any widget of the UI can be migrated to another process possibly on a remote device.

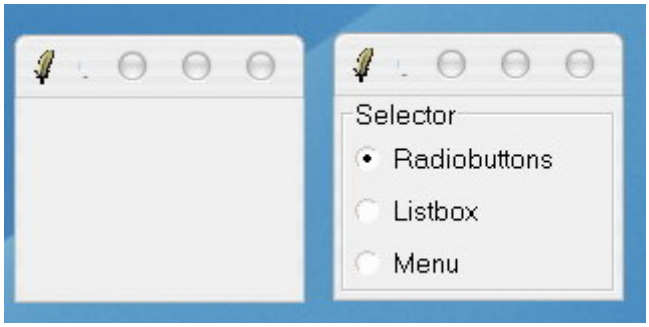


Figure 1-4 Example of a migrated UI

In this example, the UI is migrated into another process on the same computer. To migrate from the process on the left to the process on the right, they need to exchange a *migration capability*. A migration capability is a value that grants the right to pull the widget into display, hence triggering its migration. Basically, the migration capability is a universal reference to the widget that allows contacting it over the Internet in order to trigger its migration. In this example both sites run on the same computer so they can use the local file system to share the capability by using a shared file (in general the capabilities are exchanged through distributed discovery services). The left site saves the capability of the selector to a file by using the `Pickle.save` procedure of Mozart:

```
{Pickle.save {UI.selector getRef($)} "selector.cap"}
```

The right site that receives the widget must 1) create a window in which to display the widget, and 2) get the capability by using `Pickle.load`, and 3) pass it to the window:

```
UI2={Build window(name:window)}
{UI2.window show}
{UI2.window display({Pickle.load "selector.cap"})}
```

Figure 1-4 is the result of the execution of this code. The window on the left side is now empty as its content has migrated away.

As the left side still has an access to the migration capability of the selector it can use it to bring the widget back into place:

```
{UI.window display({UI.selector getRef($)})}
```

Figure 1-5 is the result of the execution of this code.

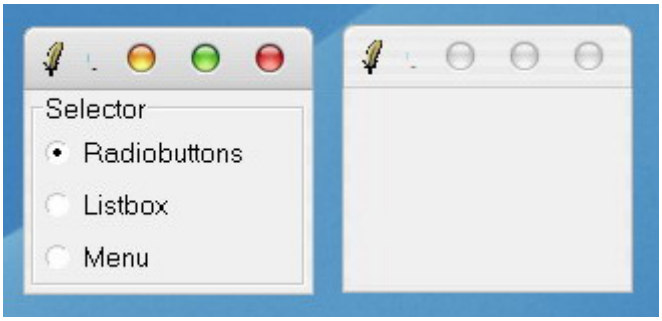


Figure 1-5 Example of a migrated UI

Note that the window on the right still exists: its content migrated away but the window itself did not budge. To make this window disappear automatically in such situation, we have to program it in an event driven fashion:

```
{UI2.window bind(event:lostWidget
                 action:proc{$} {UI2.window destroy} end)}
```

The `lostWidget` event is triggered each time a container widget loses one of its displayed widget. Here the configured action destroys the whole window in reaction to this event.

1.2.5 Implementation of the thesis statement

The example from 1.2.4 gives insight on how we implement a tool that covers the thesis statement:

G1. *Application control.* The migration is based on a capability system. It is the responsibility of the applications to exchange the capabilities for migration. The receiving end of a migration requires only the capability to trigger a migration. It does not require sharing further knowledge with the host application. The adaptation is based on a special configuration mechanism that changes the complete representation of the widget. This mechanism is triggered by the application itself.

G2. *Application feedback.* Applications can be made aware of the migrations of the widgets, using an event driven approach, like the `lostWidget` event of the example. Also typical user events (mouse click, key down...) are relayed transparently to the application. In particular the sites displaying remote widgets will not be made aware of their user events.

G3. *Platform independence.* EBL/Tk is compatible with all platforms supported by Mozart. It presents itself as a platform agnostic toolkit.

G4. *Transparent migration.* The migration process is orthogonal to the concurrent execution of the functional core of the application. In the example, there is no special precaution taken by the application regarding the fact that the widget may be local, migrated or in the process of migrating between two sites.

G5. *Transparent adaptation.* The adaptation process is orthogonal to the concurrent execution of the functional core of the application. In the example, there is no special precaution related to the current representation of the widget. Alternate

representations for a widget are independent of its application programming interface; a single interface controls them all.

G6. *Adaptation repository*. The selector widget supports 3 representations for now; if new ones become available it is trivial to extend the code of the example to support them. EBL/Tk act as a repository for these alternate representations. This repository aspect is also found with the Comets [Calv05] however EBL does not have all the introspection capability the Comets have.

1.2.5.a Benefits of this approach

This approach has several advantages:

(1) Advantages in terms of *methodology*: the approach is close enough to the classical stationary/not adaptable one that the same methodology can often apply. In particular this approach supports an incremental methodology where the UI of an application is first created as a classical single-user, single-device, non adaptative one, and then migration and adaptation are introduced where the needs appear.

(2) Advantages in terms of *reusability*: once a new component or a new renderer has been created, it can be reused.

(3) Advantages in terms of *consistency*: in a multiplatform context, it also guarantees a minimal consistency between the UI generated for different target platforms. This is not always possible when using traditional techniques where the development of each version of the UI is likely to be performed separately.

(4) Advantages in terms of *extensions*: components are defined independently of the applications, and in particular more renderers for the same component can be added later. It is possible to program the applications so that they can use any available renderer for its components: in that situation the application are able to use the new renderers automatically.

(5) Advantages in terms of *control*: the application has the complete control over the migration of its components, it can allow/disallow it as it sees fit. Also, the actual decision of what renderer should be used at a specific time is let to the application. The tool supports all the inner mechanism of adapting the component, but it is the application that takes the decision.

1.2.6 Scope and limitations

- The scope of this work is limited to Graphical User Interfaces (hereafter GUI) of the WIMP type (Window, Icon, Menu, Pointing device). These are the standard interface to most information systems, are familiar to the majority of users, and are available on almost every platform. Hence, we do not consider non visual, multimodal or 3D user interfaces. Note that the approach is probably reasonably adaptable to multimodality, but we did no research to confirm or infirm this assertion.

- The target audience of this thesis is, on the one hand, the HCI research community and, on the other hand, the distributed computing community.
- The question of how to produce usable distributed interfaces represents an important research question which is beyond the scope of this thesis.
- There exist different levels of abstraction for the programming of a GUI;
    - Low level direct access to the hardware (keyboard, mouse, GPU...). This level is strongly tied to a particular piece of hardware, and not portable to another one. Also there is no notion of WIMP interfaces.
    - Driver level: at this level the low level communication with the hardware is abstracted in a driver with a standardized application programming interface (API). This level is strongly tied to a particular API typically defined by the underlying operating system (OS)/window manager pair, and typically not portable to another OS. Also there is still no notion of WIMP interfaces.
    - Imperative toolkit level: this level introduces the notion of WIMP interfaces as imperative commands. The toolkit introduces its own API which is not directly tied to an OS/window manager pair; some toolkits run on multiple OS.
    - Model based approach (MBA): this level introduces the notion of models that grasp separate aspects of the user interaction: the dialog model (how users can interact with the objects presentation (as push buttons, commands, etc), with interaction media (as voice input, touch screen, etc) and the reactions that the user interface communicates via these objects), the task model (describes the tasks the end user performs), the navigation model (defines how the objects that a user view could be navigated through the UI), the user model (represents the different characteristics of the end user), the platform model (describes the physical devices that host the application) and so on. These models are typically expressed using XML. Most of the time, models are purely declarative data structures, however they can also be Turing complete. Runnable MBA UIs require a runtime tool that dynamically interprets the different models into a physical UI, by using an imperative toolkit.

  Several other works in this field of research (Comets [Dâas03, Calv04, Calv05, Deme06a, Deme06b], UsiXML [Limb04a, Moli05]) implements adaptation by using a model based approach to have a high-level specification of the UI at runtime whose interpretation vary according to the platform it is running on. However in this work we decided to introduce the migration and adaptation at the toolkit level instead. We argue that these approaches are complementary. Any tool providing migration and/or adaptation will need some code running at the display platform to render the actual components of the user interface. When a model based approach provides an executable runtime environment, these lines of code still have to exist. This thesis provides a repository specially designed for conveniently storing them. A model based approach can greatly benefit from this work

by having all the complex and technical aspects of actually migrating/adapting components already solved, and only focus on the general problem of interpreting the models.

- In our approach, the control of the adaptation is let to the application, there is no external meta-UI provided for this task. The idea is that the application should implement a small oracle that decides what renderer to use according to constraints relevant to the application. Note that it is also possible to create a component that embeds its own oracle so that it adapts itself automatically without further instruction from the application.

  The oracle can take several parameters into consideration:
  - *Decrease of the screen resolution and size*: this parameter has a strong influence on the structure and presentation of the user interface. Sometimes, even with a similar screen size, the available screen area may be more constrained when a part of the display is used for other purposes (e.g. virtual keyboard).
  - *Increase of the minimal size of graphical objects and the minimal spacing between them*: on some platforms, the objects included in the interface are to be larger or more distant (e.g; touch screen interfaces).
  - *Decrease of the number of available widgets*: not all toolkits are available on every platform. Furthermore, some platforms have reduced versions of the toolkit or simplified versions of the mark-up language.
  - *Decrease of the usability of available widgets*: the usability of a given widget may vary from one platform to another, for example because of the absence of a keyboard on some platforms.

  Other parameters such as the network capacity, the support of frames, images or colors, the presence and the type of pointing devices and keyboard, the storage capacity or the CPU speed can also be taken into account.

1.3 Road map

Following this introductory chapter, chapter two provides the state-of-the-art pertinent to this thesis.

Chapter 3 addresses the requirements of the problem from a graphical user interface toolkit perspective. EBL advocates the approach formerly introduced by QTk [Grol01a] to use a hybrid declarative/imperative approach to GUI programming. First we introduce this approach and the Oz data structure we rely on (3.1). Next is a justification of the geometry management in the context of migratable UI (3.2). Lastly, we discuss how we can mix object-orientation and model-based programming together thanks to EBL (3.3).

Chapter 4 addresses the requirements of the problem from a distributed graphical user interface toolkit perspective. First we introduce migration and adaptation as useful and

conservative extensions of the classical imperative toolkit paradigm (4.1). Then we present the general distributed structure of a widget (4.2), we introduce the EBL Store construct which implements the whole migration, adaptation and limited multi-user functionality (4.3), we address the receiving end of a migration (4.4), we focus on the adaptation mechanism (4.5), and finally we address distributed aspects also related to this thesis: the low level network implementation (4.6) and the security issues (4.7).

Chapter 5 details the architecture of the implemented solution so as to give the reader a good understanding on the inner working of EBL: a distribution overview (5.1), the runtime architecture (5.2), the migration capabilities (5.3), the distributed widget architecture (5.4), the display site architecture (5.5), the low level network component (5.6), the most important protocols (5.7), and finally the way EBL is interfaced to an actual toolkit is presented as a set of recipes (5.8). This chapter is focused on the "how" of the problem.

Chapter 6 presents different case studies exemplifying important aspects of EBL: the implementation of a migratable widget, and the usage of this functionality (6.1), the implementation of an adaptable widget, and the usage of this functionality (6.2), the implementation of an application that is adaptable for a PC and for a PDA (6.3), and the analysis of a multi-user application (6.4). Also this chapter further showcases the flexibility of the transparent migration (6.5), details security considerations relevant to using a distributed toolkit (6.6), analyses the performance of the tool (6.7), and finally compare this tool with other existing tools (6.8)

Finally Chapter 7 concludes this thesis, with a summary of results and contributions (7.1), future work in prospect (7.2), and final concluding remarks (7.3).

Chapter 2 State of the Art

This chapter places this thesis in context of the state of the Art. Also, this work is placed in the context of the evolution of the graphical toolkit bindings for Mozart.

2.1 Dynamic migration of running user interfaces

Distributed User Interfaces (DUIs) [Berti05, Deme05, Luyt05] apply the notion of distributing parts or whole of a user interface (UI) across several places or locations like Distributed Systems [Li03] do for general software.

2.1.1 History of the migration of running user interfaces

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 [X11], Virtual Network Computing [Vnc], and Windows Terminal Server [Wts]. 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, there is no adaptation to the target platform, the UI cannot be dissociated, and some of them rely on 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 [Bahr95]: their migratory applications are able to move from one platform to another one at run-time, provided that the operating system remains the same. This is probably the first truly migrating application; however the migration granularity is restrictive as it is not possible to benefit from devices closely situated by splitting the UI between them. The situation is similar for multi-user applications when an application should be transferred to another user as in [Dewa00]. In *The Migration Project* [Band03], 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 [Myer04] 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 [Reki97] supports migration of information between platforms, like other interaction techniques and migration environments such as *i-*

Land [Stre99], *Stanford Interactive Mural* [Guim01], *Aura* [Sous02], *ConnecTables* [Tand01]. But these solutions do not support the properties of migrating only part of the user interface, the flexibility of attaching together any user interface, and adapting it to the target platform. In addition, all the platforms should belong to the same family of devices, operating systems and software, 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.
I-AM [Cout00,Cout03b] exhibits the capabilities of platform discovery and resources sharing between compatible platforms. A meta-UI [Cout03b] is defined to control the migration process [Cout03a] across various platforms and in varying circumstances, thus releasing the user from having a predefined configuration. I-AM is mainly resource centric: the focus is on merging physical resources together, creating a bigger virtual resource for the application to run on. For example with I-AM, an application can spawn on several devices with a single mouse cursor for all the devices. We use a different approach that is application centric: the EBL applications migrate their UI into other EBL applications, possibly adapting them on the fly. The physical resources that run these applications stay separate contrarily to I-AM; for example each one of them have their own independent mouse cursor. Also there is no meta-UI for controlling the adaptation and the migration; the EBL application has the complete control over these aspects and should provide a UI suited for the task. Finally the EBL enabled applications contain the entire infrastructure required for migrating UIs, there is no dependency over an external component like a dedicated server. All devices that run EBL applications can be used for UI migration. EBL applications are written for the Mozart Programming System which is available for Windows, Mac OS/X and many flavors of Unix/Linux.
The INDIGO project [Blanch05a, Blanch05b] also proposes an architecture for running distributed UIs. Interestingly, their approach share a basic similarity with EBL: each UI component is split into a logical part for the application and a physical part for the end-user. Beyond this similarity, the techniques used are very different: the INDIGO project uses model-based techniques where both parts share a common model pertinent to the UI component, and the parts interact by making the model evolve over time. The physical part is obtained by transformation of this model. EBL does not use a model based approach, instead it provides a dedicated data structure along with specialized functionality that facilitates the extension of any existing graphical toolkit to support adaptation and migration (limited to the Mozart Programming System). Also EBL provides a complete control over migration and adaptation to the application itself, while INDIGO is limited to running a UI remotely: once the UI is created, it stays were it is; there is no dynamic migration or adaptation.

2.1.2 Current issues for DUIs

In studying DUIs, [Moli06b] has identified two main classes of problems: an ontological confusion about the various concepts and associated definitions expressing how and according to what to distribute a given UI; a practical problem of experimenting a DUI at early design time before developing it completely. The first class of problems is motivated by observing that the several recent advances in DUIs do not necessarily rely on the same concepts of distribution and, when it is the case, the definition and/or the axes

according to which the distribution may take place largely vary from one research to another. Although significant efforts exist to shed some light in this area and to structure DUI design issues, mainly in [Berti05,Deme05], the relationship between these design issues and their corresponding physical configurations are not always straightforward to establish. The second class of problems poses even more challenges because developing DUIs require a sophisticated architecture, and due to that level of sophistication it not surprising that DUIs are slow to obtain, expensive to produce, and probably equally complex to use. The aforementioned observations show that designing a DUI remains a complex problem which may prevent designers from exploring design issues because of their associated cost. If the development cost of several DUIs is too high with respect to the benefit of exploring different design issues and physical configurations for distribution, it is likely that this exploration will be abandoned soon due to lack of flexibility. In addition, the usability issues raised by distributing a UI across one or several dimensions [Grud01,Tan03a] are serious and could be hard to uncover before a really usable solution is found.

To overcome these difficulties, this thesis extends a graphical toolkit so that all its components automatically support dynamic migration:

- All components of the user interface receive the migration ability.
- This ability is expressed as a *capability*, which is a value.
- This value can be obtained by the creator of the component, which can pass it along to another device/process/component, using any means (web server, SOAP, FTP, email ...). As capabilities are just values (hence are stateless), they can be passed along by anyone that knows it.
- Some components have the capability to migrate others into themselves. Typically the table component can migrate any other widget in the different cells it has. The placement of the inner components (hence their migration into this one), is triggered by giving their migration capabilities.
- Note that container components are themselves components; as such they also have the migration capabilities. A site that uses such capability to migrate the container will also migrate the content of the container along.

Consequently, it is up to the application to define the discovery and negotiation aspects of the capabilities. This approach allows the rapid development of prototype applications, as advocated by [Moli06b]. In particular, in our approach the runtime architecture required to run DUIs is embedded directly in the tool, and does not require an extra external architecture. **The application that has created a migrated UI acts as the server for this UI, while the application currently displaying this UI acts as a client.**

2.2 Dynamic adaptation of running user interfaces

Nowadays application users are often 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, ranging from a laptop or pocket computer to a Personal Digital Assistant (PDA); or they move from one computing platform to another, 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. We characterize such *Context-Sensitive User Interfaces* (CSUIs) by first reporting on some challenges posed by this new type of user interface. 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.

2.2.1 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. Although that problem has been addressed [Thev99, Cal00, Crea01, Dey03, Eise03, Puer99, Szek95], it seems that there is neither common representation nor widely accepted technique on how to capture knowledge of these variations so as to exploit them easily and consistently at runtime. Sometimes, not all possible 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 [Eise00]. These inflexible UIs tend to rapidly become inappropriate or unusable when the context of use changes, thus leading to a questionable usability. It is therefore 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 [Szek96]. 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 user interface code portion, but

also require a dedicated software architecture receiving contextual information thanks to context-aware widgets [Crea01].

- *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 set of configurations may undesirably affect several portions of code, thus lengthening the maintenance period. The development and maintenance efforts are easily duplicated for cross-platform user interfaces, while potentially reducing consistency [Vand05a].

2.2.2 State of the art

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 [Hart93,Puer99,Szek96]:

- *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 2-1.

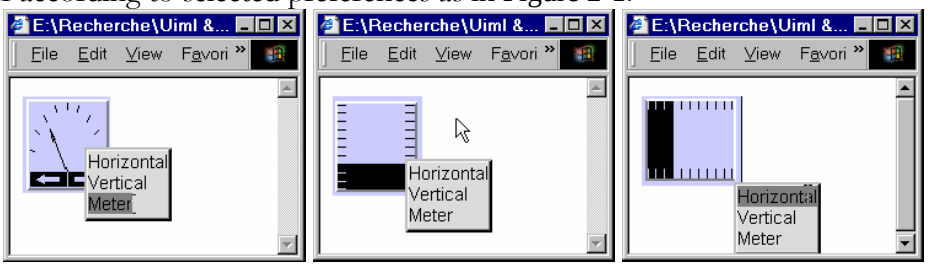


Figure 2-1. Adaptability of a widget presenting the user with a bounded value.

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 [Dey03, Thev99]. 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 [Crea00]. 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 [Calv00, Calv01, Calv02, Thev99]: a plastic UI is a particular UI type sensitive to any variation of the computing platform and/or the environment that preserves some level of usability. 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., organi-

zation structure, task allocation and role definition). A need clearly 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.

2.2.3 A general design space

To represent the types of variation that can be theoretically considered in context-sensitivity, Figure 2-2 depicts a design space expanding a design space for adaptation [Hart93] and another for plasticity [Thev99]. 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 as a 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 [Dey03]. Classically, model-based approaches are used to model these types of variations. A UI model is a declarative, editable, and analyzable representation of some predefined aspects of a UI, according to relevant abstractions [Puer97].
- Along the "What?" axis, context-sensitivity is concerned by the locus of reconfiguration: any parameter 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 defined by skills, experience, and cognitive ability) should reconfigure the tutorial (e.g., keeping advanced topics in a tutorial model).
- Along with the "For what?" axis, context-sensitivity is concerned by the four steps considered in adaptation [Hart93]. The initiative specifies the entity which initiates the need of reconfiguration. The proposal describes possible reconfigurations to be performed on the UI. The decision states the entity that 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 [Hart93]. For instance, a user may select one possible presentation style among a set of predefined ones. In Figure 2-3, the user selects and changes the presentation of a calculator, 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. 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 has 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 *mixed model* can hold both kinds of properties. For example, to accommodate multiple screen resolutions of a same computing platform, a UI may need to embark an active model to apply an appropriate reconfiguration. When models are considered only at design time, they often remain passive. "When?" and "With what?" axes are highly correlated.

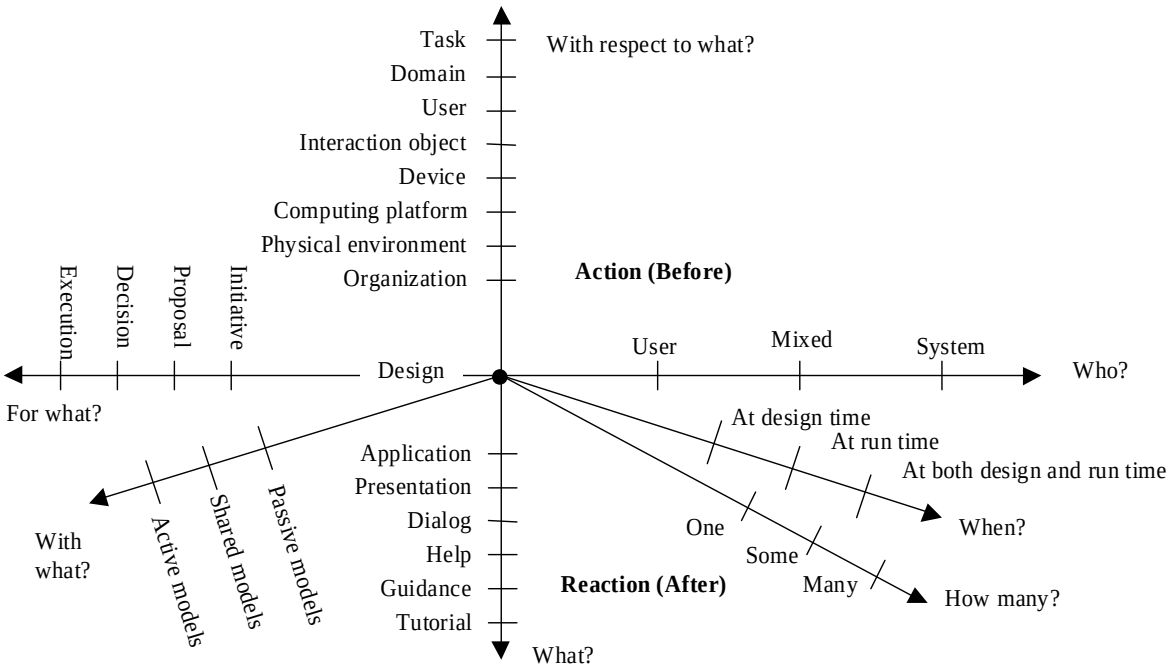


Figure 2-2. A design space for context awareness.

Figure 2-3. Multiple presentation styles for a single user interface.

2.2.4 Representative examples of context - sensitivity

The design space for context-sensitivity is able to express several important UI categories. This thesis focuses on these particular categories:

1. *Plastic UIs*: the UI is able redistribute on different devices, and remodel itself to support variations in 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 some level of usability. The usability is a set of properties specified during the requirements phase. These properties are defined in [Cal00,Thev99]. For example, the same UI can display a network load in multiple forms according to varying screen constraints [Eise00]. Figure 2-4 represents a plastic UI that accommodates different presentations of a network load while the window is being resized.

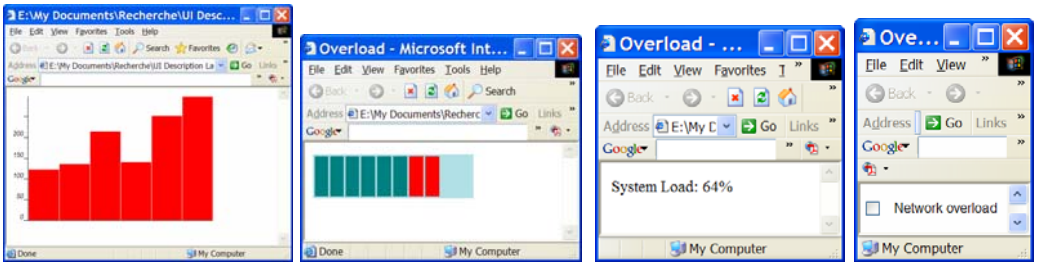


Figure 2-4. A plastic user interface for network load (inspired from [Eise01]).

2. *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 computing platforms, while preserving consistency [Paus92].
3. *Migratory UIs*: the UI is able to accommodate relocation of the user terminal [Bahr95], while maintaining the same context for the application. A UI can be migrated internally inside the application process, for example changing the position of a toolbar from the top of the window to the bottom of the window. The UI can be migrated into a remote process on the same device, for example detaching the toolbar from the main window into a window run in a separate process. In general, the UI is migrated into a remote process on a separate device, for example placing the toolbar on a PDA so as to free room estate for the work area.
4. *Mobile or nomadic UIs*: the UI is able to accommodate variations of the context of use and the change of user location.

2.3 Definition of the design space

The retained design space, after considering the thesis hypotheses is decomposed into two design axes or dimensions.

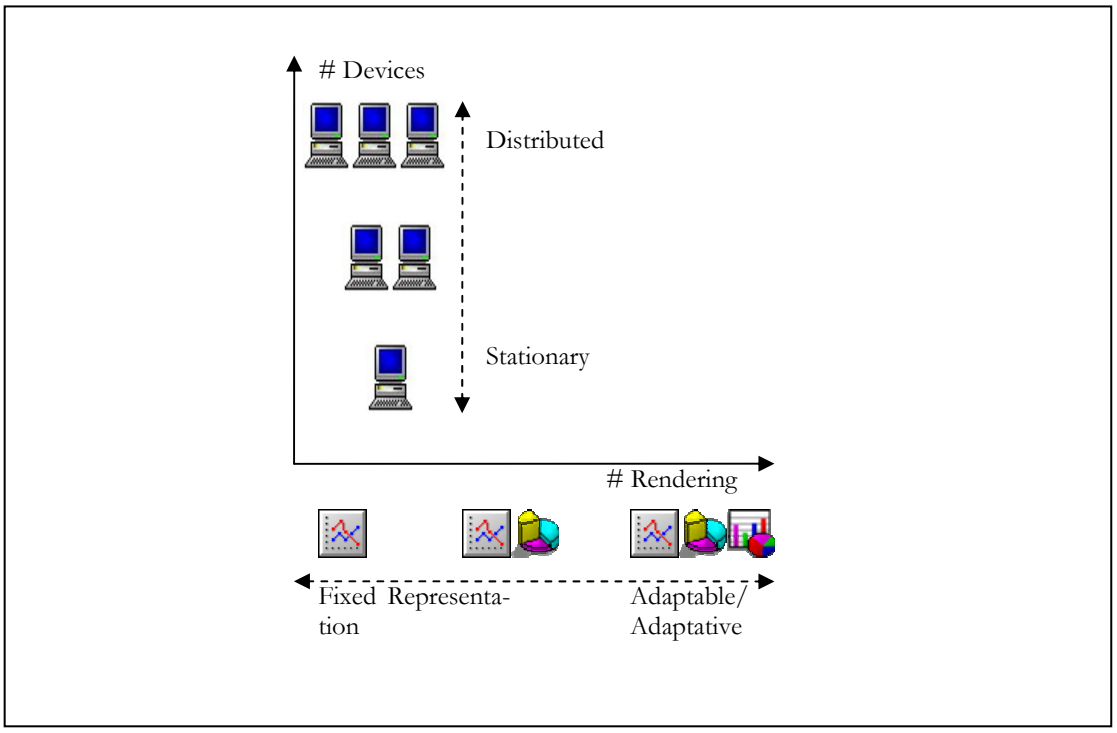


Figure 2-5. Design space.

2.3.1 Number of devices in use

This axis corresponds to the number of devices concurrently in use by a single application to display its UI. At one end of the spectrum, the application uses a single device to run its functional core along with its user interface: this corresponds to the classical stationary applications. At the other end of the spectrum, the application uses several devices to run its user interface, and we are in a distributed environment. This aspect corresponds to the migration capability.

2.3.2 Number of renderings per component

This axis corresponds to the capacity of the components to display themselves differently. At one end of the spectrum, the components have a single way of rendering themselves: this corresponds to the classical non adaptable/adaptative situation. At the other end of the spectrum, each component has several ways of displaying itself. This aspect corresponds to the adaptation capability.

2.4 Previous personal work related to this thesis

This research is the result of several years of experimentation with distributed systems, user interfaces development and design in general. It started out of my own frustration of

using a purely imperative graphical toolkit in the context of a multi-paradigm programming language [Moza]. Surely we could do better in term of expressivity, compactness and general easy of use by tapping into the other aspects of the language. At first QTk was developed to use a mixed declarative-imperative approach. The philosophy was to use the declarative paradigm for those aspects that expresses well declaratively, while keeping the imperative paradigm for all remaining aspects [Grol01a]. QTk was well accepted by the Mozart community and is now part of the standard distribution of the system [QTk]. QTk allowed us to explore different ways of programming user interfaces, and we discovered it was well adapted for the creation of adaptative user interfaces [Grol02]. At the same period of time we were also doing active research in the field of distribution algorithms, working on creating general high level abstractions for large scale distributed systems. The mixing of these two researches came naturally into two results:

- QHTML [Elan04] which was basically a graphical toolkit similar to QTk but relies on the Dynamic HTML technology to use Internet Explorer as the effective toolkit instead of Tcl/Tk.
- A migratable version of QTk [Grol04] that extends its basic functionality to support transparent migrations of widgets. This work is very experimental as it extends a toolkit in a way that it was not designed to in the first place. The result allowed us to experiment with migration; however the implementation is fragile with no way of easily making it robust in general for fault tolerance.

This thesis integrates all these results together. From the QTk and QHTML experience, we learned how to split a toolkit binding into a non-toolkit-dependent-general-purpose part and a toolkit-dependent part. From the migratable and adaptation experiences, we gained insight for designing this sort of distributed system and a way to address the adaptation problem as a migration one, effectively killing two birds with one stone.

2.4.1 The Mozart Programming System

This research relies heavily on the Mozart Programming System. Here is an excerpt from the Mozart web site that describes it nicely [Moza]:

"The Mozart Programming System is an advanced development platform for intelligent, distributed applications. The system is the result of a decade of research in programming language design and implementation, constraint-based inference, distributed computing, and human-computer interfaces. As a result, Mozart is unequaled in expressive power and functionality. Mozart has an interactive incremental development environment and a production-quality implementation for UNIX and Windows platforms. Mozart is the fruit of an ongoing research collaboration by the Mozart Consortium.

Mozart is based on the Oz language, which supports declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole. For distribution, Mozart provides a true network transparent implementation with support for network awareness, openness, and fault tolerance. Security is upcoming. Mozart is an ideal platform for both general-purpose distributed applications as well as for hard problems requiring sophisticated optimization and inferencing abilities. We have developed many applications including sophisticated collaborative tools, multi-agent systems, and

digital assistants, as well as applications in natural language understanding and knowledge representation, in scheduling and time-tabling, and in placement and configuration."

The Mozart Programming System is a great facilitator for this thesis for two main reasons:

- Mozart provides a state of the art transparent distribution mechanism. Distribution algorithms are automatically attached to entities depending on their types. Also Mozart provides advanced concurrency management like lightweight threads and dataflow synchronization. Those are great facilitators for efficiently developing complex distributed systems.
- Mozart supports imperative (procedural and object-oriented) programming and functional programming with high level data structures. One of these high level data structures is the record which is expressively equivalent to an XML tree. Obviously, being able to use this expressivity level directly inside an imperative application has a lot of advantages. This thesis takes direct benefits from the tight integration of the imperative and declarative paradigms.

To reproduce this work in a more mainstream programming language like Java, one would have to:

- Redevelop the distributed algorithms. Mainstream programming languages provide their own mechanisms for distribution very different from Mozart, e.g. RPC/RMI mechanisms versus transparent distribution.
- Redevelop the concurrency algorithms. Mozart provides lightweight threads with dataflow synchronization, mainstream programming languages use much more hard to use concepts like OO threads and semaphores.
- In the absence of a symbolic tree structure and support for functional programming, one should use an external declarative language like XML and interface it to the toolkit. Dynamically computing a record in Oz is a direct straightforward operation, but this would now become an awkward manipulation of an external language.

For efficiency reasons and the elegance of the end result, Mozart is a better platform for the problem of this thesis.

Chapter 3 EBL General Toolkit Design

The previous chapters introduce the context, state of the art and goals of the tool we are building. This chapter presents the graphical user interface toolkit aspects of EBL, independently of the migration, adaptation and in general distribution aspects that are covered in the next chapter. This is required for understanding the examples properly on one hand, but also because this still has an impact on the migration and adaptation support for applications. This aspect of EBL if largely inspired by ideas developed during my work on QTk [Grol01a].

The chapter first introduces the general approach, and the Oz data structure we rely on (3.1). Next is a discussion on the geometry management, in the context of this approach but also in the context of migration (3.2). Lastly, we discuss how we can mix object-orientation and model-based programming together (3.3).

3.1 Hybrid declarative and imperative approach

Before starting on the migration and adaptation design issues, we address a more general design issue of EBL which is required to understand the examples later on. The result of interfacing EBL with an effective graphical toolkit is a binding to this toolkit for the Mozart Programming System. This binding can be used by two different approaches:

1. **A purely imperative toolkit.** Widgets are instances of classes (in the object-orientation sense). They are created by instantiation, and then methods are used for configuration and interaction:

```
MyLabel={New ETkLabel init}
{MyLabel set(text:"Hello World")}
MyWindow={New ETkWindow init}
{MyWindow show}
{MyWindow display(MyLabel)}
```

   This example creates a label widget, and set its content to Hello World. Then a window is created, made visible (by default windows are hidden), and finally the label is placed inside the window, as shown in Figure 3-1.

Figure 3-1. Hello World example.

2. **A partly declarative, partly imperative toolkit.** EBL provides a support for creating windows using a tree data structure mixing symbolic information and references to live entities. This tree specifies a complete user interface in its initial

state, and also its behavior upon window resize. This roughly corresponds to the presentation model (the application appearance) and part of the dialog model (the behavior upon resize of the widget, and for some widgets their reaction upon activation by the user):

```
UI={Build window(name:window
                 label(text:"Hello World"
                       name:label))}
{UI.window show}
```

This example also creates a window with a `Hello World label` inside, this time using the declarative approach. The symbolic tree data structure describes the UI, and is given to the `Build` function provided by EBL. This function creates the UI, and returns an object that allows referencing the widgets of the UI (by relying on the arbitrary atom value set by the `name` feature in the description tree). Once the UI is created, we fall back to the imperative approach above, and rely on the objects for configuration and interaction.

3.1.1 Oz data structures

To fully grasp the power of the declarative approach, we describe the data structures of Oz [Moza] that we use to support this functionality.

3.1.1.a Atom

An atom is a symbolic constant that has a printable representation made up of a sequence of alphanumeric characters starting with a lower case letter, or arbitrary printable characters enclosed in quotes. Atoms are scalar values of the language that have no internal structure. For example: `a foo '=' ':=' 'OZ 3.0' 'Hello World'`. Atoms have an ordering based on lexicographic ordering.

3.1.1.b List

A list is either the atom nil representing the empty list, or is a tuple using the infix operator `|` and two arguments which are respectively the head and the tail of the list. Thus, a list of the first three positive integers is represented as: `1|2|3|nil`. A list ending by nil can also be represented by all elements between `[` and `]`, separated by a space and without the ending nil : `[1 2 3]` which is the same as `1|2|3|nil`.

3.1.1.c Strings

Another notation for a list is a sequence of characters surrounded by `"` (double quotes), for example `"Hello World"`. This is equivalent to a list of integers where each integer is the ASCII value of the corresponding character in the string. This implies that all list operations are available for calculating with strings.

3.1.1.d Records

Records are structured compound entities. A record has a label and a fixed number of components or arguments of the form `label(feat1:val1 ... featN:valN)` where label is an atom, the `featX` are atoms or numbers, and `valX` can be any valid data structure. Note that `featX` are optional. If not specified, they are implicitly numbered as: `label`

`(val1 ... valN) == label(1:val1 ... N:valN)`. Many operations can be performed on Oz records (Table 1): let `R=toto(foo:10 bar:20)`.

Operation	Example
Selection	`R.foo == 10`
Get arity	`{Arity R} == [bar foo]`
Add feature	`{Record.adjoinAt R nuk 30}` `== toto(foo:10 bar:20 nuk:30)`
Subtract feature	`{Record.subtract R bar} == toto(foo:10)`
Extract label	`{Label R} == toto`
Rename label	`{Record.adjoin R lala} == lala(foo:10 bar:20)`

Iterations on record	`Record.forAll, Record.map, Record.while, ...`
	
	`{Record.map R fun{$ V} V div 10 end}`
	`== toto(foo:1 bar:2)`

Table 1. Some Oz operations.

Many other operations are available, and if they still do not cover the needs, the functional paradigm of Oz can be used to write new ones compactly and efficiently. Because of dynamic typing, it is easy to create new record types at runtime.

3.1.2 Declarative semantics

The semantics of the declarative specification is the following:

- A widget is defined by a properly formatted record.
- The label of the record defines the type of the widget.
- By convention the widgets placed inside container widgets are defined by using numerical features. As stated above, the content of a label is implicitly numbered in the absence of a feature. As a result, the sub-widgets are generally defined with no feature prefixed: `td(label(text:"Hello world"))` instead of `td(1:label(text:"Hello world"))`.
- By convention some features are valid for all widgets, and have special meaning. For example the `name` feature defines an atom that is a feature of the object returned by the `Build` function, for further referencing of the widget.
- The rest of the features define the parameters of the widget in its initial state. Valid features are widget dependent. For example a label widget has a text feature, while a frame widget does not have this feature.

This hybrid approach allows writing examples that are easy to read and understand. The examples in this thesis will therefore use this approach.

3.2 Geometry management

As stated in 3.1.2, we use container widgets to organize the placement of the widgets on screen. A single UI description specifies at the same time the widgets of the UI and their placement. This is in opposition to a split approach where the widgets of the UI are specified separately of their organization on screen. The approach of the container widgets comes historically from QTk which did not support migration. It allows for simple UI specifications that are human readable. The changing of geometry is typically achieved

by moving widgets from one container to another, which is a simple task to perform by hand or by computation.

However we have a stronger reason to use this approach with EBL because of the migration support. The next chapter details further this distributed aspect, for now it suffices to know that the migration support is at the widget level: all widgets have the capability to dynamically migrate from one site to another. However UIs are organized in groups of widgets, and it has more sense to migrate/adapt these groups as a whole. For example in modern applications, it is common to have different toolbars that group sets of the functionality of the application together. These toolbars can often be moved around docked at the top, bottom, left side, or right side of the window, or as an independent floating window.

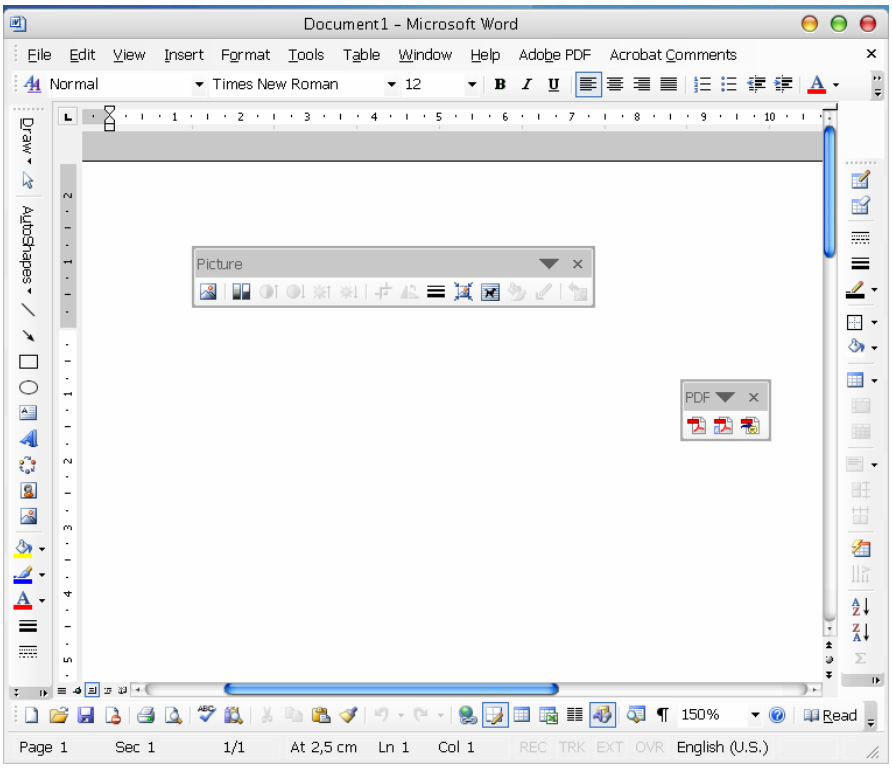


Figure 3-2 Toolbars are movable entities

These toolbars are not migratable in the sense of this thesis, as they stay at the host application. However they are movable and a good example of the functionality a migratory-capable system should offer.

We observe that there are two levels of movability:

1. Whole toolbars. The top/left handle allows to move the toolbar using a drag & drop operation. The logical relationship between the icons of the toolbar (their order) is kept when moved.
2. Independent icons inside a toolbar. The *Customize…* menu allows to modify the set of icons active in each toolbars.

Generalizing these two levels, we see that we need a system that allows migrating widgets at widgets level, or group of widgets level. The migration of a group of widgets may require a reorganization of the geometry of the place they are leaving, and of the place they are arriving to. This reorganization should follow some logical relationship between the widgets and in particular those of the group being migrated. In fact this logical relationship imposes some restrictions in the management of the placement of the widgets.

3.2.1 Absolute coordinate geometry

If widgets are placed using absolute coordinates independently of each other, the relationship information is just not there. In this case, widgets cannot be migrated in groups easily: at best the system can infer (guess) relations between them, without providing any guarantee that the result makes any sense. Another solution would be to have a complete window migration, at the pixel level, so as to be sure to keep the geometry exactly as intended by the application. This is similar to what VNC does. However, you cannot reasonably expect any kind of adaptation support with this approach. Consequently absolute geometry manager are ruled out.

3.2.2 Free rectangular splitting

From the point above, we conclude that the widgets must be placed relatively to each other. For simplification, we assume all widgets occupy rectangular areas of a window (this is generally true for all graphical toolkits). The window space is split in rectangular areas, and the widgets are placed inside them.

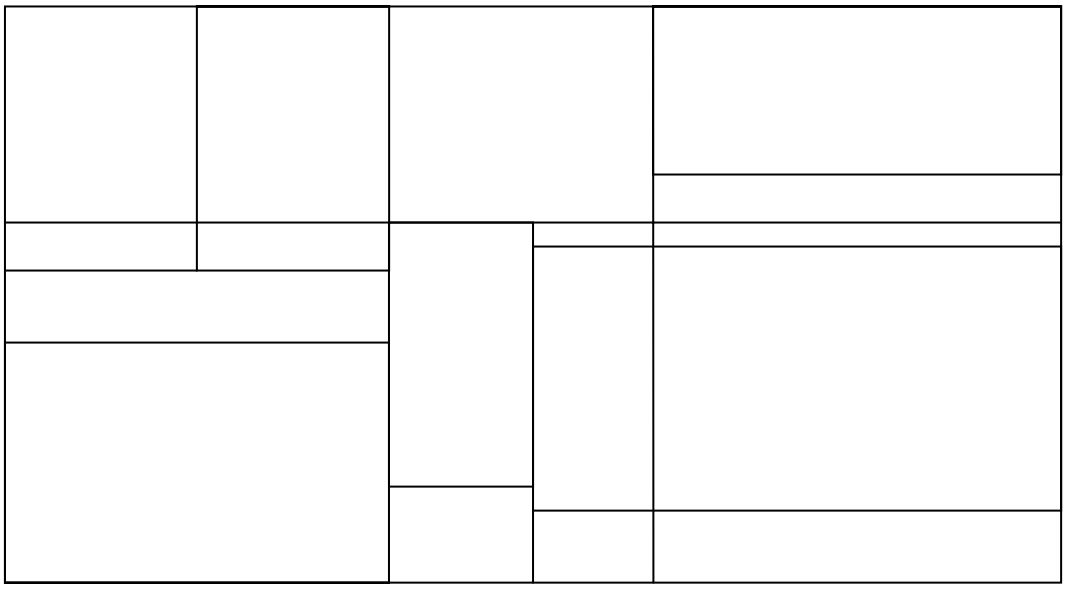


Figure 3-3 Splitting the window into rectangular areas

This is however still not complete enough for migration. First there is no direct relation between widgets that don't have contiguous borders, so we cannot determine how they should be placed into the receiving window.

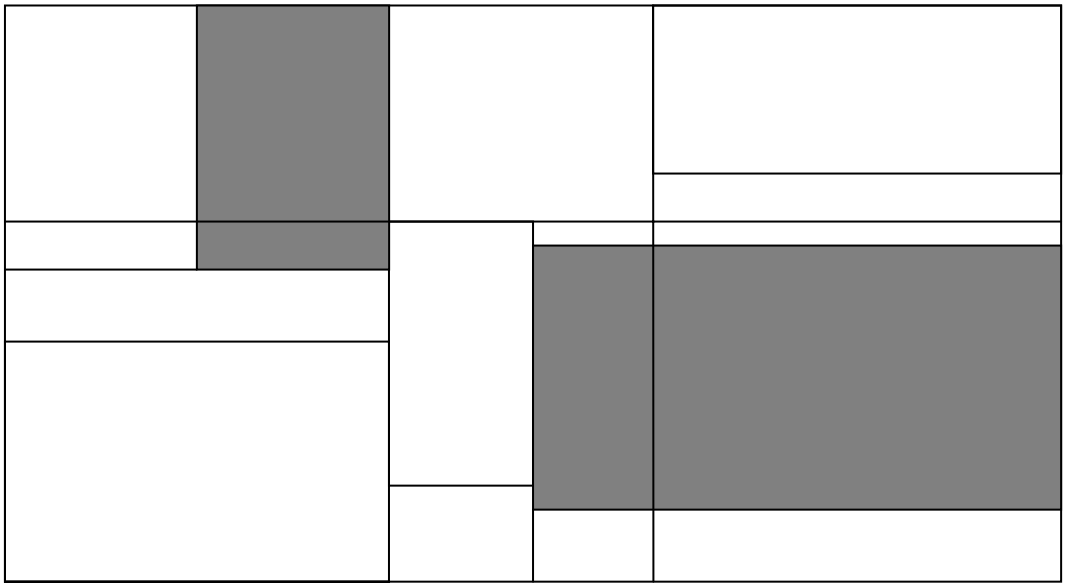


Figure 3-4 Non contiguous widgets

Second, the relation between contiguous widgets is also not complete enough: the resulting outside border can have any shape, and we cannot be sure that the receiving window can accept that shape.

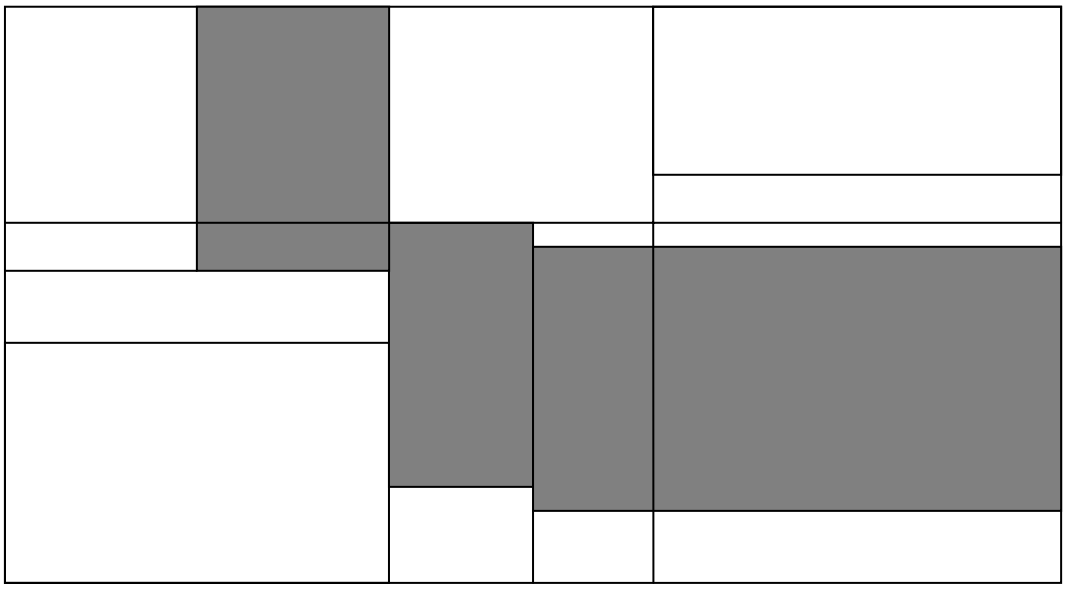


Figure 3-5 Arbitrary shape

If we assume that widgets can be stretched vertically and horizontally arbitrarily, then rectangular shaped group of widgets can be migrated into rectangular areas.

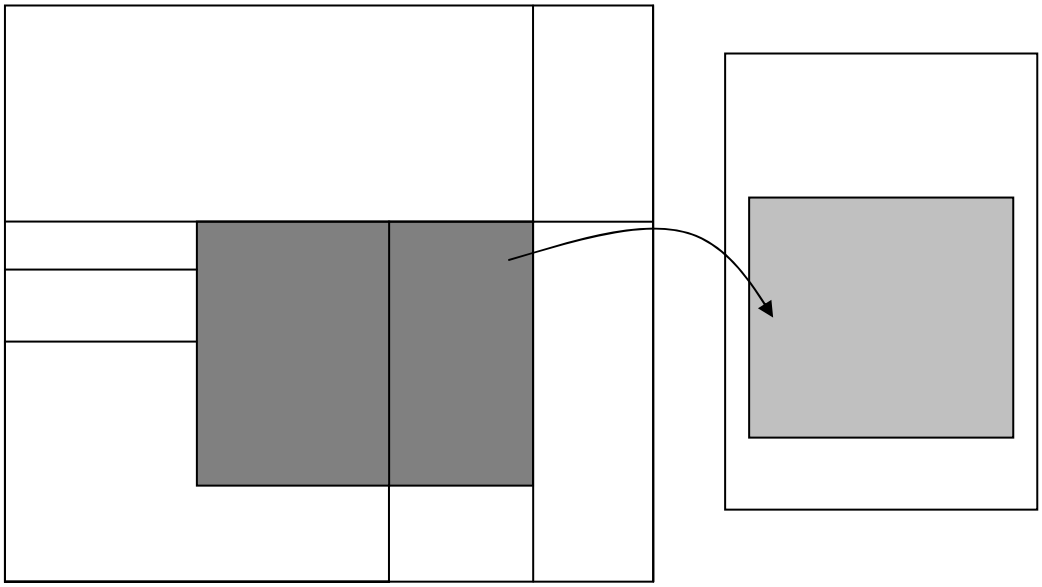


Figure 3-6 Rectangular shape

3.2.3 Hierarchical containers

Although the rectangular shape is good enough for migration, there is still a deeper problem: do the widgets inside the rectangular area have a logical relationship that has sense for migration/adaptation? There surely are situations where this is not the case. As EBL does not provide a model for specifying this relationship, we cannot add this information externally. Instead, we use the widgets themselves to carry logical relationship information, using a container widget approach for managing the geometry. In this approach, container widgets organize zero, one or more contained widgets in the rectangular area they occupy. Container widgets represent at the same time the physical organization of their contained widgets, and the logical relationship between them. For example, a toolbar container organizes physically its buttons on screen, but also represents the logical relationship between them (they form a toolbar). The migration/adaptation mechanism works at the widget level, hence in this case at the whole toolbar level, or its individual buttons.

The most generic possible container widget is the table. In a table, each column and row splits the globally available space.

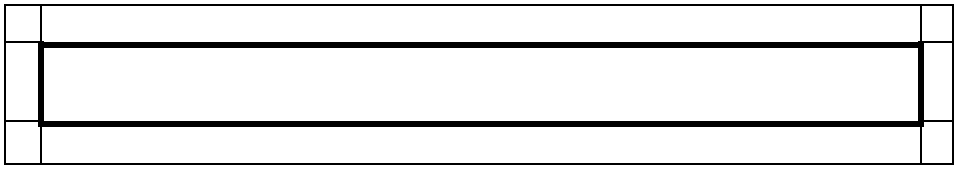


Figure 3-7 Nested tables

Widgets are placed inside cells of the table, possibly spanning over several cells of rectangular form.

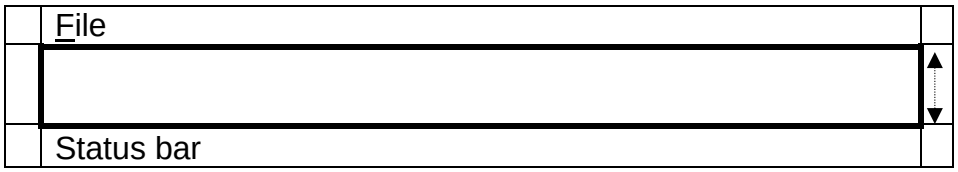


Figure 3-8 Widgets inside cells

Inside their respective cells, each border of each widget is in contact with zero or up to four of its cell border.

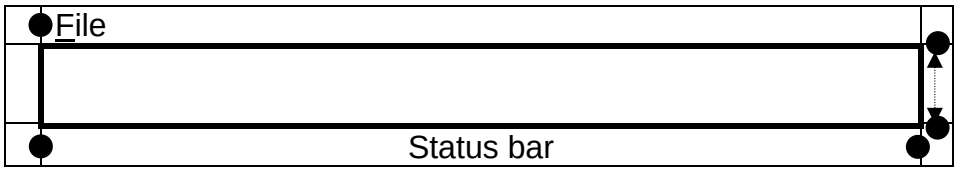


Figure 3-9 Widgets border

The black dots in the Figure 3-9 show that the File menu is configured to stay on the left side of its cell. The status bar is configured to touch the left and right side of the cell by expanding itself horizontally. The scrollbar on the right is configured so that the widget touches the top and bottom side of the cell by expanding itself.

3.2.3.a EBL td and lr widgets

EBL provides a specific support for table widgets using the declarative approach. These widgets organize the contained widgets respectively top to down and left to right.

```
UI={Build
    window(name:window
           lr(glue:nswe
              td(glue:nswe
                 menubutton(text:"File" glue:w)
                 main area widget
                 label(name:statusbar text:"Status bar"
                       relief:sunken glue:ew))
              scrollbar(glue:ns orient:vertical)))}
```

This example creates the window of Figure 3-10.

Figure 3-10 EBL geometry management example

Any of the contained widgets can recursively be a container. The `td` and `lr` widgets can split a window into packed rectangular areas. To determine what size these areas must occupy, each widget has a `glue` parameter that place constraints on them. Without going into full details, one can choose that either a widget should occupy only the size required in a specific direction, or take as much space as possible. One can also choose to stick widgets to none, one, or more of its four possible edges to "glue" its neighbors or container border.

It is possible to have a grid structure where all widgets are organized in lines or columns of the same size (Figure 3-11). The `lr` (resp. `td`) widget supports the `newline` special code which makes the following contained widgets jump to a new line (resp. column) right below the previous widgets, keeping the same column structure (resp. line) with the widgets above them. The `empty` special code leaves an empty space in a line (resp. column) and the `continue` special code spans a widget over several columns (resp. lines).

```
lr(button(text:"One" glue:we)   button(text:"Two" glue:we)
   button(text:"Three" glue:we) newline
   button(text:"Four" glue:we)  button(text:"Five" glue:we)
   button(text:"Six" glue:we)   newline
   button(text:"Seven" glue:we) button(text:"Height" glue:we)
   button(text:"Nine" glue:we)  newline
   empty button(text:"Zero" glue:we) continue)
```

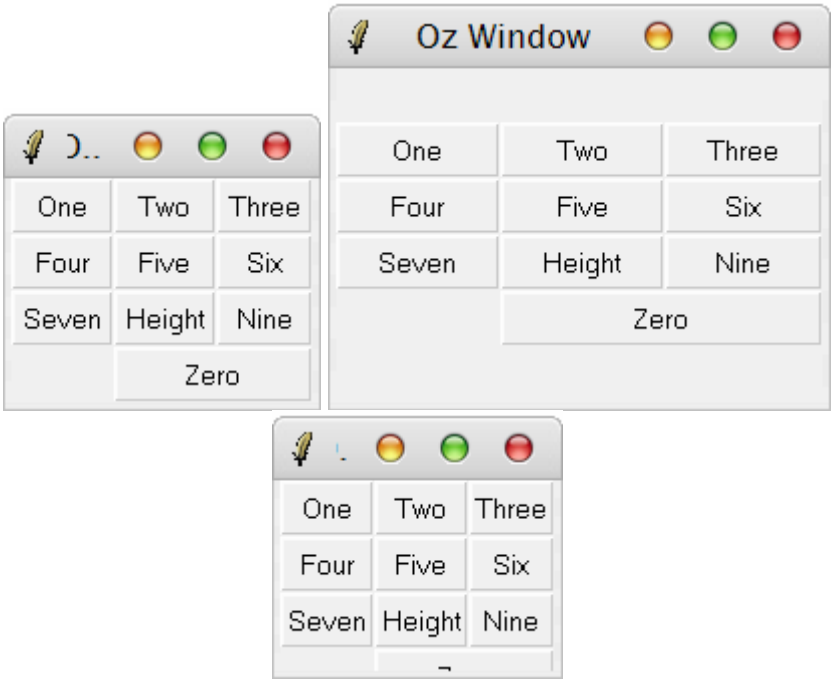


Figure 3-11 EBL geometry management with grid structure example.

When the window is stretched horizontally, the widgets grow or shrink to occupy the whole space. When the window is stretched vertically, the widgets keep their vertical size, disappearing if there is not enough space.

Graphical toolkits usually provide a table structure for geometry placement. EBL provides special functions for creating compatible `td` and `lr` widgets on top of the table widget.

3.3 Combining object-oriented and model-based approaches

Section 3.1 introduced the hybrid declarative-imperative approach supported by EBL. Now we further analyze this approach with respect to adaptation.

There is a very long tradition of developing graphical user interfaces by means of object orientation (OO). In fact it is arguably the best example of the benefits of OO. As this approach uses the full capacity of a Turing complete [Brai74] programming language, any UI can be created. However, first there is a lot of verbosity involved, and second concerns that are different in nature are intertwined together. For example the code managing the actual task of the user is mixed with the code that arranges the UI components on the screen. These yields to high development and maintenance cost. To tackle these problems, a new trend is to use model-based approaches, where each separate concern is expressed orthogonally to the others, in a declarative way. Declarative languages are not Turing complete though, so not all UI can be created by this approach. In particular it is difficult to envision all the future possible needs for the models, which can result in cumbersome extensions to the models. A well known example is the development of the web: HTML is a declarative representation of the content of a web page. However over

the time new needs appeared: client-side interactivity, separation of concern between the content and the presentation, and partial updates of the content of a page. The technologies introduced for covering these needs are:

- Client-side Turing complete scripting language (eg Javascript).
- Separate styling mechanism (CSS).
- AJAX technology relying on XML exchanges between the client and the web server for partial page updates.

Consequently modern web applications are a mix of 4 different programming languages, each one using its own concepts, syntax and semantics. The expertise required to develop these applications is huge, as is the development and maintenance cost.

In this thesis we advocate a hybrid approach, where we can bridge imperative and declarative approaches as needed depending on the situation.

3.3.1 Imperative object-oriented approach

In this approach, the components of the user interface are represented by objects. These objects are instances of classes that form a hierarchy depending on the specialization of the component. For example, a button is a specialized label that is able to receive mouse clicks. The label is itself a specialized frame that is able to display some text. From the application point of view, the user interface is a set of objects created from the classes provided by the toolkit. User inputs are retrieved in an event based fashion, and the objects are configured to execute some code when the user executes some action. All mainstream graphical toolkits follow this scheme (GTk, Windows API, Java AWT, Mac API, …). The main advantage of this approach is the complete expressivity power due to the Turing completeness of the underlying OO programming language. For this reason, EBL fully supports this approach, providing an object oriented API. At the implementation level, we thus have:

- Widgets, defined by classes, instanced into objects the application use to interact.
- Events, configured by the widgets and the user's action to execute some code inside the application.

```
Window={New ETkWindow init}
Label={New ETkLabel init}
{Window show}
{Window wm(title:"Test")}
{Label set(text:"Hello World")}
{Window display(Label)}
```

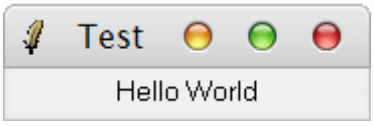


Figure 3-12. A sample ETk application using object orientation.

3.3.2 Declarative approach

Another way of developing graphical user interfaces is by use of models. Different models statically describe the different (potentially dynamic) aspects of the user interaction: layout of the widgets, reactions to user events, but also the task at hand, the user's environment, the navigation between the different parts of the UI, and so on. Model based

approaches have much deeper knowledge of the semantics of the user interaction. If the model is complete enough, it becomes possible to dynamically calculate a concrete user interface corresponding to a change of environment, i.e. automatically adapt the UI. The completeness of the model is not trivial though, particularly when facing the evolution of needs over time. Also, the complexity of the models is often inversely related to its expressivity power, and complex models tend to be hard to use. This complexity can be mostly hidden away by using modeling tools instead of directly manipulating the models. Another interesting benefit of the models is the possibility to make calculations on them directly. In particular models allow the creation of metrics, for example measuring the maximum number of mouse clicks required to achieve a particular task.

A number of researches are already exploring the model-based approach [Calv05, Limb04a], so this thesis took a different route. Model-based approaches focus first on the problem of how to describe migratable & adaptable UIs. However, they also have to solve the problem of how to actually run such interfaces. This thesis focuses mainly on this problem, even though we also introduce a bit of model-based approach. In particular we do not offer editing tools for our models.

3.3.3 Hybrid approach

EBL is provided for the Mozart Programming System, supporting the Oz programming language. This language supports symbolic programming, in particular the record data structure. This structure is expressively equivalent to XML, but fully integrated with the rest of the language. In particular, the mixing of a record inside a procedure or an object is direct. Conversely, features of a record can be procedures or objects. The imperative and declarative worlds are seamlessly intertwined together. EBL takes advantage of this to provide a model for constructing widgets in their initial state, along with their behavior upon resizing. This model covers presentation model and part of the dialog model (behavior upon window resize, and for some widgets some of their events) of classical model-based approaches.

```
UI={Build window(name:window
                 label(text:"Hello World" glue:nswe))}
{UI.window wm(title:"Test")}
{UI.window show}
```

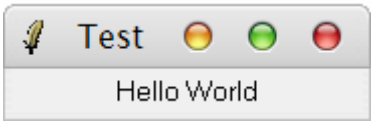


Figure 3-13. A sample ETk application using the hybrid approach.

A valid EBL UI specification is a record where:

- A widget is defined by a properly formatted record.
- The label of the record defines the type of the widget.
- By convention, numerical features define widgets that must be placed inside this one.
- By convention some features are valid for all widgets, and have special meaning. For example the `name` feature defines an atom that is a feature of the object re-

turned by the `Build` function, for further referencing of the widget. Another example is the `glue` parameter that specifies the behavior of the widget upon window resize.

- The rest of the features define the parameters of the widget in its initial state. Valid features are widget dependent. For example a label widget has a text feature, while a frame widget does not have this feature.

The `Build` function takes a valid EBL hybrid model as input, creates the user interface correspondingly, and returns an object providing control over all the created widgets. In particular, this object allows direct access to the created widgets that have used the name feature. Once a UI has been built, the developer gets back the objects controlling the created components, and falls back into the object-oriented paradigm.

EBL widgets are defined by classes. To support the hybrid approach, a one parameter function has to be attached to each widget defined. This function takes the record as received by the build function for that particular widget, and returns the corresponding object configured as requested by the widget.

3.3.4 Relation between the hybrid approach and adaptation

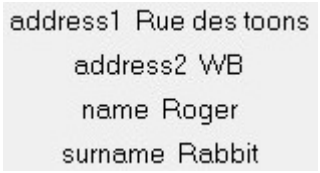
The functional paradigm of Mozart allows easy manipulations of declarative data structures. In particular, we can transform data into records properly formatted for EBL.

```
Data=data(name:"Roger"
          surname:"Rabbit"
          address1:"Rue des toons"
          address2:"WB")

fun{Transform1 D}
   {List.toTuple td
    {List.map
     {Record.toListInd D}
     fun{$ I#E}
        lr(label(text:I)
           label(text:E))
     end}}
end
```

The parameter `D` of the function is firstly transformed into a list of pairs: `featX#valX`. This list is mapped to a list where each elements have the form: `lr(label (text:featX) label(text:valX))`, where `X` is the position of the item in the list. This list is transformed back into a tuple, i.e. a record where each feature is implicitly numbered. The example record is thus transformed into (assuming implicit numbering):

```
td(lr(label(text:address1)
      label(text:"Rue des toons"))
   lr(label(text:address2)
      label(text:"WB"))
   lr(label(text:name)
      label(text:"Roger"))
   lr(label(text:surname)
      label(text:"Rabbit")))
```

This record is a valid EBL specification of this UI:



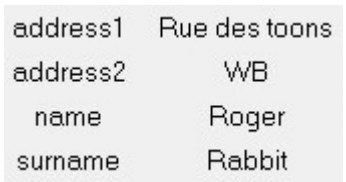
We can use a different transformation:

```
fun{Transform2 D}
   fun{Loop P}
      case P of I#E|Xs then
         label(text:I)|
         label(text:E)|
         newline|
        {Loop Xs}
      else nil end
   end
in
   {List.toTuple lr
   {Loop {Record.toListInd D}}}
end
```

Like `Transform1`, the function `Transform2` first transforms the record given as parameter into a list of pairs `featX#valX`. This list is then transformed by the `Loop` function and the resulting list transformed back into a tuple whose label will be `lr`. The `Loop` function recursively parses a list of pairs and creates another list where for one item `featX#valX` in the first list corresponds three items in the second list: `label(text:featX)|entry(init:valX)|newline`. The resulting record is:

```
lr(label(text:address1)
   label(text:"Rue des toons")
   newline
   label(text:address2)
   label(text:"WB")
   newline
   label(text:name)
   label(text:"Roger")
   newline
   label(text:surname)
   label(text:"Rabbit")
   newline)
```

This record is a valid EBL specification for this UI:



More transformations are possible, for example we could replace the label widgets on the right by entry widgets, resulting in this UI which allows edition by the user:

As we see, the hybrid approach allows the creation of application specific models, and their interpretation into runnable UIs.

3.3.5 Relation between the hybrid approach and MVC

Model-view-controller (MVC) [Kras88] is an architectural pattern used in software engineering. In complex computer applications that present a large amount of data to the user, a developer often wishes to separate data (model) and user interface (view) concerns, so that changes to the user interface will not affect data handling, and that the data can be reorganized without changing the user interface. The model-view-controller solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

It is easy to extend the approach of the previous section to have an explicit MVC approach:

- The model describing the data should be made mutable so that the controller is allowed to modify the model:

```
Data=data(name:{NewCell "Roger"}
          surname:{NewCell "Rabbit"}
          address1:{NewCell "Rue des toons"}
          address2:{NewCell "WB"})
```

The `NewCell` function creates a mutable cell entity; the parameter defines the initial value of the cell.

- With MVC, there is a view associated to the model. With EBL, this view is defined by the description record the UI is built from. The name feature of the widgets will allow the controller to control the widgets defined in the view.

```
Desc=td(lr(label(text:address1)
           label(text:@Data.address1 name:address1))
        lr(label(text:address2)
           label(text:@Data.address2 name:address2))
        lr(label(text:name)
           label(text:@Data.name name:name))
        lr(label(text:surname)
           label(text:@Data.surname name:surname))
        lr(button(text:"Ok" name:ok) button(text:"Cancel" name:cancel))
```

- With MVC, there is a controller associated to the view/model. With EBL, the controller is implemented by the imperative paradigm.

```
UI={Build window(name:top Desc)}
```

```
{UI.top show}

% clicking on Ok sets the values in the model to the current input of the
user

{UI.ok bind(event:'1'
            action:proc{$}
                      {ForAll [address1 address2 name surname]
                       proc{$ Item}
                          Data.Item:={UI.Item get(text:$)}
                       end})}
```

Note that we can still use the approach from the previous section to associate multiple combinations of views and controllers to the same model. Instead of hand-defining the associated view and controller, we use the functional paradigm of Oz to calculate them.

```
fun{Model1 D}
   {List.toTuple td
    {List.adjoin
     {List.map
      {Record.toListInd D}
      fun{$ I#E}
         lr(label(text:I)
            label(text:@E))
      end}
     [lr(button(text:"Ok" name:ok) button(text:"Cancel" name:cancel))]}}
end

proc{Control1 D UI}
   {UI.ok bind(event:'1'
               action:proc{$}
                         {ForAll {Arity D}
                          proc{$ Item}
                             D.Item:={UI.Item get(text:$)}
                          end})}
end
```

The `Model1` function takes the model as parameter and returns the description record defining its view. The `Control1` procedure takes the model and the UI built out of the corresponding view as inputs, and creates the UI control. Similarly to the previous section, we can define more functions to generate alternate views and their associated controllers.

3.3.6 Relation between the hybrid approach and Arch/Slinky

Arch/Slinky [Bass92] is another architectural pattern used in software engineering.

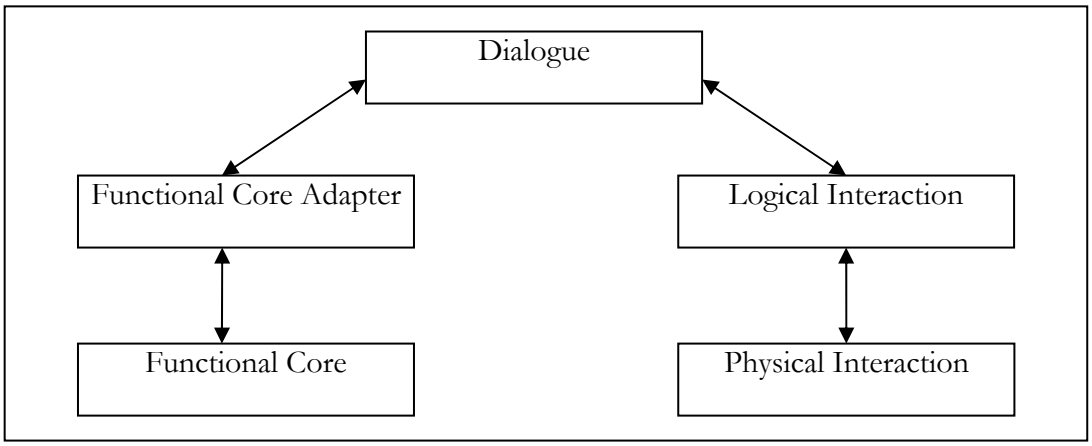


Figure 3-14. Arch/Slinky.

Arch/Slinky identifies the following five basic functions of user interface software as shown in Figure 3-14:

- Functional Core (FC). This component performs the data manipulation and other domain-oriented functions. It is these functions that the user interface is exposing to the user. This is also commonly called the Domain Specific Component, or simply the Application.
- Functional Core Adapter (FCA). This component aggregates domain specific data into higher-level structures, performs semantic checks on data and triggers domain-initiated dialogue tasks.
- Dialogue (D). This component mediates between the domain specific and presentation specific portions of a user interface, performing data mapping as necessary. It ensures consistency (possibly among multiple views of data) and controls task sequencing.
- Logical Interaction (LI) component. This component provides a set of toolkit independent objects (sometimes called virtual objects) to the dialogue component.
- Physical Interaction (PI) component. This component implements the physical interaction between the user and the computer. It is this component which deals with input and output devices. This is also commonly called the Interaction Toolkit Component.

We can use this pattern in the context of a multi-paradigm programming language with an hybrid declarative/imperative toolkit:

- The FC is typically developed using an imperative paradigm (OO or functional style).
- The FCA component is preferably developed using a functional programming style as it is well adapted to aggregate, filter and further manipulate data.
- The dialogue uses a functional programming style to generate high-level UI descriptions corresponding to the tasks, and an imperative style to control tasks sequencing.
- The LI uses a functional programming style to further transform the high-level UI descriptions from the dialogue into valid toolkit descriptions. In many cases the dialogue can generate directly to valid toolkit descriptions, and the LI layer is empty.

- The PI builds the concrete UI out of the UI description record, by calling the build method of the toolkit.

In summary: first the functional programming style is well adapted to manipulate data in general, which is very helpful to FCA, D, and LI. Second, the hybrid approach of the toolkit provides a high-level of expressivity to GUI programming that D can target directly, reducing LI to nothing and PI to a simple call to the build method of the toolkit.

Chapter 4 EBL Distributed Toolkit Design

The previous chapter introduces the EBL approach as a general graphical user interface toolkit. This chapter now covers the distributed aspects of the toolkits, which involve migration but also adaptation and limited support for multi-user applications. This gives us an overall view of the design of EBL. When pertinent, examples using the EBL/Tk toolkit are provided to give the readers insight on the impact of the design decisions over the final result. In section 4.1, we start by introducing migration and adaptation as useful and conservative extensions of the classical imperative toolkit paradigm. This section contributes the concepts of migration capability (a givable token that triggers a migration), adaptation as a configuration parameter, and migration and adaptation transparency (the process of the migration/adaptation is independent of the concurrent running of the application). Section 4.2 presents the general distributed structure of a widget. Section 4.3 introduces the EBL Store construct which implements the whole migration, adaptation and limited multi-user functionality. This is the main technical contribution of the thesis. Section 4.4 addresses the receiving end of a migration. Section 4.5 focuses on the adaptation mechanism which is a direct simple extension of the migration mechanism. Finally sections 4.6 and 4.7 address distributed aspects also related to this thesis: the low level network implementation and the security issues.

4.1 Migration and adaptation properties

EBL is designed to provide migration and adaptation functionality with a maximum of flexibility, yet with a minimum of impact on the development cost. These requirements drive the design in specific directions.

4.1.1 Granularity of migration & adaptation

A running application could have its UI migrated and/or adapted at different levels of granularity:

1. Whole screen containing the UI of the application (which may also contain the UI of other running applications).
2. Whole UI of the application, typically contained in a single window.
3. Subset of widgets of the application:
    a. Limited to a single widget.
    b. Limited to widgets that respects some placement constraints (for example respecting a rectangular shape).
    c. Any arbitrary selection of widgets.
4. Arbitrary pixel area.

Not all these levels are interesting for our purpose. In level 1, we lack information regarding the remaining of the screen which makes virtually impossible to provide interesting adaptation. In level 4, the arbitrary nature of the area also makes it virtually impossible to provide interesting adaptation. Level 2 is a particular case of level 3, where the whole UI of the application is used instead of a particular subset of it. Consequently

EBL provides migration and adaptation support at the widget level. We want a maximum of flexibility: any widget can be migrated to any platform at any time. Two widgets from the same running application can be migrated to the same platform, or to two different ones. As the migration is independent for each widget, covering 3a is enough to also cover 3b and 3c, by executing several migrations at the same time.

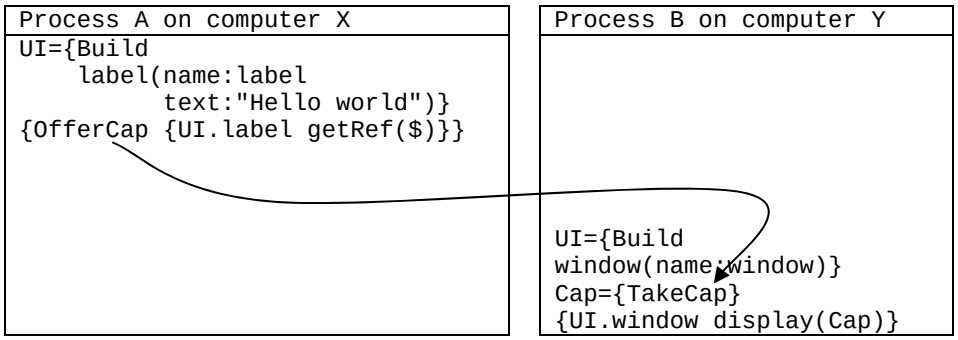
4.1.2 Orthogonal migration & adaptation

To be useful, a graphical toolkit with migration and adaptation support must still offer a functionality equivalent to a graphical toolkit with no such support. In other words the migration and adaptation are new functionality on top of the pure graphical toolkit functionality. We argue that this new functionality is important enough to be isolated from the pure toolkit functionality. EBL is consequently designed to provide the migration and adaptation functionality orthogonally to the pure graphical one.

- The migration functionality is provided as a *capability* of the widget. This capability is a value which can be passed along freely to **another process**, on **another computer**: the migration is a distributed operation between different computers connected over the Internet (local migrations on the same computer/process are of course supported). Once a *migration capability* has been passed to another process, it can be used to trigger the migration of the widget, like a PULL mechanism. To achieve this, the capability serves two purposes: 1) it contains the authority to migrate the widget, and 2) it is a reference to the home site of the widget over the Internet, like a URL for a web page. Because of 2), we often call the *migration capability* of the widget the *reference* of the widget.

  It is the responsibility of the application to pass the capabilities to interested parties: it has the complete control on who receives them. However it does not have the control on when these capabilities are used by the remote peers, i.e. when the migration really occurs. Consequently the application should be as impermeable to the migration process as possible. The only observable effect is a temporary blocking of the threads interacting with the migrated UI. For this reason we say that the migration is *transparent* to the application. Note that the application can register its interest for migration events if it wants to be notified of the process. Note also that the application has a direct access to the capabilities of the widgets it has created, and consequently can use them to migrate the widget back into its original place.

  In summary, the application that has created a widget must provide its capability and the application where this widget must be migrated to must get the capability and use it to trigger the migration. Note that no external meta-UI [Cout03b, Cout06, Roud06] is required by this process: if a UI is required so that the user can control the migration, it is part of the whole UI of the application. In other words, the meta-UI is defined by and embedded inside the application itself. This control UI could be implemented by a drop down menu for sending part of the whole UI, by detecting a drag and drop operation from the user, or any other way... As this control UI is no different than the rest of the UI, it can also be migrated!



This example illustrates the computer X offering the migration capability of a label widget, and the computer Y creating a window, getting the capability, and using it to migrate the label into its local window. Note that the `OfferCap` and `TakeCap` functions are not specified here and can be implemented in numerous different ways. This example assumes that they are able to get in touch with each other, and then exchange the piece of information given to `OfferCap`. A possible implementation is to have a shared file between the processes. Another possible implementation is to rely on emails: `OfferCap` sends an email with the capability attached to it to a mail box that is then read by `TakeCap`. Still another possible implementation is to use a DHT (distributed hash table) based P2P (peer to peer) network, and use a shared name between the two processes to place the value into the network, and obtain it back. And many more implementations are still possible.

- The adaptation of the widgets consists in changing its representation (presentation and/or interaction) while keeping a useful level of usability. In that sense, the simple reconfiguration of a visual parameter of a widget like its background color is already an adaptation of that widget. EBL pushes this view forward by introducing a special adaptation parameter to every widget. When this special configuration parameter is changed, it is the whole way the widget is displayed that is changed. Once again, this process is impermeable to the application; the only observable effect is a temporary blocking of the threads interacting with the adapted UI. For this reason we say that the adaptation is *transparent* to the application. Because of this transparency, the application is independent of the representation currently used for a particular widget. For example the target device of a migration could provide its own representation of the widget, adapting it on the fly to its own specifics.

```
UI={Build window(selector(name:selector
                          text:"Car Model"
                          items:["Ford" "Peugeot" "Renault"]))}

{UI.selector setContext(list)}

...

{UI.selector setContext(default)}
```

This example illustrates a selector widget that supports different representations. Switching between these representations is achieved by calling the `setContext` method. It is up to the application to define why and when the widget should change the representation. In this example, the representation is just changed at unspecified points in the execution of the code. In the example of Figure 1-3 on page 16, the representation is changed as a reaction to a user selection.

Coupled with the hybrid functionality of EBL, it is easy to develop a widget with multiple possible interpretations, each of them supported by a different renderer. The application can then use this widget, and dynamically switch between the renderers by the simple call of the `setContext` method.

4.2 Overview of the distributed structure of a widget

Desktop applications are often centralized applications running the functional core and the UI inside a single process of a computer. Some of them have a distributed functional core (voice over IP applications for example), but that is not what interest us in this work. Once we let parts of the UI migrate from site to site, several devices become involved in the running of the application, and we also shift from a centralized environment into a distributed one.

The way EBL introduces distribution is dictated by the design choices:

- Any widget of a running application can be migrated at any time (transparent distribution). Consequently EBL widgets are distributed entities. At anytime they may be situated at the application's process, a remote process, or even nowhere if they are not currently displayed. Later we will see that it is also possible to have several renderers connected to a single proxy, replicating the UI of the widget at several places simultaneously.
- As for the functional core of the application, EBL does not dictate if it should be distributed or stationary, nor does it offer any support for distribution.

EBL provides specific distribution support for all widgets, allowing them to dynamically migrate from one site to another. But the widgets are also used by the functional core of the application that interacts with the UI, so part of them should behave in a stationary way. The distribution scheme of widgets is composed of:

- A part that is stationary to the process that created the widget. That part is returned to the functional core of the application so that it can interact with the widget. This part is called the *proxy* of the widget.
- A part that is distributed, and run on the site actually displaying the widget. That part is the one the user can interact with. This part is called the *renderer* of the widget.

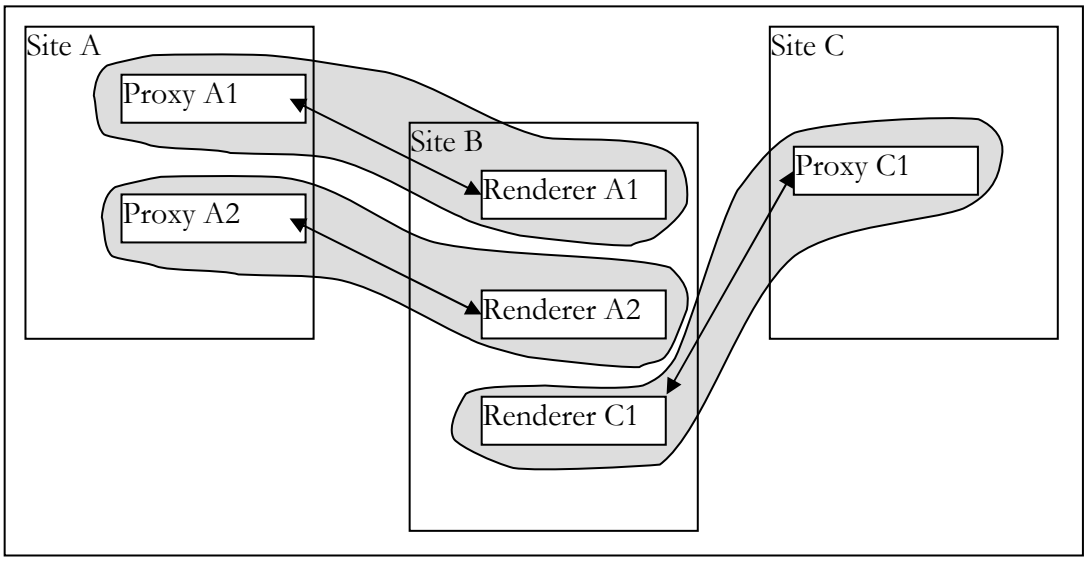


Figure 4-1 Distributed architecture example

In Figure 4-1, three sites are running on three different computers. Site A creates two widgets that are migrated into Site B. Site C creates one widget that is also migrated into Site B. Each gray area covers a widget in its distributed execution. The proxies stay at the site that created them forever. However the renderers are running at the site the widgets are migrated to. The proxies and the renderers are connected together over the Internet, so as to be synchronized.

4.2.1 Fault tolerance

One of the most difficult aspects of distribution is the fault tolerance: distributed application should be tolerant to network faults; they should have an acceptable behavior in such situation. The transparent migration of EBL means the application has very little control to when and why part of its UI becomes distributed. Obviously under this condition, the application must be very tolerant to network faults. It would be unacceptable if a transparent migration followed by a network problem resulted in the application crashing or misbehaving. Also, it would be unacceptable if the widget ended up in an inconsistent state between the proxy and the renderer which would result in inconsistencies between the application and what the user has in front of his eyes. Different types of distribution problem can happen in the proxy-renderer relationship:

1. The application running the proxy crashed. In this situation there is no reason to keep a ghost UI with no application behind it. Consequently the renderer destroys itself, removing the widget from view.
2. The application running the renderer crashed. In this situation the application should continue working on because of the transparent nature of the migration. The widget is still migratable, and when a new renderer comes in, it should be put in a state that is consistent with the application. To enforce this, EBL restricts the collaboration between the proxy and the renderer to an abstraction that implements this consistency. With this abstraction, the proxy serves as the reference of the state of the widget and the renderer follows the proxy updates to reflect them to the user. The renderer is not allowed to update the state directly; it has to submit the update to the proxy which can then apply it globally. For example,

when a user types in a text entry, the renderer submits the new text to the proxy which then updates the state of the widget globally; eventually the renderer receives the update from the proxy. To have a better user feedback, it is possible to program the renderer so that its UI reacts directly to the user input, while the update is done in the background (by anticipating the update from the proxy and apply it immediately instead of waiting for the message to bounce). However in this situation, the UI may be in a transient inconsistent state with the proxy: if the renderer fails and the widget is migrated again, the new renderer may be in a different state than the former renderer. The time taken by an update message is two times the delay of the network between the proxy and the renderer: this is often measured in the hundred of milliseconds. Consequently the potential inconsistency of the renderer is typically very limited. In summary the EBL approach allows to have a UI that reacts instantaneously to the user input, while keeping an eventual consistency between the proxy and its UI. In case of network faults, small updates may be lost though.

3. Indefinite/transient network problems between the proxy and the renderer. On the Internet, there is typically no information available when a remote site fails: it does not notify that it is down, it just stops responding. Similarly, there is no information if a network link is down; all traffic on this link is just lost. In the first scenario, the remote site is gone forever (the application may be restarted, but it is a new process, the old one is gone forever), while in the second scenario the remote site can come back (the network gets reconnected). The Mozart distribution layer provides support for detecting when a remote site stops responding, but of course is unable to detect if the site will come back later or not. EBL interprets this information differently at the proxy and renderer side:
    a. If a proxy detects there is a communication problem with a renderer, it cuts its connection with it. Even if the communication problem was transient, the connection is terminated. As a result the renderer disappears. We consider this solution to be acceptable because it is always possible to migrate the widget again when the connection is more stable. And in case of a permanent failure, the connection is rightly cut.
    b. If a renderer detects there is a communication problem with its proxy, it cuts its connection with it. Even if the communication problem was transient, the connection is terminated. As a result the renderer disappears. We consider this solution to be acceptable because it is always possible to migrate the widget again when the connection is more stable. And in case of a permanent failure, the connection is rightly cut.

4.3 EBL store

Widget proxies collaborate distributively with their renderers at runtime. The purpose of this collaboration is to maintain consistency between the state of the widget at the proxy, and the actual rendering of the widget at the renderer. To fix the idea, let's consider a simple button widget. Its state is defined by different parameters:

- The text displayed inside the button.
- The colors used to draw the text and the background of the button, the color used once the mouse cursor is over the button, or when the button is pushed...
- The width and height of the button.

- The image displayed inside the button, if any.
- The type of border drawn outside the button (relief, sunken, flat…) and its width.
- And so forth...

Typically the effective back end toolkit offers a way to dynamically change the value of these parameters. They are named, and the widget maintains a dictionary associating the parameter name with the parameter value. The possible parameter values depend on the type of the corresponding parameter name. For example, a text parameter expects a text string while a width parameter expects an integer. **EBL provides a distributed entity for storing such dictionary; we call this entity a *store***:

- The store maintains name->value associations. Also it is possible to associate default values for specific names.
- The store associates type checking to specific name->value pairs. For example, the store can be configured so that only integer values are accepted for the width parameter of a button. Illegal updates of the store trigger exceptions.

The stores are distributed entities:

- Each peer that has an access to a store can get or update values from that store.
- However one of these peers has the special master role, while the other peers have role of slaves. The master act as the reference for the content of the store. Each update must transit by the master and only then is the update really applied. The master runs at the proxy while the slaves run at the renderers. The disappearance of the master breaks the store (it cannot work correctly anymore). However when that happens, the proxy itself also disappears and the whole widget is gone: there is no more reason for this store to exist at the renderers.
- Implementation wise, a store is a dictionary that is replicated at each peer that has a reference to it. At the master, the content of the local dictionary is the official content of the store. At the slaves, the content of the local dictionary is a cache to the currently known state of the store by this slave. Consequently obtaining information from the store is always performed on the local dictionary. Updating information on the other hand must always transit to the master, which then broadcasts the information to the slaves so that their cache is updated accordingly.
- Eventually each entry of the cached dictionaries used by the slaves has the same value as the dictionary of the master. In other words, the store guarantees eventual consistency of the name->value association betweens its connected peers.
- There is no transactional mechanism to update multiple keys of the store, and guarantee that if one of them is really applied, then all of them are. See the next point for why this is not a restriction.
- The store also provides asynchronous messaging between the master and the slaves. The content of a message can be any Oz value, which can be very complex data structures (lists, trees...). Mozart guarantees the integrity of each message: they arrive completely or not at all. Consequently a slave can send a message instructing multiple updates of the store to the master. Either the master receives this message and then applies each update in turn, or it disappears in the process. In the later case, the whole store disappears. This scheme implements a transaction that is either committed, or the whole store disappears.

- Each peer can be notified when the content of the store is updated. For example, the renderer can configure its store to trigger a particular action when the store is updated by the proxy.
- Each peer can specify translation rules for the content of the local dictionary representing the store, and what is really exchanged over the network. The transparent distribution of Mozart translates automatically many of the types of Oz, the store provides a support for those situations not directly covered by Mozart.
- Finally, stores are attached to widgets, and for convenience widgets can use as many stores as they need; the stores are created on the fly.

In summary **the store is the basic data structure that implements the proxy-renderer relationship**.

Here is an actual example of the usage of a store:

```
{Store setParametersType(t(text:'String' relief:'Relief'))}
{Store setTypeChecker(t('String':String.is#"A String"
                        'Relief':fun{$ L}
                                    {List.member L [flat sunken raised]}
                                 end#"The atom flat, sunken, or raised"))}
{Store setProxyMarshaller(t('String':m(u2s:fun{$ V}
                                              {String.toByteString V}
                                           end
                                        s2u:fun{$ V}
                                              {ByteString.toString V}
                                           end)))}
{Store setRenderMarshaller(t('String':m(u2s:fun{$ V}
                                               {String.toByteString V}
                                            end
                                         s2u:fun{$ V}
                                               {ByteString.toString V}
                                            end)))}
{Store setDefaults(t(text:"" relief:flat))}
```

This example configures a store:

- The allowed keys for this store are `text` and `relief`. Attempt to access other keys will raise an exception. The type of text is `'String'`, and the one of relief is `'Relief'`.
- The type definition of `'String'` and `'Relief'` is composed of two parts: a unary function that returns a Boolean telling if the parameter checks the type, and a string that is displayed in case of type error.
- The proxy side of the store is configured to automatically translate the data provided for the type `'String'` before putting in into the store. One translation is defined when the application sets the value (`u2s:` user to store), and reflexively another translation is defined when the application gets the value (`s2u:` store to user). In this example, the store uses the more compact `ByteString` type for putting strings into the store. Note that the actual translation of the `ByteString` type for transfer over the network is done by the distribution layer of Mozart. The marshal configuration of the proxy allows translating types that are not directly supported by the distribution layer of Mozart.
- The render side of the store is similarly configured to automatically translate the `'String'` type.

- Finally, default values are specified for this store.

```
{Store set(text "Hello World")}
```

This example sets the value of the text parameter to "Hello World".

```
V={Store get(relief $)}
```

This example puts the current value of the relief parameter into the variable V.

```
{Store set(relief wrongvalue)}
```

This example raises an exception because `wrongvalue` is not accepted by the type of `relief`.

```
Event={Store createEvent(action:proc{$} {Show 'Value Changed'} end
                         code:$)}
{Store registerVirtualEvent(text Event)}
```

This example registers an event at the store that displays a `'Valued Changed'` message. The store triggers this event each time the `text` key is modified.

4.3.1 Simple multi-user functionality

Note that there is no condition on the number of renderers connected to a store at any time. It could be zero, one or more, and that can evolve dynamically over time. The eventual coherency ensures that the proxy and all its connected renderers will eventually agree on the state of the widget. In other words, the store implements the independence on the number of connected renderers.

If one renderer is connected to a proxy then there is one physical widget linked to the logical widget viewed by the application. If there is no renderer connected to a proxy then the application still sees its logical widget, and thus can continue working on, but there is no physical counter part for the user to interact with. If there is more than one renderer connected to a proxy, the application still sees a single logical widget, though there are several physical representations of this widget that mirror this widget. These representations can be running on different devices, in front of different users. This is a lightweight and limited way of introducing multi-user functionality that we call *simple multi-user* ability. This functionality is configured by the connection policy of the proxy. When a new renderer joins, this connection policy is executed: by default it disconnects from all previous renderers, ensuring only one renderer is connected at all time. By changing the default connection policy, the proxy can accept the new renderer without disconnecting the others, effectively allowing multiple simultaneous renderers.

4.3.1.a Updated design space

The design space of Figure 2-5 on page 31 is extended to reflect this new functionality.

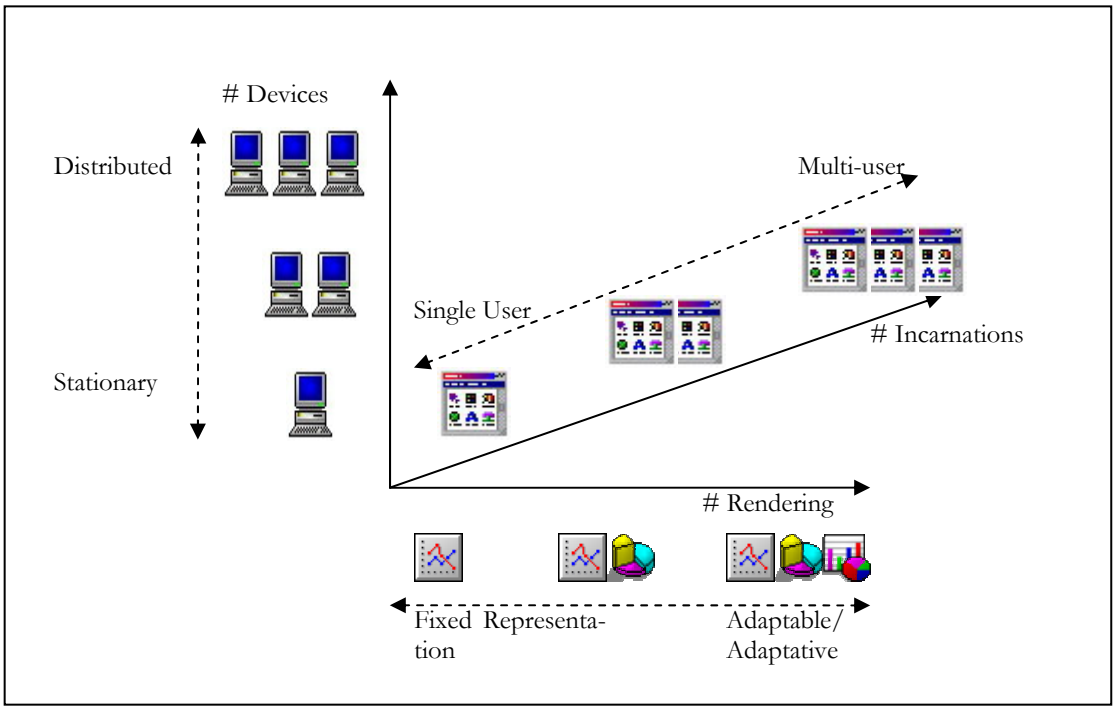


Figure 4-2 Extended design space

Figure 4-2 displays the extended design space where a third dimension is introduced. This axis corresponds to the number of times each component is concurrently displayed. At one end of the spectrum each component is displayed exactly once: this corresponds to he classical single user situation. At the other side of the spectrum the components are concurrently displayed several times, allowing multiple users to interact with them concurrently.

4.3.1.b Ubiquitous widgets

Some widgets are particular in the sense that they are not attached to a particular container. Calling them widgets is a bit of a stretch; they are usually called resources. Text fonts are a typical example: a "Courier 10" font is not a widget per se, still it is a resource defined at the application side, but consumed at the widget side. Other examples are images and drop down menus, as they can be displayed/used multiple times. EBL treat these resources as special widgets, that are able to be displayed multiple times.

4.3.2 Event bindings

Displaying the content of a widget is only half of the interaction between the user and the application. The user must also be able to communicate to the application, typically by using the mouse or keyboard to interact with the widgets. Graphical toolkits manage user interactions in an event driven way: the application registers the user events it is interested in with some callback code; the toolkit triggers the code in reaction to the user inputs. In the case of EBL, the applications have access to the proxies of the widgets and not directly to their renderers. Consequently the events are registered to the proxies, even though these events in fact happen at the renderers. The information relevant to each event (event type like mouse click, parameters to get like the coordinates of the mouse, and a unique id for the callback code) is placed in an EBL store, which is eventually rep-

licated at the renderers. The renderers configure the events using the low level toolkit to trigger a code that reports the event back to the proxy. In turn the proxy triggers the code configured by the application.

Proxy	Renderer

meth bindOnClick(Action)
  Event={Store createEvent(
  event:'<MouseClick>'
  action:Action
  code:$)}
  {Store bind(Event)}
end
	meth bind(Event P)
	  {Widget bind(event:Event
	  action:P)}
	end

This example shows the event management at the proxy-renderer level. The EBL store provides a method to create a distributed representation of an event. Basically, this method creates a unique code that contains the type of the event, its parameters (if any), and replaces the callback code by a unique id, while mapping this id to the actual code in a separate dictionary. The bind method of the EBL store then places this event into the store, which is eventually replicated at the renderer(s). When an EBL renderer manager receives a new event in a store, it automatically calls the bind method of the renderer of this widget. The first parameter of this method is the type and parameters of the event, while the second parameter embeds a callback code to execute. This code is executed at the renderer, and embeds the sending of a message to the proxy to notify of the event. This message contains the id corresponding to the callback code configured by the application for this event. Eventually, the EBL proxy manager receives this message, and triggers the callback code configured by the application.

4.3.3 Causality management of events

With EBL, each widget is an independent entity, migrating from site to site independently of other widgets. However, a running UI does not behave in an independent way: each user action triggers a response that can span on different parts of the UI. For example, clicking on the OK button of a dialog box closes it. For coherency reasons, it is fundamental to respect the causality relationship: successive user inputs should be treated in the order they were entered.

With EBL, it is possible to split the UI of an application on multiple devices. We consider that respecting the global causality of the UI on all these devices to be:

1. Very inefficient: we would need a global clock mechanism to order all events precisely. Also the processing of each event would have to wait long enough to make sure no other event arrived on another device, but the network message was slow to arrive. This would create a constant lag between the user interaction, and the reaction of the application.
2. Not very useful: if different components of the UI have a causality relationship (the Ok button of a form that must be filled completely), there is probably little

reason to split them on different devices. Also the UI can often force the causality when needed (enabling the Ok button only when the form is filled).

For these reasons, EBL restricts its guarantee for causality only for the parts of the UI that are migrated inside the same window. From the application point of view, this means that user events should be piped in a stream that respects the FIFO order. For example, if a user clicks on the buttons 1, then 2, then 3, at the application level, the events 1, 2 and 3 should be raised in that particular order. This result requires to order the events at the display, and to keep this ordering at the proxy side down to the actual action configured for this event.

Another issue is related to the management of events: when running on multiple devices, combination of events become possible that cannot be on a single device. For example, if an application has two separate buttons, when running on a single device, the mouse cursor can pass over only one of them at a time. However when the buttons are split on separate devices, each of these devices have their own mouse cursor, and now the buttons can both have a mouse cursor over them simultaneously. This is a limitation of the transparency provided by EBL: the transparency is at the widget level, and not at the whole UI level. In my experience, this limitation has never been a problem because once again when such widget interdependency occurs, it is logical to migrate them as a whole.

4.3.4 Functional core of the widget

The stores contain the state of the widget. The content can be changed by the application through the API provided by the proxy, internally by the proxy itself (update the time of a clock widget automatically for example), or even by one of the renderers. The functional core of the widget implements the functions that make its state evolve, hence change the content of its stores. Depending on the widget semantics, the functional core can have different level of complexity:

- A simple button widget supports functions for changing its displayed text, color and so forth.
- A complex text widget supports functions for changing characters fonts, search&replace, image insertion, and so forth.

This functional core is typically implemented by the low-level toolkit. However this one runs at the renderer and not at the proxy. Yet the proxy-renderer relationship supposes that the proxy serves as the canonical reference for the full state of the proxy. Consequently **the functional core that evolves the state should also be replicated at the proxy**. This can require a lot of reimplementation work. **EBL offers a technique for avoiding this rewriting by delegating part of the functional core to the renderer**, using an RPC (remote procedure call) like mechanism:

- The proxy provides a method for updating the state of the widget to the application.
- When called, this method submits the request to the renderer.
- The renderer receives the request:
  - It uses the low level toolkit to apply it.
  - It gets back the resulting state.
  - It submits the new state to the proxy.

The proxy ends up receiving the new state, and updates its state. Note that if several renderers are connected, they all receive the request from the proxy. However the proxy only accepts the first answer it receives and ignores the rest. Anyway when accepting the first answer, the new state is broadcasted to all the renderers and they eventually end up synchronized.

This delegation mechanism creates a dependency on the renderer that does not exist otherwise. To guarantee consistency, the delegation is encapsulated in a transaction: if the renderer survives long enough to submit the resulting state to the proxy, the transaction commits. When it is not the case, the transaction is suspended until a new renderer comes in, its state is synchronized with the proxy and then the transaction is tried again. From the application perspective, a call to a method of the proxy may suspend until a renderer survived long enough for the call to be fully processed. In summary, the delegation mechanism provides an interesting trade-off to widget developers:

- Invest more time in the development of the widget, reproducing as much as possible the functional core of the widget at the proxy side. The widget will have better performance (no round-trip necessary between the proxy and the renderer), and offer better behavior in case of network faults (no thread suspended).
- Invest as little time as possible in the development of the widget, using delegation as much as possible to avoid reproducing the functional core completely. The widget will have bad performance (due to round-trips between the proxy and the renderer), and offer bad behavior in case of network faults (suspension).
- Use a mix between these two approaches, balancing the cost of development with the benefit in terms of performance and behavior in case of network faults.

When creating an EBL binding for a new toolkit, it is possible to start with a minimum development effort by using delegation as much as possible, and then add more functionality to the proxies as needed by real world situations. This minimizes greatly the initial development cost for binding EBL to a toolkit, while keeping the possibility to enhance performance and behavior upon network faults later if needed.

Proxy	Renderer

`meth setComplexVariableTo(V)`
`  if {Not {Store askSet(cv V $)}}`
`  then raise exception end`
`  end`
`end`
	`meth askSet(Key V R)`
	`  try`
	`  {Widget configure(Key V)}`
	`  R=true`
	`  catch Error then`
	`  R=false`
	`  end`
	`end`

In this example, the proxy provides a method for setting a particular variable to a value. However the functional part that checks that the value is acceptable for this variable has

not been implemented. Instead the proxy uses the `askSet` method to rely on the renderer. This method is similar to the `set(Key Value)` method with a third parameter that will be eventually bound to `true` or `false` depending on what the renderer reports. Eventually the `askSet` method of the renderer is called: it uses the actual widget to try and use the parameter, and reports the success of the operation.

4.4 The receiving end of a migration

So far we explained the relationship between a proxy and its renderers when they are already running. Proxies are created by the application itself, but we have not addressed who creates the renderers yet. We could have a dedicated generic application just for that purpose: it would open an empty window, receive a migrated widget and display it inside the window. However that seriously limits the possibilities of migration: for example we could not display multiple migrated widgets in a single window. Also it is quite natural to expect that a widget migrated away from a window can later get back into its original place. To achieve maximum flexibility, we would like to be able to migrate any widget into any place of any window. EBL forces the container widgets to use the migration mechanism for all their content, even if the content is created at the same place as the container. The content of the container widgets is managed by a special EBL store: each contained widget receives a unique name, and its value is composed of its universal reference and its placement information inside the container (row and column coordinates for example). For each entry of the content store, a special method is called at the renderer site by the renderer manager. The first parameter of this method is the renderer actually created by using the universal reference, and the second parameter is the placement information of this widget. The method must implement the effective low level toolkit commands to place the renderer according to the placement information.

```
meth importHere(SubWidget PlacementInstructions)
   {Toolkit place(SubWidget PlacementInstructions)}
end
```

This example shows the method that must be implemented by a container widget. It is called automatically by EBL when another widget must be placed inside this one. Symmetrically, the container widget must also implement a method for removing a contained widget:

```
meth remove(SubWidget)
   {Toolkit remove(SubWidget)}
end
```

4.4.1 Container composition

Now that we know how a renderer is created when its content is migrated inside a container, we can consider what happens when the migrated widget is itself a container with some content already migrated inside. For example, let's consider the most common container: the table widget. It splits a rectangular area into rows and columns, each contained widget occupying one or more cells. When the table is migrated, the renderer store synchronizes with the proxy store, and eventually the content of the special store referencing its contained widgets is restored. As a side effect all its contained widgets are then migrated along. As a result, it is possible to dynamically compose user interfaces, by succes-

sively migrating widgets into place. Also, a table can be a target for the migration of widgets from different sites, gathering them together at a single place. This table can in turn be migrated to yet another site: the contained widgets are also migrated along, see Figure 4-3.

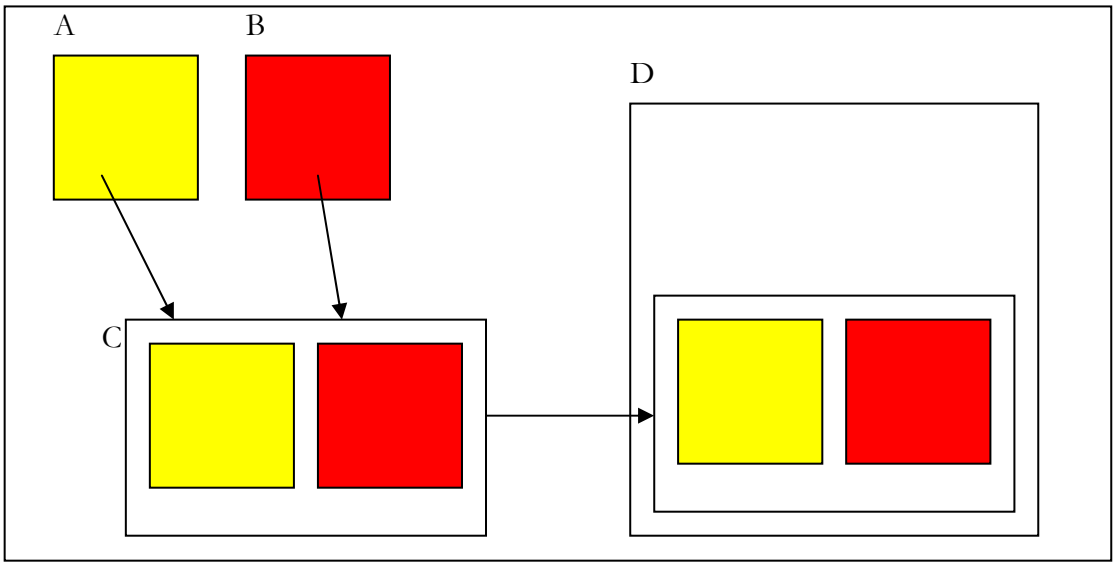


Figure 4-3 Composition of UI by successive migrations

4.4.2 Toplevel widgets

Widgets are migrated inside other container widgets. However we need a toplevel widget: one that is a container, but that is not itself contained inside another one; the root of the tree of contained widgets. EBL provides a way to create a local toplevel widget which is displayed directly at the site that creates it, and cannot be migrated away from this location. Toplevel widgets are containers, and any other widget can be migrated inside them.

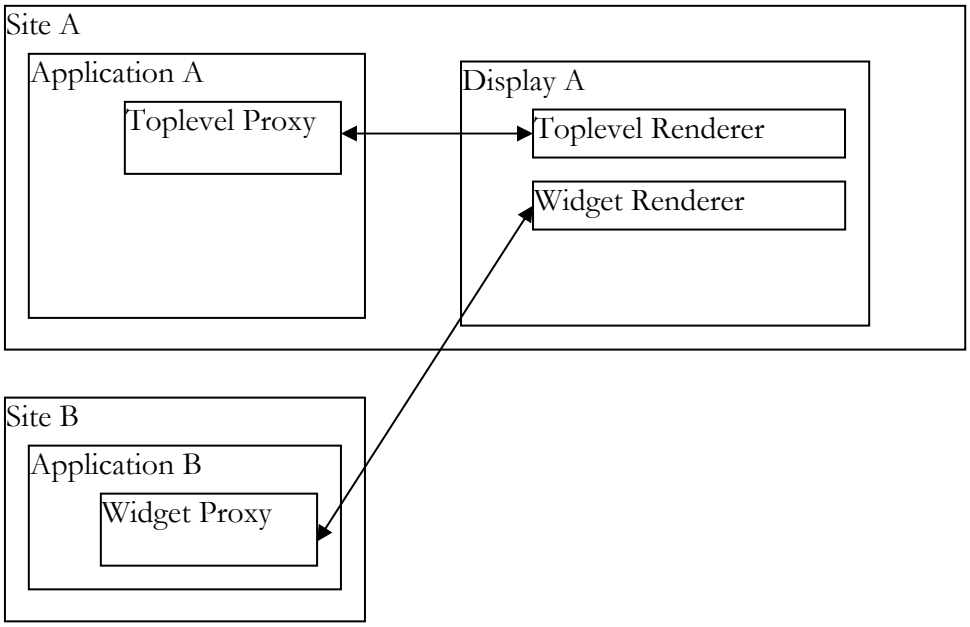


Figure 4-4 Toplevel widget

In Figure 4-4, the application at site A creates a toplevel widget. This widget is displayed directly at the site A, and is not migratable. However, widgets from another site (here B) can be displayed inside this toplevel.

Toplevel widgets also serve other purposes:

- They link to an actual toolkit resource, and provide this resource to their contained sub-widgets.
- They provide an event stream for serializing all events for all widgets they contain in order to keep their causality order *for the events happening in this window only*.

For example, the EBL/Tk binding uses the Tk module [TkMod] for implementing a toplevel window. When a widget is migrated inside this window, the window passes the reference to the Tk module to this widget's renderer: the renderer uses it to create its widget. If this widget is itself a container, when another widget is migrated inside itself, it will also pass the reference to Tk along. In other words, the toplevel widget defines the resource that is used for the actual UI, and the migrated renderers consume this resource for achieving their task. This has very interesting consequences:

- Versioning: each renderer definition is attached a `rendererClass` parameter. The migration of widget A inside container B can succeed if and only if their `rendererClass` parameters are identical; otherwise the migration fails and nothing happens. This `rendererClass` parameter makes sure that the site that provides widgets and the one that imports them are using the same version of the toolkit.
- Heterogeneity: it is possible to define more than one toplevel widgets that use different `rendererClass` values. If other widgets define at least one renderer per `rendererClass` value, then the UI can be migrated (and adapted to the underlying toolkit) between these different toplevel widgets successfully. For example, a universal toolkit could provide a unified version of the table, label, button, radiobox

and checkbox widgets for GTk and Tcl/Tk. The system would allow the transparent migration between a GTk window and a Tcl/Tk window; the correct renderer would be automatically used. Note that the heterogeneity is at the granularity of the toplevel widget; it is not possible to mix different low level toolkits inside the same window. This feature has not been fully implemented though.

4.4.3 Migration trigger

An important aspect of the migration mechanism is its trigger. This trigger is a distributed operation between the application's site and the site effectively receiving the widget's renderer:

- The application site could push the migration to the receiving site. This requires that the application knows a willingly accepting receiving site.
- The receiving site could pull the migration from the application. This requires that the receiving site knows a willingly offering application site.

In both situations, both sites need to get in touch.

One can see this problem as a discovery service problem. The receiving site offers a service for accepting migrated widgets, while the sending site offers a service for offering widgets. There are many different ways of implementing discovery services, each one of them having its advantages and problems. The purpose of this thesis is not to promote one of them over others; instead we assume a discovery service already exists and allows sites to know about each other, and focus on the actual migration trigger instead.

Conveniently Mozart offers extensive support for distributed applications, including a distributed connection mechanism. The distribution support allows sharing language entities among remote sites, with distributed protocols automatically attached to them depending on their types (not all types support distribution though), along with configurable fault detection and recovery. To share an entity between two or more sites, one can either pass its reference on another already shared entity, or use the ticket mechanism. This mechanism prepares a local entity to be shared by other sites, and creates a text string (the ticket) that points to it. This string can be passed by any medium, including a URL on a web server, ftp, by telling people on the phone, by SMS, and so on. Another Mozart site can use this string to get a reference to the entity it points to.

There is another valid interpretation to this mechanism: a ticket is a *capability* granted on the reference of the entity. Any site that has this ticket can use this capability and have the reference to the entity. EBL extends this philosophy to widgets: the proxy of the widget can grant the capability to migrate the widget, in the form of a universal reference which in fact is a ticket. The application can give this capability away to remote sites any way that suits its purpose. Remote sites that receive this capability can use it to pull the widget, i.e. trigger the migration. It is the responsibility of the application to give access to these capabilities wisely. For example it could use a secure web server that requires the user to identify himself. However once a capability has been given away, it cannot be taken back. In particular, the site receiving the capability could pass it along to another site; after all it is just a text string. For the proxy site, the capability would still be valid, and this other site can pull the widget. At first this looks like a security issue, however this is a pertinent property for dynamically migrating widgets: a widget A is sent into a

container widget B of a remote site. B is itself sent into a third remote site. To keep the consistency, the content of the container should be migrated also, and so the widget A is now at the third site too.

One could envision different levels of capability for a widget:

- Full migration capability, where the widget is migrated and the user can interact with it
- Read only version capability, where the widget is migrated, reacts to the application's updates, but doesn't allow the user to interact with it.

For now EBL supports the full transferable capability only, further work is required to add support for less restrictive capabilities. EBL also implements a basic discovery service for publishing and receiving tickets based on IP/Socket number combination. Mozart itself provides a discovery service over local area networks (LANs).

4.5 Adaptation

So far we focused mainly on the migration aspect of EBL. However, we also want support for adaptation at the widget level. This is defined by the capability of a widget to have different representations and user interactions that support the same semantics. For example, a clock widget with an analog representation or a digital representation is said adaptable. Fortunately the proxy-renderer mechanism of EBL provides a straightforward solution for this problem:

- The semantics of the widget is defined by its proxy, which is fixed at the application's site.
- The actual representation and user interaction is defined by its renderer, which is running at the receiving site.

The adaptation is trivially achieved by allowing a single proxy to work with different-but-semantically-compatible renderers.

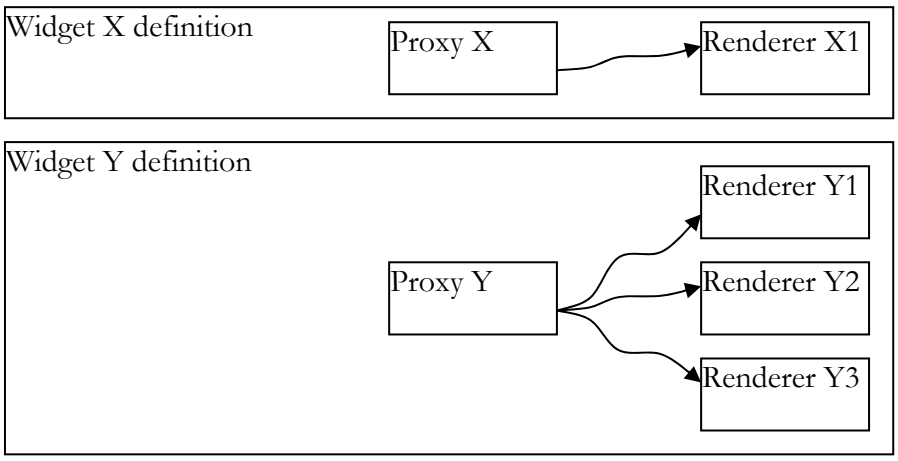


Figure 4-5 Adaptable widget definition example

Figure 4-5 shows two widget definitions. The widget X has only one possible renderer: all instances of X will be displayed using this unique renderer. Consequently the widget X is not adaptable. On the other hand, the widget Y has three possible renderers: the in-

stances of Y will be able to use any of these three renderers, and switch between them dynamically at runtime. Consequently the widget Y is adaptable.

Note that the possible renderers are defined at the proxy side and not at the side where the renderer runs. The renderer is defined as a class which is sent by the proxy to the renderer site during the migration protocol. Consequently it is possible for an application to add new renderer definitions and use them at runtime; the display does not have to be changed or linked to a newer version of the GUI library to use these new definitions.

The action of adapting a widget consists in migrating the widget into the place it is already occupying, but using a different renderer definition. The renderer currently in use is a configuration parameter of the proxy changed by the `setContext` method. The adaptation automatically occurs when this parameter is changed.

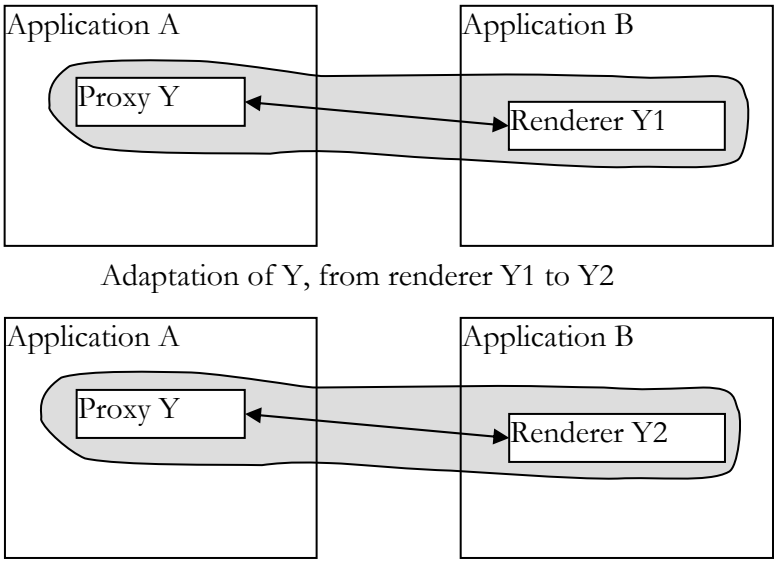


Figure 4-6 Runtime adaptation example

In Figure 4-6, an instance of the widget Y has been created at the application A, and migrated into the application B. At first, the widget was configured to use the renderer Y1 for this widget. At any time, the application A can change the adaptation parameter: consequently, the renderer Y1 is replaced by a new renderer Y2 at the application B. This is achieved by re-migrating the widget into the application B, but this time using the renderer Y2 instead of Y1.

Multiple renderers connected to the same proxy could be adapted independently, however EBL does not implement this functionality yet. Currently all the renderers are adapted similarly.

The possible values for the adaptation parameter depends on each widget, however they must all define at least a default one otherwise the widget can not be displayed at all. EBL provides the support for defining the extra renderers for a widget and also provides the adaptation protocol itself. In summary the concept of stores provided by EBL allows si-

multaneously the migration, adaptation and simple multi-user functionality for graphical user interfaces.

```
UI={Build window(name:window
                 selector(name:selector
                          text:"Pick a number"
                          items:["One" "Two" "Three"]))}
...
{UI.selector setContext(listbox)}
...
{UI.selector setContext(menu)}
...
{UI.selector setContext(default)}
```

The `selector` widget of this example supports three renderers. The `setContext` method switches between them (when and why this happen is unspecified in this example).

4.6 Low level network implementation independence

The proxy-renderer scheme requires a communication mechanism that:

- Allows two kinds of peers (the proxy and the renderer(s)), pretty much like a client-server architecture.
- Provides a fault reporting mechanism and explicit removal of peers.
- Supports one-to-one communication so that renderers can notify the proxy of user events for example.
- Supports one-to-many communication so that the proxy can send state update messages to all the renderers simultaneously for example.

To maximize flexibility, EBL completely isolates this functionality; however it also relies on another assumption about its properties:

- Communications between two peers are FIFO ordered. This order is true for one-to-one messages, for one-to-many messages, but also for mixes between one-to-one and one-to-many. In other words, if peer A sends messages M1 then M2, and peer B receives them: B will receive them in the M1->M2 order no matter if M1 and M2 were sent as one-to-one or one-to-many. Note that there is no order for messages between different targets, and consequently there is no global causality. If A sends M1 to B and M2 to C, and B upon reception of M1 sends M3 to C, there is no guarantee that M2 will arrive before M3 at C.

This assumption allows relying on the ordering of the messages when implementing the network protocols between the peers.

This low level network library can be almost trivially implemented using a client-server approach. However other possible implementations exist, for example using a peer to peer network instead of a client-server one. EBL provides a default client-server implementation, but another one can be provided and used if needed. All EBL high level protocols between distributed components rely on this low level network library.

4.7 Security issues

There are security issues at different levels:

- Trust of the network. On trusted networks (typically private local area networks and virtual private networks), all devices and services on the network are considered trustable. In this scenario, there is no need to protect migration capabilities and connections between EBL peers. On networks that are not trustable (typically the Internet), the migration capabilities could be captured and abused by attackers. To prevent this, the discovery service used by the EBL peers should use encryption to pass the capabilities securely. Similarly the communication between the proxies and their renderers should be encrypted to prevent man in the middle attacks. It is not the purpose of EBL to provide a discovery service, be it secure or not. Also, encrypted communications should be done at lower communication levels, the Mozart distribution layer in this case. For these reasons EBL does not provide security at the network level. Note that multiple devices physically close to each other typically run on a private trusted LAN, and this scenario is covered by EBL. In the case of the Internet, using a Virtual Private Network, or SSH tunneling techniques can secure EBL applications.
- Trust of the good usage of the migration capability. Once an application gives a migration capability away, it loses control over 1) what sites have access to this capability and so where the UI will be displayed and 2) when and how many times will the capability be used to trigger a migration. Note that this enables migrated containers to restore their content as the migration capabilities of their content is passed to their new renderer site. However this property is not acceptable from a security point of view. To fix this problem, proper control over the capability should be given to the application: revocation (the migration capability does not function anymore) and/or restriction (the migration capability works only if triggered by specified site(s)). These controls have not been implemented by EBL so far.
- Trust of the remote proxy. When an application triggers a migration, it receives the renderer definition from the proxy and then executes it using local resources. Executing remote code using local resources is always a huge security threat. There are two ways to mitigate this threat:
    - Sandboxing: the code is executed with very restricted access to the local machine, so that it cannot possibly do any harm. With EBL the local resources available to the renderers at runtime are defined by the renderer environment. When creating a toolkit binding with EBL, one can make sure this environment is as restricted as possible. However a reference to the low level toolkit is typically required for the renderer to function, and low level toolkits often permit a lot more than we would like, including an access to the file system.
    - Trusted computing: an authority certifies that the remote proxy is trustable from a security perspective. The renderer site accepts a migration only after making sure the proxy is trustable. EBL does not implement this level of security directly; EBL bindings should use an extra certification mechanism to use this security model.
- Trust of the remote renderer. When a UI is migrated away, the proxy sends the renderer definition to the remote site, and then receives an active connection with the newly created renderer. However there is no guarantee that it is indeed

the renderer definition that is used by the remote site. It is a situation similar to web applications: it is trivially simple to change a web page so that it sends arbitrary data to the server. In particular all the type checks of the web page can be bypassed, and the server receives invalid data. For this reason, data input validation should always be done at the server side even if they are also done at the client side. The EBL store implements this scheme: updates originating from the renderers are validated by the proxy before being applied. This security is still quite weak, for example it does not prevent a denial of service attack where a false renderer floods its proxy with updates so as to make the UI unusable. For stronger security, we need trusted computing like in the previous bullet.

- Trust at the EBL binding high level. Let's suppose that we have EBL, a secure EBL toolkit binding that uses a secure discovery service, encrypted data channels between EBL peers, and a trusted platform to ensure that proxies and renderers are all trustable. Let's consider the login application A whose UI is migrated to the remote application B. Even with EBL, B could introspect its UI to steal the login information. Such introspection is not allowed by EBL: the proxy of a container only knows the migration capabilities and placement information of its content, and is completely unable to access its contained widgets. However if B has an access to the low level toolkit (which is typically the case), and the low level toolkit offers introspection (which is also typically the case), B can use it to access the displayed widgets. To fully solve this security issue, we also need trusted computing at the application level.

Chapter 5 Implementation

Chapter 3 and 4 present *what* the problem EBL solves is. Chapter 5 now focuses on *how* this problem is solved. We detail the most important technical aspects of EBL. EBL is a toolkit agnostic middleware written for the Mozart Programming System. EBL has to be linked to an actual toolkit, resulting in another toolkit supporting the functionality of the original one, plus:

- Full support for dynamic transparent migration, using the migration capability approach.
- Full support for dynamic transparent adaptation, using the configuration approach.
- Support for simple multi-user interfaces.
- Support for mixed declarative/imperative approach to GUI programming.

In particular, EBL has been linked to Tcl/Tk to create the EBL/Tk graphical toolkit. Examples of use of EBL/Tk are available in chapter 5.

Section 5.1 defines what the distributed infrastructure EBL is built on. Section 5.2 gives an overview of the distributed architecture, at the granularity of complete EBL applications. Section 5.3 focuses on migration capabilities, and contributes their routing through the different distributed entities of EBL applications. Section 5.4 gives detailed information of the distributed architecture, this time at the granularity of EBL widgets. Section 5.5 details the architecture of the receiving end of a migrated UI. Section 5.6 describes the low level network component of EBL. Section 5.7 describes the most important network protocols. And finally section 5.8 describes a set of recipes to follow when binding EBL to a graphical toolkit.

5.1 Distribution overview

All distributed operations are implemented by an asynchronous FIFO message passing system by using the port data type of Mozart. A port is composed of

- A receiving end which is stationary (future versions of Mozart may allow distribution for this type) and cannot be closed.
- A sending end which can be distributed among multiples sites so that they can all send messages on the same port.

Network faults are detected by the distribution layer of Mozart, which gives us an oracle approximating the current status of the network link. EBL is configured to manage these faults lazily, that is only when an actual network operation is performed. The consequence of a network fault is always a disconnection of the distributed part of the widget (the renderers). Because of the laziness, when a network fault occurs between an application and its remote UI, only when the remote UI tries to interact with the application (usually in response to a user event or an application update), will it disconnect from the application, resulting in its disappearance. This laziness is advantageous though, because transient network errors do not necessarily result in a disconnection, only if the UI is in use will it be the case. Also Mozart provides the *transparent distribution* of some of its data type: a distributed network protocol is automatically attached to the entity, and it can be used remotely as if it were a local entity. For EBL this functionality is limited to values,

which are transparently copied over the network when a reference is passed from a site to another one. As Mozart supports high order programming, procedures and class definitions are values, and as such they can be transparently passed from one site to another.

5.2 Runtime architecture

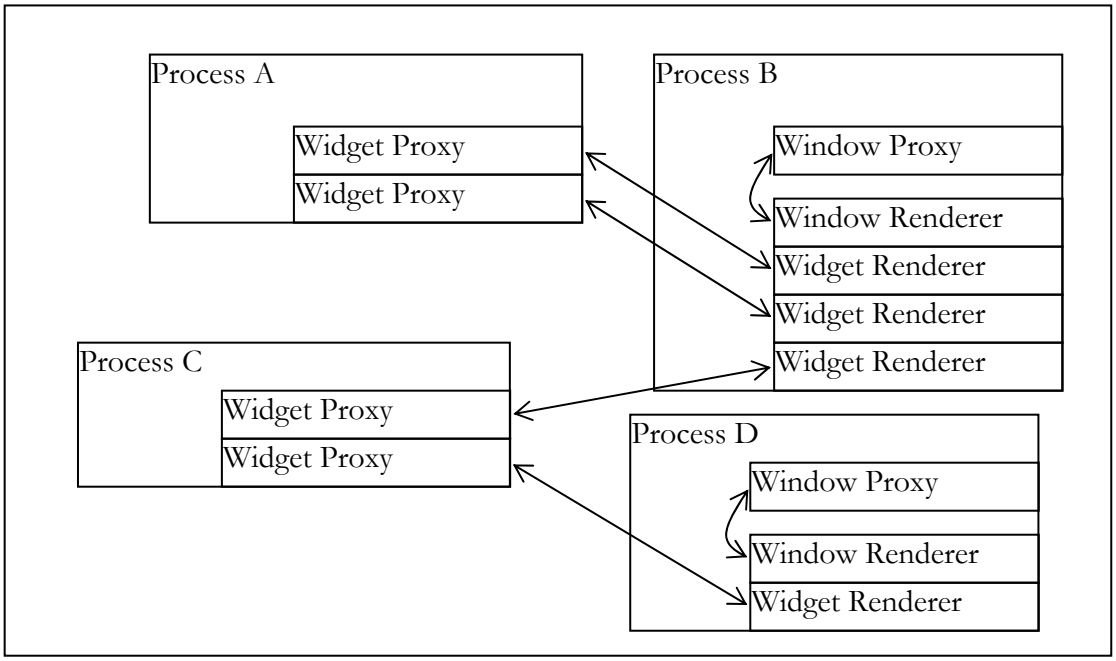


Figure 5-1 General running architecture

At runtime, each widget is split in two parts: the stationary part that stays at the creator site (the proxy), and the migratable part that is run at a remote site to actually display the widget (the renderer). A notable exception is the toplevel window widget: the migratable part stays at the creator site; it is created immediately along with the proxy and cannot migrate away. The renderer part of a widget needs a window to be displayed inside, so it can only run at a site were a window proxy is running. Note that the content of a window is a separate widget that can be migrated away. In other words, toplevel windows provide the physical hook where widgets can be displayed. Also Note that there is no dependency on an external server for this architecture to work. Widget proxies act as servers for their renderers. This is based on the distribution layer of Mozart.

5.3 Migration capabilities

In order for the sites to get in touch and start working together, we need a way of referencing the widgets over the Internet. This is achieved by the migration capabilities that serve 1) as references and 2) as authority for pulling the widget. The proxy of each widget (except the toplevel window that is not migratable) has a migration capability that consists in a combination of bytes that encapsulates all the information required to connect to the widget through the Internet. Migration capabilities are given to the proxies of container widgets, triggering the creation of a renderer for the proxy corresponding to the capability inside the renderer of the container widget.

5.3.1 Trajectory of a universal reference

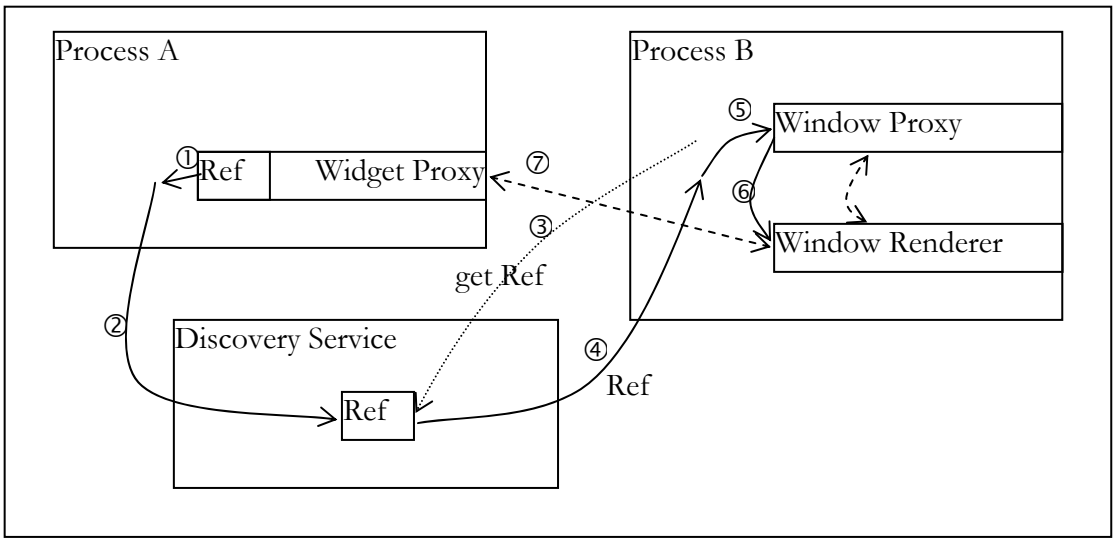


Figure 5-2 Universal reference trajectory

Dashed arrows are actual connections, plain arrows are the trajectory of the universal reference

The universal reference is a capability the creator of the widget can give to a remote site. Typically, an intermediate discovery service allows the sites to exchange these values. Figure 5-2 is a typical scenario:

1. Process A running on computer X creates a widget and asks for its migration capability.
2. Process A stores this capability at the discovery service.
3. Process B running on computer Y asks the discovery service for the capability of the widget it wants to display.
4. Process B receives the answer
5. Process B passes it to the proxy of a container widget, here a window.
6. The proxy forwards the capability to its renderer.
7. And lastly the renderer opens a connection with the proxy corresponding to the capability. In section 4.7.1 we show how this connection is used for creating a new renderer for this proxy.

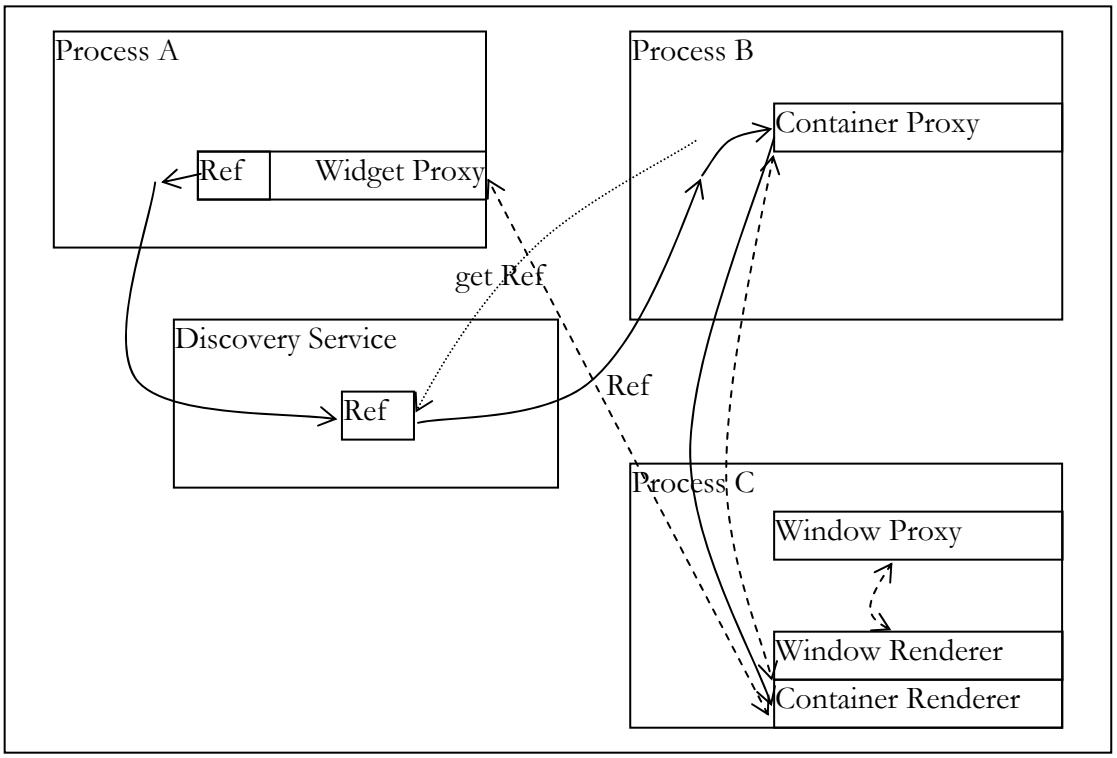


Figure 5-3 Complex trajectory

Figure 5-3 displays a more complex scenario where the process B migrates the widget inside a container that is currently displayed at the process C. The migration capability follows the same route as in Figure 5-2, except that the container proxy forwards the capability to its renderer at process C and not locally anymore.

5.3.2 Discovery service

The discovery service can be implemented in a number of different ways:

- By phone with the operator in front of the offering sites dictating the content of the reference to the operator in front of the receiving site.
- By email sent by the offering site and read by the receiving site.
- By using an intermediate web site where the offering site puts the reference and the receiving site gets back.
- By using an FTP server.
- The offering site can start a server on a specific socket which serves the capabilities. The receiving site knows the IP and socket numbers to connect to this server. EBL provides a simple implementation of this service.
- On local area networks, the offering applications can listen to specific broadcasts, and the receiving site broadcast its request. This is implemented by the Discovery module of Mozart.
- The offering and receiving sites can connect to a peer to peer network, and use it for registering and discovering capabilities. For example, we can use the P2PS module [P2PS] to register capabilities to a distributed hash table (DHT), and allow looking up for them.

As we see, the discovery service is orthogonal to the problems solved in this thesis, and there are many possibilities depending on the actual application needs, so EBL does not try very hard to provide a proper one.

5.4 Distributed widget architecture

Widgets are split in two parts: the proxy that always stay at the application site, and the renderer that is dynamically migrated between sites. EBL provides a high level service that is well fitted for synchronizing these parts together.

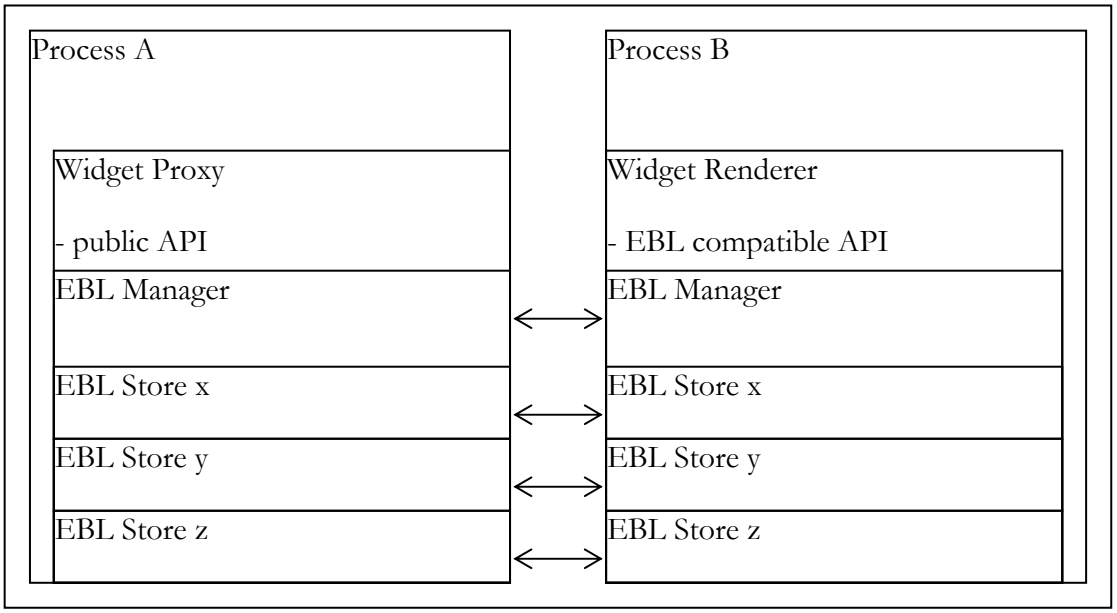


Figure 5-4 Widget architecture

5.4.1.a Specifics of the widget proxy

Widget proxies implement the application side of the widgets, as classes. These classes are public to the application, and the visibility of their methods directly defines what the application can use or not. For that reason EBL does not impose anything regarding the definition of the proxies, in particular it does not provide a master proxy class all proxy classes should inherit from. Instead the proxy functionality of EBL is accessed through the EBL proxy manager; widget proxies should create an instance of this class, and keep a reference to it. The proxy manager provides different services to the proxy (see Annex A for the complete reference):

- Migration capability management
- Support for container widgets
- Support for migration, adaptation, and simple multi-user functionality.
- Support for toplevel widgets
- Communication medium with the renderers, including direct message passing, and simple transactional mechanism.
- Destruction of the widget.
- Creation of stores that manage the state of the widget and the user events.

In particular, the manager is able to create an arbitrary number of stores. A store is a distributed dictionary that is automatically synchronized with all connected peers, à la Linda [Leler90]. Using dictionaries as the basic communication medium between the proxies and the renderers has several benefits: first, there is independence on the number of connected peers, which allows working when zero, one or more renderers are connected to the proxy. Second, it corresponds to the needs of widgets that mainly associate values to specific parameters. When a renderer is created, its stores contain the current known state at the proxy, and it has to configure the actual widget accordingly. Further updates to the stores trigger dedicated methods at the store, so that it is notified of the updates and is able to reflect them at the actual widget. Stores are highly configurable: the allowed keys can be restricted to a known set, each key can be strongly typed, and the serialization of the value between the proxy and the renderer(s) can also be configured. Further, the store provides transactional operations to update keys, were the renderer becomes the actual responsible for updating the value. This is particularly useful when the proxy does not implement the complete functional core of the widget: it can rely on the renderer to apply state updates and get back the resulting state. Transactions automatically succeed if the renderer survives long enough to send back the response to the update, otherwise the transaction suspends until a new renderer arrives, and it is submitted there back again. As a result, transactions never fail, but they can be suspended forever. When a renderer is connected to the proxy, all transactions eventually succeed, granted the renderer survives for long enough. Pending transactions are kept in order with the known state of the widget, so as to make sure that the state of the widget is always the same when a particular transaction is attempted. Finally stores also manage the user events, like responding to a mouse click. The complete reference is at the Annex A.

5.4.1.b Specifics of the widget renderer

Widget renderers implement the migrated side of the widget, the one that has an actual physical incarnation. EBL creates instances of the renderers as required by a migration or an adaptation. Renderers are also defined as classes. EBL forces the renderer to follow an interface that implements a specific set of methods. These methods are called by EBL automatically when needed. Some of these methods are notification of updates of the store, so that the renderer can reflect them on the actual widget. Some of these methods are for container widgets, and required to create the environment of the children widgets. And finally some methods are related to the user events, like responding to a mouse click. In particular, the init method receives the renderer manager as parameter. Contrarily to the proxy which has to create it explicitly, the renderer receives an already configured and fully working manager that is already connected to the proxy. In particular, the renderer can ask the manager to have access to the stores. These stores already contain the current state as known by the proxy at the time of the migration. Lastly, the renderer manager provides an access to the environment object (typically contains the reference to the local toolkit, see below for more information). The complete reference is at the Annex A.

5.5 Display site architecture overview

The renderer part of a widget is a distributed agent that links to the resource of the host site while collaborating with its proxy:

- The renderer manager provides a way to synchronize with the proxy, by means of the stores.
- The renderer manager provides an access to the local resources, by means of a shared state with its container that we call *migration environment*.

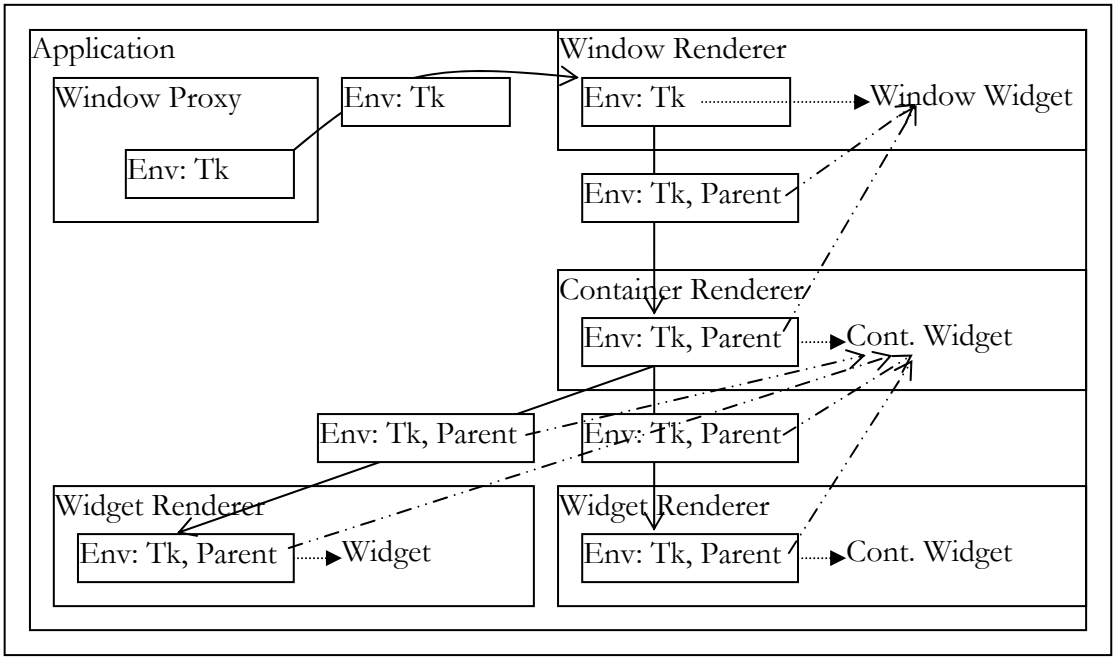


Figure 5-5 Runtime environment architecture

Plain arrows are the trajectory of the environments, finely dashed arrows points to widgets actually created from a local environment, large dashed arrows points to the actual widget the parent is referring to. Tk is the toolkit in use in this example.

The toplevel proxy creates a migration environment from scratch, typically putting a reference to the actual toolkit to use, and gives it to its local renderer. Each container renderers passes the environment down to their contained widgets, adding a reference to themselves as the parent of the contained widget. Each renderer receives the environment from their manager, and uses its content to gain an access to the actual toolkit, and their parent widget. Using that resource, they are able to create the actual widget to put inside the parent widget.

5.6 Low level network component

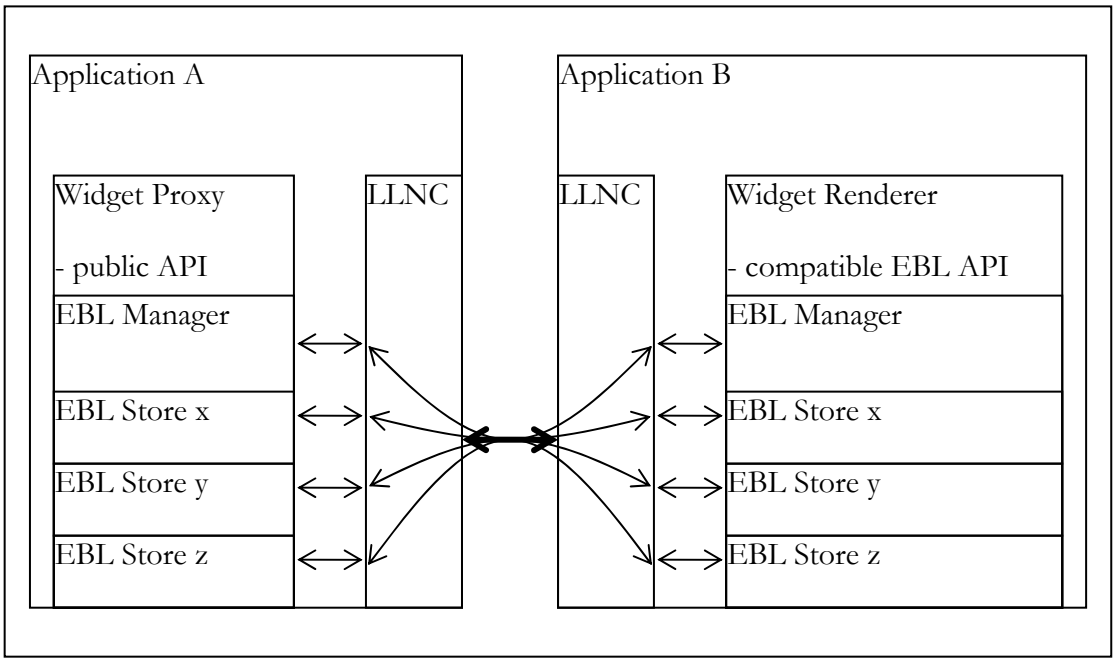


Figure 5-6 Runtime environment architecture

All network protocols of EBL (store protocols, migration protocols ...) are built on top of a single low level network component (LLNC in Figure 5-6). A default client-server implementation is provided; others implementation could be used, for example to run on a peer to peer network.

5.7 Protocols

This section presents the most important protocols used by EBL.

5.7.1 Migration protocol

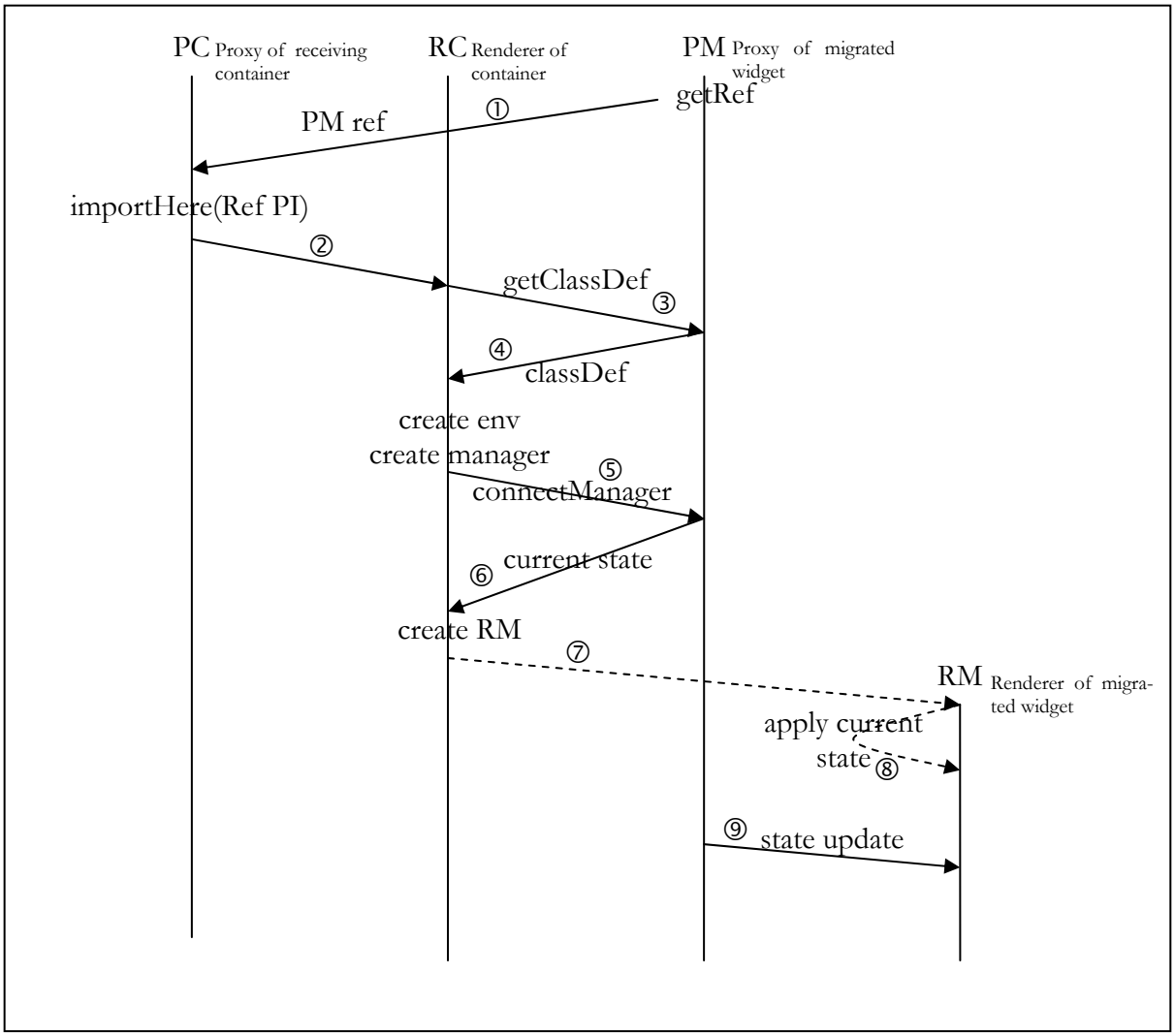


Figure 5-7 Migration protocol

The migration protocol is a negotiation between the proxy of the receiving container (PC), the proxy of the migrated widget (PM), the renderer of the container (RC) and the renderer of the migrated widget (RM). First, the migration capability of PM has to be given to PC somehow ①. The migration starts at PC, by using the `importHere` method of its manager using the reference given by PM (the second PI parameter is further placement instructions for example the row/column coordinates of a table container). This method stores this new child; stored children are automatically given to RC ② (either at the child's creation or at the RC creation). RC connects to PM using the reference contained in the capability ③, and in returns PM sends the class definition of the widget renderer ④. RC creates an environment and then asks EBL to create RM using the class definition just received. If RC fails to create RM (due to PM not responding, or an error while creating RM), RC tells PC to drop this particular child. To create RM, EBL first creates its manager, connects back to PM ⑤, gets the actual state of all stores ⑥, and then creates the RM object with the manager as parameter ⑦. The initialization of RM

should create the actual widget, and update its state according to the current content of the store ⑧ (parameters & event bindings). Once initialized, EBL automatically calls the methods of RM according to the updates of the store ⑨. If the migrated widget is itself a container, the information necessary to restore its content is in the store it receives from PM, and RM reacts to it like RC after step ②. As a result its content is also migrated along.

5.7.1.a Negotiation phase

The step ③ of the protocol above asks a class definition for the renderer and is returned the one currently selected by PM. Indeed there can be different renderers possible for this widget, and the process running PM has selected one of them in particular (using the `setContext` method). However we can extend this protocol further by adding a negotiation phase where RC uses the knowledge of its own available resources (keyboard/mouse presence, screen size...) to hint PM so that it is able to override the current selection for the renderer with another one that is more fit to the device. The scheme would require:

- A model for describing the platform running the UI.
- Introspection capabilities for renderers determining their level of compatibility with specific platforms.

Another option is for RC to use its own renderer definition, ignoring the one sent by PM. This may result in an incorrect renderer that is unable to behave correctly with its proxy, however this would open up the possibility of having a target device that adapts the UIs it receives even if the process running those UIs do not know how to adapt them !

The current implementation of EBL is limited to the protocol of the Figure 5-7 though.

5.7.1.b Fault tolerance

Network failures can happen at any time, between any of the sites:

- Between PC and PM. There is no direct connection between these two sites: the capability of PM can be brought to PC by a third site.
- Between PC and RC. If message ② cannot be sent be cause of a network failure or because there is currently no RC, then the migration cannot be executed. Nevertheless, the migration instruction is now part of the store of the widget. When a new RC comes in, it will then proceed with the migration of PM. As a result, there may be an arbitrary time between the application command to migrate a widget, and when this command is really executed. If message ② was sent, and there is a network failure between PC and RC then RC eventually disappears. This can happen while the migration protocol is still running, or afterwards. In all cases, the disappearance of RC will result in a disconnection with either PM or with RM. In both situations the migration of RM is cancelled, and it is destroyed if it exists.
- Between RC and PM. The only time this network connection matters is between messages ③ and ⑥. If there is a network failure there, then the migration of PM is aborted. Also RC removes PM from the migration store shared with PC, so that PM is no more considered as a contained widget of PC.
- Between PM and RM. This is the same as between PC and RC, see above.

5.7.2 Adaptation protocol

With EBL an adaptation is the migration of a widget into the place it currently occupies, this time using a different renderer. The migration protocol is described at the point above. However, a protocol is still needed to trigger this re-migration.

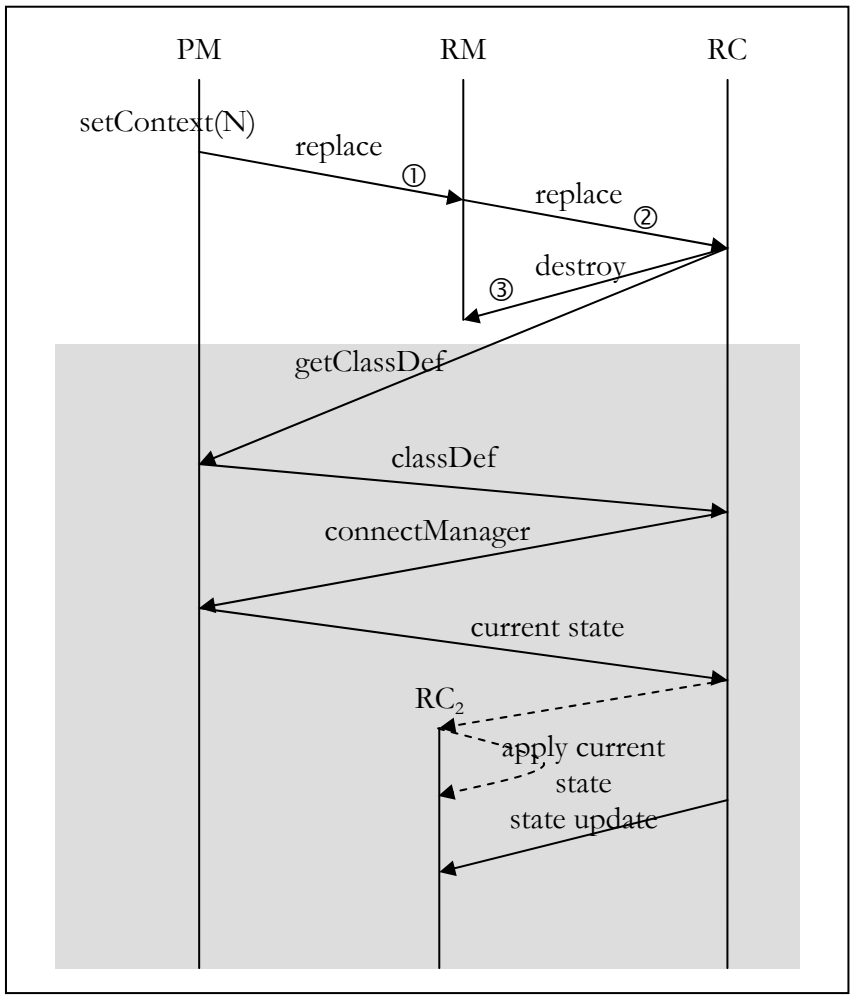


Figure 5-8 Adaptation protocol

When no renderer is connected, the adaptation trivially consists in storing the new renderer to use for the next migration, which is served at step ③ of Figure 5-6. When renderer(s) are connected, the proxy must make sure they are replaced by the new one. First PM sends a replace message to RM in reaction to the change of renderer ①. RM cannot replace itself directly, it must contact its container widget and forward the replace request ②. RC reacts by destroying the current RM ③, and starting the whole migration process for RM again (gray area). When PM is asked for the class definition of RM, this time it sends the new one. As a result, the old renderer is replaced by the new one.

5.7.3 Single user and simple multi-user variations

The ability to have one or more renderers concurrently connected to a single proxy is configured at the proxy level. This is controlled when responding to the `connectManager` request during migration ⑤ in Figure 5-7. When a single renderer is concurrently allowed, PM first disconnects its currently connected renderer(s) (if any), and then responds to the message. When multiple renderers are allowed concurrently, PM simply accepts the new renderer, leaving the previously connected ones alone. The simple multi-user functionality is quite limited. For example all renderers have equal rights in updating the widget state, or responding to a proxy delegation. However the nature of the collaboration is so that all the stores ends up synchronized to the proxy content, so all the renderers end up displaying the same information. If users are using different renderers concurrently, the management of the consistency between their actions depends on the way the renderer-proxy interaction is implemented, which is let to the widget developer. A current limitation of EBL is that there is a single current active renderer class, which means that a widget with multiple renderers cannot have them adapted differently. This is an implementation limitation that could be overcome.

5.7.4 User event management

User events are the events related to the running user interface. Widgets are normally independent agents; changing the color of a widget does not interfere with the state of other widgets. However in the case of user events, one expects the causality of the events to be kept. If a user clicks successively on button 1, button 2, and then button 3, the application listening to these events expects them to arrive in this order. The user events introduce a dependency between the widgets. The causality of the events **happening in a single EBL window** (UIs split over different windows/devices do not respect the causality between the windows/devices) is kept by serializing them at the renderer side before sending them to the proxy, and by keeping this serialization at the proxy up to the actual triggering of the action configured at the application.

Mozart provides a data type that supports a FIFO stream behavior: ports. A single stream is associated to a port that serializes all the messages sent on it.

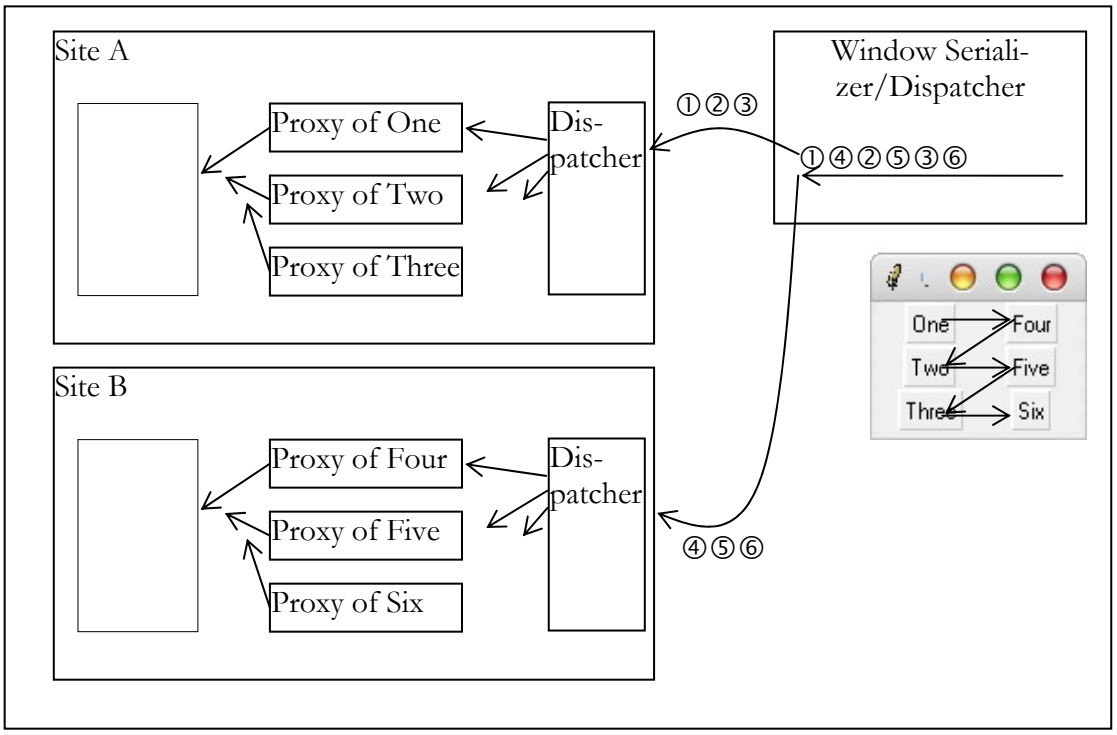


Figure 5-9 Events ordering

When an application configures an action to be executed in reaction of an event, the proxy of the widget uses a store to associate the action with the event, and possibly further instructions concerning the parameters returned with the event like the coordinate of the mouse for example. The synchronized store at the renderer executes a method to bind the event on the actual widget. When the user triggers the event, it is encapsulated with its parameters in a single message that is sent to the toplevel widget. The toplevel widget has a single thread for receiving these messages (hence, it serializes them all), and sends them to the proxy corresponding to the message. This message is received on the event port associated to the proxy. A single thread reads all the messages from the stream of this port, to execute the associated action. When several widgets need to keep the causal ordering between them, they have to be configured to use the same event port.

By default all widgets constructed out of a single declarative data structure will use the same event port; consequently if these widgets are displayed in a single window, the causality of the events is respected down to the application level.

5.7.5 Stores

The main communication medium between the proxy and its renderer(s) is the store. A single widget can have an arbitrary number of named stores concurrently. Methods are provided for reading and writing to these stores (see annex A for a complete reference):

- `set(I K V)`: sets the key `K` of store `I` to `V`
- `get(I K ?V)`: returns current value `V` in the store `I` for the key `K`

Each time the store is updated, a method is triggered at the renderer side. For that reason the class defining a renderer must respect a specific API. For example, the renderer class

must implement the `set(I K V)` method. This method is automatically called by the renderer manager in reaction to an update of the key `K` of the store `I` with the new value of `V`. This method should implement the update of the actual widget corresponding to this update of the store. In some sense, this is a remote method invocation (RMI) mechanism, with proper storage of the data at the proxy. Another interpretation is that the stores are repositories of the state widget at the proxy, and as a side effect trigger methods at the renderer so that it can reflect the state changes to the actual widget.

In summary the renderer manager triggers methods of the renderer object as needed by the store updates. To simplify concurrency issues, the renderer manager uses a single thread for invoking methods, which are implicitly serialized.

5.7.6 Delegation protocols

EBL allows the proxy to delegate (part of) its functional core to the renderer. As the renderer is the place where the real toolkit is, and since the toolkit often already implements the functional core of the widget, this is a convenient way of avoiding re-implementing that part at the proxy. The delegation is at two different levels of granularity:

- At the whole widget level, by means of the `ask(Q R)` method of the proxy manager. `Q` is the question submitted to the renderer, and `R` is eventually bound to a response to this question. On the other side, the renderer manager automatically invokes the `ask(Q R)` method of the renderer, and expects this method to eventually bind `R` to the response of `Q`. Once bound, the response is routed back to the proxy.
- At the key level of each store, by means of:
  - The `remoteGet(I K R)` method. `I` is the name of the store, `K` is the name of the value, and `R` is eventually bound by the value returned by the renderer for this key.
  - The `remoteSet(I K V R)` method. `I` is the name of the store, `K` is the name of the value, `V` is the new value proposed by the proxy itself for this key, and `R` is eventually bound to true if the renderer confirms it accepts this value (typically by checking the actual toolkit accepts it), or false otherwise.

Delegation operations are implemented by transactions:

- If a renderer and the proxy survive long enough for the response to get back to the proxy, then the transaction commits. For store transactions, the value update becomes official, ie enters the store. As a side effect if there is more than one renderer, they all receive the updated value.
- If a renderer does not survive long enough for the transaction to complete, then the transaction is suspended until a new renderer comes in. When this happens, first the state of the renderer is synchronized with the one of the proxy, and then all pending transactions are automatically retried in order.
- If the proxy does not survive long enough for the transaction to complete, then the widget itself disappears, and the transaction is lost forever as there is no store anymore to apply it to.

As stated in 3.4.3, the keys of a store are independent of each other. The store maintains the consistency of each individual key, but not the consistency of multiple key updates. However it is trivially simple to achieve this effect by making sure it is the proxy that makes the multiple updates: either it survives all the updates, and then they will be eventually applied also at the renderers, or it disappears in the process but then the whole widget and its store also disappear. For the delegation mechanism, this translates into using the key granularity for independent key updates, and the whole widget granularity for multiple updates. We did not refine this granularity further as the need did not arise for it. Consequently EBL maintains the consistency of the store by the following strategy:

- At the whole widget level the transactions are FIFO ordered. As EBL uses a single thread at the renderer side, it means that `ask(Q R)` methods are called successively in the order of the transactions.
- Each key has its own transaction lock that serializes the transactions in the FIFO order, while transactions on different keys are completely independent. As EBL uses a single thread at the renderer side, it means that all the `remoteGet/remoteSet` methods will be submitted to the renderer successively in the order of the transactions.

It is up to the developer to make sure that the key granularity is used correctly, i.e. that the key state is independent from the rest of the store. Under this assumption, the strategy makes sure that the state of the store over which the transactions are applied stay coherent in case of a retry. Indeed in this situation, the stores of the new renderer are first set to the current state of the proxy (i.e. last known state before the pending transactions), and then the transactions are applied in order.

As a result, the proxy is not dependent on the survival of the renderer: it maintains the complete state of the widget plus the pending transactions. As a result the renderers can be disconnected at any time with no ill effect on the proxy. This ensures network fault tolerance.

5.8 EBL toolkit binding recipes

To offer interesting functionality to the applications, EBL has to be interfaced to an actual graphical toolkit, resulting in a Mozart binding for this toolkit plus the EBL enabled functionality. The previous sections introduced separately the different artifacts provided by EBL for supporting the migration, adaptation, and simple multi-user functionalities. This section will now show how to use them together, and details the actual binding of a toolkit with EBL, as a set of recipes depending on the kind of widgets to interface.

5.8.1 Simple widget

A simple widget is a widget that corresponds to (1) an actual toolkit widget, and (2) is composed of a single graphical entity to configure and interact with. Buttons, labels, radio buttons, and checkboxes are examples of simple widgets. Because of (2), a single store is enough for all configuration parameters, and all event bindings.

5.8.1.a Proxy

```
class ProxyOfMyWidget
   feat
      widgetName:myWidget
      Manager
      Store
   meth init
       self.Manager={NewEBLProxyManager self.widgetName}
       self.Store={self.Manager getStore(default $)}
       configuration of the store
   end

   functional core of the widget

   meth destroy
       {self.Manager destroy}
   end
   meth getRef($)
       {self.Manager getRef($)}
   end
   meth setContext(C)
       {self.Manager setContext(C)}
   end

end
```

A class defines the proxy side of the widget. Its API is not imposed by EBL and is completely free. To work with EBL, the proxy must create an EBLProxyManager, which controls the distributed widget. This manager gives an access to the stores of this widget.

The stores are highly configurable:

- Specification of the accepted keys, and their type:

  ```
  {self.Store setParametersType(t(key1:type1 ... keyN:typeN))}
  ```

  Only the keys `key1`...`keyN` will be accepted by the store. The special key `'...'` can be specified as a wildcard for accepting any other keys:

  ```
  {self.Store setParametersType(t('...':typeX))}
  ```

- Specification of the type verification functions, and their error message:

  ```
  {self.Store setTypeChecker(t(type1:CheckFun1#ErrorString1 ...))}
  ```

  `CheckFunX` is a unary function that returns a Boolean telling if the parameter is of the valid type. `ErrorStringX` is the string displayed in the error message in case of a type error.

- Specification of default values for the parameters:

  ```
  {self.Store setDefaults(t(key1:Val1 ... keyN:ValN))}
  ```

- Specification of automatic translations before putting or getting a value in the store, at the proxy and at the renderer side:

```
{self.Store setProxyMarshaller(t(type1:m(u2s:Fun1a s2u:Fun1b) ...))}
{self.Store setRenderMarshaller(t(type1:m(u2s:Fun1a s2u:Fun1b) ...))}
```

Values are translated by the function `u2s` (user to store) when putting them into the store, and translated by the function `s2u` (store to user) when getting them from the store.

The proxy widget should also provide an interface for destroying the widget, getting its migration capability, and changing its renderer. The methods simply forward the request to the proxy manager.

Typically the functional core of the widget provides an API for 1) changing/accessing its parameters and 2) listen to user events:

```
meth set(K V)
   {self.Store set(K V)}
end
meth get(K V)
   {self.Store get(K V)}
end
meth bind(…)
   Event={self.Store createEvent(…)}
in
   {self.Store bind(Event)}
end
```

The methods simply apply the request to the store. In this example, we rely completely on the configuration of the store for checking the validity of the request. If it is not possible to configure the store completely, for example because the checking of the validity of a parameter is too complex, then we can use the delegation mechanism instead:

```
meth set(K V)
   if {Not {self.Store askSet(K V)}} then
      raise exception end
   end
end
meth bind(…)
   Event={self.Store createEvent(…)}
in
   if {Not {self.Store askBind(Event $)}} then
      raise exception end
   end
end
```

The `askSet` and `askBind` methods submit the requests to the renderer, and eventually returns if it agrees on them or not. They use the transactional protocol described in section 4.7.6.

5.8.1.b Renderer

The renderer is defined by a second class. This class has to follow a template imposed by EBL.

```
class RendererOfMyWidget
   feat handle manager store env toolkit parent eventPort
```

```
   meth init(M)
      self.manager=M
      self.store={M getStore(default $)}
      configuration of the store
      self.env={M getEnv($)}
      self.toolkit={self.env get(toolkit $)}
      self.parent={self.env get(parent $)}
      self.eventPort=(self.parent).eventPort
      self.handle=self.toolkit command for creating the actual widget
      configuration of the initial state of the widget
   end
   meth set(I K V)
      toolkit command for changing the parameter K of the store I to V
   end
   meth remoteSet(I K V ?R)
      toolkit command for trying to change the parameter K of the store I
      to V, and return the success of the operation in R
   end
   meth remoteGet(I K ?R)
      toolkit command for returning the current value of the parameter K
      of the store I into R
   end
   meth bind(I Event P)
      toolkit command for configuring the widget to listen to the event
      Event of the store I and trigger P
   end
   meth askBind(I Event P ?R)
      toolkit command for trying to configure the widget to listen to the
      event Event of the store I and trigger P, and return the success of
      the operation in R
   end
   meth ask(Q ?R)
      respond to the question Q in R
   end
   meth send(M)
      apply the message M
   end
   meth destroy
      toolkit command for destroying the widget
   end
end
```

The init method receives the RendererManager as parameter. Similarly to the ProxyManager, the RendererManager gives an access to the stores of the widget. Stores are also configurable at the renderer side, however in most situations they have already been configured by the proxy. Differently from the ProxyManager, the RendererManager provides an access to the migration environment, which contains a reference to:

- The local graphical toolkit to use.
- The parent container widget. This is required by the local toolkit when creating a widget inside another one.
- The `eventPort` that serializes the user events.

The init method must create the actual widget for this renderer, and set it in its initial state by getting the content of the store. Typically, this is achieved by:

```
      {ForAll {self.store getState($)}
       proc{$ K V} {self set(default K V)} end}
      {ForAll {self.store getBinding($)}
       proc{$ K#V} {self bind(default K V)} end}
```

Besides the init method, the Renderer class must define several other methods:

- `set(I K V)`: automatically called by EBL when the key `K` of the store `I` is set to the value `V`.
- `remoteSet(I K V ?R)`: automatically called by EBL when the proxy relies on the renderer for setting the key `K` of the store `I` to the value `V`. This method must return the success of the operation in `R`.
- `remoteGet(I K ?R)`: automatically called by EBL when the proxy uses the `remoteGet` function of its manager. The renderer must return in `R` the current value of the widget corresponding to the key `K` of the store `I`.
- `bind(I Event P)`: automatically called by EBL to configure a user event. The renderer must use the local toolkit to configure the widget to listen to `Event` of the store `I`, and trigger an action that sends `P` on the `eventPort`, so that events are forwarded to the proxy while respecting their causality order.
- `askBind(I Event P ?R)`: automatically called by EBL when the proxy relies on the renderer for checking the validity of `Event` of the store `I`. The event must be configured like for `bind(I Event P)`, and the method must report the success of the operation in `R`.
- `ask(Q ?R)`: automatically called by EBL when the proxy submits a question to the renderer by means of the `ask(Q R)` method of its manager. The renderer must interpret `Q`, and return the answer in `R`.
- `send(M)`: automatically called by EBL when the proxy sends a message to the renderer by means of the `send(M)` method of its manager.
- `destroy`: automatically called by EBL when the widget must be destroyed.

5.8.2 Windowing information

Even for simple widgets, application often needs windowing information: information regarding their actual rendering in a window that is not an actual parameter of the widget. The color depth of the screen displaying the widget and the absolute position of the widget on the screen are examples of windowing information. This information is dynamic in essence, and should not be cached in a store. Instead the proxy can directly ask the renderer. This question is an EBL transaction that is eventually answered when a renderer survives long enough to send to answer.

```
class WInfoProxyOfMyWidget
   from ProxyOfMyWidget
   meth winfo(K ?V)
       {self.Manager ask(winfo(K) V)}
   end

end
```

```
class WInfoRendererOfMyWidget
   from RendererOfMyWidget
   meth ask(Q R)
      R=case Q of winfo(P) then
           toolkit command for returning the P windowing information
        end
   end
end
```

5.8.3 Compound widget

A compound widget is a widget that is composed of several simple widgets, but is considered as a single widget from the point of view of the application. For example a monthly calendar that is created as a table of labels. The widget itself is the calendar; its implementation however uses several simple widgets. In practice, a single proxy will have to manage several physical widgets; each of them is associated to its own store to access and modify it.

5.8.4 An item container widget

An item container widget is a widget that is composed of different identifiable entities not detachable from the widget, and the application has a direct access to these entities. This is somewhat similar to a container widget, except that the sub-entities are not real widgets. For example the graphical items drawn on a vector based drawing area. The items are not proper widgets; in particular they can exist only inside a drawing area. For that reason, they are not programmed as individual widgets. Similarly to the compound widget, we associate a store per item. At the proxy side, the proxy creates individual objects for each item, so that they know their identity and use the widget's manager to communicate.

```
class ItemProxy
   feat Id Store
   meth init(I S)
      self.Id=I self.Store=S
   end
   functional core of the item using self.Store
end

class ItemContainerProxyOfMyWidget
   from ProxyOfMyWidget
   meth createItem(... return:R)
      Id={NewName}
   in
      R={New ItemProxy init(Id {self.Manager getStore(Id $)})}
      configuration of R
      notification of the existence of R to the renderer, for example
        by using a special key/store combination that maintains the list
        of the items
   end
end
```

5.8.5 Container widget

A container widget allows other widgets to migrate and be displayed inside its renderer. A very common container is the table widget. At the proxy side, a container provides a command for receiving other widgets. The renderer must implement a method that displays the migrated widget using the low level toolkit. Also, the renderer must implement a method that creates the child migration environment. In particular, this method must pass the reference to the toolkit to the child.

```
class ContainerProxyOfMyWidget
   from ProxyOfMyWidget
   meth display(Ref PlacementInstructions)
      possibly the proxy checks if another widget is already
```

```
      at the PlacementInstructions place to remove it from there
      {self.Manager importHere(Ref PlacementInstructions)}
   end

end
```

```
class ContainerRendererOfMyWidget
   from RendererOfMyWidget
   meth importHere(Ob PlacementInstructions)
      toolkit command for placing Ob according to PlacementInstructions
   end
   meth setChildEnvironment(E PlacementInstructions)
      {E put(toolkit self.toolkit)}
   end
end
```

5.8.6 Toplevel widget

A toplevel widget creates a resource for receiving other widgets. Typically, the toplevel widget is a window. From the point of view of EBL, a toplevel renderer receives an access to the local resource directly from its proxy; toplevel widgets are also container widgets and as such this resource is passed down to each individual migrated widget for the whole window. Also container widgets create the unique user `eventPort` associated to the site for keeping the causality order of user events. Messages received on the stream of this port are zero parameter procedures encapsulated by EBL that forwards the user event to the corresponding proxy when they are applied.

```
class WindowRendererOfMyWidget
   from ContainerRendererOfMyWidget
   meth init
      ContainerProxyOfMyWidget,init
      EventPort
      thread
         {ForAll {NewPort $ self.eventPort}
          proc{$ M} {M} end}
      end
      Env={self.Manager createRemoteEnvironment($)}
      {Env put(toolkit Toolkit)}
      {Env put(eventPort EventPort)}
   in
      {self.Manager createRemoteHere(Env)}
   end
end
```

The init method creates the `eventPort`, and applies the procedures received from it. Also it creates an environment from scratch, and enters the toolkit and `eventPort` inside. Finally, the renderer is created locally with this environment by the `createRemoteHere` method of the ProxyManager.

5.8.7 Simple multi-user functionality

EBL supports a simple multi-user functionality by letting a single widget having multiple renderers concurrently. By default, widgets are configured to have only one renderer connected at all time. This can be easily reconfigured.

```
class MultiUserProxyOfMyWidget
```

```
   from ProxyOfMyWidget
   meth allowMultipleRenderers(B)
      {self.Manager setConnectionPolicy(
       proc{$ M}
          case M of incoming(Id) then
             if {Not B} then
                {ForAll {self.Manager getRenderIds($)}
                 proc{$ I}
                   {self.Manager disconnect(I)}
                 end}
             end
             {self.Manager connect(Id)}
          else skip end
       end}
   end
end
```

The `setConnectionPolicy` method of the proxy widget configures a code that is applied when a renderer attempts to connect to this proxy. The code defined here accepts new incoming connections by calling the `connect` method of the ProxyManager. However when `B` is false, this code first disconnects all currently connected renderers, by using the `disconnect` method of the ProxyManager.

5.8.8 Global resource

A global resource is a simple multi-user widget that can be present several times at the same site; however there should be only one renderer created for it. For example, a font is a physical resource:

- It is a physical object one can interact with, to get information about its metrics for example.
- It can be used by several different widgets simultaneously.
- However if several widgets at the same site use the same font, it is better to have only one renderer for this font at that site.

This effect is achieved by using a unique storage for global resources at each site. The toplevel window creates the storage, which is passed down to all migrated widgets through the environment. The marshallers of the proxy and renderer stores are configured so that fonts use this unique storage automatically.

```
Globalizer

GlobalDict={Dictionary.new}

fun{Globalizer Env Ref}
   Id={VirtualString.toAtom Ref}
   O N
   H={Env get(render $)}
in
   {Dictionary.condExchange GlobalDict Id unit O N}
   if O==unit then
      %% create it
      E={H.manager createRemoteEnvironment($)}
      {H setChildEnvironment(E unit)}
      {E put(proxy Ref)}
   in
      {H.manager createRemoteHere(E render:N)}
   else
      %% already created, return the current occurence
```

```
      N=O
   end
   {N getWidget($)}
end
```

The `Globalizer` function takes an environment and a reference (migration capability) as input. It uses a dictionary for mapping the global resources to their reference, creating the entry in the dictionary when not already present.

The toplevel widget puts the `Globalizer` in the environment before creating its renderer:

```
{Env put(global Globalizer)}
```

Container widgets (including the toplevel itself) pass this resource to their children, in the method `setChildEnvironment`:

```
{E put(global {self.env get(global $)})}
```

Widgets have parameters of the type of font. A font is itself implemented by a proxy-renderer class pair. At the application side, when a parameter is set to a font, it is in fact set to the proxy of the font. This proxy cannot enter the store of the widget because it is not a distributable entity. Instead, the store is configured to transparently put the reference (migration capability) to the font instead. At the renderer side, the marshaller of the store is configured to automatically translate this reference into an actual renderer. We use the `Globalizer` function for this last step, so that a single renderer will represent all the occurrences of the same font at the renderer site.

```
fun{ObjectToRef O}
   if {Object.is O} then
      {O.Manager getRef($)}
   else
      O
   end
end

fun{RefToHandle O M}
   E={{M getManager($)} getEnv($)}
in
   {{E get(global $)} E O}.handle
end
```

A store can be configured to marshall/unmarshall the `globalResource` type automatically at the proxy side:

```
{ProxyStore setMarshaller(p(globalResource:m(u2s:ObjectToRef)))}
```

And finally at the renderer side:

```
{RendererStore setMarshaller(p(globalResource:m(s2u:RefToHandle)))}
```

Chapter 6 Case Studies and Evaluation

Chapter 3 and 4 define *what* the problem EBL solves is. Chapter 5 explains *how* EBL solves this problem and finishes by recipes for linking EBL to an actual toolkit. The Tcl/Tk graphical toolkit was interfaced with EBL to create the EBL/Tk toolkit [ETk]. Chapter 6 now focuses on *how to use* this EBL-enabled toolkit. This chapter presents a series of case studies using EBL/Tk and exemplifying how it can be used to support runtime migration, adaptation, and multi-user interaction. It also discusses how the design principles stated in Chapter 3 and 4 are materialized into relevant facilities. The case studies are: a migratable clock (6.1), an adaptable clock (6.2), an application adaptable to a PC and a PDA (6.3), and a multi-user game (6.4). The next section (6.5) explains the UniversalReceiver application that takes advantage of the generality of the migration capability mechanism. The last sections of this chapter evaluate different aspects of the approach: software engineering issues (6.6), performance measurements (6.7) and finally a comparative analysis with other solutions (6.8).

6.1 Case study #1: A migratable clock

This case study revolves around a simple clock application. Contrarily to the well known xclock application, our application will support migration.

Figure 6-1 xclock

We define the clock so that it uses the home site time, and not the time of the site currently displaying it. There are many different ways to implement this clock; this section will present some on them.

We first consider the clock as a stand-alone application whose functional core implements the clock (6.1.1). This functional core is then moved into the proxy of a special-

ized widget to give birth to a proper clock widget (6.1.2). For efficiency reasons, the functional core is then moved to the renderer, by using the delegation mechanism (6.1.3). So far the clock widget is a specialization of another widget. This limits the possible adaptations for the clock to this kind of widget. We remove this limitation by implementing a proper clock widget that relies directly on the low level toolkit (6.1.4). Relying on the low level toolkit forced us to use a different abstraction level for the programming of the user interface. EBL provides a technique that allows using the high level abstraction of EBL/Tk also at the renderer side (6.1.5).

6.1.1 The clock as a stand-alone application

The first approach is to develop a stand-alone application; the application's functional core is setting the displayed text to the current time every second.

```
UI={Build window(name:window
                 label(name:clock))}

proc{RunClock Clock}
   thread
      proc{Loop}
         Time={OS.localTime}
      in
         {Clock set(text:{Format Time.hour}#":"#
                         {Format Time.min}#":"#
                         {Format Time.sec})}
         {Delay 1000}
         {Loop}
      end
   in
      {Loop}
   end
end

{RunClock UI.clock}
{UI.window show}
```

Figure 6-2 EBL clock

The user interface is trivial: a label inside a window. The thread runs a loop (implemented by recursion using a functional style) that sets the text of the label to the current time each second.

To migrate this clock away from the display of the site that created it, we need a place to receive it. Let's call this place the rendering site, and pass the migration capability of the clock over there. EBL provides a simple discovery mechanism based on the IP address of the site offering the migration capabilities, and a TCP port number. Note that other discovery services could be used instead.

```
ClockReference={UI.clock getRef($)}
Pu={NewPublisher 15632}
{Pu.subscribe clock ClockReference "The Clock"}
```

The `getRef` method of the label widget returns its migration capability. This capability is a text string containing a reference to this widget over the Internet. It looks like **`^3#35^E^#E3x-ozticket://10.0.0.4:9000:eEKaiy:Ve/W:w:w:m:zR4OtvE^m*`**.
The `NewPublisher` function of EBL creates a publisher object bound to the TCP port provided as parameter. The publisher object is a dictionary that maps a key to a value and a string describing this value. The `subscribe` feature of the publisher object enters an entry into this dictionary.

The rendering site first has to create a window that will be the container in which to display this widget. Then, it must get the migration capability, and pass it to the container where to put it.

```
% the rendering site: another process running on another computer

UI={Build window(name:window)}
{UI.window show}

% the IP/port combination is hardcoded below
L={GetFromPublisher "10.0.0.4" 15632}

% L is a list of pairs Reference#Description. Here we need the first item of
the pair that is the first item of the list, hence L.1.1

{UI.window display(L.1.1)}
```

Once migrated, the label widget is still migratable by any other site that obtains its migration capability.

```
% yet another rendering site

UI={Build window(name:window
                 td(name:table
                    label(text:"Migrated Clock ")))}

{UI.window show}
L={GetFromPublisher "10.0.0.4" 15632}
{UI.table display(L.1.1 g(row:1 column:0))}
```

Figure 6-3 EBL clock

Figure 6-3 is a screenshot of migrating the clock inside a cell of a table instead of inside the window itself.

6.1.2 The clock as a widget, v1

We will now build a clock widget instead of having an application whose functional core implements one. We will have a proper clock widget which updates itself; the functional core of the application is no more concerned with the update of the clock. The new widget consists in three parts that have to be registered to EBL/Tk:

- The proxy of the widget. In this example, the clock is a specialized label widget. The clock proxy is defined by this class:

```
ETkLabelProxy={GetWidgetClass label} % proxy definition of the label widget

class ClockProxy from ETkLabelProxy
   meth init
      ETkLabelProxy,init
      {RunClock self}
   end
end
```

- The renderer of the widget. As the widget is a specialization of the label widget, we simply reuse the label widget renderer.

```
ETkLabelRenderer={GetRenderClass label default}
```

- The hybrid approach of EBL also needs a build function for mapping the description record of the UI into the actual widget. Once again this widget specializes the label widget, and we will reuse its build function.

```
ETkLabelBuild={GetBuildFun label}
```

The clock widget is registered by:

```
{RegisterAs clock ClockProxy ETkLabelBuild}
{SetRenderContextClass TCLTK clock default ETkLabelRenderer}
```

Once registered, the clock widget can be directly used:

```
UI={Build window(name:window clock(name:clock))}
{UI.window show}
```

As the clock is a specialized label, it supports its migration mechanism like we did before:

```
ClockReference={UI.clock getRef($)}
Pu={NewPublisher 15632}
{Pu.subscribe clock ClockReference "The Clock"}
```

Note that nothing changes at the renderer site as it still uses the capability registered by this proxy to migrate the clock widget. The fact that the migration capability now refers to a clock widget instead of a label one does not matter, the migration still occurs.

6.1.3 The clock as a widget, v2

The clock widget of 5.1.2 is still a label widget. In particular nothing prevents the application from changing the displayed text. Each second the clock widget would reset the text to the time, but in the meantime the application can still make it behave strangely. To address this problem, we will now create a proper clock widget which is not a direct specialization of another one. The API of this widget is limited to the clock functionality only.

```
class ClockProxy
   feat Clock !Manager
   meth init
      UI={Build label(name:clock)}
   in
      self.Clock=UI.clock
      self.Manager=self.Clock.Manager
      {RunClock self.Clock}
   end
   meth getRef($)
      {self.Clock getRef($)}
   end
end
```

The solution that requires the least changes over version 1 above is to have the `ClockProxy` delegate to a label widget instead of directly inheriting from it. To still be a valid EBL/Tk proxy, this version must provide the `getRef` method and pass the capability of the label. Also the `Manager` feature of the label must be replicated and point to the EBLProxyManager of the label widget.

Note that nothing changes at the renderer site as it still uses the capability registered by this proxy to migrate this new clock widget.

6.1.4 The clock as a widget, v3

The two first versions of the clock explicitly rely on the label widget. Their implementation is simplified and in particular it avoids creating a clock renderer class. However it limits the adaptation ability of the clock as we are stuck with the renderer of the label widget. To address this problem, we create a proper proxy-renderer pair for the clock widget.

```
class ClockProxy
   feat !Manager
   meth init
      self.Manager={NewProxyManager clock}
      thread
         Store={self.Manager getStore(clock $)}
         {Store setParametersType(t('...':'...'))}
         {Store setTypeChecker(t('...':fun{$ _} true end#""))}
         proc{Loop}
            {Store set(time {OS.localTime})}
            {Delay 1000}
            {Loop}
         end
      in
         {Loop}
      end
   end
   meth getRef($)
      {self.Manager getRef($)}
   end
end
```

This time the clock proxy creates its own EBL ProxyManager. According to the definition of this clock, the time is updated by the proxy. A thread runs a loop that updates a

key of a store to the current time of the proxy each second. Note that the store is configured to accept any value for any key; as it is used only internally by the proxy we can always assume the correctness of the updates.

```
class ClockRenderer
   feat
      manager handle tk parent eventPort
   meth init(M)
      self.manager=M
      self.tk={{M getEnv($)} get(tk $)}
      self.parent={{{M getEnv($)} get(parent $)} getWidget($)}
      self.eventPort=self.parent.eventPort
      self.handle={New (self.tk).label tkInit(parent:self.parent.handle)}
      {ForAll {{self.manager getStore(clock $)} getState($)}
       proc{$ K#V} {self set(clock K V)} end}
   end
   meth set(I K V)
      {self.handle tk(configure
                      text:{Format V.hour}#":"#{Format V.min}#":"#{Format
V.sec})}
   end
   meth destroy
      try
        {self.handle tkClose}
      catch _ then skip end
   end
end
```

The renderer must respect a skeleton imposed by EBL. With this simple example, the renderer must implement these methods:

- `init(M)`: when the clock widget is migrated, EBL creates an instance of this class, and calls the `init` constructor with the renderer manager as parameter. The manager gives an access to the local environment, in particular to the local graphical toolkit and to the parent widget. These allow the creation of the widget (self.handle). Finally the widget is initialized to the current state of the store.
- `set(I K V)`: this method is called by EBL each time the key `K` of the store `I` has been changed to `V`. This method must implement the consequence of this update on the actual widget. In this example, a single key of a single store is used for setting up the current time. Note that this method uses the low level commands of the toolkit, so that the portion of the code of the `RunClock` procedure that accomplishes this task cannot be reused directly.
- `destroy`: this method is called by EBL each time this actual renderer is no more needed, for example because it is migrated to another site, or because the widget is destroyed.

After registering the proxy and the renderer to EBL:

```
{RegisterAs clock ClockProxy proc{$ _} skip end}
{SetRenderContextClass TCLTK clock default ClockRenderer}
```

We end up with a functional clock widget.

Note that nothing changes at the renderer site as it still uses the capability registered by this proxy to migrate this new clock widget.

6.1.5 The clock as a widget, v4

The renderer of the version 3 of the clock had to be implemented using low level toolkits command. In particular, the code updating the label widget according to the time is different with the one from the `RunClock` procedure:

RunClock	v3 Renderer

`{Clock set(text:{Format Time.hour}#":"#`
`{Format Time.min}#":"#`
`{Format Time.sec})}`
	`{self.handle tk(configure`
	`text:{Format V.hour}#":"#`
	`{Format V.min}#":"#`
	`{Format V.sec})}`

Although they are pretty close in this example, they are different because they use different abstraction levels: RunClock uses the EBL/Tk abstraction level while the v3 renderer uses the Tcl/Tk abstraction level. This difference adds supplementary complexity when creating EBL widgets. For this reason EBL provides a way to work around the problem, by giving an access to the declarative build function at the renderer side. This function will create widgets at the EBL/Tk level of abstraction at the renderer side. These widgets can then be displayed locally there, by using a technique similar to the one for the toplevel widgets: create an environment from scratch, and display these widgets locally inside the renderer. So now the renderer is in fact a container for the widgets he himself creates. We hide the complexity of this mechanism into a generic renderer class. This class is dependent on the low level toolkit, and as such cannot be part of EBL itself. It is part of EBL/Tk however. Here is its definition:

```
class GenericRenderer
   feat
      manager handle tk parent content eventPort
   meth init(M)
      self.manager=M
      self.tk={{M getEnv($)} get(tk $)}
      {{M getEnv($)} put(render self)}
      {{M getEnv($)} put(handle self.handle)}
      self.parent={{{M getEnv($)} get(parent $)} getWidget($)}
      self.eventPort=self.parent.eventPort
      self.handle={New (self.tk).frame tkInit(parent:self.parent.handle)}
      {(self.tk).send grid(columnconfigure self.handle 0 weight:100)}
      {(self.tk).send grid(rowconfigure self.handle 0 weight:100)}
      {self createContent}
      {M displayHere({self.content.top.Manager getRef($)} unit)}
   end
   meth importHere(Ob P)
      Tk=self.tk
   in
      {Tk.send grid(Ob.handle column:0 row:0 sticky:nswe)}
   end
   meth remove(Ob)
      {(self.tk) grid(forget Ob.handle)}
   end
   meth destroy
      try
        {self.handle tkClose}
      catch _ then skip end
   end
   meth setChildEnvironment(E _)
      {E put(tk self.tk)}
      {E put(system {{self.manager getEnv($)} get(system $)})}
```

```
      {E put(global {{self.manager getEnv($)} get(global $)})}
   end
end
```

As this renderer is now a container, it must also implement the `importHere`, `remove` and `setChildEnvironment` methods. Now we can implement a clock renderer based on this class:

```
class ClockRenderer from GenericRenderer
   meth init(M)
      GenericRenderer,init(M)
      {ForAll {{self.manager getStore(clock $)} getState($)}
       proc{$ K#V} {self set(clock K V)} end}
   end
   meth createContent
      self.content={self.manager build(label(name:top) $)}
   end
   meth set(I K V)
      {self.content.top set(text:{Format V.hour}#":"#
                                 {Format V.min}#":"#
                                 {Format V.sec})}
   end
end
```

And now the renderer is working at the EBL/Tk abstraction level instead of the low level Tcl/Tk one.

There is a last implementation detail that needs to be solved for this example to work: EBL allows defining several widget repositories concurrently, each one using their own set of proxy class definitions, while the renderer definitions are shared among all these repositories. For the build function of the renderer manager to work right, EBL needs to know which proxy repository should be used. This information is known when the application constructs the widget, by the build function associated to the proxy for supporting the declarative approach for building UIs. This function must pass this information to the manager of the proxy, which relays it automatically to the connected renderer. Consequently the clock proxy must now be registered like this:

```
{RegisterAs clock ClockProxy
 proc{$ E} {E.handle.Manager setBuilder(E.builder)} end}
```

6.2 Case study #2: An adaptable clock

Let us extend the clock v4 of section 5.1 by adding new renderers. This will allow an application to dynamically switch between them, for example to dynamically adapt the clock with respect to the size available to display it. To offer adaptation to the application, the clock widget must provide a method to support it:

```
meth setContext(C) {self.Manager setContext(C)} end
```

We begin by simple alternate textual adaptations for the clock (5.2.1). Then we continue with a complete different type of representation for the clock: an analog clock (5.2.2), and show examples of further possible adaptations (5.2.3).

6.2.1 Adaptation example #1

Let us first create other textual representations of the time, which can still use a label widget:

```
F=[hourmin#fun{$ V} {Format V.hour}#":"#{Format V.min} end
   hourminsecdate#fun{$ V}
                     {Format V.hour}#":"#{Format V.min}#";"#
                     {Format V.sec}#"\n"#
                     {Format V.mDay}#"/"#{Format V.mon+1}#"/"#V.year+1900
                  end]

{ForAll F
 proc{$ Name#Fun}
    class LabelRenderer from ClockRenderer
       meth set(I K V)
          {self.content.top set(text:{Fun V})}
       end
    end
 in
    {SetRenderContextClass TCLTK clock Name LabelRenderer}
 end}
```

Two functions are created to transform a date into different string representations. These functions are paired with names, and placed in a list. The list is parsed by the `ForAll` procedure, and for each name & function pair, a class is created that specializes the `ClockRenderer` of v4 so that the `set` method now uses the function to map the time to a string. This class is registered with the associated name.

We can now build a clock as usual, and then switch between the default representation defined by v4, and these two new ones:

```
UI={Build window(name:window clock(name:clock))}
{UI.window show}

...

{UI.clock setContext(hourminsecdate)}

...

{UI.clock setContext(hourmin)}

...

{UI.clock setContext(default)}
```

It is interesting to note how we used a mixed declarative/imperative approach for creating these two new renderers:

- The list is a declarative data structure that contains high level specification of a problem.
- This list is parsed to create the individual renderers.

In other words, the list is a specification in a model, while the second part is the dynamic interpretation of this model into an executable entity. This approach is possible because Oz is a multi-paradigm programming language. With this approach, new textual render-

ers can be added by simply extending a list. Alternate renderers are created at a very low development cost. In other words, this approach facilitates the creation of adaptable widgets.

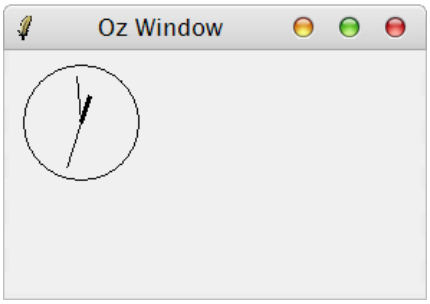
6.2.2 Adaptation example #2

In the adaptation example #1, the renderers we added were well suited for a mixed declarative/imperative approach. Let us now add a renderer which is not well suited for this approach: an analog clock that cannot be implemented by a label widget. The only Tcl/Tk widget that supports displaying such information is the canvas widget: a vector based drawing area that is piloted by imperative commands. Drawing an analog clock in a EBL/Tk canvas is achieved by these procedures:

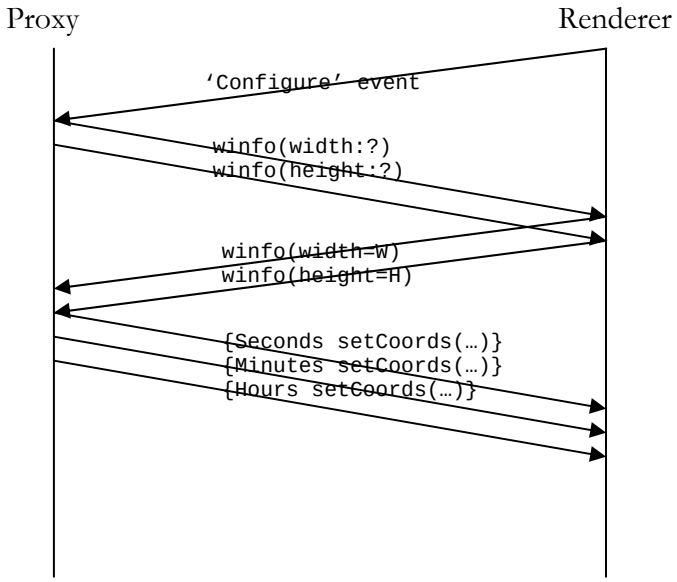
```
PI2={Float.acos 0.0}

fun{InitCanvas Canvas}
   Ring Hour Min Sec
   {Canvas create(oval [0.0 0.0 0.0 0.0] handle:Ring)}
   {Canvas create(line [0.0 0.0 0.0 0.0] width:3 handle:Hour)}
   {Canvas create(line [0.0 0.0 0.0 0.0] width:1 handle:Min)}
   {Canvas create(line [0.0 0.0 0.0 0.0] width:1 handle:Sec)}
in
   r(ring:Ring hour:Hour min:Min sec:Sec)
end

proc{SetTime AC Time Width Height}
   CM={Int.toFloat Time.min}/60.0
   CH={Int.toFloat (Time.hour mod 12)}+12.0+CM/12.0
   CS={Int.toFloat Time.sec}/60.0
   S={Max {Min Width Height} 40.0}
   S2=S/2.0
   S23=S2*2.0/3.0
   S25=S2*2.0/5.0
in
   {AC.ring setCoords([10.0 10.0 S-10.0 S-10.0])}
   {AC.sec setCoords([S2 S2
                      S2+S23*{Float.cos CS*4.0*PI2-PI2}
                      S2+S23*{Float.sin CS*4.0*PI2-PI2}])}
   {AC.min setCoords([S2 S2
                      S2+S23*{Float.cos CM*4.0*PI2-PI2}
                      S2+S23*{Float.sin CM*4.0*PI2-PI2}])}
   {AC.hour setCoords([S2 S2
                       S2+S25*{Float.cos CH*4.0*PI2-PI2}
                       S2+S25*{Float.sin CH*4.0*PI2-PI2}])}
end
```

The `InitCanvas` function creates the artifacts of the analog clock. The `SetTime` procedure changes the coordinates of these artifacts according to a time, a width, and a height. We can use these functions directly on a canvas:

```
UI={Build window(name:window canvas(name:canvas))}
{UI.window show}
AC={InitCanvas UI.canvas}
{SetTime AC {OS.localTime} 100.0 100.0}
```



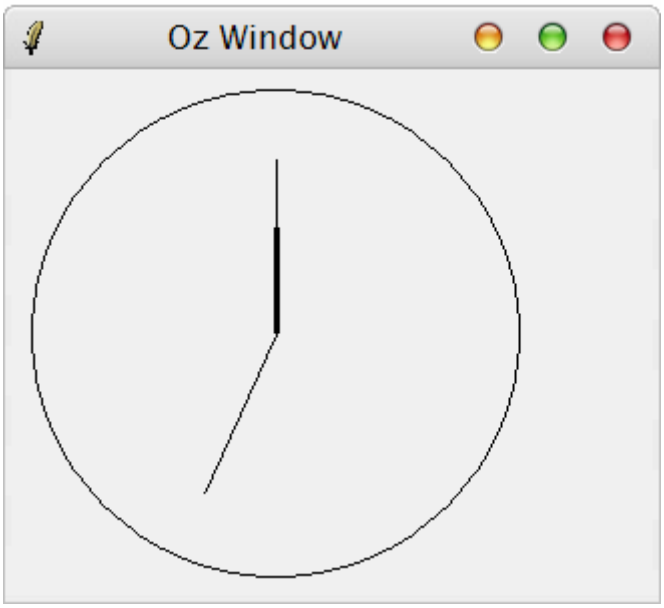
To create a proper live analog widget, we need to extend this code to update the time each second: this is achieved by the clock proxy. Also we need to take the display size of the canvas into account, so that the analog clock resizes itself accordingly. We could implement that part at the proxy side. However in that situation when the canvas is resized, the action that resizes the clock would be executed by the proxy and not by the renderer. This action must first obtain the new width and height of the canvas, and then update the clock accordingly: these are all remote operations.



This places an unnecessary overhead on the network. To avoid this problem, the renderer will manage the resize of the canvas directly:

```
class AnalogClockRenderer from GenericRenderer
   attr Width Height Time
   feat AC
   meth init(M)
      GenericRenderer,init(M)
      %% The InitCanvas function creates the clock artifacts
      self.AC={InitCanvas self.content.top}
      %% The Configure event below is triggered when the widget is resized
      {self.content.top
       bind(event:'Configure'
            action:proc{$}
                      %% Obtains the new width and height of the widget
```

```
                      Width:={Int.toFloat {self.content.top winfo(width:$)}}
                      Height:={Int.toFloat {self.content.top
                                            winfo(height:$)}}
      %% The SetTime procedure configures the clock artifacts to display
      %% a specific time at a specific width and height
                      {SetTime self.AC @Time @Width @Height}
                   end)}
      %% The stores are preconfigured with the current state of the widget
      {ForAll {{self.manager getStore(clock $)} getState($)}
       proc{$ K#V} {self set(clock K V)} end}
   end
   meth createContent
      self.content={self.manager build(canvas(name:top) $)}
   end
   meth set(I K V)
      Time:=V
      %% The SetTime procedure configures the clock artifacts to display
      %% a specific time at a specific width and height
      {SetTime self.AC V @Width @Height}
   end
end

{SetRenderContextClass TCLTK clock analog AnalogClockRenderer}
```



This example shows two important points of EBL:

1. It is possible to use the high level toolkit (EBL/Tk) when defining a renderer. This makes it possible to directly move part of the functional code between the proxy and the renderer.
2. The hybrid declarative/imperative approach allows using a model-based approach when the situation is favorable and an imperative approach otherwise. In particular, there is no restriction of expressivity on what is possible to implement.

6.2.3 More adaptation examples

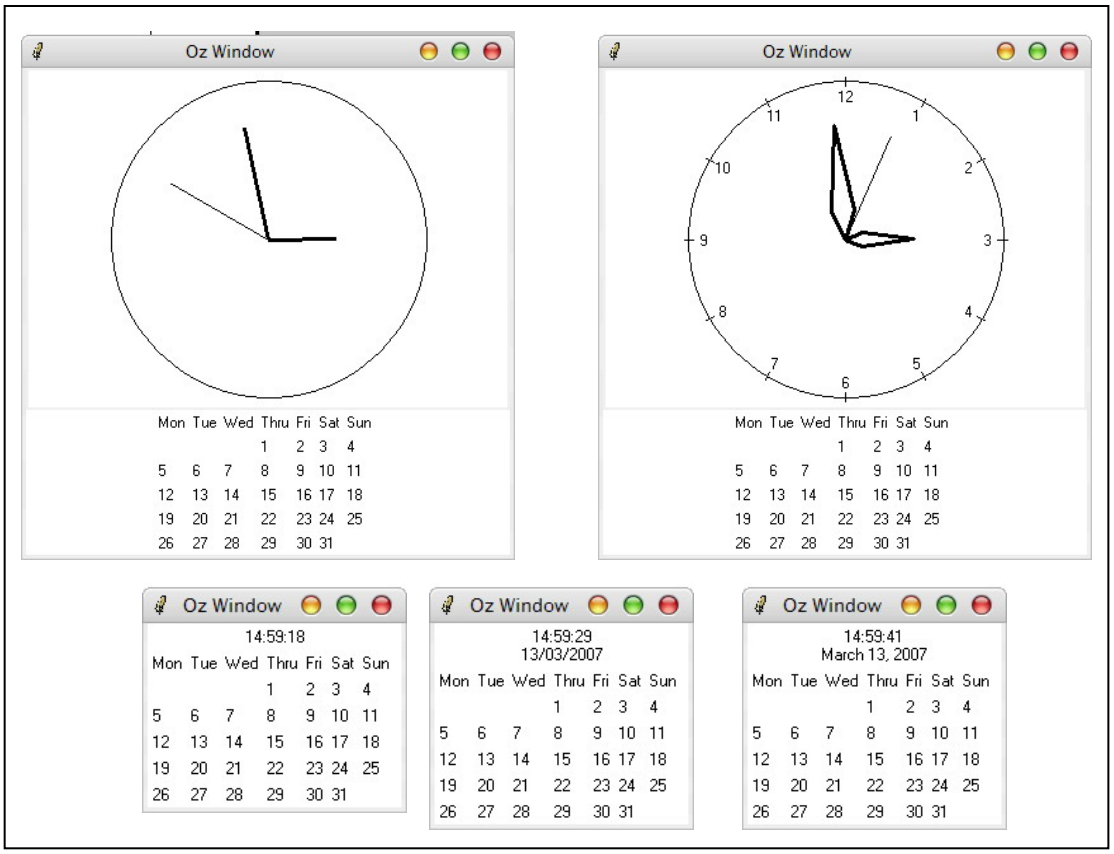


Figure 6-4 Adaptable clock with calendar

Figure 6-4 shows more adaptation examples for the clock. Once a widget has been defined, any application can use it and benefit from its adaptations. EBL can be used as a repository for adaptable widgets, allowing reusability for all applications.

6.3 Case study #3: An adaptable application

Some of the widgets of EBL/Tk support natively different representations. Let us show an example of an application for entering information about a movie, using adaptation.

```
UI={Build window(name:window
                 td(navigator(name:navigator
                              lr(label(text:"Director:" glue:w)
                                 entry(name:director glue:we) newline
                                 label(text:"Writers:" glue:w)
                                 entry(name:writes glue:we) newline
                                 label(text:"Release Date:" glue:w)
                                 entry(name:releasedate glue:we))
                              lr(selector(name:genre
                                          text:"Genre"
                                          items:["Action" "Crime"
                                                 "Drama" "Comedy"]))
                              td(label(text:"Plot outline")
                                 text(name:text width:40 height:10
```

```
glue:nswe)))))}
```

Each adaptable widget defines its own names for the different possible renderings. We can force them to use shared names for the combination of renderings that interest us. We will define two of them:

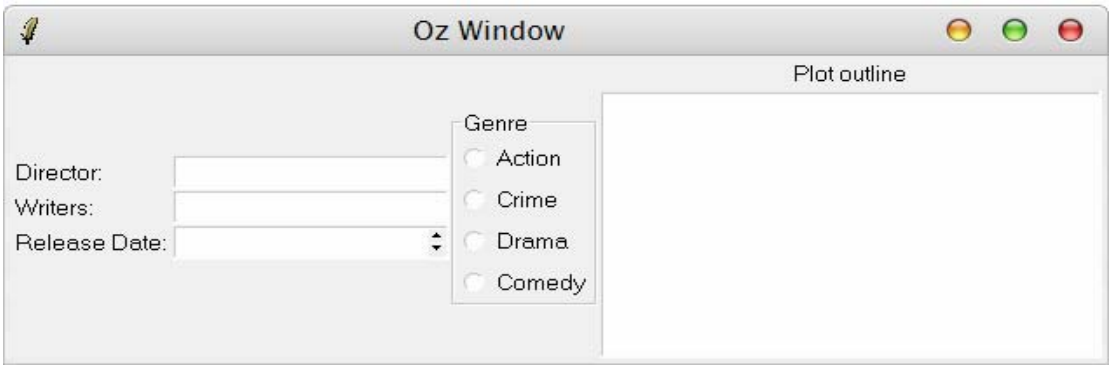
- One for use on a PC where a large screen and a keyboard are present.
- One for use on a PDA where the screen estate is small, and there is no keyboard.

```
AdaptationMap=[navigator#flat#default entry#default#virtualkb
               spinbox#default#virtualkb selector#default#menu]

{ForAll AdaptationMap
 proc{$ Widget PC PDA}
    {SetRenderContextClass TCLTK Widget pc {GetRenderClass Widget PC}}
    {SetRenderContextClass TCLTK Widget pda {GetRenderClass Widget PDA}}
 end}
```

The `AdaptationMap` list is parsed and for each of the specified widget, the two renderer definitions we are interested in are obtained, and re-registered using the `pc` and `pda` names.

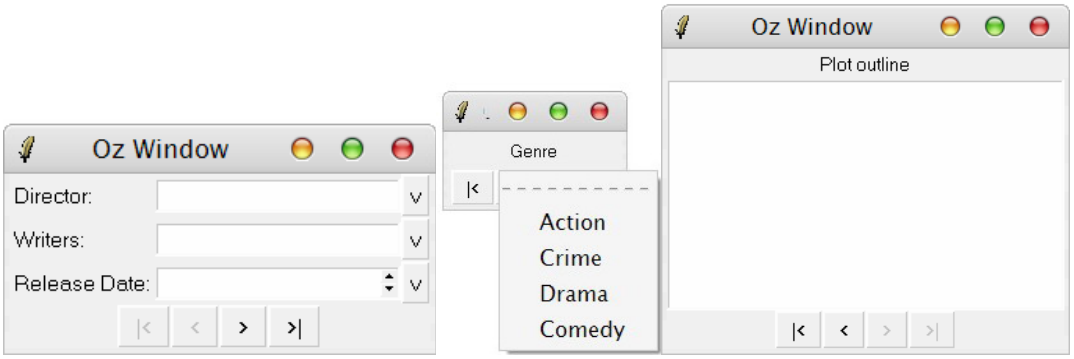
This allows us to use the `setContext` method on `UI` itself, which applies the context to all the widgets it manages.

```
{UI setContext(pc)}
```



The PC version offers a side by side presentation for the three groups of input fields. The left part uses the normal text and number input widgets. The middle part uses a set of radio buttons for selecting the genre. The right part uses a normal text widget.

```
{UI setContext(pda)}
```



On the other hand the PDA version only display one part at a time, with navigational buttons at the bottom. Further the text and number input widgets have an arrow that displays a virtual keyboard for entering the data:

The middle part uses a menu to select between the items instead of radio buttons. This widget also has a listbox renderer that we could use instead:

And finally the right part uses a text widget for which no alternate renderer is currently provided unfortunately. If a renderer was created with support for a virtual keyboard, then we could use it by specifying it in the `AdaptationMap`, without changing any other part of the code.

6.4 Case study #4: A multi-user application

Let's consider a digital Pictionary-like game. The game is run on three devices: a PDA, a laptop, and a computer connected to a video projector as illustrated in Figure 6-5.

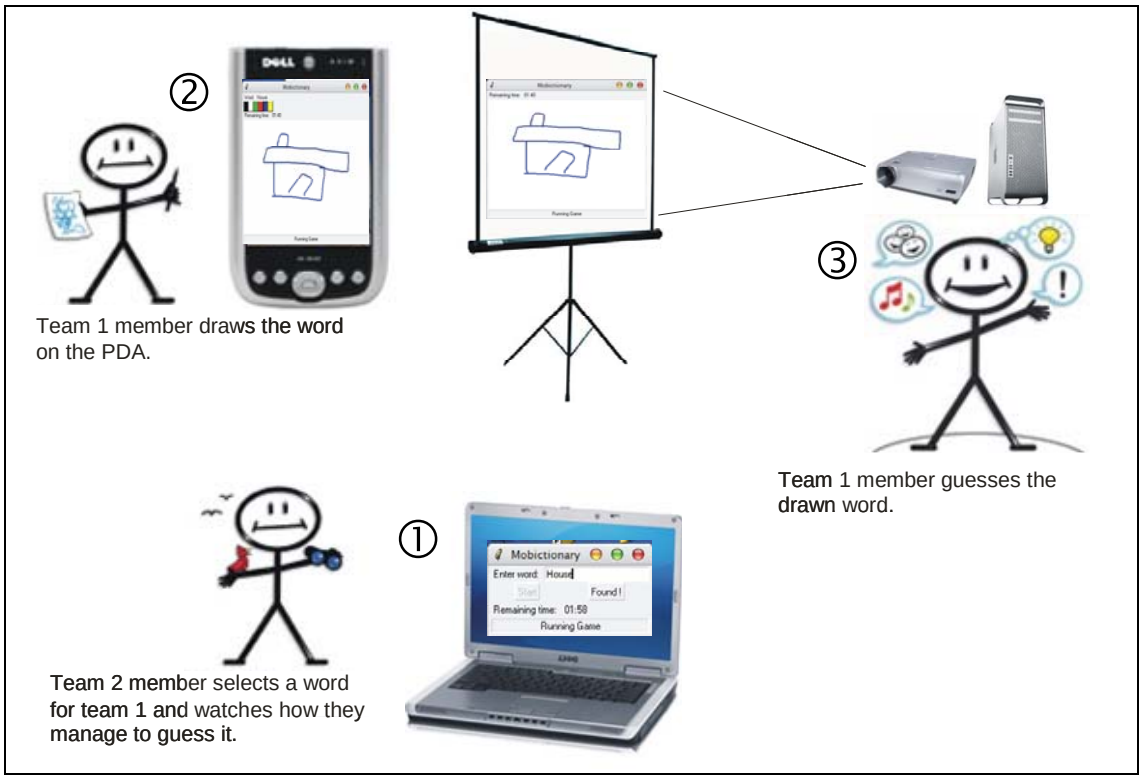
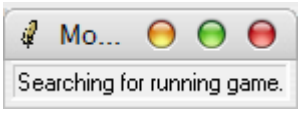


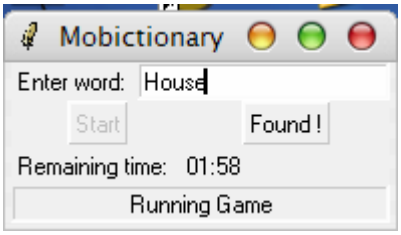
Figure 6-5 Mobictionary scenario

To start the game, the application must be run on the three devices.

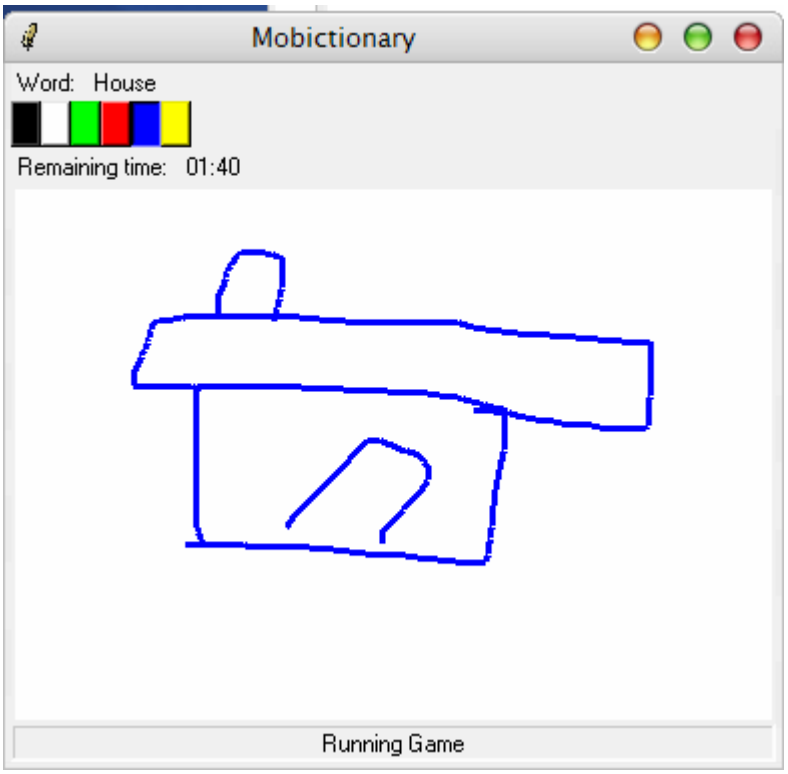


Once the connection is established,

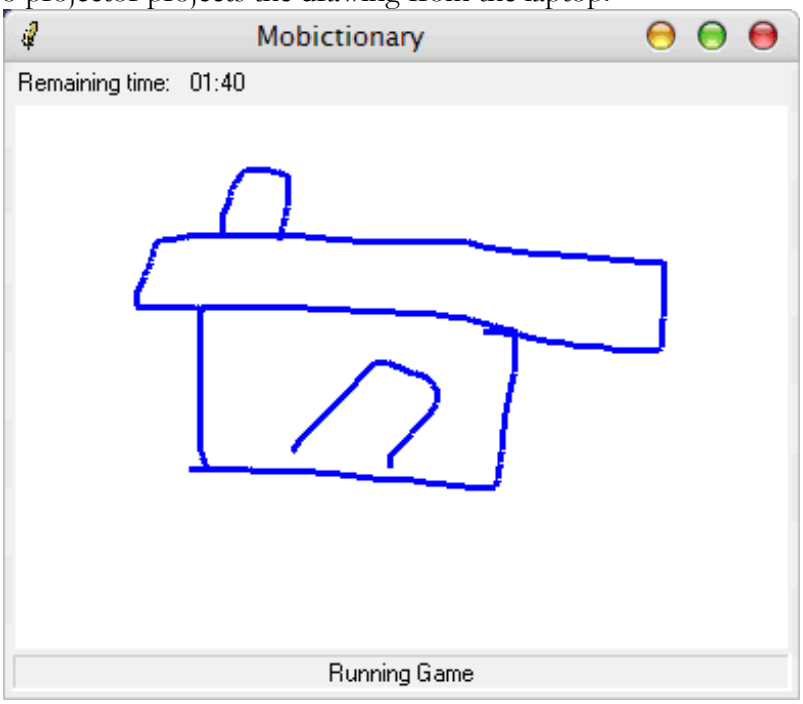
- The laptop is given to one team for them to enter a word. Once entered, a countdown starts.



- The drawer from the other team is given the PDA: the word appears on it, and he can draw whatever he wants on the screen.



- A video projector projects the drawing from the laptop.



Let's design this application with our toolkit, using the simple multi-user aspects of the toolkit. Even if three different devices are used in a distributed fashion, we must think of this application as a single process application, with transparently distributed user interfaces. The widgets used are:

- A text field for entering the text (Laptop)

- A label for displaying this text (PDA)
- A start button for accepting the text input and starting the countdown (Laptop)
- A clock (PDA, Laptop, and PC)
- A free drawing area (PC and PDA)
- A toolbar for selecting the drawing tools (PDA)
- A "Win" button the click if the word is guessed on time (Laptop)

We end up with 7 different components, some of them present only on one device, others at different devices simultaneously. Let's get back to the proxy-renderer relationship: the underlying principle is that the proxy serves as the reference for the state of the widget while the renderer follows the instructions of the proxy for updating its incarnation. This principle allows the existence of several renderers; each of them following the same instructions. Basically they all act as mirrors views of the same proxy. Note that this is against the principle of not disrupting the stationary behavior as we introduce multi-user capabilities. New complexity is introduced because of concurrency and coherency problems. The toolkit itself provides a very basic way of dealing with this complexity: each widget can be configured so as to have at maximum one renderer at a time (the default), or to let an arbitrary number of renderers connected at the same time. This thesis is not going to explorer further the complexity of managing multi-user interfaces that way; however we plan on doing it on future research. Nevertheless for the Mobictionary application, we have enough:

- The application can be run from any device, including the PC, the laptop, the PC or even another computer. For our demonstration, we use the laptop.
- The text field, Start and Win buttons are created and migrated to the laptop
- The clock is created, configured to support multiple renderers and displayed on all the devices
- The label and the toolbar are created and migrated to the PDA.
- The canvas is created, configured to support multiple-renderers and displayed on the PDA and the PC.

When the start button is clicked, the content of the text entry is placed into the label, and a countdown thread is created: each second the clock is updated to reflect the remaining time. The toolbar chooses the active drawing tool, while the clicks on the drawing area issue commands to apply it at that place. And that's it; we have a functional simple multiplayer game!

Note that the three processes find each other using the Discovery module of Mozart, which uses a broadcasting message on a LAN to find providers.

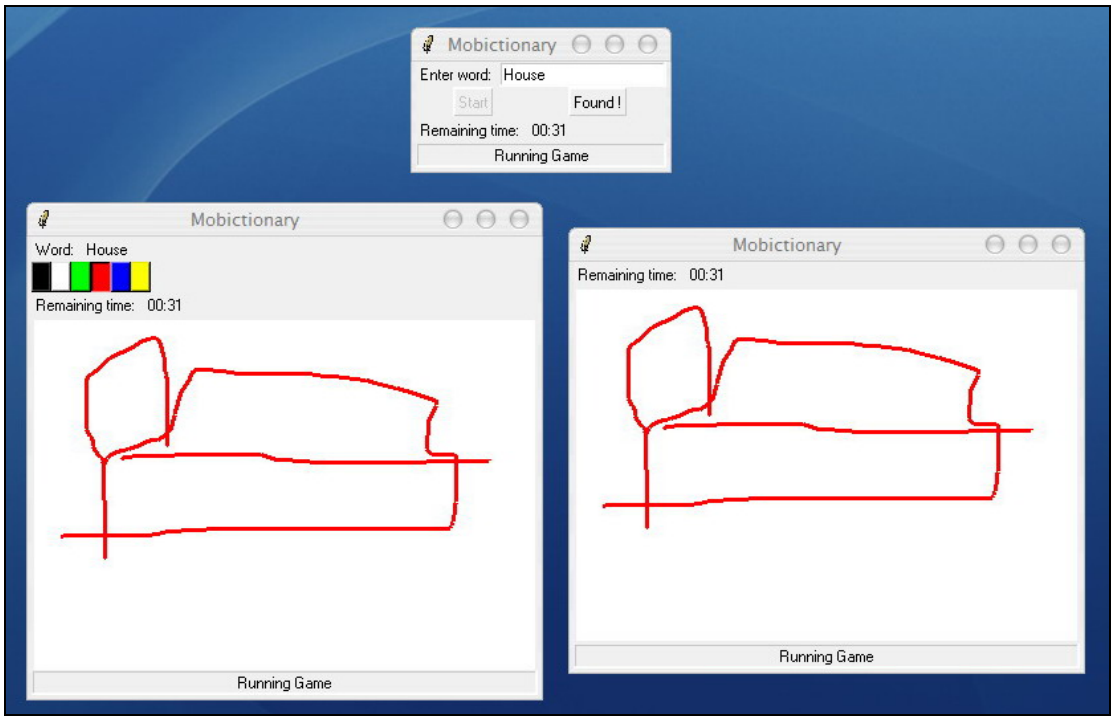


Figure 6-6 The Mobictionary application

The code of this application is available at Annex B. This case study is a good example of how EBL enables a simple multi-user application to be designed as a single application.

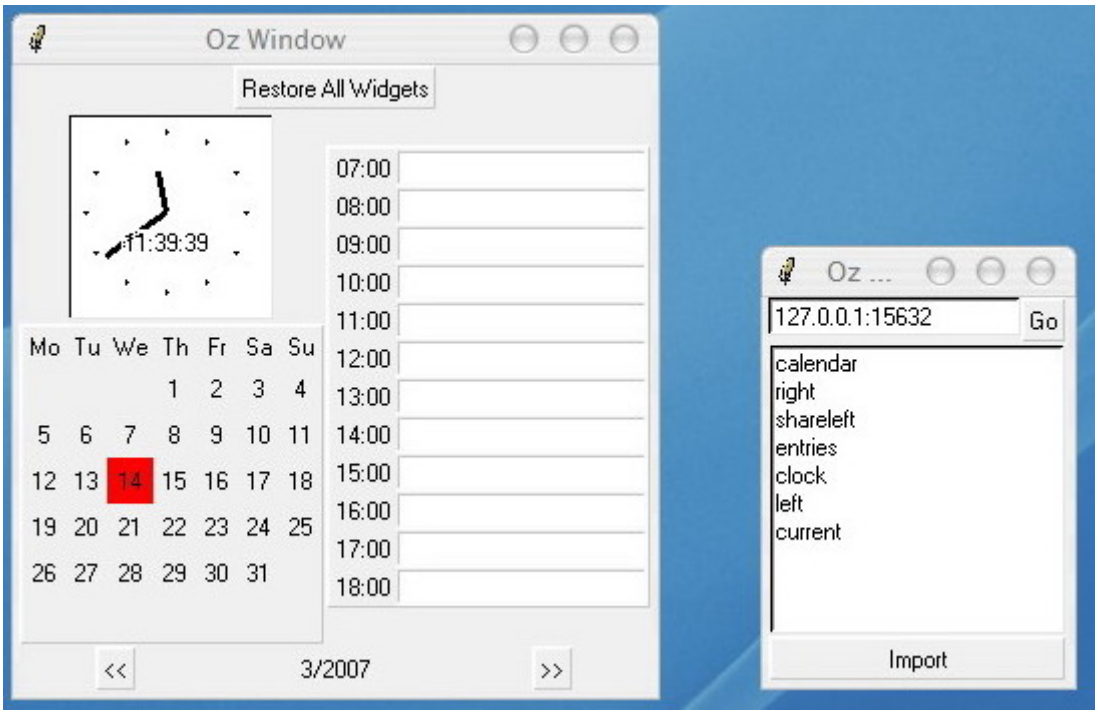
6.5 Flexible & transparent migration: UniversalReceiver

With this toolkit, all the widgets have automatically a migration capability. This capability is controlled by the universal reference of the widget. This universal reference is a simple text string encoding the information needed to find the widget on the Internet. As long as the widget exists, this reference implements the migration capability of the widget. Typically, passing a reference from an application A to an application B is achieved by a discovery service. This service can be implemented in many different ways:

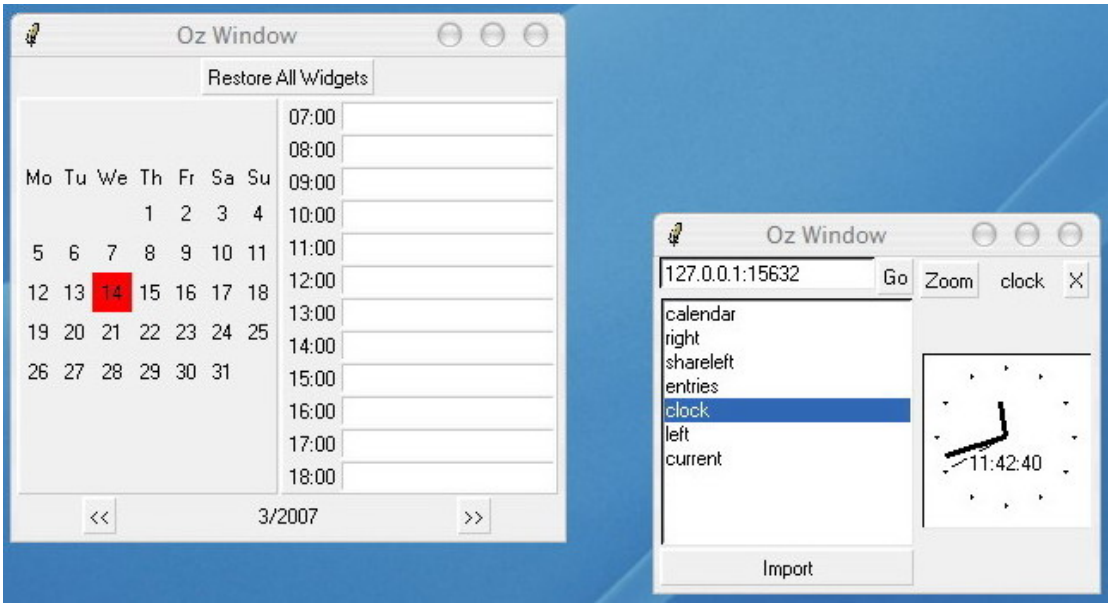
- By human beings, dictating the reference over the phone.
- By email, sending the reference inside an email.
- By using an Internet server, where A registers the reference and B gets it back. This server can be a Web server, an FTP server, or the simple socket server provided by EBL itself.
- By broadcasting messages over a LAN, allowing B to find A and get the reference. This can be implemented by the Discovery module of Mozart.
- By registering to a peer to peer network and using its functionality to get the reference. This can be implemented by the P2PS module for Mozart.

The reference mechanism completely hides the actual functionality of the associated UI. In particular the application B does not need to be specifically designed to fit with the application A in order for the migration to work. The application could B designed to work hand in hand with some part of the UI of the application A: this is a design issue and not a requirement. Thanks to that property, we have created a UniversalReceiver that uses the embedded EBL discovery functionality for migrating any UI offered by this ser-

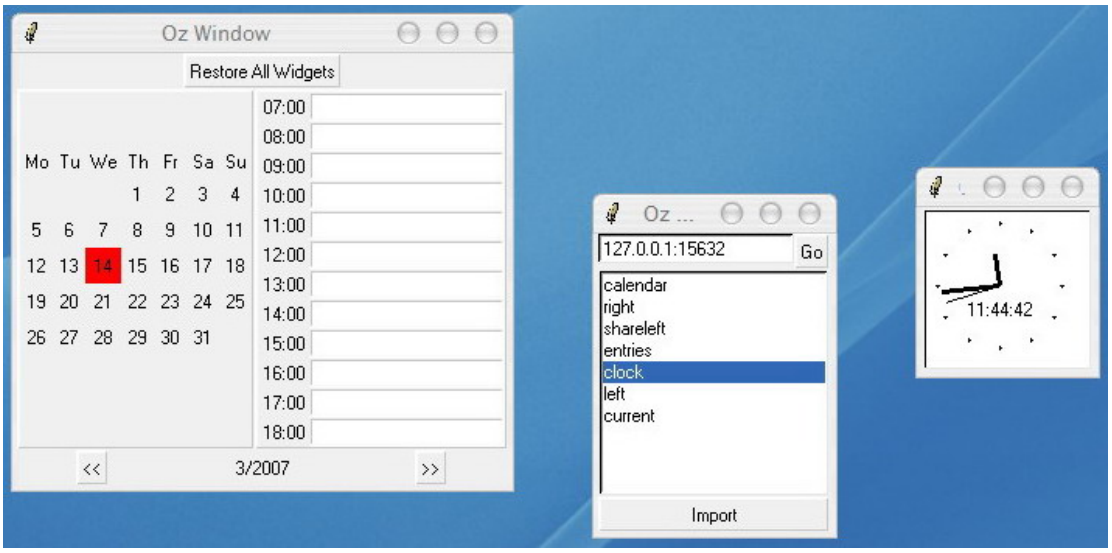
vice. The screenshots are executed on the display of a single computer for clarity; the remote applications are different processes on the same computer. For EBL it would work exactly the same if the processes were on other computers.



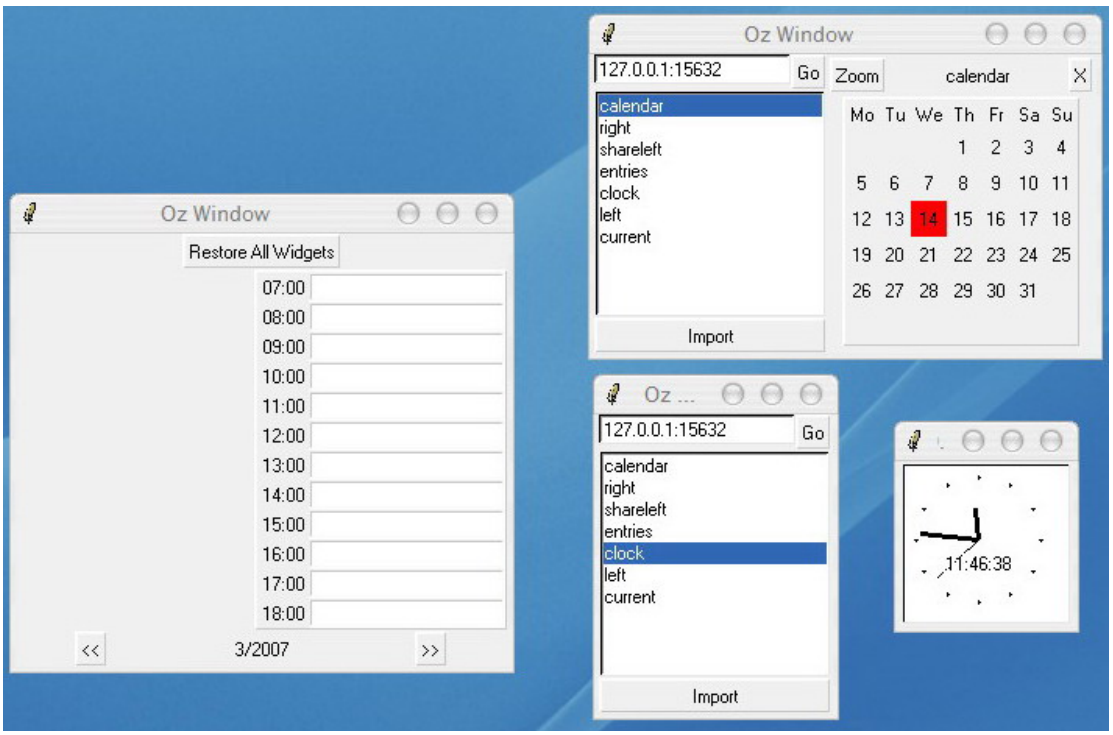
The Calendar application at the left offers different parts of its UI using the EBL discovery module. The UniversalReceiver application at the right gets the information over these parts. One can select one of them, and import it.



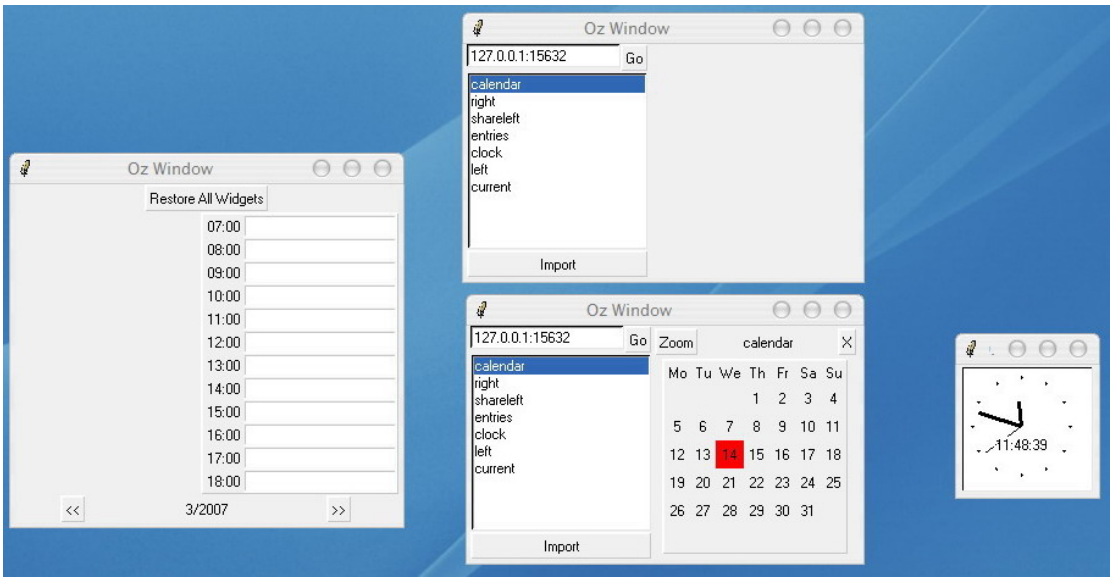
The Zoom button in the UniversalReceiver window creates a new window (in the same process as the UniversalReceiver) and migrates the widget there.



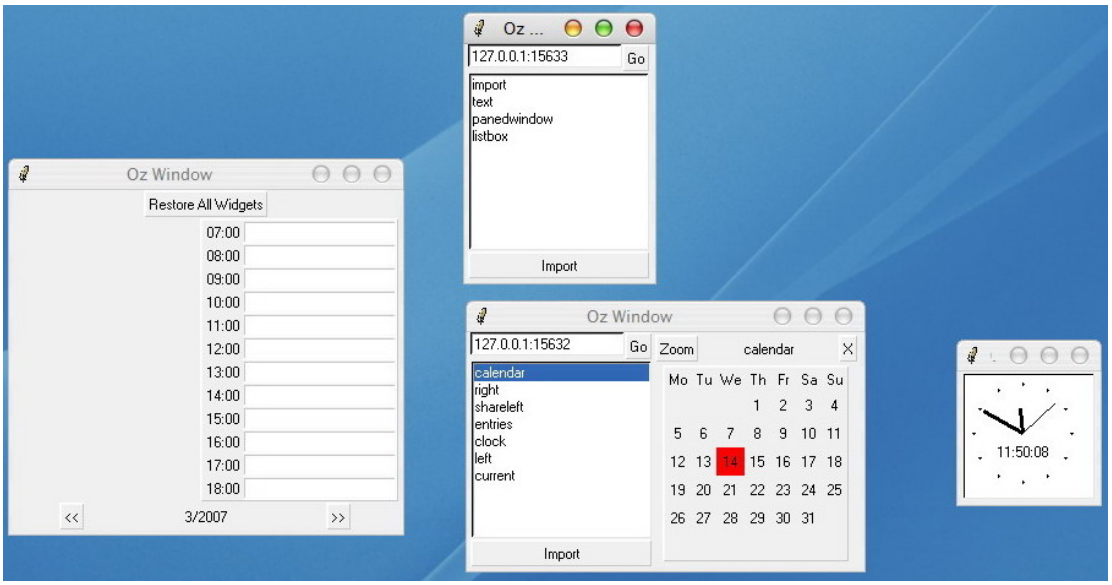
A second UniversalReceiver can be concurrently launched, and get the references from the Calendar.



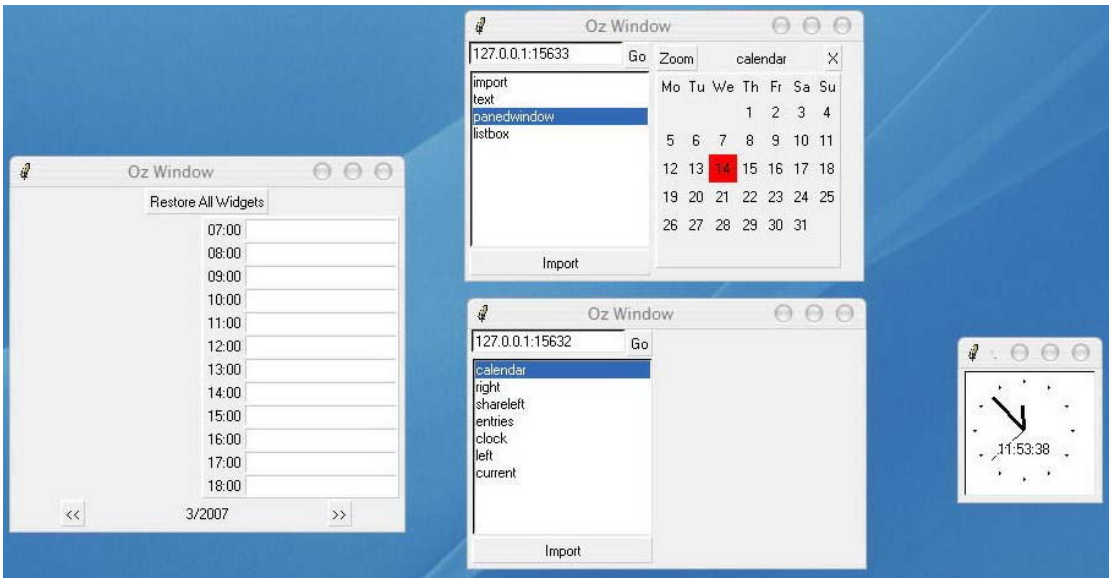
A UI migrated at the second UniversalReceiver can still be migrated at the first one.



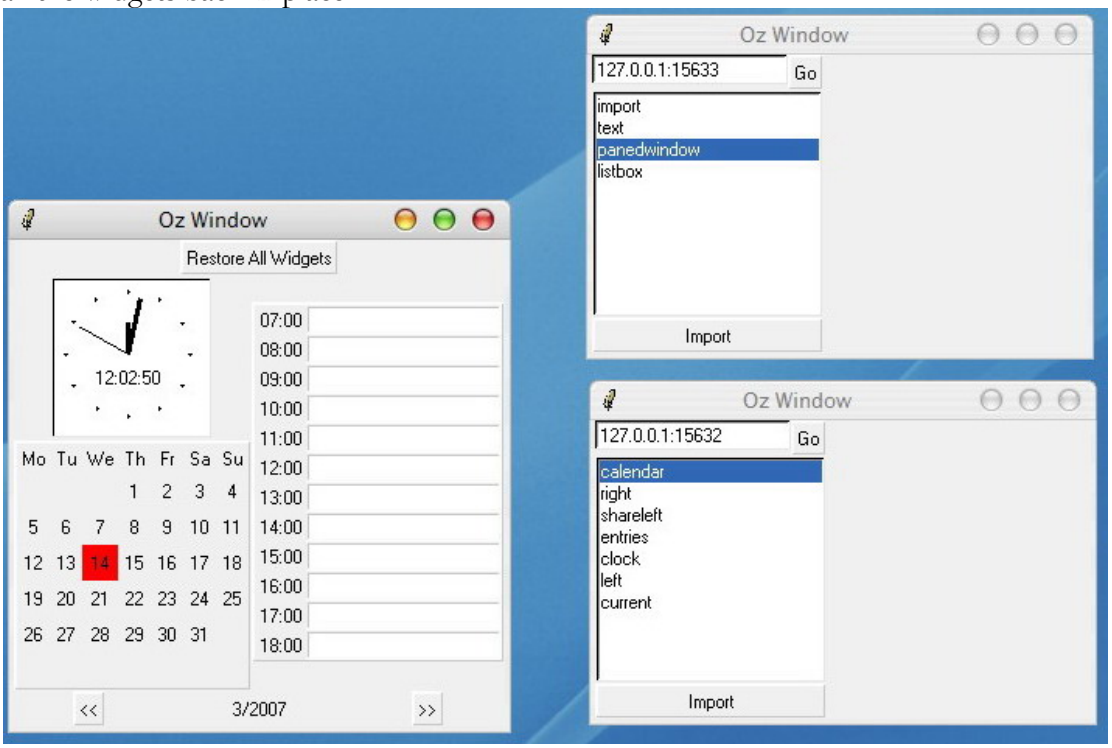
Now let's connect the second UniversalReceiver to the first one (note the different port number).



And we can migrate the right part of the first UniversalReceiver (the panedwindow) into the second UniversalReceiver. Note how the calendar is migrated along.



And finally we can use the Restore All Widget button of the Calendar application to gets all the widgets back in place.



This example demonstrates the great flexibility of the migration provided by EBL.

6.6 Software engineering issues

The purpose of this thesis is to demonstrate the EBL approach to providing migration, adaptation and simple multi-user for graphical user interfaces. We do not have any particular claims regarding the software engineering aspects of EBL enabled toolkits. In particular EBL is not designed to enforce the developer into a style deemed appropriate for

GUI programming. The focus is to add the extra functionality on top of the concepts of usual graphical toolkits (widgets, event bindings) in a mostly conservative way. Because of this, EBL is by default no better or worse than typical toolkits from a software engineering point of view. However we can make several observations that hints that EBL has in fact several interesting properties from this point of view:

- The MVC (model-view-controller) pattern is usually considered the correct way to program complex UIs. The mixed declarative/imperative approach naturally favors this approach: using a functional approach, it is natural to isolate the model, and infer a view (the description record in a declarative paradigm) and the associated controller (the functional core in an imperative paradigm). The switching of paradigm forces a clean split between the view and the controller. Creating alternate representations for supporting adaptation is implemented by creating alternate functions that infer other view/controller pairs.
- The management of user events by the toolkit often forces the programmer into a bad programming style:
  1. Typically toolkits associate one event to one piece of code, however in general the real association must also take a context into account. For example, clicking on an unselected rectangle selects it, while clicking on an already selected rectangle initiate a move operation by drag and drop. The context of the rectangle is characterized by its selection state, which changes the code to execute in reaction to the same user event. Except some toolkits like HsmTk [Blanch05b] designed to specifically deal with this situation, most toolkits ignore the context and only allow user event to code direct associations.
  2. Some mainstream toolkits (e.g. AWT for Java) force managing user events in specific groups. For example, a group will handle all mouse related events; the developer will have to manage all these events specifically in this group.

  These two reasons force the place where the events are treated inside the code of the application. This place is not based on modularity or locality considerations, but on the place imposed by the toolkit. This results in spaghetti like coding, where pieces of code logically connected together are split throughout the rest of the code.

  EBL provides a single `bind` method for all events configuration. This method works at the individual event granularity, hence does not suffer from point two above. However EBL does not manage contexts for the user events, hence suffers from the limitation of point one above. Context management of user events can be added in future versions of EBL.

6.7 Performance

The speed tests are executed on a 2 GHz Pentium-M laptop with 1 GB of memory running Windows XP SP2. All tests are executed inside the Mozart OPI using a fresh virtual machine. The time is measured in milliseconds (ms).

6.7.1 Toolkit speed analysis

As EBL relies on an external library for the actual display of the widgets, its performance is the one of that library plus the overhead it introduces itself. This overhead occurs each time the widget has to communicate with the application, which could happen:

1. when the proxy sends an update to the renderer (simple send)
2. when the proxy asks the renderer to apply an update and return back the result (send & receive)
3. when the renderer notifies the proxy of an update, for example when the user types a letter in an entry widget (simple send)
4. when the renderer notifies the proxy of an event (simple send)

Also, the migration process itself is a distributed operation that may require a lot of information to be sent to the renderer. Note that when the renderer is displayed at the same site as its proxy, then the distribution feature of Mozart makes sure that all the communication is purely local; in this situation the overhead is very minimal.

It is very hard to have a useful metrics to measure the overhead introduced by EBL, because a lot of factors beyond our control take part of the global overhead:

1. Respective speed of the computers involved in the distribution of the UI. In particular, embedded devices like handhelds computers have very little computational capability when compared to desktop computers.
2. The network latency often introduces a huge perceivable delay. Working on a LAN reduces this delay.
3. Each widget is implemented as a dual proxy-renderer entity. The functional core of the widget can often be arbitrarily placed more at the proxy or more at the renderer. This greatly influences the final performance of the widget. Also, the implementation of the proxy can have different degree of reliance on the renderer. In particular, the functional core of the widget can be duplicated at the proxy with a great deal of flexibility: this reduces greatly the amount of work required to create a proxy class, however the dependance introduced is an overhead directly dependent on the network latency.

6.7.1.a Asynchronous speed comparison

First we survey the speed of ETk in a centralized environment, for commands that do not rely on rely on the renderer to complete. We compare the ETk speed with these two other bindings for Mozart:

- The Tk module is a direct object-oriented binding to Tcl/Tk. Commands are sent asynchronously to Tcl/Tk; errors appear at the Tcl/Tk side and are not reported back to the Oz application.
- The QTk module is a high level binding to Tcl/Tk, which supports a mixed declarative/imperative approach for building user interfaces. Commands are sent synchronously to Tcl/Tk; errors at the Tcl/Tk side are reported back to the Oz application.

The benchmark code consists in changing the foreground color of a button widget 20 000 times.

```
declare

Win={New Tk.toplevel tkInit}
Button={New Tk.button tkInit(parent:Win text:"Button")}
{Tk.send pack(Button)}

proc{Loop I}
```

```
   if I<10000 then
      {Button tk(configure fg:"red")}
      {Button tk(configure fg:"black")}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

Tk version

```
declare

[QTk]={Module.link ["x-oz://system/wp/QTk.ozf"]}
Button
Win={QTk.build td(button(handle:Button text:"Button"))}
{Win show}

proc{Loop I}
   if I<10000 then
      {Button set(fg:red)}
      {Button set(fg:black)}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

QTk version

```
declare

[ETkMod]={Module.link ["ETk.ozf"]}
ETk=ETkMod.etk

Button
All={ETk.build window(name:top button(handle:Button text:"Button"))}
{All.top show}

proc{Loop I}
   if I<10000 then
      {Button set(fg:ETk.color.red)}
      {Button set(fg:ETk.color.black)}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

ETk version

The measure times are:

Toolkit	Time in ms
Tk	422
QTk	24 297
EBL/Tk	750

The synchronous approach of QTk drastically slows it down. However in this situation EBL/Tk is still running asynchronously. The overhead introduced by EBL over Tk is 328ms for 20 000 operations.

6.7.1.b Synchronous speed comparison

We repeat the previous test, this time using a command implemented synchronously in EBL/Tk.

```
declare

Win={New Tk.toplevel tkInit}
Canvas={New Tk.canvas tkInit(parent:Win)}
{Tk.send pack(Canvas)}
Item={Canvas tkReturn(create(rect 10 10 100 100) $)}

proc{Loop I}
   if I<10000 then
      {Canvas tk(itemconfigure Item outline:"red")}
      {Canvas tk(itemconfigure Item outline:"black")}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

Tk version

```
declare

[QTk]={Module.link ["x-oz://system/wp/QTk.ozf"]}
Canvas
Win={QTk.build td(canvas(handle:Canvas))}
{Win show}
Item={Canvas create(rectangle 10 10 100 100 handle:$)}

proc{Loop I}
   if I<10000 then
      {Item set(outline:red)}
      {Item set(outline:black)}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

QTk version

```
declare

[ETkMod]={Module.link ["ETk.ozf"]}
ETk=ETkMod.etk

Canvas
All={ETk.build window(name:top canvas(handle:Canvas))}
{All.top show}
Item={Canvas create(rectangle [10.0 10.0 100.0 100.0] handle:$)}

proc{Loop I}
   if I<10000 then
      {Item set(outline:ETk.color.red)}
      {Item set(outline:ETk.color.black)}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

ETk version

The measured times are:

Toolkit	Time in ms
Tk	453
QTk	12 188
EBL/Tk	13 968

The synchronous approach of QTk and EBL/Tk drastically slows them down. The difference in speed with QTk is directly due to the overhead of the proxy-renderer mechanism; here it is 1780ms for 20 000 operations.

6.7.1.c Migration speed

The next speed measurement consists in migrating a single widget in a single process 20 000 times.

```
declare

[ETkMod]={Module.link ["ETk.ozf"]}
ETk=ETkMod.etk

Label
All1={ETk.build window(name:top label(text:"Label" handle:Label))}
{All1.top show}
All2={ETk.build window(name:top)}
{All2.top show}
Ref={Label getRef($)}

proc{Loop I}
```

```
   if I<10000 then
      {All2.top display(Ref)}
      {All1.top display(Ref)}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

To ensure the fault tolerance, the proxy contains the complete state of the widget. As a result, the migration process between two sites consists in dropping the old site, and migrating to the new one. There is no dependency between these two sites, and in particular the old site can be dropped even during the migration process. As a result, the migration process is asynchronous, and the code above is executed in only 203ms.

To have a more interesting figure, we can issue a command that requires that the migration process is completed before migrating away.

```
declare

[ETkMod]={Module.link ["ETk.ozf"]}
ETk=ETkMod.etk

Label
All1={ETk.build window(name:top label(text:"Label" handle:Label))}
{All1.top show}
All2={ETk.build window(name:top)}
{All2.top show}
Ref={Label getRef($)}

proc{Loop I}
   if I<10000 then
      {All2.top display(Ref)}
      {Wait {Label winfo(width:$)}}
      {All1.top display(Ref)}
      {Wait {Label winfo(width:$)}}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

To isolate the time taken by the migration process, we have to remove the time taken by the synchronous operation itself.

```
declare

[ETkMod]={Module.link ["ETk.ozf"]}
ETk=ETkMod.etk

Label
All={ETk.build window(name:top label(text:"Label" handle:Label))}
{All.top show}
```

```
proc{Loop I}
   if I<10000 then
      {Wait {Label winfo(width:$)}}
      {Wait {Label winfo(width:$)}}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

The first test executed in 93 281 ms while the second in 12 844 ms and so it took 80 437 ms to migrate the widget 20 000 times or around 4 ms per migration. Each migration required to destroy the old widget, and create a new one. We can measure the time this takes with the Tk module.

```
declare

Win={New Tk.toplevel tkInit}

proc{Loop I}
   if I<20000 then
      Button={New Tk.button tkInit(parent:Win text:"Button")}
   in
      {Tk.send pack(Button)}
      {Wait {Tk.return update(idletasks)}}
      {Button tkClose}
      {Loop I+1}
   end
end

T1={Property.get 'time.total'}
{Loop 0}
T2={Property.get 'time.total'}
{Show T2-T1}
```

This code executed in 27 313 ms. Consequently the migration overhead introduced by EBL/Tk is 53 124 ms for 20 000 migrations or around 2.6 ms per migration.

In conclusion we argue that EBL is very well fitted with UIs that do not constantly change their state constantly. This covers WIMP desktop applications at large. However this does not cover multimedia applications with video, nor does it cover 3D games where the view is also constantly changing.

6.7.2 Development cost of EBL/Tk

The EBL/Tk toolkit was developed to validate the EBL approach. The code is composed of core proxy and renderer classes that are valid for all widgets supported by Tcl/Tk. These classes are defined in around 800 lines of code. Each widget specialize these classes, to further reflect their functionality at the proxy level. The stores are configured automatically by getting the type information of the widget parameters from Tcl/Tk when compiling EBL/Tk. Consequently, the specialization is often very cheap, for example the button widget is defined in only 33 lines of code. The most complex

widget is the canvas widget which is defined in around 500 lines of code; this code also introduces objects for representing the items in the canvas. The global development effort for EBL/Tk was around a single man/month of work.

Filename	# lines
ETk.oz	833
ETkButton.oz	32
ETkCanvas.oz	510
ETkCheckbutton.oz	82
ETkDialogbox.oz	216
ETkEntry.oz	335
ETkFont.oz	111
ETkImage.oz	283
ETkLabel.oz	44
ETkLabelframe.oz	51
ETkListbox.oz	281
ETkMenubutton.oz	449
ETkMessage.oz	18
ETkMisc.oz	1225
ETkNavigator.oz	404
ETkPanedwindow.oz	256
ETkRadiobutton.oz	116
ETkReceiver.oz	19
ETkScale.oz	92
ETkScrollbar.oz	191
ETkSelector.oz	207
ETkSpinbox.oz	128
ETkTable.oz	150
ETkTest.oz	165
ETkText.oz	136
ETkWindow.oz	191

6.7.3 Development cost of applications using EBL/Tk

Not enough data is available to draw accurate conclusions on the impact of EBL/Tk on the development cost of applications. However we can tell that the EBL/Tk abstraction is pretty similar to the QTk abstraction as they both use a similar mixed declarative/imperative approach, and both use the same low level Tcl/Tk toolkit. The mixed approach greatly reduces the code size required for building UIs, some personal experiments showed a factor of up to 3 times smaller code. Also the code is more readable and easier to maintain.

6.8 Comparative analysis

We will now compare EBL to five other technologies supporting migration and/or adaptation as well as Tcl/Tk that is used by EBL/Tk. We selected seven criteria that place the benefits of EBL into relief. We assess the different technologies according to these criteria, and display the result in Kiviat diagrams for an immediate visual comparison.

6.8.1 Places

This dimension covers the numbers of places where the user interface extends itself. The possible values are:

- Single place, like a single window on a single computer.
- Several places on the same computer, like an application composed of a main window and a floating toolbox.
- Several places on several computers, like an application that has parts of its UI on a laptop, and part of it on a PDA connected wirelessly to the laptop.

6.8.2 Dynamicity of the distribution

This dimension covers the nature of the distribution of the user interface. This point is related to the previous one. The possible values are:

- Stationary, not distributed.
- Fixed remote display, the application runs at one site, the UI appears at another, both sites are fixed for the whole duration of the application.
- Dynamic remote display, the application runs at one site, the UI appears at another which can change at runtime.
- Full dynamic migration, the application can completely migrate from one site to another at runtime.

6.8.3 Control

This dimension covers who is in charge of the migration/remote display ability. The possible values are:

- No control, there is no migration/adaptation possible.
- External control, the mechanism is provided outside the application which cannot control it in any way.
- Application control, the mechanism is controlled by the application itself.

6.8.4 Reproduction

This dimension covers the possibility of the UI to be physically present several times. This point is related to the previous one. The possible values are:

- One and only one incarnation at all time, the elements of the UI are restricted to be physically present once and only once.
- Many incarnations at the same time, individual elements of the UI can exist several times concurrently.

6.8.5 Users

This dimension covers the concurrent number of users of an application. The possible values are:

- Single user.
- Simple multi-user interfaces like in this thesis.
- Real multi-user support.

6.8.6 Adaptation

This dimension covers the ability of adapting the user interface. This point is related to the previous one. The possible values are:

- No adaptation.
- Widget level adaptation, different representations are available for the widgets, and the adaptation mechanism selects the best one depending on the situation.
- Global level adaptation, there exists an adaptation model for the complete UI that allows to calculate different representations according to different situations. This approach is quite disruptive with usual UI programming techniques, as it requires a model-based approach instead of the more classical imperative direct approach.

6.8.7 Granularity level

This dimension covers the level of granularity for splitting the UI among different places. The possible values are:

- Complete UI, the UI cannot be split.
- Widget level, the UI can be split at the individual widget's level. Usually, it's the logical functionality that is reproduced on remote sites (for example a remote text entry uses the local text entry widget of the distant site, ie it's the logical functionality of the text entry that is shared between the sites).
- Pixel level, the UI can be split at any pixel level. Usually it's the physical incarnation of the application's home site that is mirrored at the remote site.

6.8.8 Relation of the criteria with migration, adaptation, and multi-user functionality

	Places	Distribution	Control	Reproduction	Users	Adaptation	Granularity
Migration	✓	✓	✓				✓
Adaptation			✓			✓	✓
Multi-user	✓	✓	✓	✓	✓		

6.8.9 Comparison charts

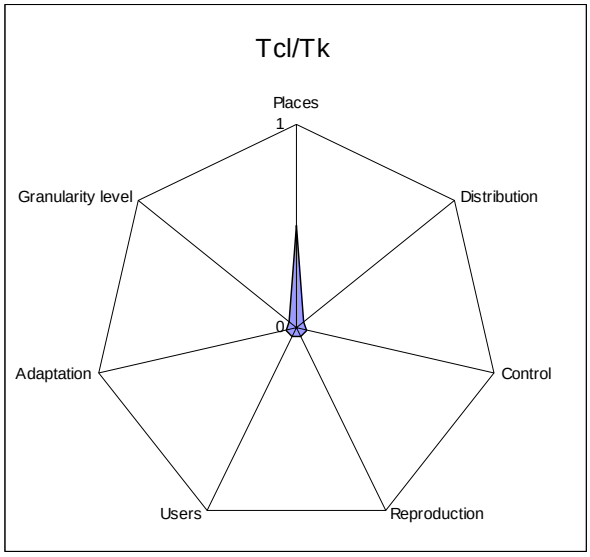
This section presents visual representations for the categorization of different tools. This representation is to be interpreted as a convenient way for grasping the capabilities of these tools, but not as a scientific tool of measure. In particular, the area covered by the

tools does not have any particular unit, and should not be used as a measure of anything. Also the values used on each axis are arbitrary as they represent discrete capabilities in a continuous representation. Nevertheless, we think this representation is a convenient way of graphically showing the tools capabilities. The criteria are related to the functionality EBL adds, which is why it fares well in the comparison. Depending on the usage one has of a particular toolkit, other criteria may apply, where EBL may not fare well.

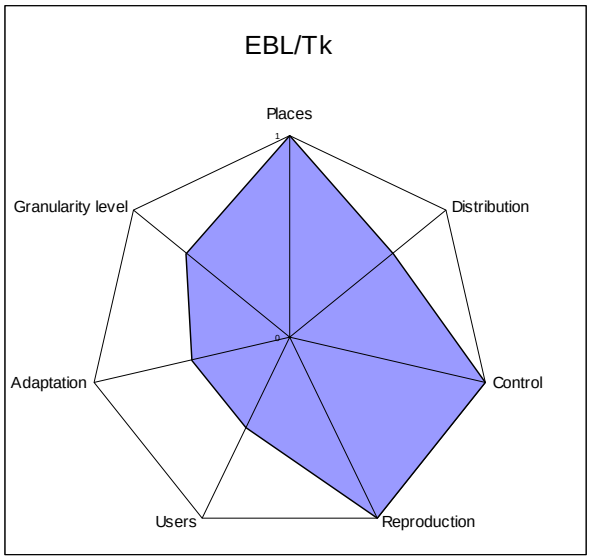
We use these values for the categorizations:

Places	0,05=Single, 0,5=many on single computer, 1=many
Users	0,05=single, 0,5=simple, 1=full multi-user support
Distribution	0,05=stationary, 0,33=fixed remote UI, 0,66=dynamic remote UI, 1=full appli migration
Reproduction	0,05=single, 1=many
Granularity level	0,05=NA, 0,33=full UI, 0,66=widget level, 1=pixel level
Adaptation	0,05=none, 0,5=widget level, 1=model based
Control	0,05=NA, 0,5=external, 1=application

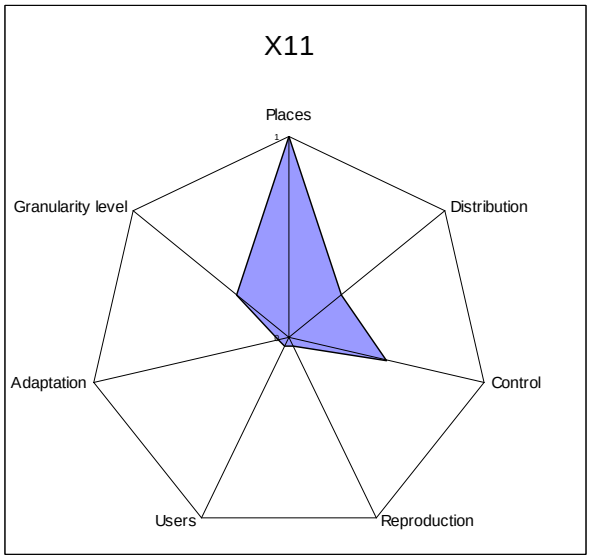
Tcl/Tk allows the opening of several windows, besides that it doesn't provide any support for remote display/migration/adaptation/multiple users.



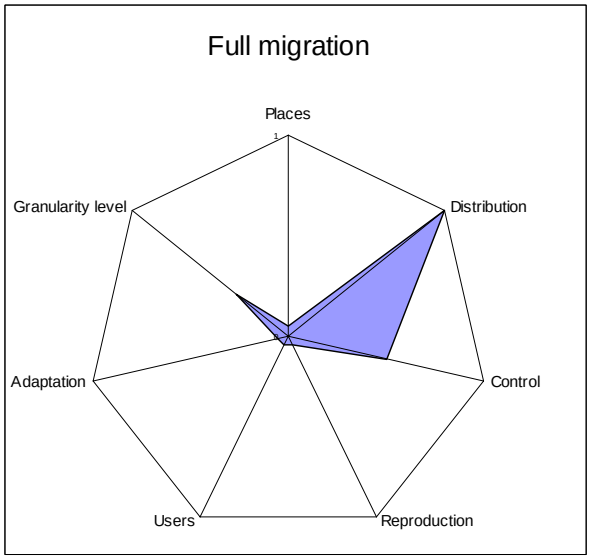
EBL/Tk offers a complete application control for migration and adaptation at the widget level. Also widgets can be rendered simultaneously at different places for a basic multiple users support.



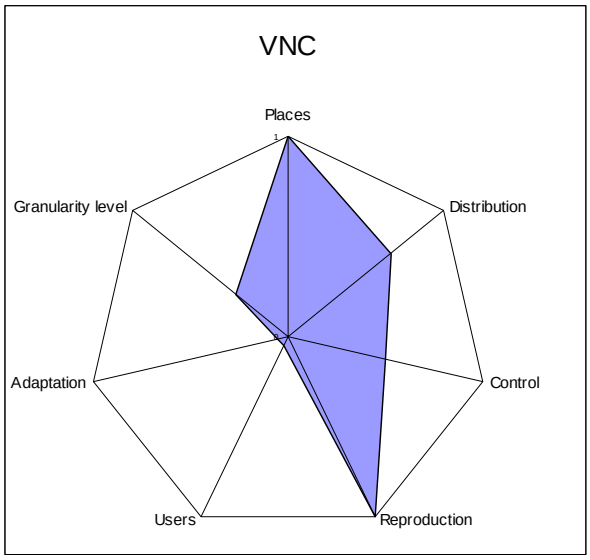
X11 provides a remote display mechanism external to the application. Once the application is started, the UI cannot be migrated away from where it is displayed. Also there is no adaptation nor multi-user functionality.



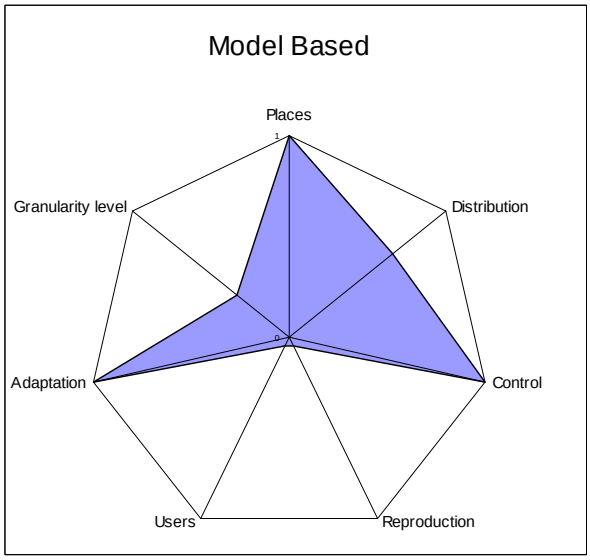
Systems allowing a complete dynamic migration of a running application do not provide any benefit regarding adaptation and multi-user support. Note that the UI is situated at a single device at a time, hence the score for places.



VNC allows controlling remotely another computer, by mirroring exactly the remote display at the pixel level. It's possible to remote control several times the same computer at the same time, but it doesn't introduce real multiple user possibilities. Once again, there is no adaptation possible.



The model based approach is a very different approach for creating a user interface. All aspects are described using high level models, and the system is capable to dynamically infer a concrete user interface from these models and the current context. This graph corresponds to a model-based approach that provides adaptation and migration, but no multi-user support. Other model-based approaches may exist.



Chapter 7 Conclusion

7.1 Summary of results

The EBL middleware was developed to provide migration, adaptation, and simple multi-user capability to an effective toolkit. The EBL/Tk toolkit was created on top of EBL using the Tcl/Tk back-end toolkit. This toolkit is functionally equivalent to Tcl/Tk; any UI created with Tcl/Tk can be implemented using EBL/Tk. Further the extra functionality provided by EBL is orthogonal to the basic toolkit functionality; there is no penalty in term of development cost in using EBL/Tk instead of directly using Tcl/Tk. On the contrary, the mixed declarative/imperative approach of EBL/Tk often reduces the development cost. Any UI created by EBL/Tk is migratable by default; this migration is *transparent* to the application that created in the sense that a) the running of the application is independent on the physical location of the widget and b) the migration mechanism is independent on the concurrent execution of the application. All widgets can have several different physical representations, and they can change between them at runtime. The adaptation mechanism is similarly *transparent* to the application. Finally widgets can be displayed several times simultaneously, offering a simple multi-user functionality. The flexibility of the *transparent* migration/adaptation allows adding migration and adaptation as an afterthought to a stationary/non adaptable UI. This allows the rapid prototyping of migratable/adaptable applications.

7.1.1 Contributions

This thesis contributed the following:

- Idea of the migration as a widget capability and its validation.
- Idea of the adaptation as a special migration case and its validation.
- Idea of simple multi-user interfaces as a special migration case and its validation.
- Specification and implementation of a middleware supporting these ideas.
- Implementation of an actual, usable module for use by the Mozart community.

In particular, we know no other work providing a ready to use toolkit supporting transparent migration and adaptation of any part of the UI.

7.2 Future work

7.2.1 Hybrid approach

In papers predating this work [Grol01] [Grol02] [Elan04], we explored the idea of mixing together an imperative approach and a declarative one to the creation and running of UIs, in the context of a multi-paradigm programming language. The declarative part uses a model-based approach for the specification of the presentation and (part of) the dialog models. The dynamic behavior of the UI is then implemented using a classical object-oriented imperative approach.

Compared to a pure imperative approach, we observed that this hybrid approach greatly reduces the amount of work required for the creation of the UI. Also as we are in the context of a multi-paradigm programming language that supports functional programming, we are able to dynamically calculate the content of a UI by writing a function, a task that is hard to achieve using a pure imperative approach.

Compared to a pure model-based approach, this hybrid approach does not benefit from the design and verification tools generally available to model-based approaches. However, the hybrid approach does not suffer from any expressivity restriction as the imperative guarantees the Turing completeness. This is often not the case with model-based approaches that are domain specific, and may not support all future new requirements, and/or may introduce a lot of verbosity when going to the limit of its expressivity.

EBL provides the migration/adaptation/simple multi-user functionality, but also provides a support for the hybrid approach. With the support of the multi-paradigm programming language, it is possible to switch between a more declarative and a more imperative world constantly, as best fits the situation at hand. We think this yield a very interesting way of quickly developing UI, in general for prototyping purposes. The methodology would be to develop application-specific models for all parts of the functional core that are easily expressible into models, and stick to an imperative approach for the rest. These models would not benefit from the design and validation tools of more general domain-specific models, nevertheless their expressivity can be made as to the point as possible, in a as compact as possible way.

One possible direction for future work is to further explore this methodology with EBL in the context of migratable/adaptatble/multi-user applications.

7.2.2 Tool extensions

EBL/Tk is available in version 1.0. The implementation is fully functional granted these limitations:

- The independent adaptation of multiple renderers connected to a single proxy is not supported. This is a limitation of the current implementation that could be easily removed.
- The current EBL implementation does not support several toolkits to be concurrently used. This limitation can be removed by introducing a negotiation phase during the migration protocol so that the renderer class is also selected according to the toolkit offered by the receiving container. In other words, the widget would transparently adapt to the toolkit offered by its container.
- The security aspect of the migration is only partially managed; it should be extended to support real world situations. To achieve proper security, all communication channels should be encrypted. Further, the migration capabilities should also be encrypted to prevent attackers from reading them over the network and then using them. This security can already be achieved by using a virtual private network or SSH tunneling. Embedding the encryption directly into EBL requires some more work.
- The end user module (EBL/Tk) uses Tcl/Tk as the actual toolkit which is quite aging compared to more recent toolkits. We intend on replacing Tcl/Tk by a

newer toolkit like GTk. Also we would like to use the Ajax and DHTML techniques for creating a toolkit running in a web browser.

7.2.3 Future exploration areas

The simple multi-user aspect of EBL is largely unexplored. It appeared as a side product of the migration/adaptation design; still it has an interesting research potential in itself. The concurrent use of a single widget by multiple users raise questions on the consistency of the state of the widget, and how this is reflected to the user. Should the users be restricted in their concurrent interactions, so as to prevent them for executing contradictory changes? How do we manage that? What support can EBL offer? What methodology is best adapted for designing such widgets, and then what methodology is best adapted for creating applications that use them?

Further research is also needed to assess the limits of this approach for more complex migratable/adaptable applications. In particular, the ergonomic aspects of migratable/adaptable applications will have an impact on the methodology used to create them. I personally think that the EBL approach is helpful in dealing with this problem, thanks to 1) the transparent aspect of the migration and the adaptation, and 2) the hybrid approach that allows an easy integration with model-based design. However further research is required to verify that.

7.3 Concluding remarks

This work is mainly an *enabler*: it gives a new functionality not previously available to applications. This functionality has a disruptive potential, in the sense that it could yield to new ways of thinking and developing applications, or even to completely new kinds of applications. However this functionality is provided to the developers in a *non* disruptive fashion: if you don't use them, they don't come into the way. And you can use them as an afterthought of an already developed application. Hopefully this thesis convinced you that this goal is achieved.

References

A

B

[Band03]
Bandelloni, R., Paternò, F., *Platform Awareness in Dynamic Web User Interfaces Migration*, Proceedings of International Conference on Human-Computer Interaction with Mobile and Handlheld Devices MobileHCI'2003.

[Bass92]
Len Bass and UIMS Tool Developers Workshop, *A Metamodel for the Runtime Architecture of an Interactive System*, SIGCHI Bulletin, 24(1), 32-37. 1992.

[Berti05]
Berti, S., Paternò, F., and Santoro, C., *A Taxonomy for Migratory User Interfaces*, in Proc. of 12th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2005 (Newcastle upon Tyne, 13-15 July 2005), M. Harrison (ed.), Lecture Notes in Computer Science, Vol. 3941, Springer-Verlag, Berlin, 2005.

[Bahr95]
Bharat, K.A., Cardelli, L., *Migratory Applications Distributed User Interfaces*, Proceedings of ACM Conference on User Interface Software Technology UIST'95 (Snowbird, October 1995), ACM Press, New York, 1995, pp. 133–142.

[Blanch05a]
Renaud Blanch, Michel Beaudouin-Lafon, Stéphane Conversy, Yannick Jestin, Thomas Baudel and Yun Peng Zhao, *INDIGO : une architecture pour la conception d'applications graphiques interactives distribués*, in Proceedings of IHM 2005, Toulouse - France, September 2005.

[Blanch05b]
Renaud Blanch, *Facilitating post-WIMP Interaction Programming using the Hierarchical State Machine Toolkit*. Rapport de Recherche 1410, Laboratoire de Recherche en Informatique, Université Paris-Sud, France, April 2005.

[Brai74]
Brainerd, W.S., Landweber, L.H., *Theory of Computation*, Wiley, 1974.

C

[Calv00]
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.

[Calv01]
Calvary, G., Coutaz, J., Thevenin, D., *A Unifying Reference Framework for the Development of Plastic User Interfaces*, Proceedings of 8th IFIP International Conference on Engineering for Human-Computer Interaction EHCI'2001 (Toronto, May 11-13, 2001), Little, R., Nigay, L. (eds.), Lecture Notes in Computer Science, Vol. 2254, Springer-Verlag, Berlin, 2001, pp. 173–192.

[Calv02]
Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., Vanderdonckt, J., *Plasticity of User Interfaces: A Revised Reference Framework*, Proceedings of 1st Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2002 (Bucharest, July 18-19, 2002), Academy of Economic Studies of Bucharest, INFOREC Printing House, Bucharest, 2002, pp. 127–134.

[Calv04]

References

Calvary, G., Coutaz, J., Daassi, O., Balme, L., Demeure, A., *Towards a new generation of widgets for supporting software plasticity: the 'comet'*, Proceedings of the 9th IFIP International Working Conference on Engineering for Human-Computer Interaction, Jointly with the 11th International Workshop on Design, Specification and Verification of Interactive Systems EHCI-DSVIS'04 (Hamburg, July 11-13, 2004).

[Calv05]
Calvary, G., Daassi, O., Coutaz, J., Demeure, A., *Des widgets aux comets pour la Plasticité des Systèmes Interactifs*, Revue d'Interaction Homme-Machine, 6(1), 2005, pp. 33–53.

[Chae04]
Chae, M.W., Kim, J.W., *Do size and structure matter to mobile users? An empirical study of the effects of screen size, information structure, and task complexity on user activities with standard web phones*, Behaviour & Information Technology, 23(3), 2004, pp. 165–181.

[Chu04]
Chu, H., Song, H., Wong, C., Kurakake, S., Katagiri, M., *Roam, a seamless application framework*, Journal of Systems and Software, 69(3), 2004, pp. 209–226.

[Cout00]
Coutaz, J., Lachenal, C., Calvary, G., Thevenin, D., *Software Architecture Adaptivity for Multisurface Interaction and Plasticity*, Proc. of IFIP Workshop on Software Architecture Requirements for CSCW–CSCW'2000 Workshop (Philadelphia, December 2-6, 2000), ACM Press, New York, 2000. Accessible at http://iihm.imag.fr/coutaz/ifipcscw2000/workshop.html

[Cout03a]
Coutaz, J., Balme, L., Lachenal, Ch., Barralon, N., Software Infrastructure for Distributed Migratable User Interfaces, Proceedings of UbiHCISys Workshop on UbiComp 2003, 2003.

[Cout03b]
Coutaz, J., Lachenal, Ch., Dupuy-Chessa, S., *Ontology for Multi-Surface Interaction*, Proceedings of 9th IFIP TC 13 International Conference on Human-Computer Interaction INTERACT'2003 (Zurich, September 1-5, 2003), Rauterberg, M., Menozzi, M., Wesson, J. (eds.), IOS Press, Amsterdam, 2003.

[Cout06]
Coutaz, J., *Meta-User Interface for Ambient Spaces*, invited talk of the 5th Internal Workshop on Task, Models and Diagrams for UI design TAMODIA'2006 (Hasselt, October 23-24, 2006).

[Crea00]
Crease, M., Gray, P., Brewster, S., *A Toolkit of Mechanism and Context Independent Widgets*, Proceedings of the International Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2000 (Limerick, June 5-6, 2000), Springer-Verlag, Berlin, 2000.

[Crea01]
Crease, M., A Toolkit of Resource-Sensitive Multimodal Widgets, Ph.D. Thesis, Department of Computing Science, University of Glasgow, December 2001.

D

[Dâas03]
Dâassi, O., Calvary, G., Coutaz, J., Demeure, A., *Comet : Une nouvelle génération de « Widget » pour la Plasticité des Interfaces*, Actes de la 15ème Conférence Francophone sur l'Interaction Homme-Machine IHM'2003 (Caen, November 2003), Hermès, Paris, 2003, pp. 64–71.

[Deme05]
Demeure, A., Calvary, G., Sottet, J.-B., Ganneau, V., Vanderdonckt, J., *A Reference Model for Distributed User Interfaces*, Proceedings of 4th International Workshop on Task Models and Diagrams for user interface design TAMODIA'2005 (Gdansk, 26-27 September 2005), ACM Press, New York, 2005, pp. 79–86.

[Deme06a]
Demeure, A., Calvary, G., Coutaz, J., Vanderdonckt, J., *The Comets Inspector: Manipulating Multiple User Interface Representations Simultaneously*, Proceedings of 6th International Conference on Computer-Aided De-sign of User Interfaces CADUI'2006 (Bucharest, June 6-8, 2006), Chapter 13, Springer-Verlag, Berlin, 2006, pp. 167–174.

[Deme06b]
Demeure, A., Calvary, G., Coutaz, J., Vanderdonckt, J., *The Comets Inspector: Towards Run Time Plas-*

References

ticity Control based on a Semantic Network, Proceedings of 5th Int. Workshop on Task Models and Diagrams for User Interface Design TAMODIA'2006 (Hasselt, October 23-24, 2006), Coninx, K., Luyten, K., Schneider, K. (eds.), EDM-Luc, Hasselt, 2006, pp. 199–206.

[Dewa00]
Dewan, P., Choudhary, R., *Coupling the User Interfaces of a Multiuser Program*, ACM Transactions on Computer-Human Interaction, 2(1), 2000, pp. 1–39.

[Dey03]
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.

E

[Elan04]
El-Ansary, S., Grolaux, D., Van Roy, P., Rafea, M., *Overcoming the Multiplicity of Languages and Technologies for Web-based Development Using a Multiparadigm Approach*, Proceedings of 2nd International Mozart/Oz Conference MOZ'2004 (Charleroi, October 2004), Lecture Notes in Computer Science, Volume 3389, Springer-Verlag, Berlin, 2004, pp. ?. Accessible on-line at http://www.cetic.be/moz2004.

[Eise00]
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.

[Eise01]
Eisenstein, J., Vanderdonckt, J., Puerta, A., *Applying model-based techniques to the development of UIs for mobile computers*, Proceedings of the 6th International ACM Conference on Intelligent User Interfaces IUI'2001 (Santa Fe, January 14-17, 2001), ACM Press, New York, 2001.

[Eise03]
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.

[ETk]
The EBL/Tk homepage, accessible at http://gforge.info.ucl.ac.be/projects/ebl/.

F

G

[Grol01a]
Grolaux, D., Van Roy, P., Vanderdonckt, J., *QTk: A Mixed Model-Based Approach to Designing Executable User Interfaces*, Proceedings of 8th IFIP Working Conf. on Engineering for Human-Computer Interaction EHCI'2001 (Toronto, May 11-13, 2001), Lecture Notes in Computer Science, Vol. 2254, Springer-Verlag, Berlin, 2001, pp. 109–110. Accessible at ftp://ftp.info.ucl.ac.be/pub/publi/2001/ehci2001final.ps.gz or at http://www.mozart-oz.org/papers/abstracts/ehci2001.html

[Grol02]
Grolaux, D., Van Roy, P., Vanderdonckt, J., *FlexClock, a Plastic Clock Written in Oz with the QTk toolkit*, Proceedings of 1st Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2002 (Bucharest, 18-19 July 2002), Academy of Economic Studies of Bucharest, INFOREC Printing House, Bucharest, 2002, pp. 135–142.

[Grol04]
Grolaux, D., Van Roy, P., Vanderdonckt, J., *Migratable User Interfaces: Beyond Migratory User Interfaces*, Proceedings of 1st IEEE-ACM Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services MOBIQUITOUS'04 (Boston, August 22-25, 2004), IEEE Computer Society Press, Los Alamitos, 2004, pp. 422–430.

[Grud01]
Grudin, J., *Partitioning digital worlds: focal and peripheral awareness in multiple monitor use*, Proc. of ACM

References

Conference on Human Aspects in Computing Systems CHI'2001 (New Orleans, April 2001), ACM Press, New York, 2001, pp. 458–465.

[Guim01]
Guimbretière, F., Stone, M., Winograd, T., *Fluid Interaction with High-resolution Wall-size Displays*, Proceedings of ACM International Conference on User Interface Software Technology UIST'2001, ACM Press, New York, 2001, pp. ? –?.

H

[Hart93]
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.

J

K

[Kras88]
Krasner, G . E . & S . T. Pope, *A cookbook for using model-view-controller user interface paradigm in Smalltalk-80*, Journal of Object Oriented Programming, 1,3, 26-49, 1988.

L

[Leler90]
W. Leler, *Linda Meets Unix*, Computer, vol. 23, no. 2, pp. 43-54, Feb., 1990

[Li03]
Li, B., Tsai, W.-T., and Zhang, L.-J., *A Semantic Framework for Distributed Applications*, Proc. of the 5th Int. Conf. on Enterprise Information Systems ICEIS'2003 (Angers, 22-26 April 2003), Volume IV - Software Agents and Internet Computing, pp. 34-41.

[Limb04a]
Limbourg, Q., Vanderdonckt, J., *UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence*, in Matera, M., Comai, S. (Eds.), "Engineering Advanced Web Applications", Rinton Press, Paramus, 2004, pp. 325–338.

[Luyt05]
Luyten, K., Vandervelpen, Ch., and Coninx, K., *Task Modeling for Ambient Intelligent Environments: Design Support for Situated Task Executions*, Proc. of 4th Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2005 (Gdansk, 26-27 September 2005), ACM Press, New York, 2005, pp. 87-94.

M

[Moli05]
Molina, J.P., Vanderdonckt, J., Montero, F., Gonzalez, P., *Towards Virtualization of User Inter-faces based on UsiXML*, Proceedings of 10th ACM International Conference on 3D Web Technology Web3D'2005 (Bangor, March 29-April 1, 2005), ACM Press, New York, 2005, pp. 169-178.

[Moli06b]
Molina, J.P., Vanderdonckt, J., González, P., Fernández-Caballero, A., Lozano, M.D., *Rapid Prototyping of Distributed User Interfaces*, Proceedings of 6th International Conference on Computer-Aided Design of User Interfaces CADUI'2006 (Bucharest, 6-8 June 2006), Chapter 12, Springer-Verlag, Berlin, 2006, pp. 151–166.

[Moza]
The Mozart Programming System, accessible at http://www.mozart-oz.org/

[Myer04]

References

Myers, B.A., Nichols, J., Wobbrock, J.O., Miller, R.C., *Taking Handheld Devices to the Next Level*, IEEE Computer, 37(12), December 2004, pp. 36–43.

N

O

[Oust94]
Ousterhout, J., *Tcl and TK Toolkit*, Addison-Wesley, Reading, 1994.

P

[Paus92]
Pausch, R., Conway, M., DeLin, R., *Lesson Learned from SUIT, the Simple User Interface Toolkit*, ACM Transactions on Information Systems, 10(4), 1992, pp. 320–344.

[Pier04]
Pierce, J.S., Mahaney, H.E., *Opportunistic Annexing for Handheld Devices: Opportunities and Challenges*, Proceedings of Human-Computer Interface Consortium 2004, accessible on-line at: http://www-static.cc.gatech.edu/~jpierce/papers/OA-HCIC2004.pdf

[P2PS]
Peer to peer library available at http://gforge.info.ucl.ac.be/projects/p2ps/ .

[Puer97]
Puerta, A.R., *A model-based interface development environment*, IEEE Software, 14(4), 1997, pp. 40-47.

[Puer99]
Puerta, A.R., Eisenstein, J., *Towards a General Computational Framework for Model-Based Interface Development Systems*, Proceedings of the 4th ACM International Conference on Intelligent User Interfaces IUI'1999 (Los Angeles, January 5-8, 1998), ACM Press, New York, 1999.

Q

[QTk]
Donatien Grolaux, *QTk - Graphical User Interface Design for Oz*, http://www.mozart-oz.org/documentation/mozart-stdlib/wp/qtk/html/index.html

R

[Reki97]
Rekimoto, J., *Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments*, Proc. of ACM International Conference on User Interface Software Technology UIST'97 (Banff, Alberta, Canada, October 14-17, 1997), ACM Press, New York, 1997, pp. 31–39.

[Roud06]
Roudaut, A., Coutaz, J., *Méta-IHM ou comment contrôler l'espace interactif ambient*, Actes des Troisièmes Journées Francophones sur la Mobilité et l'Ubiquité UBIMOB'2006 (Paris, September 5-8, 2006), ACM Press, New York, 2006.

S

[Shne83]
Shneiderman, B., *Direct manipulation: A step beyond programming languages*, IEEE Computer, 16(8), 1983, pp. 57–69.

[Sous02]
Sousa, J., Garlan, D., *Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments*, Proceedings of IEEE-IFIP International Conference on Software Architecture 2002.

[Stre99]
Streitz, N., et al., *i-LAND: An interactive Landscape for Creativity and Innovation*, Proceedings of ACM International Conference on Human Factors in Computing Systems CHI'99 (Los Angeles, April

References

1999), ACM Press, New York, 1999, pp. 120–127.

[Szek95]
Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E., *Declarative interface models for user interface construction tools: the MASTERMIND approach*, Proceedings of the IFIP Working Conference on Engineering for Human-Computer Interaction EHCI'1895 (Grand Targhee, August 14-18, 2005).

[Szek96]
Szekely, P., *Retrospective and Challenges for Model-Based Interface Development*, Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'1996 (Namur, June 5-7, 1996), Vanderdonckt, J. (ed.), Presses Universitaires de Namur, Namur, 1996, pp. xxi-xliv.

T

[Tan03a]
Tan, D.S., Czerwinski, M., *Effects of Visual Separation and Physical Discontinuities when Distributing Information across Multiple Displays*, Proceedings of 9th IFIP TC 13 International Conference on Human-Computer Interaction INTERACT'2003 (Zurich, September 1-5, 2003), Rauterberg, M., Menozzi, M., Wesson, J. (eds.), IOS Press, Amsterdam, 2003.

[Tand01]
Tandler, P., et al., *ConnecTables: Dynamic coupling of displays for the flexible creation of shared workspaces*, Proceedings of ACM International Conference on User Interface Software Technology UIST'2001, ACM Press, New York, 2001, pp. 11–20.

[Thev99]
Thevenin, D., Coutaz, J., *Plasticity of user interfaces: Framework and research agenda*, Proceedings of the 7th International Conference on Human-Computer Interaction INTERACT'99 (Edinburgh, August 30-September 3, 1999), IOS Press, Amsterdam, 1999, pp. 110–117.

[Thev01]
Thevenin, D., *Adaptation en Interaction Homme-Machine: le cas de la Plasticité*, Ph.D. thesis, Université Joseph Fourrier, Grenoble, 2001.

[TkMod]
Christian Schulte, *Window Programming in Mozart*, http://www.mozart-oz.org/documentation/wp/index.html

U

V

[Vand05a]
Vanderdonckt, J., *A MDA-Compliant Environment for Developing User Interfaces of Information Systems*, Proc. of 17th Conf. on Advanced Information Systems Engineering CAiSE'05 (Porto, June 13-17, 2005), Pastor, O., Falcão e Cunha, J. (eds.), Lecture Notes in Computer Science, Vol. 3520, Springer-Verlag, Berlin, 2005, pp. 16–31.

[Vanr04]
Van Roy, P., Haridi, S., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, hardcover, 900pp+xxix, ISBN 0-262-22069-5, March 2004.

[Vnc]
Virtual Network Computing, accessible at http://www.uk.research.att.com/vnc/.

W

[Weis91]
Weiser, M., *The Computer for the Twenty-First Century*, Scientific American, September 1991, pp. 94-10.

[Weis93]
Weiser, M., *Some Computer Science Issues in Ubiquitous Computing*, Communications of the ACM, 36(7), 1993, pp. 75–84.

[Wts]

References

Windows Terminal Server, accessible at http://www.microsoft.com/windows2000/technologies/terminal/default.asp.

X

[X11] The X11 Consortium, accessible at http://www.x.org/.

Y

Z

Annex A: EBL reference

This is the reference of the EBL middleware which provides a factory for binding a graphical toolkit in Mozart. An example of such toolkit is EBL/Tk which is documented at http://gforge.info.ucl.ac.be/plugins/wiki/index.php?id=24&type=g.

EBL Module

- NewWidgetRepository
  This zero parameter function creates a new widget repository. A widget repository stores proxy widget definitions, the build function, and more. Several widget repositories can coexist simultaneously, each of them having their own set of proxies. Note that renderer definitions are global to all widget repositories, beware of name clashes. This function returns a record with the following features:
  - register
    The first parameter of this procedure is a proxy class definition. The second parameter is the procedure to call when creating the widget from a description record. The proxy is registered under the name provided by the widgetName feature of its class.
  - registerAs
    The first parameter is the name under which this widget proxy must be registered. The second parameter is a proxy class definition. The third parameter is the procedure to call when creating the widget from a description record.
  - registerAlias
    The first parameter is the name under which this widget must be registered. The second parameter is a unary function which parameter is the description record used to create this widget. The function must return another record to use instead of the original one. For example registerAlias could be used to create an alias called predeflabel, and the function replaces the description record for predeflabel by a label with some predefined parameter values.
  - getWidgets
    This zero parameter function returns the names of the proxy widgets defined in this repository.
  - getWidgetClass
    Same as getProxyClass
  - getProxyClass
    This one parameter function returns the proxy class corresponding to the parameter.
  - getBuildFun
    This one parameter function returns the procedure to call when creating the widget named after the parameter.
  - setGathererClass

The build function returns a gatherer object that centralizes information regarding all the widgets created in this build. The single parameter of this procedure is a class that further specialize the class of the gatherer object. The final gatherer object returned by a build function has features corresponding to the name features defined in the construction record. It also has features that are Oz names, one per widget created. It also supports these methods:

- getAllIds(?L): binds L to a list of all the Oz names corresponding to the widgets created by build.
- getAllItems(?L): binds L to a list of all the proxy objects created by build.
- getAllNames(?L): binds L to a list of all the names specified in the construction record.
- getDefaultLook(?L): binds L to the default look given to all the widgets created.
- restoreInitialGeometry: places all the widgets back into the position were they were first placed by the construction record.

Note that when using the build method of the EBL renderer manager, the gatherer class is **never** specialized according to the setGathererClass function.

- setRenderContextClass
  Same as the SetRenderContextClass below
- build
  The build function takes a construction record as input, creates the corresponding user interface, and returns the gatherer object.
- defaultLook
  A look that is used by default for all widgets created by this repository.
- getRenderClass
  A two parameter function that returns the renderer class corresponding to the widget named after the first parameter, in the context named after the second parameter.

- SetRenderContextClass
  A four parameter procedure that registerers a renderer class. The first parameter is the toolkit identity, the second is the widget name, the third is the context for this renderer, and the last is the renderer class. The toolkit identity is an Oz name that corresponds to a particular back-end graphical toolkit. A container renderer will accept a child renderer if and only if their toolkit identities are the same.
- NewEBLProxyManager
  This zero parameter functions returns a proxy manager. See below.
- NewLook
  This zero parameter functions returns a look. A look is a special dictionary that allows specifying default parameter values for widgets. This is a supplementary functionality provided by EBL that is independent of the migration/adaption/simple multi-user one. A look is a record composed of the features:
  - set

The single parameter of this procedure specifies a record whose label corresponds to a widget, and the features to default values for the corresponding parameters of the widget.

  - get
    The single parameter of this function is the name of a widget; it returns the current record defining the default value for this widget.
  - getWidgetLook
    The single parameter of this function is the name of a widget; it returns the current widget look defined for this widget (see below).

- NewWidgetLook
  A widget look if the part of a look that focuses on a single widget. The NewWidgetLook function is a zero parameter function that returns a record whose features are:
  - set
    The single parameter of this procedure specifies a record whose features correspond to default values for the corresponding parameters of this widget.
  - get
    This zero parameter function returns the current record defining the default value for this widget.
  - register
    The parameter of this function specifies an Oz port. Each time the look of this widget is set, the new record is sent on this port. The function returns a record with a single feature named unregister, which is a zero parameter functions that stops the updates from being sent on the port.
- NewRadioListeners
  Radiobutton widgets are grouped together so that a single radiobutton can be selected at a time. In the EBL context, radiobuttons of the same groupe can be migrated into different devices, so we cannot rely on the back-end toolkit synchronisation mechanism for radiobuttons. The NewRadioListeners function is a zero parameter function that creates a structure for synchronizing radiobuttons (and assimilated widgets). The function returns a record with the following features:
  - register
    This procedure registers a widget in a group. The first parameter is an Oz name that specifies a group (widgets registering using the same name will be part of the same group). The second parameter is a reference to the widget proxy. The third parameter is a zero parameter procedure that is called when the widget must be deselected. The selection of the widget must be implemented by the widget itself, it is not specified in the RadioListener.
  - unregister
    This procedure removes a widget from a group. The first parameter is an Oz name that specifies the group; the second parameter is the reference to the widget proxy.
  - setActive

When a radiobutton is selected by the user, the proxy of the widget must make sure that first the widget appears selected, and two that the setActive procedure is called. The first parameter is the name of the group; the second parameter is the reference of the proxy widget calling the setActive procedure. This procedure makes sure that all the other widgets in this group are deselected.

- AddLookSupport
  This function takes a proxy class definition as the single parameter, and returns a specialization of that class that adds look support. This is a supplementary functionality provided by EBL that is independent of the migration, adaptation, and simple multi-user one. This function assumes that the class given as parameter supports the method set(K V) for setting the value of the parameter K to V, unset(K) for restoring the parameter K to its default value, init(...) for initiliazing the object, and destroy for finalizing it. The returned class adds the setLook(L) method that can be called to specify a look L to use for this widget.
- AddSynonymSupport
  This function takes a proxy class definition as the single parameter, and returns a specialization of that class that adds synonym support for the parameters of the widget. This is a supplementary functionality provided by EBL that is independent of the migration, adaptation, and simple multi-user one. This function assumes that the class given as parameter supports the method set(K V) for setting the value of the parameter K to V, unset(K) for restoring the parameter K to its default value, init(...) for initiliazing the object, and destroy for finalizing it. The returned class is so that the set(K V) method also works for synonyms of K. The synonyms are specified by the synonyms feature of the class, as a record where feature#value pairs specify synonym#realParameterName combinations.
- AddMultiSetGetSupport
  This function takes a proxy class definition as the single parameter, and returns a specialization of that class that adds multiple parameter setting and getting in a single method call. This is a supplementary functionality provided by EBL that is independent of the migration, adaptation, and simple multi-user one. This function assumes that the class given as parameter supports the method set(K V) for setting the value of the parameter K to V, and get(K ?V) for getting the current value for parameter K. The returned class supports set(feat1:Val1 ... featX:ValX) and get(feat1:Val1 ... featX:ValX) methods for setting and getting all the parameters feat1, ..., featX in a single method call.
- CreateWidgetClass
  Combines AddLookSupport, AddSynonymSupport and AddMultiSetGetSupport, and register the default renderer class, and also other renderers if provided.
- NewPublisher
  This function creates a simple discovery service for passing references to remote sites. This service is based on a local server that binds on the port specified as the single parameter of the NewPublisher function. This function returns a record where the features are:
  - getIP
    This zero parameter function returns the IP address of this device.

  - getPN
    This zero parameter function returns the port number this publisher is bound to.
  - close
    This zero parameter procedure closes this publisher server.
  - subscribe
    The first parameter of this procedure is a unique key for the item to subscribe. The second parameter is the item to subscribe (generally the reference of a widget). The third parameter is a string describing this item.
  - unsubscribe
    The single parameter of this procedure is the key of the item that should not be provided by this publisher server anymore.
- GetFromPublisher
  This function takes a string representing an IP adress as first parameter, and a port number as second parameter. If a publisher server is responding at this adress, the function returns a list of Items#Description, otherwise it raises an exception.

EBL Construction Record Procedure

The build function takes a description record and creates the corresponding user interface. The name of the record determines the type of the widget. An instance of the proxy is created, and then the corresponding procedure specified in register or registerAs is called. This procedure is given a single parameter that is a record with the following features:

- build
  If the widget is a container, and the description record specifies the content to be created inside, one must use this build function instead of the build of the widget repository. This allows the gatherer to register all the created widgets and their initial position.
- builder
  If the renderer of this widget needs to use the build function of the EBL renderer manager, this builder reference must be passed to the proxy manager of this widget by calling the setBuilder method of the manager and giving it this builder reference as parameter.
- handle
  References the newly proxy widget for this widget
- gatherer
  References a temporary version of the gatherer, which is specialized by the setGathererClass, but does not yet provide the normal gatherer functionality (features are absent, getAllIds, getAllNames, ... methods are absent too).
- eventPort
  The build function creates a single eventPort which should be passed to the method setEventPort of the proxy manager so that all the widgets created by a single build command uses a single eventPort for processing their events.

- id
  An Oz name that is the unique id associated to this widget inside the gatherer.
- desc
  The description record that specifies how this widget should be created.

EBL Proxy Manager

- setBuilder(S)
  Allows the renderer to use its build method. The S parameter is from the builder feature of the parameter given to the procedure called when the widget is created from a description record.
- getStore(Name ?O)
  Binds O to a direct reference to the store named Name.
- refToId(R ?I)
  Binds I to the id of them reference R.
- setRenderContextClass(C R)
  Sets the C render class for the context R for this particular widget.
- setContext(C)
  Sets the current context of the widget to C.
- getContext(?C)
  Binds C to the current context of the widget.
- getRenderClass(?R)
  Gets the render class corresponding to the current context.
- destroy
  Destroys the widget, terminating all its renderers.
- setConnectionPolicy(P)
  P is a one parameter procedure that determines the connection policy for this widget, ie the actions to undertake when a rendererer connects to the proxy. The parameter of this procedure determines the event to handle:
  - incoming(Id)
    A new renderer whose id is Id just connected to this proxy.

  Typically, the P procedure disconnects from the currently connected renderers (this list is obtained by the getRenderIds method and the disconnection is executed by calling the disconnect method) and then connects to the incoming renderer (by using the connect method).
- setEventPort(P)
  Sets the port were all user events are sent to P.
- createRemoteEnvironment(?E)
  Binds E to an environment. Typically this method is used by toplevel widgets that have to create a local renderer for which they require a preconfigured environment.
- getRef(?R)
  Binds R to the reference of this proxy. This reference can be used by another container proxy to trigger a migration of this widget.
- getRenderIds(?R)
  Binds R to the list of the ids of all the currently connected renderers.

- getChildrenIds(?R)
  Binds R to the list of the ids of all the children widgets of this container.
- getChildInfo(I ?R)
  Binds R to unit if there is no child with id I, or to a pair Ref#PlacementInstructions corresponding to this child.
- disconnect(Id)
  Disconnects the renderer whose id is Id.
- connect(Id)
  Connects the renderer whose id is Id. This should be only called from the policy procedure defined by setConnPolicy.
- createRemoteHere(Env)
  This method instructs EBL to create a renderer connected to this proxy locally, using the environment Env. This is typically used by toplevel windows.
- importHere(Ref PlacementInstructions id:?Id<=_)
  This method instructs EBL to migrate the widget whose reference is Ref, using the placement instructions specified by PlacementInstructions. Id is bound to the id of the newly created child.
- restoreHere(Ref PlacementInstructions)
  If no child widget already corresponds to Ref#PlacementInstructions, then importHere this widget, otherwise does nothing.
- dropClient(I)
  This method instructs EBL to disconnect the child widget whose id is I.
- execEvent(E Args)
  Executes the event E with parameters Args. E can be a procedure, a pair object#method or a pair port#message.
- ask(Q ?R)
  Creates a transaction that sends the query Q to the renderer(s) of this proxy, and waits for the response which is eventually bound to R. If several renderers are connected, the first one to provide the response is taken into account. If no renderer is connected (long enough for having the response sent back), then the ask(Q ?R) blocks until a new renderer comes in and send the response. All ask(Q ?R) request are serialized. In particular if one of them suspends due to a lack of renderer, when a new renderer connects its state if first completely updated, and then pending transactions are resumed keeping the serialisation at all time.
- send(M)
  Sends the message M to the renderer(s) of this proxy in the FIFO order. If no renderer is connected, the message is dropped.
- getState(?S)
  Used internally to obtain a serialisation of the complete current state of the widget.

EBL Proxy Store

- setProxyMarshaller(P)

Defines how items are transformed at the proxy before putting them into the store, or getting them back from the store. P must be a record where each feature is a type name, and the value is also a record. The value record can have a feature u2s (applied when the user sends a value to the store) and a feature s2u (applied when the user gets a value from the store). The u2s feature specifies a function which returns the transformed version of its parameter. The s2u feature is similar, but can also receive a second parameter which is bound to the store object that is applying this transformation.

- setRenderMarshaller(P)
  Defines how items are transformed at the renderer when interacting with the store. See setProxyMarshaller for the structure of P.
- setTypeChecker(P)
  Defines a map between a type name and a function validating a value for this type. P must be a record where each feature is a type name, and the value is a pair Function#Description. Function is a one parameter function that returns true if the parameter is of the valid type, false otherwise. Function can also return the atom remote, which instructs EBL that the value of this store can only be validated by a renderer; in this situation, an update of a parameter of this type will require a renderer to be able to apply the update, and if it does not raise an exception, then this update is confirmed at the proxy, otherwise the update is refused and an exception is raised. Description is the text that is displayed in the error message raised when an invalid type is used.
- setParametersType(P)
  Defines a map between parameters name and their type. P must be a record where each feature is a parameter name, and the value is the type associated to this parameter. The feature named '...' can be also used to associate a default type to parameters not present in P.
- setDefaults(P)
  Defines default values for parameters. P must be a record where each feature is a parameter name, and the value is the default value.
- set(K V)
  Sets the parameter K to the value V. Raises an exception if V is not acceptable for K. This method follows the type instruction specified by setParametersType and setTypeChecker, which can make the check purely local, or executed remotely.
- localSet(K V)
  Sets the parameter K to the value V. Raises an exception if V is not acceptable for K. This methods does not completely follow the type instruction specified by setParametersType and setTypeChecker, because only local checks are executed, remote checks are assumed to succeed.
- remoteSet(K V ?R)
  Sets the parameter K to the value V. Binds R to true if the update succeeds, false otherwise. This methods does not completely follow the type instruction specified by setParametersType and setTypeChecker, because even if local checks succeed, a remote check is also executed before accepting the update.
- unset(K)

Sets the parameter K back to its default value as specified by setDefaults.

- getManager(?M)
  Binds M to the proxy manager this store is part of.
- get(K ?V)
  Binds V to the current value of the parameter K. This method depends on the configuration specified by setDefaults: a value that has been previously set, or one that has a default value defined will use a localGet, otherwise a remoteGet will be used.
- localGet(K ?V)
  Binds V to the current value of the parameter K, as known by the proxy which is either the last value set for this parameter, or its default value as defined by set-Defaults. Raise an exception when EBL is unable to obtain such value.
- remoteGet(K ?V)
  Binds V to the current value of the parameter K, as known by the renderer.
- createEvent(event:E<=unit action:A args:G<=nil unbind:?U<=_ code:?C)
  Creates an event, whose event string is E, action is A, and arguments are the list G. Binds U to a zero parameter procedure that destroys this event when applied, and binds C to the code id of this event.
- registerVirtualEvent(Virtual Code)
  Real events are events related to the actual toolkit, while virtual events are internal events managed at the Oz level. This method associates a virtual event named Virtual (an atom or Oz name) to the event whose id is Code.
- triggerVirtualEvent(Virtual FullArgs)
  Triggers the virtual event named Virtual (which triggers all events associated to this virtual event by the registerVirtualEvent method), with the list of arguments provided by FullArgs.
- triggerEvent(Code Args<=nil)
  Triggers the event whose id is Code with the list of arguments provided by Args.
- askBind(Code ?R)
  Sends the event whose id is Code to the renderer, which then creates a proper event binding using the back-end toolkit. If this operation is successful, R is bound to true, otherwise R is bound to false. Note that askBind also use the transactional mechanism like the ask method.
- bind(Code)
  Sends the event whose id is Code to the renderer, which then creates a proper event binding using the back-end toolkit. This method assumes that the event binding will succeed at the renderer.
- removeBind(Code)
  Tells the renderer that it must no more having an event configured for Code.
- getState(?S)
  This method is used internally to serialize the complete state of the store.
- destroy
  Destroys the store and its content.

EBL Renderer API

Annex A

EBL Renderers must follow this specific API:

- init(M)
  Initialization method, the parameter M gives a reference to the manager of this renderer. See below for the functionalities of this manager. The init method creates the actual widget, and initializes its state according to the store (parameters and event bindings).
- set(I K V)
  The key K of the store I must be updated to V: reflect this update in the widget.
- remoteSet(I K V R)
  Try to set the value of K of the store I to V. If it is possible, apply the update to the widget, and return R=true, else return R=false.
- remoteGet(I K R)
  Ask for the current value of the key K of the store I, and return it in R.
- ask(Q R)
  Reply to the question Q in R
- send(M)
  Receive the message M and apply it
- setChildEnvironment(E PlacementInstructions)
  If this widget is a container, this method specifies further configuration for the environment of the child widget E
- importHere(Ob PlacementInstructions)
  If this widget is a container, the child widget Ob has to be migrated inside this one, using the placement instructions given by PlacementInstructions.
- bind(I E P)
  Bind the event whose id is E of the store I according to the specification of P.
- askBind(I E P R)
  Try to bind the event whose id is E of the store I according to the specification of P. If it is possible, returns R=true, else returns R=false and the binding is canceled.
- removeBind(I E)
  Remove the binding whose id is E of the store I.
- destroy
  Destroy the widget

EBL Renderer Manager

- getEnv(?E)
  Bind E to the environment of this renderer. The environment is a special dictionary containing renderer side information necessary for the good running of the widget. In particular, the environment contains a reference to the effective toolkit the renderer should use. Also the environment contains a reference to the event stream of the toplevel widget.
- getStore(Name ?O)

Bind O to a direct reference to the store named Name.

- getWidget(?R)
  Bind R to the renderer object this object is the manager of.
- refToId(R ?I)
  Bind I to the id of them reference R.
- createRemoteHere(Env)
  This method instructs EBL to create a renderer connected to this proxy locally, using the environment Env. Use this method to display widgets created by the build method.
- createRemoteEnvironment(?E)
  Bind E to an environment. Typically this method is used by widgets that have to create a local renderer for which they require a preconfigured environment.
- importHere(Ref PlacementInstructions id:?Id<=_)
  This method instructs EBL to migrate the widget whose reference is Ref, using the placement instructions specified by PlacementInstructions. Id is bound to the id of the newly created child.
- build(Desc V) & displayHere(R P)
  For symmetry reasons, the EBLRemoteManager also provides an access to the mixed declarative/imperative approach. The build method is equivalent to the build function provided by EBL, however the widgets created are running at this renderer side, and not at the application side. Because of this, the universal references of these widgets should be used only at this renderer, by using displayHere instead of importHere. In practice, the technique consists in three steps:
  - Create a renderer widget that is a single widget container.
  - At the renderer's side, use the build method of its EBLRemoteManager to create a complex UI using the high level approach provided by EBL.
  - Get the universal reference of the toplevel widget of this newly created UI, and use the displayHere method of the renderer
    the renderer should also implement the importHere(Ob PlacementInstructions) method to finally put the toplevel widget in place
- destroy
  Destroy the widget, terminating all its renderers.

EBL Renderer Store

- setRenderMarshaller(P)
  Same effect as the method from the EBL Proxy Store.
- setParametersType(P)
  Same effect as the method from the EBL Proxy Store.
- set(K V)
  Set the parameter K to the value V. Access to the K parameter by the renderer are blocked until the proxy has applied this update.
- get(K ?V)
  Binds V to the current value of the parameter K.
- getName(?N)

Binds N to the name of this store.

- getManager(?M)
  Binds M to the manager this store belongs to.
- createEvent(event:E<=unit action:A args:G<=nil unbind:?U<=_ code:?C)
  Creates an event, whose event string is E, action is A, and arguments are the list G. Binds U to a zero parameter procedure that destroys this event when applied, and binds C to the code id of this event.
- registerVirtualEvent(Virtual Code)
  Real events are events related to the actual toolkit, while virtual events are internal events managed at the Oz level. This method associates a virtual event named Virtual (an atom or Oz name) to the event whose id is Code.
- triggerVirtualEvent(Virtual FullArgs)
  Triggers the virtual event named Virtual (which triggers all events associated to this virtual event by the registerVirtualEvent method), with the list of arguments provided by FullArgs.
- triggerEvent(Code Args<=nil)
  Triggers the event whose id is Code with the list of arguments provided by Args.
- removeBind(Code)
  Tells the renderer that it must no more have an event configured for Code.
- getState(?S)
  This method is used internally to serialize the complete state of the store.
- destroy
  Destroys the store and its content.

EBL Renderer Environment

The RemoteEnvironment provides these functionalities:

- put(K V)
  Sets the key K to the value V
- get(K ?V)
  Binds V to the value of K, raise an exception if absent.
- condGet(K D ?V)
  Binds V to the value of K, binds V to D if K is absent.
- destroy
  Empties the environment.
- entries(?L)
  Returns the list of pairs Key#Value defined in this environment.
- clone(?C)
  Returns a new environment that contains the same information as this one.

Annex B: Mobictionary source code

There are 3 peers in a mobictionary game:
A. Team A that chooses a word
B. A drawer from team B that knows the words, and makes drawing of this word
C. The rest for team B that try and guess the word from the drawing

A single piece of code runs the three GUIs for each type of peers. The code starts by creating a window, and then search for running games. The discovery service searches for games on the local area network. If a game is found, then the main part of the window and the status bar at the bottom of the window are migrated, and that's it: the functional core behind the UI is executed at the host of the game. If a game is not found, then we create a new host for one. Creating a game consists in creating the different components of the UI including the components for the two other peers. Some of them are present at several peers simultaneously:

- A text field for entering the text (A)
- A label for displaying this text (B)
- A start button for accepting the text input and starting the countdown (A)
- A clock (A, B, and C)
- A free drawing area (B and C)
- A toolbar for selecting the drawing tools (B)
- A "Win" button the click if the word is guessed on time (A)

Once created, this game host waits for another peer to connect, and provides it the migration capabilities for the UI of A. Then the host waits for another peer to connect, and provides it the migration capabilities for the UI of B. The host itself displays the UI of C. When all peers are connected, the host runs the game: it waits for a word to be entered by A, starts the timer and displays the word at B, updates the timer each second, and when it falls to zero terminates the game. Alternatively, the "Win" button terminates the game sooner in case of victory.

```
declare

ETkModule={Module.link ["ETk.ozf"]}.1
ETk=ETkModule.etk

PortNu=15435

Master

{DPInit.init init(ip:"127.0.0.1") _}

Win={ETk.build window(name:top
```

Annex B

```
                  td(glue:nswe name:inner
                    td(name:main glue:nswe bg:ETk.color.white)
                    label(glue:swe name:info relief:sunken
                       text:"Searching for running game.")))}

{Win.top show}
{Win.top wm(title:"Mobictionary")}

Client={New Discovery.client init(port:PortNu)}
Master=case {Client getOne(timeOut:2000 info:$)} of timeout then unit [] X
then X end
try {Client close} catch _ then skip end

if Master==unit then
   %% nobody is running a game => create a fresh one

   {Win.info allowMultipleRenderers(true)}
   {Win.info set(text:"Waiting for first team")}
   CanvasB={ETk.build canvas(name:canvas bg:ETk.color.white borderwidth:0)}
   Canvas=CanvasB.canvas
   {Canvas allowMultipleRenderers(true)}
   ClockB={ETk.build lr(name:clockline glue:nwe
                        label(glue:w text:"Remaining time: ")
                        label(glue:w name:clock text:"02:00"))}
   ClockLine=ClockB.clockline
   Clock=ClockB.clock
   {ForAll {ClockB getAllItems($)}
    proc{$ H} {H allowMultipleRenderers(true)} end}
   {Win.main display(ClockLine o(row:0 column:0 sticky:nwe))}
   {Win.main rowconfigure(1 weight:100)}
   {Win.main columnconfigure(0 weight:100)}
   {Win.main display(Canvas o(row:1 column:0 sticky:nswe))}
   Int1={ETk.build td(name:top
                      lr(glue:nwe
                         label(text:"Enter word: ")
                         entry(name:word glue:we))
                      lr(glue:we
                         button(text:"Start" name:start
                                state:disabled glue:n)
                         button(text:"Found !" name:stop
                                state:disabled glue:n))
                      )}
   Colors=[black white green red blue yellow]
   ColorsDesc={List.toTuple lr
               {List.map Colors
                fun{$ C}
                   button(name:C text:"  " bg:ETk.color.C glue:w)
                end}}
   Int2={ETk.build td(name:top
                      lr(glue:nwe
                         label(text:"Word: " glue:w)
                         label(name:word glue:w))
                      {Record.adjoin ColorsDesc lr(glue:nwe relief:sunken)}
                      )}
   Pu1={New Discovery.server
        init(port:PortNu
             info:{Connection.offer {Win.info getRef($)}#
                                    {Int1.top getRef($)}})}
   Pu2
   Start Go Won
   {Int1.top bind(event:'connect'
                  action:proc{$}
```

Annex B

```
                          {Pu1 close}
                          Pu2={New Discovery.server
                               init(port:PortNu
                                    info:{Connection.offer
                                          {Win.info getRef($)}#
                                          {Int2.top getRef($)}})}
                            {Win.info set(text:"Waiting for second team")}
                            {Int1.top display(ClockLine
                                              o(row:2 column:0 sticky:nwe))}
                        end)}
 {Int2.top bind(event:'connect'
                action:proc{$}
                           {Pu2 close}
                           {Int2.top display(ClockLine
                                             o(row:2 column:0 sticky:nwe))}
                           {Int2.top rowconfigure(3 weight:100)}
                           {Int2.top columnconfigure(0 weight:100)}
                           {Int2.top display(Canvas
                                             o(row:3 column:0 sticky:nswe))}
                           Start=unit
                        end)}

 %% waits for everybody to connect

{Wait Start}

 %% first step : wait for a word

 DrawColor={NewCell black}
 {Win.info set(text:"Waiting for a word")}
 {Int2.black set(relief:sunken)}
 {Int1.start set(state:normal)}
 {Int1.start bind(event:default
                  action:proc{$}
                             Go=unit
                             {Int1.start set(state:disabled)}
                          end)}
 {Wait Go}

 %% second step : run game

 {Win.info set(text:"Running Game")}
 {Int1.stop set(state:normal)}
 {Int1.stop bind(event:default
                 action:proc{$}
                           try Won=true catch _ then skip end
                           {Int1.stop set(state:disabled)}
                        end)}
 {Int2.word set(text:{Int1.word get(text:$)})}

 %% this thread updates the time each second for 120 seconds
 %% or until the game is won

 thread
    fun{ToTime T}
       M=T div 60
       S=T mod 60
    in
       if M<10 then 0#M else M end#":"#
       if S<10 then 0#S else S end
    end
    proc{Loop T}
```

Annex B

```
         if {IsFree Won} then
            {Clock set(text:{ToTime T})}
            {Delay 1000}
            if T>0 then
               {Loop T-1}
            else
               try Won=false catch _ then skip end
            end
          end
      end
   in
      {Loop 120}
   end

   %% let the drawer select the active color

   {ForAll Colors
    proc{$ C}
       {Int2.C bind(event:default
                    action:proc{$}
                              {Int2.{Access DrawColor} set(relief:raised)}
                           {Assign DrawColor C}
                           {Int2.C set(relief:sunken)}
                          end)}
    end}

   %% let the drawer draw

   CX={NewCell 0.0}
   CY={NewCell 0.0}
   UnEvent1 UnEvent2
   {Canvas bind(event:'1'
                args:[float(x) float(y)]
                action:proc{$ X Y}
                          Col={Access DrawColor}
                       in
                          {Canvas create(oval [X-1.0 Y-1.0 X+1.0 Y+1.0]
                                         fill:ETk.color.Col
                                         outline:ETk.color.Col)}
                          {Assign CX X}
                          {Assign CY Y}
                        end
                unbind:UnEvent1)}
   {Canvas bind(event:'B1-Motion'
                args:[float(x) float(y)]
                action:proc{$ X Y}
                           {Canvas create(line [{Access CX} {Access CY} X Y]
                                         width:3
                                         fill:ETk.color.{Access DrawColor})}
                          {Assign CX X}
                          {Assign CY Y}
                       end
                unbind:UnEvent2)}
   {Wait Won}

   %% last step: game over

   {UnEvent1}
   {UnEvent2}
in
   {Win.info set(text:"Game Over")}
else
```

Annex B

```
   %% connected to a site that is running a game => get my UI

   P={Connection.take Master}
in
   {Win.inner display(P.1 o(row:1 column:0 sticky:swe))}
   {Win.main display(P.2 o(row:0 column:0 sticky:nswe))}
end
```