IST BASIC RESEARCH PROJECT
SHARED COST RTD PROJECT
THEME: FET DISAPPEARING COMPUTER
COMMISSION OF THE EUROPEAN COMMUNITIES
DIRECTORATE GENERAL INFSO
PROJECT OFFICER: THOMAS SKORDAS

Gloss°

Global Smart Spaces

Final Reference Framework for Interaction Surfaces

D19

27/10/2003, UNIV. JOSEPH FOURIER/WP7/VERSION 2.0

J. COUTAZ, N. BARRALON, C. LACHENAL, G. REY

IST Project Number	IST-2000-26070	**Acronym**	GLOSS
Full title	Global Smart Spaces		
EU Project officer	Thomas Skordas		

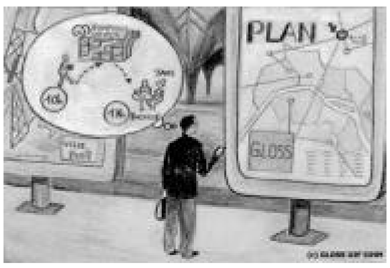
Deliverable	**Number**	D19	**Name**	Final Reference Framework for Interaction Surfaces			
Task	**Number**	T	**Name**	n/a			
Work Package	**Number**	WP7	**Name**	Interaction Techniques			
Date of delivery	**Contractual**		October 2003	**Actual**		Oct. 2003	
Code name	n/a			**Version** 2.0 draft ☐		final ☑	
Nature	Prototype ☑ Report ☑ Specification ☐ Tool ☐ Other:						
Distribution Type	Public ☑ Restricted ☐ to:						
Authors (Partner)	J. Coutaz, N. Barralon, C. Lachenal, G. Rey						
Contact Person	J. Coutaz						
	Email	Joelle.coutaz@imag.fr	**Phone**	+33 4 76 51 48 54	**Fax**	+33 4 76 44 66 75	
Abstract (for dissemination)	This document describes the final version of a reference framework for understanding, reasoning and implementing user interfaces for global smart spaces. This framework includes an ontology that makes explicit the concepts of multi-surface interaction and, based on this ontology, I-AM, a software infrastructure that supports the dynamic composition of heterogeneous interaction resources to form a unified space. In this space, users can distribute and migrate whole or parts of user interfaces as if they were handled by a unique computer. I-AM provides users with the illusion of a unified space at no extra cost for the developer.						
Keywords	Multi-surface interaction, distributed user interface, migratory user interface, user interface development tool, Interaction Abstract Machine.						

Table of Content

1 Introduction

The state of the art in ubiquitous computing shows early examples of interactive systems, motivated by human-centered concerns, that are based on the dynamic composition of interaction resources: Distributed UI as in Rekimoto's Pick and Drop [Rekimoto 97], dynamic docking of multiple displays to enlarge real screen estate (e.g., the ConnecTable [Tandler 01b] or Hinckley's work [Hinckley 03]), Migratable UI as in I-land [Streitz 99] and Seescoa [Luyten 02], or the Personal Server approach that promotes the borrowing of near-by interaction resources [Want 01]. As far as we know, all of these prototypes have been developed as concept demonstrators reusing and hacking the current foundational tools of the GUI technology.

A number of European projects (e.g., Cameleon[1] and Consensus[2] are currently concerned with the definition of models and tools that support the design and development of plastic UIs, i.e., UIs that can dynamically adapt to context of use while preserving usability [Thevenin 99]. Tools such as Teresa [Paternò 02], Vaquita [Vanderdonckt 01] and ARTStudio [Calvary 01] have been developed to support the specification and generation of UIs that can adapt to PDAs, cell phones and workstations. These tools support well-established methods that ensure the usability of the resulting UI. But they do so for centralized UIs where a virtual machine like JVM or a Web browser is sufficient for supporting the portability of the generated code. When it comes to address distributed plastic UI over a dynamic set of heterogeneous resources, these tools are impeded by the existing windowing systems and toolkits.



In the GLOSS vision, users have a number of devices available, some of which they own and which travel with them, and others that are available from their current location. A GLOSS user may want to use GLOSS services, plug them together in order to create new services and arrange their interaction resources to suit the activities at hand. In addition, several users may discover themselves in a place and may decide to work together.

In summary, state of the art in distributed, migratable, composable, and plastic UI's as well as UI development tools are limited by existing windowing systems and toolkits designed at a time where user interfaces where confined to a single screen using a single pointing and text entry device. To address the problem, either we hack the current foundational tools and develop short-term demonstrators, or we aim at general

[1] http://giove.cnuce.cnr.it/cameleon.html

[2] http://www.consensus.upv.es

foundational solutions that support the new requirements imposed by global smart spaces. If the scientific community aims at supporting a sound approach to the development of distributed migratable, plastic UI's, we need to revise our foundational tools, which, by definition, set the basis for the development of higher level of abstraction tools. In turn, experience shows that these high level tools increase the capacity to develop usable user interfaces.

In WP7, we are concerned with the development of a software infrastructure that enables users to borrow and lend local interaction resources in an opportunistic manner as they move in the global fabric of networked computation. The multi-surface interaction ontology presented in Year 2 (Cf. D17), now integrated in the GLOSS ontology (See D9), has provided a sound rationale for eliciting the limitations of current foundational tools. It is briefly recalled in Section 2. Next, we analyse how the state of the art addresses our requirements (Section 3). In Section 4, we describe I-AM, our proposal for a middleware run-time infrastructure that addresses the limitations of current tools. I-AM complements the local infrastructure as defined in D8 by St Andrews, and draws upon the proximity-group model developed by TCD. It is not intended for traditional centralised UI's. Instead, it is aimed at facilitating the development of distributed and migratable UI's *at no extra cost for the developer while providing users with a unified view* as if the user interface were handled by a single computer. In Section 5, we illustrate the integration of I-AM within an infrastructure that supports the run time adaptation of plastic user interfaces. This infrastructure has been designed by UJF for the CAMELEON R&D IST-2000-30105 project.

2 MULTI-SURFACE INTERACTION ONTOLOGY

A more detailed description of the Multi-surface Interaction ontology is available in D9. Figure 1.1 is here to recall the elements of the model.

The concepts necessary to understand our software solution are the following: platform, interaction resources, surfaces and instruments, spatial relationships, coupling interaction resources with digital content, distributed UI and migratable UI.

A platform may be elementary or a cluster. An *elementary platform* is a set of physical and software resources that function together to form a working computational unit whose state can be observed and/or modified by a human user. None of these resources is able per se to provide the user with observable and/or modifiable computational function. A personal computer, a PDA, or a mobile phone, are elementary platforms. On the other hand, resources such as processors, central and secondary memories, input and output interaction devices, sensors, and software drivers, are unable, individually, to provide the user with observable and/or modifiable computational function.

Some resources are packaged together as an immutable configuration called a core configuration. For example, a laptop, which is composed of a fixed configuration of resources, is a core configuration. The resources that form a core configuration are *core resources* or Machine-bound [Johanson 02] resources. Other resources such as external displays, sensors, keyboards and mice, can be bound to (and unbound from) a core configuration at will. They are *extension resources*.

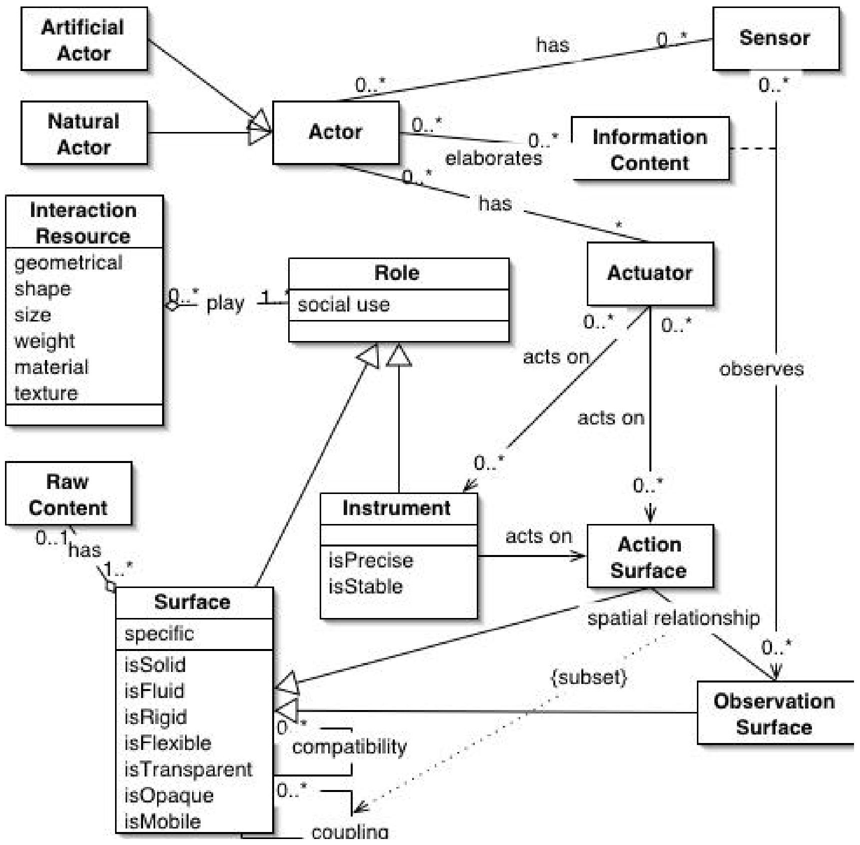
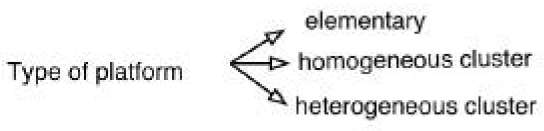


Figure 1.1. The GLOSS ontology for multi-surface interaction.

A *cluster* is a composition of elementary platforms. The cluster is homogeneous when it is composed of elementary platforms of the same class. For example, the DynaWall is an homogenous cluster composed of three electronic white boards running the same operating system [Streitz 99]. The cluster is heterogeneous when different types of platforms are combined together as in Rekimoto's augmented surfaces [Rekimoto 99].

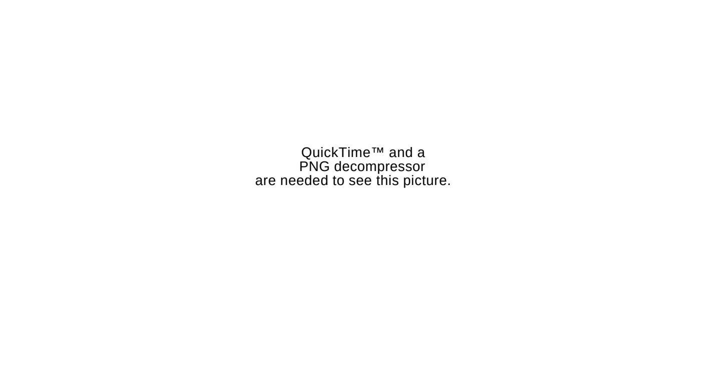


Interaction resources are subclasses of core and extension resources. They are mediators between an artificial actor (e.g., a GLOSS system) and a natural actor (e.g., a user). An interaction resource may serve as an *instrument* (e.g., a pen and a mouse) and/or as a *surface* (e.g., a screen display or a wall). As an instrument, an interaction resource mediates the actions of an actor. As a surface, the outmost boundary of a physical entity serves as a recipient for making information observable to an actor. Physical surfaces and instruments are characterised by attributes (grounded in the physical world) as well as by relations. An *action* surface is a subset of a physical surface on which an actor can act directly with actuators and/or indirectly with instruments. Similarly, an *observation surface* is a subset of a physical surface that an actor can observe with sensors.

Among these relationships, geometric spatial relations are central. They form a *physical topology* that results from the way surfaces and instruments are assembled. The physical topology can be projected on digital content in many ways. For example, a set of screen displays can be used as multiple physical magic lenses that can be moved independently over the digital space. Alternatively, the displays, although not contiguous in the physical space, may partition the digital space in a continuous manner in order to show the informational content without any hole. In other words, coupling interaction resources onto (with) digital content is an important issue for HCI design. It is therefore important that the technical solution provides coupling mechanisms while, at the same time, it is able to support different application-dependent policies.

A *user interface is distributed* when its interaction components (e.g., windows and panels) are allocated (whether it be statically or dynamically) to different platforms of a cluster. For example, using the painter metaphor, tools palettes are displayed on a PDA held in the non-dominant hand whereas a stylus is held in the dominant hand. Like a painter artist, the user picks the appropriate tool on the palette with the stylus, and then draws on the canvas supported by a wall-size electronic screen. Distribution of user interfaces can be performed at multiple grains. Granularity of user interface distribution may vary from application level to pixel level:

- At the *application level*, the user interface is fully replicated on the platforms of the target cluster. If the cluster is heterogeneous (e.g., is comprised of a mixture of PC's and PDA's), then each platform runs a specific targeted user interface. All of these user interfaces, however, simultaneously share the same functional core.
- At the *workspace level*, the user interface components that can be allocated to different surfaces are workspaces. A workspace groups together a collection of interactors that support the execution of a set of logically connected tasks. In graphical user interfaces, a workspace is mapped onto the notions of window and panels. The painter metaphor is an example of UI distributed at the workspace level.



- At the *domain concept level*, the user interface components that can be distributed between platforms are interactors that allow users to manipulate domain concepts. In Rekimoto's augmented surfaces, domain concepts can be distributed between laptops and horizontal and vertical surfaces. (In their demo, tables and chairs interactors can be moved between physical surfaces.).
- At the *pixel level*, any user interface component can be partitioned across multiple platforms. For example, in I-land, a window may simultaneously lie over two contiguous white boards.

Distribution is complemented with the capacity of the user interface to migrate at run time. A *user interface is migratable* when all or parts of its components can be transferred at run time between platforms. Migration may be total or partial:

- *Migration is total* if the user interface migrates entirely to a different platform.

- *Migration is partial* when a subset only of the user interface moves to different platform(s). The subset that can migrate is consistent with the granularity of distribution: partial migration may be performed at the workspace level, concept level or pixel level.

The *GLOSS I-AM aims at supporting* **the essence of multi-surface interaction**, *that is, the dynamic composition of heterogeneous clusters within which user interfaces can be distributed and migrated in a continuous way at multiple levels of granularity* **without imposing any extra programming on the developer**. In the next section, we analyse the state of the art in the lights of our requirements.

3 STATE OF THE ART

From a software perspective, the problem space described in Section 2 implies that the software that implements an interactive system be dynamically reconfigurable, possibly distributed across a dynamic set of heterogeneous resources, and migratable across these resources. Thus, we need to consider how to address:

- dynamic reconfiguration which, in turn, requires system integrity,
- heterogeneity of system resources, and
- detection of arrival/departure of interaction resources.

These issues are discussed next along two dimensions: first, the general proposals from the state of the art followed by current UI-oriented infrastructures.

3.1 DYNAMIC SOFTWARE RECONFIGURATION

Dynamic software reconfiguration primarily covers the following issues: 1) modification of the structure of the system (i.e., by adding, removing, substituting components, and/or by modifying their connection) and 2) modification of the geographical distribution of the components across the currently available comptutational resources. In the context of this discussion, a component is considered to be a software unit responsible for implementing a set of services that can be composed with other components. For doing so, it includes a description of the inputs it requires and the outputs it supplies.

As discussed in [Oreizy 99], dynamic reconfiguration may be close-adaptive or open-adaptive:

- When *close-adaptive*, the system includes all of the mechanisms and data to perform adaptation on its own: it is self-contained (autonomous).
- *Open-adaptiveness* implies that adaptation is performed by mechanisms and data that are external to the system.

Whether it be close-adaptive or open-adaptive, dynamic reconfiguration is best supported by a component-connector approach [Oreizy 99, Garlan 01]. Components that are capable of reflection (i.e., components that can analyse their own behaviour and adapt) support close-adaptiveness. Components that are capable of introspection (i.e., components that can describe their behaviour to other components) support open-adaptiveness.

In I-AM, we have adopted a component-connector approach. The notion of component provides the foundations for system flexibility. In addition, system reconfiguration implies that links between the components change over time, and that components may be eliminated or replaced in the reconfiguration process. This flexibility requires that components should not reference their peers explicitly, nor should they know the communication protocol expected by their peers. In a connector-based approach, where connectors are in charge of linking the components and supporting their communication protocol, a clear distinction between the functional aspects of the system (the components) and the communication aspect of the system (the connectors) is achieved. However, in a connector-based approach, the overall structure of the system must be described. An ADL (Architecture Description Language) as described in D11 is a promising way to go.

3.2 DYNAMIC DISCOVERY OF INTERACTION RESOURCES

The dynamic discovery of system resources is a complex problem. Jini, based on a client-server model is appropriate for small-scale applications. For ubiquitous environments, the existence of a centralised server is not an option. Rendez-Vous[3] from Apple, based on the ZeroConf standard [Guttman 01], supports mutual discovery between the IP resources connected to a network. UPnP from Microsoft goes in the same direction [Microsoft 00]. These are useful techniques for managing the arrival and departures of elementary platforms in a cluster. However, they do not address the specific case of interaction resources that are not IP devices.

In I-AM, we have used contextors [Coutaz 02], an infrastructure aimed at acquiring contextual information about the physical environment and the platform, to discover the arrival and departure of interaction resources.

3.3 HETEROGENEITY OF INTERACTION RESOURCES

In software engineering, resources heterogeneity has been addressed with the notion of abstract machine.

Current windowing systems and UI toolkits are based on this technique but reveal a number of limiting factors for multi-surface interaction:

1. *The window model is biased by the workstation screen. Instead, the concept of window must be replaced with that of a physical surface with an explicit model of its physical attributes*. Today, windowing systems model windows as rectangular drawables whose borders are constrained to be parallel to that of the display. This model is based on the (wrong) the assumption (for multi-surface interaction) that users keep facing a vertical screen and that the rendering surface is rectangular. With the proliferation of video projectors, it is increasingly popular to project window contents on horizontal surfaces such as circular tables. In this situation, users should be able to rotate the digital content

[3] http://www.arstechnica.com

so that everyone can share information without twisting their neck (Cf. our Year2 GLOSS demonstrator). Today, rotating windows and user interfaces must be implemented from scratch from low-level graphics primitives. Jazz/PAD++ [Bederson 00] and DiamondSpin[4] are toolkits that offer the basics for implementing rotative, zoomable user interfaces but they do so for centralised UIs only. Similarly, physical attributes of rendering surfaces, such as size, shape and colour, must be made available to the application program: one does not want to render information on a rectangular white surface in the same way as on a circular table covered with a red cloth. In turn, the capture of the physical attributes of a surface calls for new requirements on the sensing technology. The contextors infrastructure can serve as a basis for implementing colonies of perceptual processes [Crowley 02].

2. *Geometrical relationships between physical displays are poorly modelled. Instead, the (possibly 3D) spatial relationships between the interaction resources and the user should be explicitly modelled, dynamically acquired and maintained.* Windowing systems are able to support a limited number of screens. In addition, the relative location of the display screens must be set up by the user through dedicated system forms. As a result, the user is in charge of additional articulatory tasks. Again, because they are screen-centric, windowing systems do not support topologies that include the surrounding environment (e.g., walls, tables, users' location with respect to the display surfaces, etc.). However, for many novel interactive systems, the topology of the rendering surfaces matters: for example, 3D rendering on a vertical surface and 2D presentation on a horizontal surface [Rekimoto 99].
3. *Only single instances of input instruments are supported. Instead, the system should be able to support any number of instances of an instrument class.* In current windowing systems, the reference workstation is supposed to have one single mouse and keyboard. On systems like MacOS, it is possible to plug two physical mice. Unfortunately, they are linked to the same interruption level and are modelled by the event manager as a device type, not as a device instance. As a result, multi-user applications such as MMM [Bier 92] whose users share the same screen with multiple mice, require the underlying toolkit and event manager to be revisited as in MID [Hourcade 99].
4. *Interaction is confined to the resources of a single workstation. Instead, we need to distribute UI's across a set of interaction resources managed by a cluster of possibly heterogeneous machines.* Applications like i-LAND [Streitz 99] and Rekimoto's augmented surfaces [Rekimoto 99] require the aggregation of multiple computers. Pocket-size computers can play the role of input devices to control information displayed on wall-mounted electronic boards as in Pebbles [Myers 98]. In addition, UI migration requires additional functionalities such as mechanisms for state recovery.

[4] http://www.merl.com/projects/diamondspin

5. *Absence of dynamic discovery of interaction resources. Instead, in ubiquitous computing, interaction resources appear and disappear opportunistically*: users can upload situated information on their private PDA as they pass by a public active wall. Two users who meet serendipitously in the street, may want to start a collaborative activity by bringing together their PDA's to form a larger interaction space. GaiaOS [Viswanathan 01] and Aura [Sousa 02] aim at supporting heterogeneity and dynamic migration. However, they address centralised user interfaces only.

In the next paragraph, we analyse state of the art infrastructures intended to support some aspects of our concept of multi-surface interaction. We will primarily look into those aimed at supporting clusters and/or UI migration and distribution. Solutions like the MID toolkit and Pebbles are interesting improvements over current windowing systems but they boil down to the dynamic connection of input devices to a single core configuration.

3.4 USER INTERFACE INFRASTRUCTURES

There are two approaches to user interface-oriented infrastructures: those where the focus is on developing an infrastructure at the scale of the Planet as in Gaia [Viswanathan 01] and Aura [Sousa 02]. Those where the infrastructure is adapted to a room as for BEACH [Tandler 01a], iROS-PointRight [Johanson 02a] and Easyliving [Brumitt 01]. In the following paragraph, we discuss Aura as an example of a run time infrastructure in the large, BEACH and iROS-PointRight as run time infrastructures in the small.

3.4.1 INFRASTRUCTURE IN THE LARGE: AURA

The goal of the Aura project is to find solutions to the following competing goals: 1) to maximise use of available resources in a ubiquitous-enabled computing world, 2) to minimise the distraction and drains on user attention that stem from managing those resources.

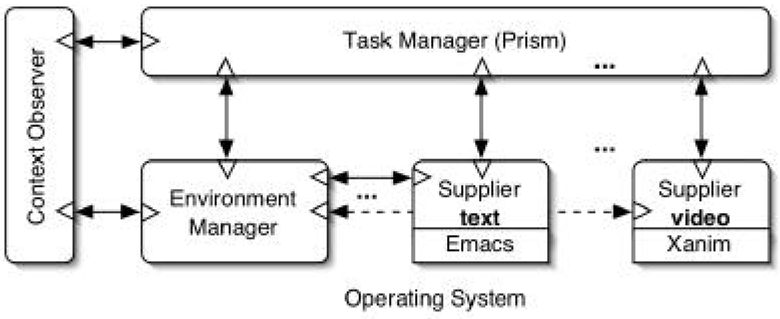


Figure 3.1. Functional decomposition of an Aura environment (from [Sousa 02]).

The approach to the problem is to provide an infrastructure that configures itself automatically for the mobile user, "potentially using whichever computing capabilities are available or reachable from the current location" [Sousa 02]. When a user moves to a different platform, Aura attempts to reconfigure the computing infrastructure so that the user can continue working on tasks started elsewhere. Figure 3.1 shows the functional decomposition of an "Aura environment".

An Aura environment is supposed to run in multiple places on the Planet such as home, office, car, etc. In the GLOSS terminology, it is a local architecture. The context observer of an Aura environment detects events of interest that occur in the physical local place (e.g., user is entering, user is leaving, etc.) and informs the local environment manager as well as the local task manager of these facts. The local environment manager is in charge of modeling the computing and interaction resources locally available. The task manager, called Prism, checkpoints the state of the running suppliers (i.e., services needed to support users' tasks) at a high level of abstraction. For example, for a text editing supplier, the task manager saves the file name and well as the current insertion point in the text of the source document.

When the current local context observer detects the user is leaving the Aura environment, it informs the local task manager which checkpoints the local suppliers and causes the local environment manager to pause those services. When the user enters another Aura environment, the new local context observer detects the fact and informs the new local task manager. In turn, the task manager re-instantiates the tasks that were suspended by finding and configuring services suppliers in the new Aura environment. So, for example, a user who was working at home using Word can carry on the task in the new environment but possibly using a different text editor such as Emacs.

Aura is intended to address all types of platforms and sources of context changes. However, adaptation is addressed by software component substitution and the granularity for substitution is a whole interactive system (i.e., a supplier in the Aura terminology). As a result, only total migration is possible with a state recovery at the task level. In other words, Aura addresses UI migration, but does not support UI distribution across heterogeneous interaction resources if the supplier did not do so in its own code. Therefore, Aura is an infrastructure for open-adaptivity.

3.4.2 INFRASTRUCTURE IN THE SMALL: BEACH

The BEACH framework (Basic Environment for Active Collaboration with Hypermedia) [Tandler 01a] provides functionalities for synchronous cooperation and interaction for roomware. As shown in Figure 3.2, it can be viewed as a four layer abstract machine. (Actually, there are others views of the BEACH framework that we have not been able to fully understand, despite code inspection.)

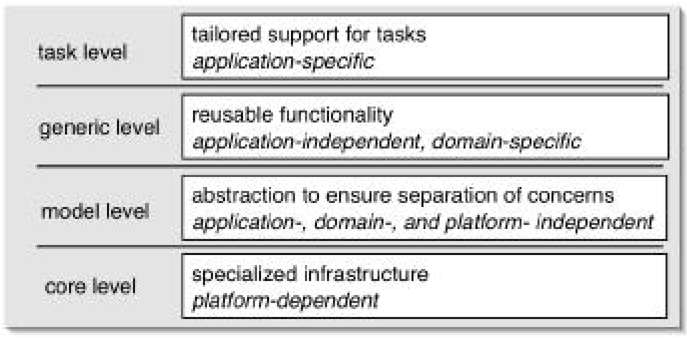

task level
tailored support for tasks
application-specific
generic level
reusable functionality
application-independent, domain-specific
model level
abstraction to ensure separation of concerns
application-, domain-, and platform- independent
core level
specialized infrastructure
platform-dependent

Figure 3.2. BEACH four conceptual levels of abstraction (From [Tandler 01a]).

The core level, which uses the COAST framework [Tandler 01a], encapsulates platform dependent details including multi-user events handling, sensor drivers, and access to shared data. The Model level provides higher levels with abstract classes that are application, domain and platform independent and that allow different implementations for different platforms. The Generic level includes generic components such as the notion of document, data types, private and public workspaces that are common to collaborative situations. The task level groups the high-level abstractions specific to a particular application. From an implementational point of view, BEACH relies on a client-server paradigm where the server serves as the reference for all of the BEACH clients running in a room. To be part of the cluster, elementary platforms must run a BEACH client.

A BEACH infrastructure addresses homogeneous dynamic clusters where all machines run a uniform Smalltalk software environment. User interfaces built on top of this environment can be distributed and migrated at the pixel level. Typically, a window can overlap two surfaces. From the developer's perspective, BEACH provides the programmer with a single logical output display mapped onto multiple physical displays. However, it is unclear whether BEACH includes an explicit model of the topology between interaction surfaces (the implicit hypothesis is that the interaction resources in a cluster are compatible and can be coupled in a predefined known way). In addition, BEACH does not seem to address the dynamic connection of input devices.

3.4.3 INFRASTRUCTURE IN THE SMALL: IROS AND POINTRIGHT

iROS[5] is an open source middleware developed at Stanford University as a basic infrastructure for roomware. As opposed to BEACH where every application has to be developed from a uniform Smalltalk software platform, iROS aims at supporting legacy applications. It includes core components for data storage, service management, and communication (the Event Heap).

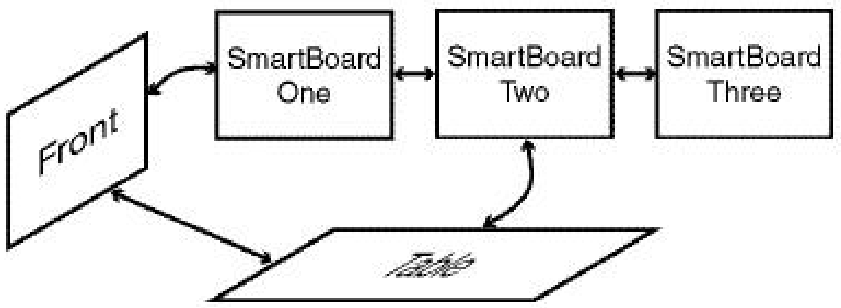


Figure 3.3. The iRoom screen topology (from [Johanson 02a]).

The Event Heap is derived from the tuplespace model [Johanson 02b]. In the tuplespace model introduced by [Gelernter 92], applications communicate and coordinate through a commonly accessible tuplespace. Tuples, which are a collection of ordered type-value fields, may be posted to the space, or read from the space in either a destructive or non-destructive manner. A tuple is chosen by a template tuple specified by the retrieving application. The template contains precise values for the fields to be matched, and wild cards for fields containing data to be retrieved. Event Heap has extended the tuplespace model to satisfy interactive rooms requirements. In particular, tuples have been extended with extra fields such as type and self description for extension and semantic checking, routing attributes to improve performance, event

5 http://iros.sourceforge.net

sequencing, a TimeToLive attribute to garbage collect unconsumed events, and a publish-subscribe mechanism to complement the polling mechanism of the original tuplespace model. On top of iROS, a number of applications have been built. PointRight is one of them.

PointRight allows the mouse cursor to cross the display screens of an heterogeneous cluster based on the geometric topology description of the space (see Figure 3.3). It is implemented as a client of the Event Heap server and is supposed to run on each of the elementary platforms of the cluster.

A PointRight client includes a sender and a receiver:

- The sender redirects the mouse and keyboard events from the local input devices. It uses the geometric topology of the screen displays of the room to direct input to the appropriate screen (i.e., to the elementary platform that owns the target display) and sends the redirected local mouse and keyboard events using the Event Heap.
- The receiver accepts remote mouse and keyboard events from the Event Heap. It is responsible for rescaling cursor motions to fit the characteristics of the local display.

The topology description of the room space consists of machines, screens and connections. Machines are the elementary platforms that currently run PointRight. A screen description includes the dimensions of the physical screen, the identification of the elementary machine it is managed by, and the set of connections to other screens. A connection represents a valid transition of pointers between screens. They are represented by an edge (top, bottom, left, right) and the region of the edge through which the pointer can transition. There can be multiple connections to a single screen edge as long as the regions of the connections do not overlap.

In the current implementation,

- The topology description of the room space is read from a configuration file. By monitoring events on the Event Heap, it is possible to know whether the machines of the configuration file are running.
- PointRight provides a unified multi-display space for pointing and keying: as the user moves the mouse, the cursor crosses the screens seamlessly across the space of displays as though there were a single surface connected to a single machine. However, windows and icons cannot be moved across screen boundaries.

PointRight is similar in spirit to VNC [Richardson 98], x2x[6] and x2vnc [Hubinette 02]. VNC (Virtual Network Computing) allows a user on one computer to interact with applications running on another computer. It does so by mirroring the remote application display and by forwarding the local keyboard and mouse events to the remote application. However, users can control a single screen at a time. To control multiple screens, one would need to launch a VNC window per remote screen and switch among the VNC windows. x2x and x2vnc are similar in spirit to PointRight, but they are specific to X-Window and do not support arbitrary configurations.

[6] http://ftp.digital.com/pub/DEC/SRC/x2x/

PointRight goes one step further than VNC, x2x and x2vnc, by redirecting pointer and keyboard events in an heterogeneous cluster. It includes a model of the geometric topology of the screens but the model is inherently 2D. In particular, horizontality and verticality are not addressed. PointRight does not support the dynamic discovery of interaction resources, nor does it support the overlap of windows and icons across screen boundaries. This is where the illusion of a unified workspace breaks. The authors "would like to find a method that elegantly extends PointRight to moving information around the iRoom, while maintaining [the] focus on general, heterogeneous applications and operating systems" [Johanson 02, pp. 233].

In summary, to our best knowledge, none of the current infrastructures, currently available or under development, covers all aspects of multi-surface interaction. I-AM presented next, aims at such goal.

4 I-AM (INTERACTION ABSTRACT MACHINE)

Our approach to the problem is to develop an Interaction Abstract Machine grounded on a sound model such as that of X Window, and extend it to support the key concepts of multi-surface interaction. I-AM extends the functional coverage of current windowing systems to distributed migratable user interfaces across a dynamic set of heterogeneous resources. More specifically:

- It hides away the low-level details of hardware and software heterogeneity at the appropriate level of abstraction,
- It supports interaction resource discovery and models their spatial relationships,
- It permits the distribution and migration of UIs across multiple surfaces at the pixel level. From the developer's perspective, this facility is provided at no extra cost. From the user's perspective, all the surfaces and instruments form a unified interactive space despite they are being driven by different machines.

A detailed description of I-AM is presented next. First, we outline the key principles of I-AM (4.1), its overall structure (4.2), and the limitations of the current implementation (4.3). The following sections (4.4, 4.5, 4.6) describe the key components of the structure in detail. We conclude the presentation of I-AM with a program example that gives a flavor of the developer's view (4.7).

4.1 PRINCIPLES

Figure 4.1 illustrates the principles of I-AM. The very bottom of the figure shows an example of a user's view where the GLOSS logo is shown seamlessly across a set of three surfaces handled by three distinct elementary platforms. The logo can be moved across surfaces boundaries using any pointing device of the cluster. At the top of the figure, the developer's view. Here, the user interface of an application consists of several windows as if they were part of a unique interaction space handled by a unique computer.

As shown at the bottom of the figure, the platform is a cluster composed of three elementary platforms. Each one handles a unique surface and runs a different operating system (e.g., MacOS X, Windows XP, Windows NT). Through surfaces links, surfaces are composed in a plane using, possibly different, orientations in the plane. Similar to the PointRight notion of connection, surfaces links are reference points located on the edge of a surface. They can take the form of a physical sensor (e.g., infrared sensors, accelerometers as in Hinckley's example of synchronous gestures for connecting tablets [Hinckley 03]). They can also be painted dots tracked by a computer vision system. Surfaces link allow I-AM to dynamically compute the topology of the surfaces. They also concretise our notion of compatibility introduced in the multi-surface ontology: surfaces can be composed only through links that are technically compatible.

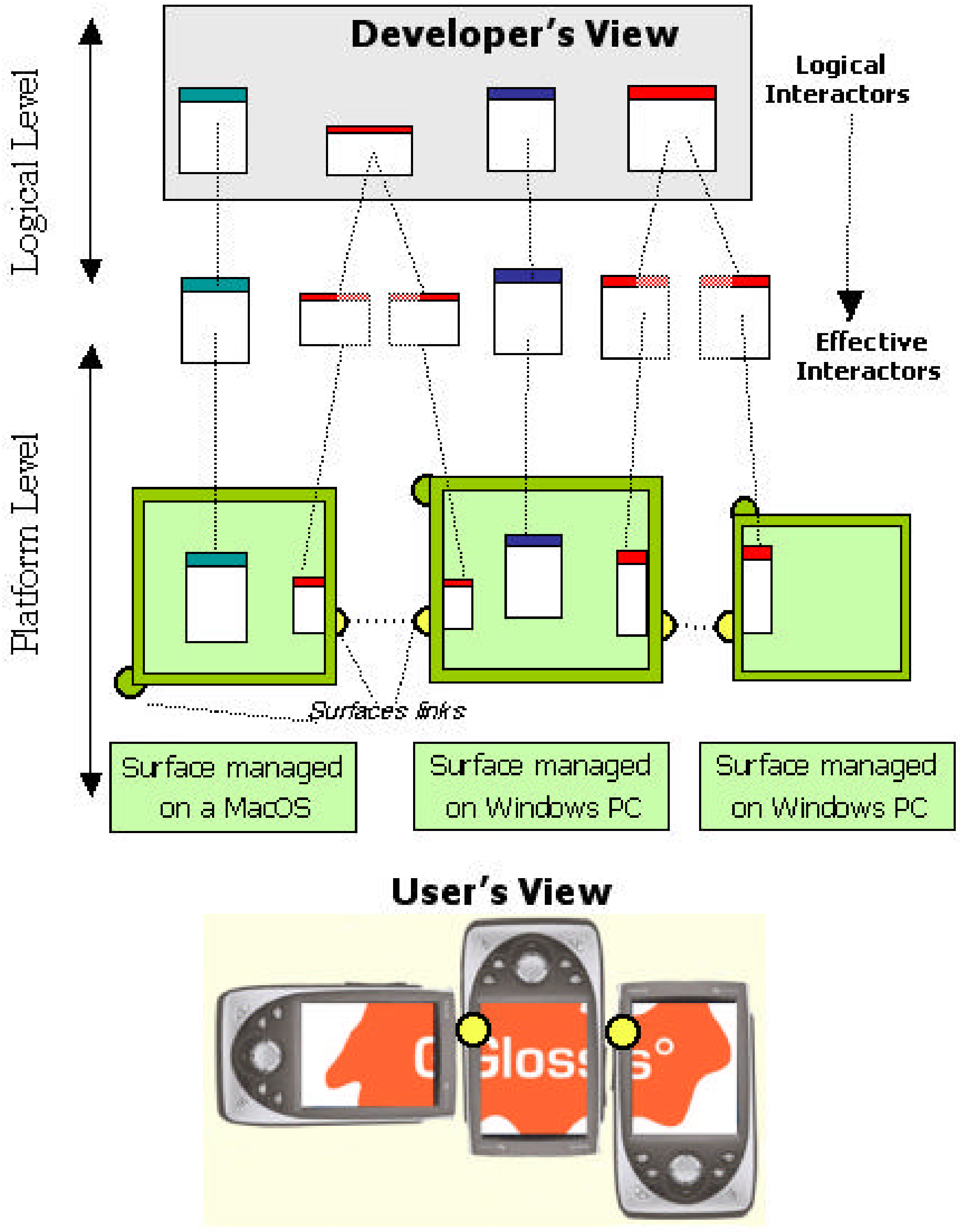


Figure 4.1. The principles of IAM.

The bottom of the figure shows the distribution of the user interface across three surfaces. Some interactors such as the top left window of the developer's view, are fully rendered within a single surface whereas other interactors, such as the right most window of the developer's view, are split across two surfaces. In the latter case, the logical interactor of the developer's view is mapped into two effective interactors whose rendering is tightly coupled to entertain the illusion of a unified space: as the user moves one of the effective interactors using any pointing device of the cluster, the other "twin" effective interactor is moved and resized accordingly as if the twins were one single piece. Of course, the number of effective interactors that correspond to a logical interactor depends on the position of the logical interactor in the logical space and on the topology of the surfaces onto which the logical space is projected.

The role of I-AM is to continuously maintain the mapping between the logical view of interactors as handled by the developer and the effective interactors as manipulated by the user. As stressed earlier, PointRight does not address the problem of interactors transition across surface boundaries at the pixel level. BEACH offers a similar illusion to I-AM but it does not support any free form of dynamic composition of surfaces, nor does it support different underlying operating systems. In its current implementation, I-AM supports heterogeneous clusters composed of MacOS X, Windows NT and Windows XP.

Having presented the principles of I-AM, we are now able to describe the technical structure of I-AM.

4.2 OVERALL STRUCTURE OF I-AM

As shown in Figure 4.2, I-AM is structured into three levels of abstraction sitting on top of the hardware and operating systems that form the legacy basis of the cluster:

- The Platform level hides the heterogeneity of the underlying level and manages the interaction resources in a normalised way. This level is implemented by the IAMPlatform Java package.
- The Logical Level provides applications with a customized abstract view of the physical platform layer. This level is implemented by the IAMApp Java package.
- The Interactor level implements the basic graphic interactors such as windows and widgets that populate the logical view of the Logical Level. This level is implemented by the IAMInteractor Java package.

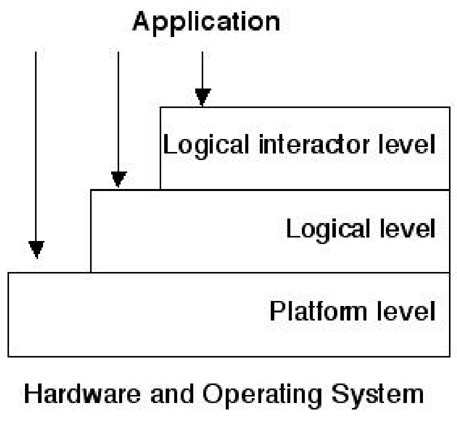


Figure 4.2. I-AM as a the three levels of abstraction machine.

Before we describe the functions of the three layers of I-AM, we need to make clear the hypotheses and limitations of the current implementation.

4.3 I-AM: CURRENT LIMITATIONS AND HYPOTHESES

4.3.1 CURRENT LIMITATIONS

- Surfaces are screen displays including video-projected displays,
- Surfaces are rectangular,
- Surfaces are assembled (coupled) within a plane but this plane can be folded as in iRoom (see Figure 3.3). 3D topology will be integrated when the sensing technology will be available. The steerable video projector described in [Borkowski 03], which tightly couples a camera (sensor) and a video projector (actuator), offers a promising way to compute the 3D orientation of any flat surface in a room. This sensor-actuator device will be used in a future version of I-AM.
- Legacy applications are not supported. In other words, the user interface of next generation applications is supposed to be developed (or generated by tools) using I-AM interactors. However, provided that their source code is available, legacy applications may be automatically reverse-engineered using Aspect Oriented Programming [Gregor 97]. In the past, we have used this approach successfully to transform an AWT user interface into a rotating Jazz-based User interface without re-programming the original UI.

4.3.2 HYPOTHESES

- Naming: an elementary platform is identified by its IP address. An interaction resource is uniquely identified by the IP address of the elementary platform it is managed by, followed by an integer that is unique for this platform. For example, the ID of a surface on a screen S handled by a processeur P is a triplet: <IP address of P, video card output number of S, unique integer>. Unique identifiers are provided by a dedicated package of IAM not described in this document.
- Communication: within I-AM components, processes communicate asynchronously through TCP sockets. Within the contextors infrastructure used for detecting the arrival/departure and composition of interaction resources, processes (e.g., contextors) use multicast sockets and subscribe to the same multicast group.
- Access control: A group management facility is supposed to identify the elementary platforms that have the right to participate in the cluster. In other word, the boundary of a cluster is known. This service can be provided by the proximity-group software developed at TCD.

4.4 THE PLATFORM LEVEL: THE IAMPLATFORM PACKAGE

Every processor that belongs to the cluster runs an IAMPlatformManager. An IAMPlatformManager is “elementary platform” centric:

- It manages the resources that are local to the elementary platform it runs on, and

- It has no knowledge of the existence of its peers IAMPlatformManagers. *The absence of explicit reference to any other IAMPlatformManager supports our requirements for scalability, flexibility and reconfigurability.* The glue between the machines is performed at the Logical Level of I-AM.

An IAMPlatformManager supports the following functions:

- It discovers the interaction resources that are locally connected to the processor it runs on,
- It maintains a description of these resources, and
- It provides the world[7] with the basic means for using the interaction resources[8]. This includes the publication of (a) the existence of the interaction resources that may interest future consumers (e.g., IAM applications), and (b) the communication ports that will allow consumers to send requests to use a particular interaction resource, and if successful, to obtain dedicated communication ports to exchange messages directly with the requested resource.

More precisely, an IAMPlatformManager running on an elementary platform P:

- Creates a *SurfacesContextor* whose role is to gather and publish the existence of all of the interaction resources that reside on P,
- Creates one *SurfaceManager* per physical surface connected to P,
- Creates an *InstrumentManager* that handles all of the instruments connected to P.

These components (SurfacesContextor, SurfaceManager, and InstrumentManager) are presented in more detail in the following paragraphs. Refer to Figure 4.5 for a detailed representation of their relations.

4.4.1 SURFACESCONTEXTOR

There is one SurfacesContextor per elementary platform.

A *SurfacesContextor* is a kind of contextor. A contextor is a software component whose role is to sense contextual information at the appropriate level of abstraction [Coutaz 02]. In I-AM, the SurfacesContextor of an elementary platform P receives data from the software SurfaceSensor attached to each one of the surfaces handled by P. It bundles this information at the appropriate level of abstraction so that, through a publish-subscribe mechanism supported by the contextors infrastructure, the world (e.g., the Logical Level of I-AM) can be informed about the resources P is able to provide.

At the opposite of PointRight whose resources are statically described in a configuration file, I-AM includes a powerful mechanism for dynamically discovering the surfaces, their properties, and their physical composition.

[7] « The world » denotes any software component that is not part of the IAMPlatformManager. IAM applications are examples of such software components.

[8] Current limitation: only the existence of surfaces is published

4.4.2 *SURFACE MANAGER*

There is one SurfaceManager per surface S of an elementary platform P. The SurfaceManager of a surface S covers the following functions:

- It gets and maintains the physical characteristics of S. As shown in Figure 4.3, the attributes of a surface include: height and width (both in pixels and millimeters) where digital information can be rendered, color and texture; for each of its edges, the width of the physical border (in millimeter); a coordinates system located at the top-left corner of the surface; and surface links that denote the physical connecting points of the surface with other surfaces. As mentioned above, links may be physical or logical sensors. Their location is specified in millimeters relatively to the top-bottom-left-right edges of surfaces. Figure 4.4 shows an example of two surfaces composed via two links A and B. For core surfaces, the SurfaceManager computes the physical characteristics of S by using the API from the underlying legacy Operating System.

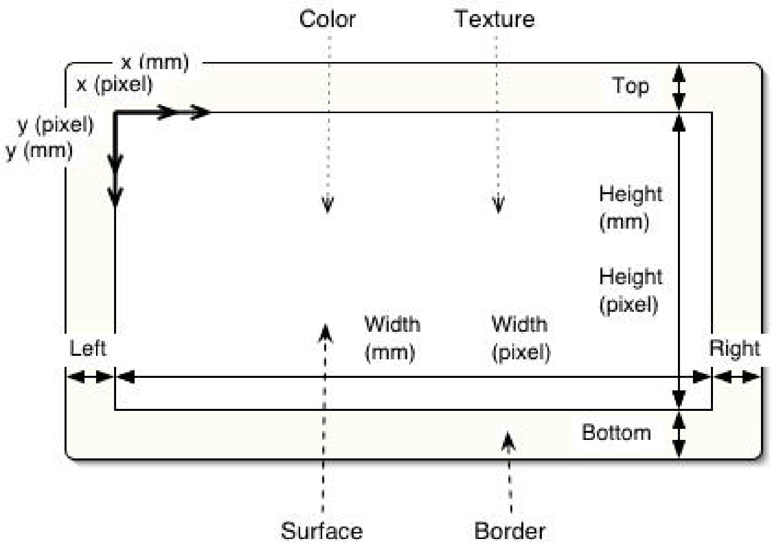
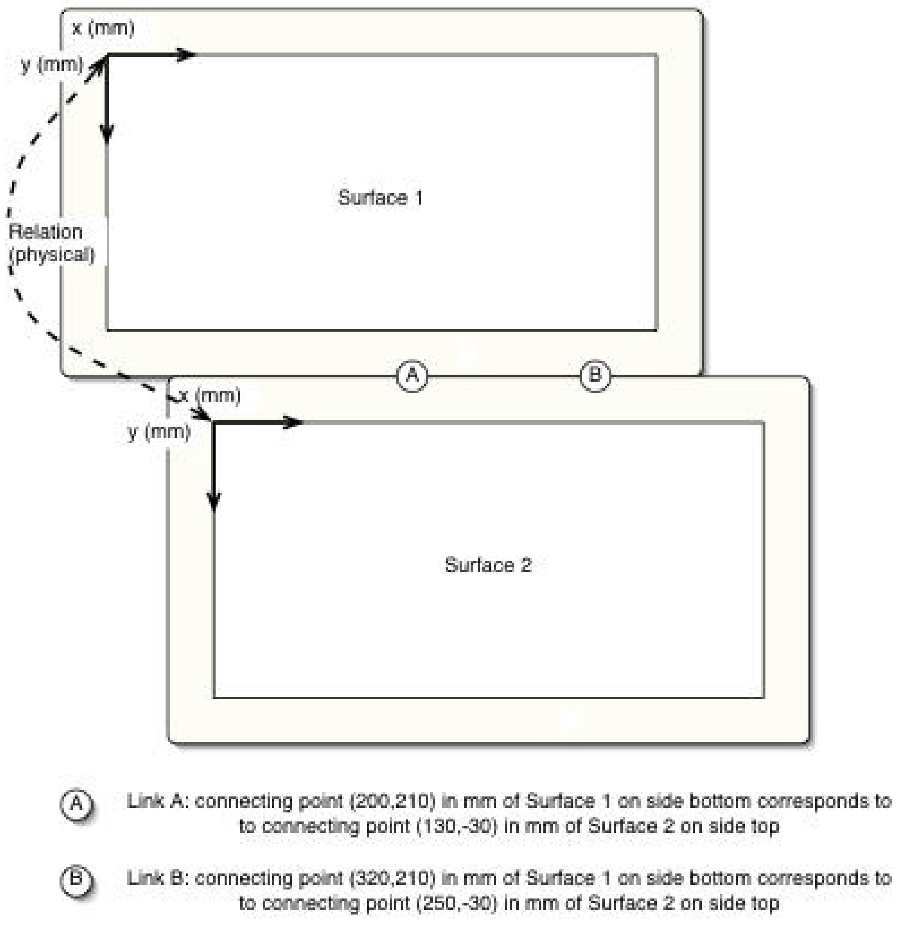


Figure 4.3. The attributes of an I-AM Surface.

- The SurfaceManager of S creates a SurfaceConnectionManager whose role is to open two communication ports on which it will listen for IAM applications requests such as “open connection with S to use S as an observation surface” or “open connection with S to use S as an action surface”.
- The SurfaceManager of S informs the SurfaceSensor of S about the characteristics of S (physical characteristics, surfaces links, communication ports, etc.). In turn, the SurfaceSensor of S transmits the information it receives to the SurfacesContextor that resides on P. As mentioned above, the SurfacesContextor aggregates the information it receives from every SurfaceManager of P into a global view for future resource consumers, e.g., IAM applications.
- The SurfaceManager of S creates a SurfaceDefaultPresentation whose role is to maintain the connection between the IAM application A that uses S and the interactors that effectively render the logical interactors of A: If A, an IAM application, is successful at opening a connection with S through the ActionSurfaceSocket or the ObservationSurfaceSocket of S, the

SurfaceDefaultPresentation of S creates an IAMAppEffectiveInteractorManager that binds S and A. This IAMAppEffectiveInteractorManager creates a local proxy to communicate with the possibly remote A. This includes two communication ports, depending on the way S is used. If S is used as an observation surface, then an IAMAppObservationProxy is created to send observation events to A about what has been observed on the surface (e.g., mouse clicks). If the surface is used as an action surface, then an IAMAppActionProxy is created to receive action events from A (e.g., create an IAMInteractor).



Link A: connecting point (200,210) in mm of Surface 1 on side bottom corresponds to to connecting point (130,-30) in mm of Surface 2 on side top

Link B: connecting point (320,210) in mm of Surface 1 on side bottom corresponds to to connecting point (250,-30) in mm of Surface 2 on side top

Figure 4.4. Two surfaces connected through two links A and B.

As discussed in Section 4.6, an IAM interactor, which is a logical interactor, is mapped, at the platform level, into a non empty set of Effective Interactors. As shown in 4.1, these effective Interactors belong to different surfaces. An IAMAppEffectiveInteractorManager, which binds together a surface S and an IAMApp A, memorizes the IAMEffectiveInteractors that are owned by A and rendered on S. By doing so, when the application A disappears, all of its “belongings” on S can be destroyed.

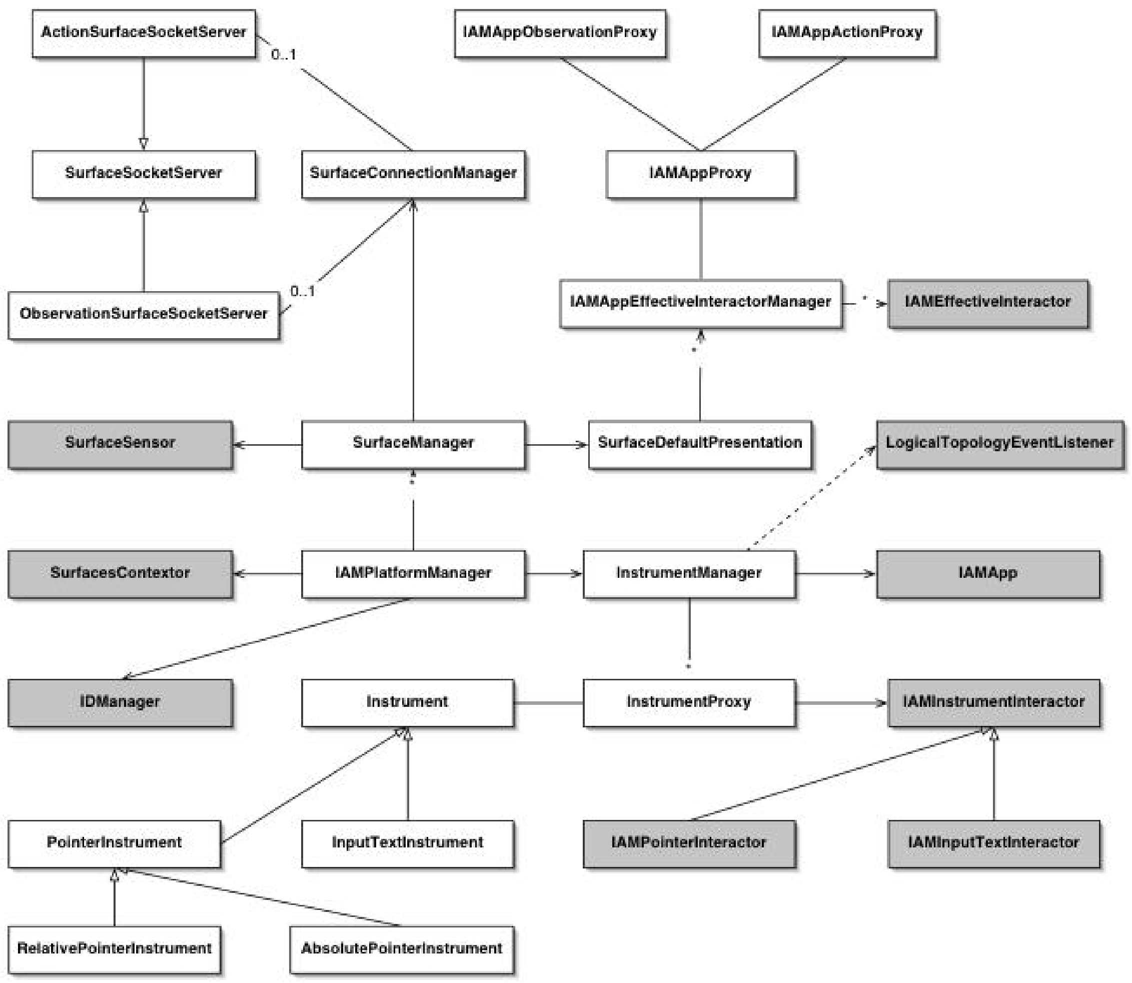


Figure 4.5. UML class diagram of the IAMPlatformManager package.

4.4.3 INSTRUMENT MANAGER

There is one InstrumentManager per elementary platform.

The role of the InstrumentManager of an elementary platform P is to manage the instruments handled by P and to perform the appropriate redirection of events generated with these instruments. Whereas PointRight performs events redirection based on the mechanisms of the underlying operating system and windowing system, I-AM re-uses for inputs, the same mapping technique developed for output. This technique is presented in Section 4.5.

As shown in Figure 4.5, instruments are of two types: InputTextInstruments and PointerInstruments. InputTextInstruments generate characters. Typical subclasses include physical keyboards and speech recognition systems. PointerInstruments include two subclasses: relative pointing instruments that provide relative positioning information (e.g., mouse has moved dx-dy pixels) while absolute pointing instruments such as laser beams and pens, provide a position in the coordinates system of a surface. New types of instruments such as the iStuff instruments [Ballagas 03], can be modeled as subclasses of the Instrument class.

Instruments that are core resources (e.g., mouse and keyboard) are discovered by the InstrumentManager using the API of the local underlying operating system. Extension instruments such as wireless iStuffs, are discovered through contextors. For every instrument handled by P, the InstrumentManager creates an InstrumentProxy as an abstract normalized representative of the physical instrument. In turn, this proxy creates an IAMInstrumentInteractor, that is, an IAMPointerInteractor if the physical instrument is a pointer instrument, or an IAMInputTextInteractor if the physical instrument is a text input instrument.

An IAMInstrumentInteractor (including its subclasses) is a logical interactor. Its association to an InstrumentProxy fulfills two goals:

. It allows I-AM to make the state of the physical instrument observable to the user. Typically, for a pointer instrument, the rendering of an IAMPointerInteractor is an arrow cursor.

. It allows I-AM to implement the events redirection by reusing the mapping that applies for output between logical and physical interactors. This mapping is presented in Section 4.5.

By default, the core InputTextInstrument and the core PointerInstrument of a platform P, are tightly coupled. This means that characters generated by an InputTextInstrument is automatically directed to the logical interactor that is the current focus of its associated PointerInstrument. For example, if a user, using the mouse of an elementary platform A selects a window rendered on the screen of an elementary platform B, then the characters typed with the keyboard of A, appears in the window of B. This default coupling can be overridden.

4.4.4 SUMMARY

In summary, the Platform Level of IAM hides away the heterogeneity of the underlying elementary platforms of the cluster. It provides the next level of IAM, i.e., the Logical Level, with:

(a) a normalized view of the set of physical interaction resources that are available on each elementary platform, and
(b) networked communication means to use these resources.

While the platform level is “elementary platform” centric, the Logical Level of IAM is application centric. This is where the illusion of a unified space is created.

4.5 THE LOGICAL LEVEL: THE IAMAPP PACKAGE

Applications that use an I-AM cluster may each have its own way to exploit and interpret the physical configuration of the cluster. Therefore, the Logical Level of I-AM provides applications with a means to build their own view on top of the physical platform level. To get and exploit such a view, an application must be an instance of an IAMApp. Figure 4.6 shows the UML class diagram of an IAMApp. First, we describe the functions provided by an IAMApp, then we discuss the mapping problem, a key issue to support the illusion of a unified space.

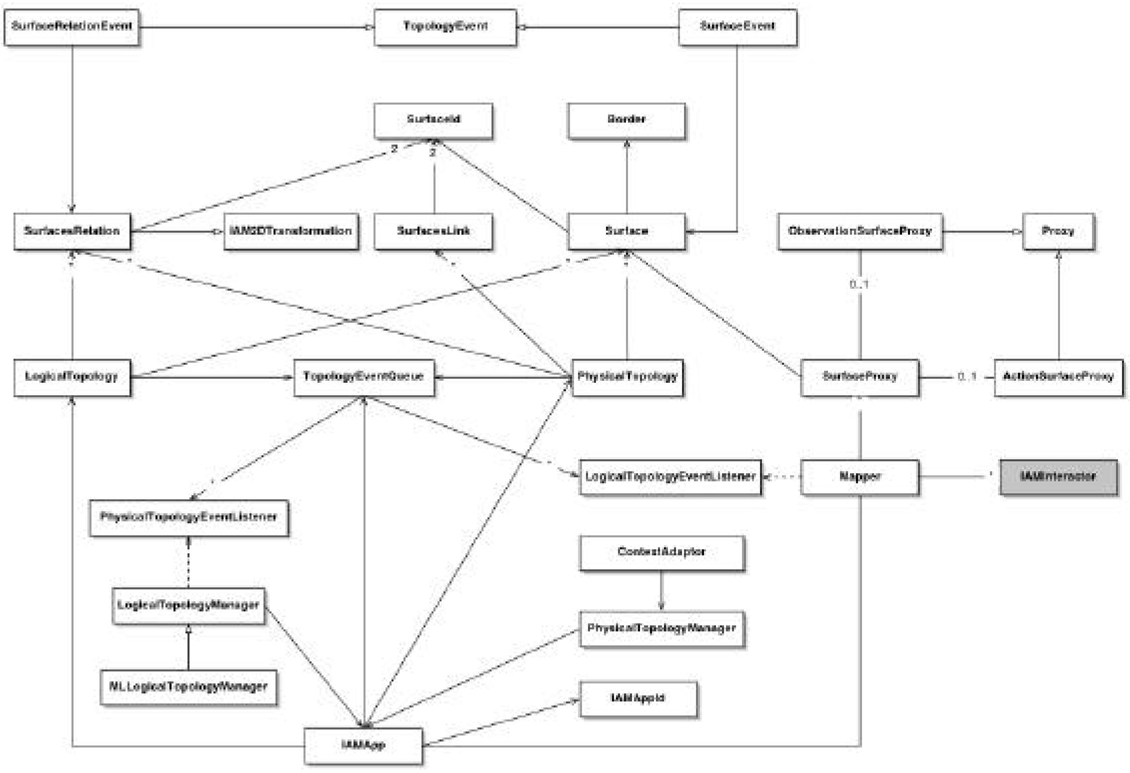


Figure 4.6. UML class diagram of the Logical Level of I-AM: an IAMApp.

4.5.1 FUNCTIONAL COVERAGE OF AN IAMAPP

An IAMApp A provides the following functions:

- It discovers the interaction resources[9] that are currently available in the cluster. To do so, an IAMApp expresses its interests through its ContextAdaptor. Expressions of interest include "give me a large surface", "give me the list of surfaces available in the cluster", "tell me when a new surface arrives or leaves" etc. As presented in [Coutaz 02], a ContextAdaptor serves as a gateway between an application and the contextors infrastructure. Contextual information about the physical interaction resources is compiled from the information produced by the SurfacesContextor that runs on each of the elementary platforms (Cf. Section 4.4.1). As illustrated in 4.5.2, the ContextAdaptor of A provides A with events that may have an impact on the topology of the surfaces used by A.
- It creates the communication channels with the interaction resources (surfaces) it is interested in. When the ContextAdaptator of an IAMApp A informs that a new surface S matches its expression of interest, A may request the SurfaceConnectionManager of S to open a communication channel for action and/or for observation. If the connection is successful, A creates a SurfaceProxy as its own representative of the (possibly remote) S. As presented in Section 4.4.2, a surface, at the platform level, maintains an IAMAppProxy per application that uses it. As a result, the port ObservationSurfaceProxy of S in A exchanges messages with its corresponding port IAMAppObservationProxy

[9] Current limitations: "interaction resources" should be understood as "surfaces" only.

associated with S at the platform level. Similarly, an ActionSurfaceproxy of the Logical Level exchanges messages with its corresponding IAMAppAction proxy of the Platform Level.

- It includes the mechanisms and politics to map the physical world with the digital LogicalSpace (and vice versa). The LogicalSpace is an infinite Cartesian space. The mapping problem and our solution are discussed next (Sections 4.5.2 and 4.5.3).
- It populates the LogicalSpace with IAMInteractors using the facilities provided by the IAMInteractor package (e.g., create, destruction, etc.) presented in Section 4.6.

4.5.2 THE MAPPING PROBLEM

The central problem that the Logical Level addresses is to entertain the illusion of a unified space while the user is interacting with physically disjoint surfaces that may differ in size, orientation, resolution, and so on. The following simple examples illustrate the problem of rendering a window interactor across two surfaces.

In the situation depicted in Figure 4.7, the window overlaps two surfaces S1 and S2 whose physical characteristics are strictly identical. In addition, the axes of their coordinates space are aligned. Let P1 be the intersection of the top-border line of the window with the right border of S1. To entertain the illusion of continuity, P1 must have a corresponding point P2 in S2. Let (x1, y1) be the coordinates of P1 in the coordinates system of S1 and (x2, y2), the coordinates of P2 in the coordinates system of S2. Then, for the situation depicted in Figure 4.7, P2.x2=P1.x1+1-W1 where W1 is the width of S1, and P2.y2=P1.y1. This simple example shows that a point P that belongs to an interactor in the LogicalSpace, is mapped in the physical world, as 2 points P1 and P2 related by geometric relationships (in our example, a translation on the X axis).

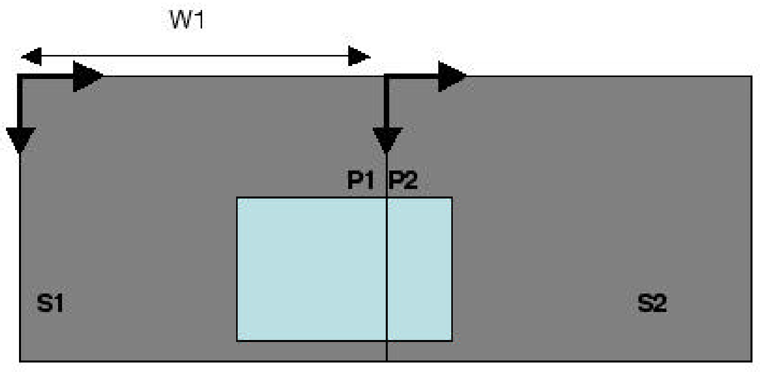


Figure 4.7. Mapping a window on two identical surfaces S1 and S2 whose coordinates systems are aligned.

Figure 4.8 illustrates the situation where S1 and S2 are composed as in 4.7 but where the resolutions of S1 and S2 differ. As shown in Figure 4.8, a translation is not enough. Rescaling is necessary to insure visual continuity.

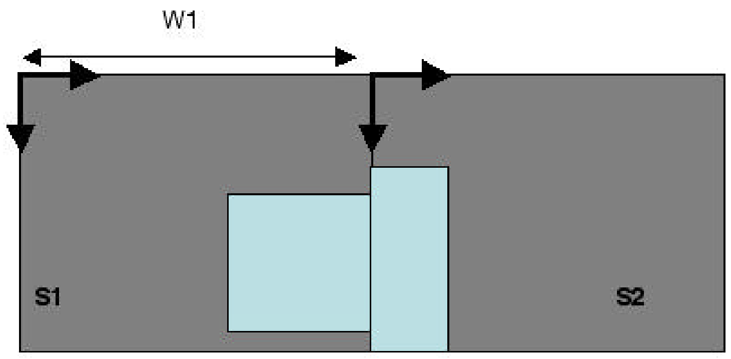


Figure 4.8. Mapping a window on two surfaces S1 and S2 whose coordinates systems are aligned, but whose resolutions are different.

Having illustrated the problem of entertaining the illusion of a unified space, we are now able to present the process used in I-AM to map the LogicalSpace of an IAMApp application to the physical space and vice versa.

4.5.3 *OUR SOLUTION TO THE MAPPING PROBLEM*

In I-AM, the mapping between the logical and the physical spaces is implemented as a three-step process:

. *From physical to normalized-physical spaces*: this step is performed by the PhysicalTopologyManager component. The PhysicalTopologyManager is in charge of creating and maintaining the PhysicalTopology, a data structure that models the spatial relationships of the surfaces used by an IAMApp.

. *From normalized-physical to logical spaces*: this step is ensured by the LogicalTopologyManager component. This component is in charge of creating and maintaining the LogicalTopology. The LogicalTopology is a data structure that defines the projection of the PhysicalTopology onto the LogicalSpace of an IAMApp.

. *From logical to physical spaces*: this step is performed by the Mapper component. Based on the LogicalTopology, this component is in charge of rendering the logical interactors of an IAMApp into their corresponding effective interactors.

The *PhysicalTopology* contains the list of surfaces (and their characteristics) used by A, as well as the two-by-two SurfacesLinks and SurfacesRelations.

. The surfaces used by A are those for which A has obtained a communication channel. (An application does not necessarily use all of the surfaces of the cluster.)

. As discussed in 4.4.2, a SurfacesLink denotes a physical composition point between 2 surfaces. For every SurfacesLink that connects two surfaces S1 and S2, the PhysicalTopologyManager computes a SurfacesRelation using the description of the SurfacesLinks and the geometric characteristics of S1 and S2 (height, width, borders size). This is how the mapping between “physical to normalized-physical” spaces occurs.

. A SurfacesRelation between two linked surfaces is an affine transformation (e.g., translation, rotation, shear, and scaling) between the coordinates systems of the two surfaces. Reusing the illustration of Figure 4.7, the SurfacesRelation between S1 and S2 is a translation on the X axis (where W1, the value of the translation, is expressed in millimeters). As demonstrated above, this transformation is necessary to produce the illusion of continuity between surfaces as users drags interactors and mouse cursors across surface boundaries.

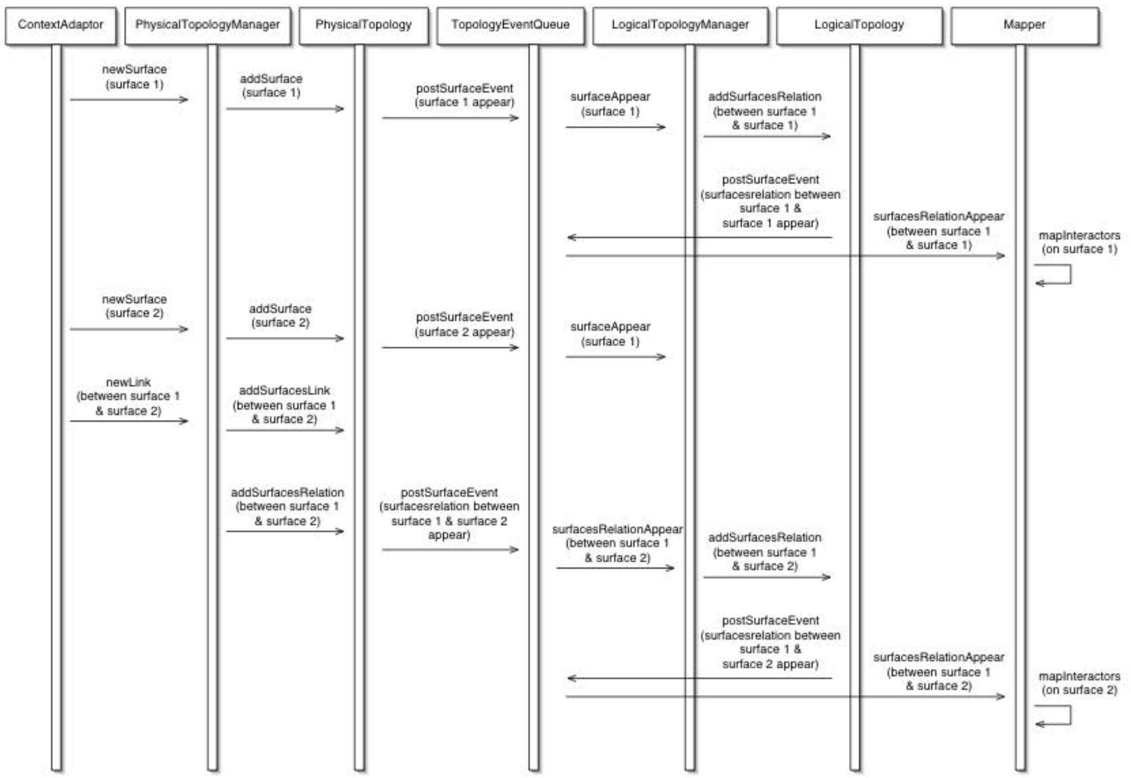

ContextAdaptor
PhysicalTopologyManager
PhysicalTopology
TopologyEventQueue
LogicalTopologyManager
LogicalTopology
Mapper
newSurface (surface 1)
addSurface (surface 1)
postSurfaceEvent (surface 1 appear)
surfaceAppear (surface 1)
addSurfacesRelation (between surface 1 & surface 1)
postSurfaceEvent (surfacesrelation between surface 1 & surface 1 appear)
surfacesRelationAppear (between surface 1 & surface 1)
mapInteractors (on surface 1)
newSurface (surface 2)
addSurface (surface 2)
postSurfaceEvent (surface 2 appear)
surfaceAppear (surface 1)
newLink (between surface 1 & surface 2)
addSurfacesLink (between surface 1 & surface 2)
addSurfacesRelation (between surface 1 & surface 2)
postSurfaceEvent (surfacesrelation between surface 1 & surface 2 appear)
surfacesRelationAppear (between surface 1 & surface 2)
addSurfacesRelation (between surface 1 & surface 2)
postSurfaceEvent (surfacesrelation between surface 1 & surface 2 appear)
surfacesRelationAppear (between surface 1 & surface 2)
mapInteractors (on surface 2)

Figure 4.9. Example of a sequence diagram when new surfaces and surfaces connections occur.

The *LogicalTopology* is built by the LogicalTopologyManager based on the knowledge of the PhysicalTopology. It contains the list of the surfaces that are mappable onto the LogicalSpace and, for each mappable surface S, a SurfacesRelation between S and a reference surface.

. By default, the *reference surface* is the core display of the elementary platform where A is launched. If the platform has no screen, then the first discovered screen becomes the reference surface. The reference surface is a mappable surface.

. Surfaces of the PhysicalTopology that are mappable onto the LogicalSpace are those that form a path that includes the reference surface. Surfaces that are not part of the path, are not mappable. A non-mappable surface S will become mappable when a SurfacesLinks appears between S and any mappable surface.

. A SurfacesRelation is an affine transformation (expressed in millimeters) between the coordinates systems of the reference surface and a mappable surface. This is how the mapping between "normalized-physical to logical" spaces occurs. And this is how I-AM allows developers to customize their own view of the physical space by letting them specify their own SurfacesRelation. For example, one politics is to ignore the borders of the physical surfaces as well as the space between the surfaces, while another politics would not. One may refer to D18 for a comparative discussion on the different types of transformation. In particular, we show the effect of surface borders on visual continuity, an attribute ignored in current graphical tools, including iRoom and i-LAND.

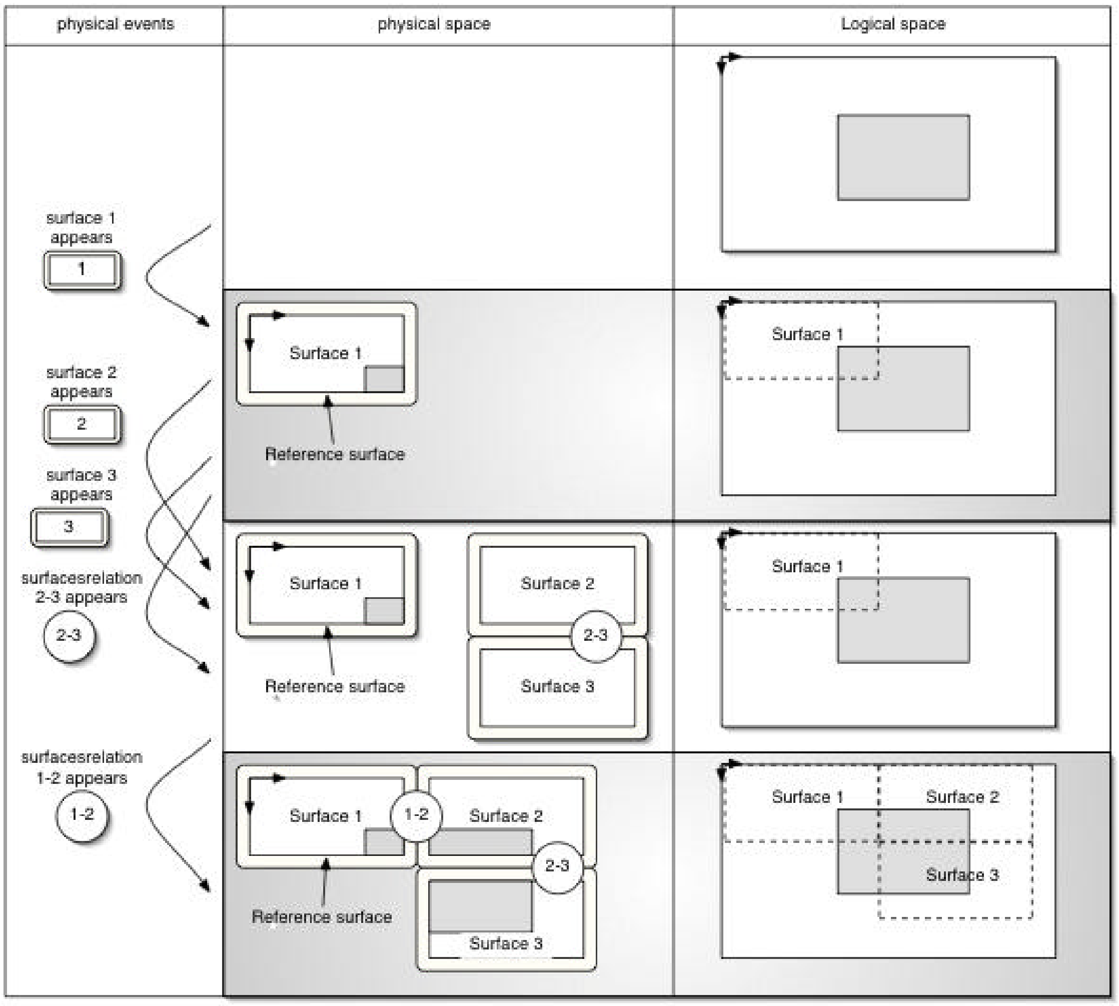


Figure 4.10. Mapping between the LogicalSpace and the PhysicalSpace. The left most column corresponds to a sequence of messages that notify the arrival of surfaces as well as their composition through SurfacesLinks. The right most column shows the mapping of the surfaces onto the LogicalSpace. The central column shows the resulting physical space as perceived by the user. The first row corresponds to the initial state: the LogicalSpace contains a grey rectangle and there is no surface to render the LogicalSpace. Then, surface 1 appears. By default, it becomes the reference surface and its coordinates system is aligned with that of the LogicalSpace. A portion of the LogicalSpace can be rendered. Surface 2 arrives in the cluster but is not composed with the reference surface: it is not mapped onto the LogicalSpace. Therefore, the arrival of the surface has no perceivable side effect. Idem for Surface 3. Surfaces 2 and 3 are now connected. Because none of them is connected to the reference surface, they are not mapped to the LogicalSpace. The last row shows the physical space that results from the composition of Surface 2 with the reference surface.

Having described the data structures and the role of the components involved in the mapping process, we now describe the dynamic aspect of the process. Any modification to the PhysicalTopology or to the LogicalTopology is reported as a TopologyEvent queued in the TopologyEventQueue. To receive the TopologyEvents of interest, the PhysicalTopologyManager and the LogicalTopologyManager are PhysicalTopologyEventListeners: both of them must be notified about changes in the physical world. On the other hand, the mapper, whose job is grounded on the LogicalTopology, is a LogicalTopologyEventListener.

The sequence diagram shown in Figure 4.9 makes explicit the effect of the arrival of the first surface of the cluster (surface 1); followed by the arrival of a second surface (surface 2) which is then connected physically to surface 1. Roughly speaking, any modification in the physical world that is detected by the ContextAdaptor of the IAMApp triggers the mapping process; the PhysicalTopologyManager translates the modification of the physical world into a modification of the PhysicalTopology which, in turn, triggers the LogicalTopologyManager. The arrival of a surface that is not physically connected to any mappable surface has no further effect. On the other hand, the arrival of the surface that is physically linked to a mappable surface has an impact on the Mapper: this new surface defines an additional peephole on the LogicalSpace. Therefore, the content of the LogicalSpace "hidden" behind the new surface must be rendered. Figure 4.10 provides another illustration of this process.

From the event "surfacesRelationsAppear between the reference surface 1 and the mappable surface 2", the Mapper gets the physical characteristics of surface 2 including its pixel size, as well as the projection of surface 2 in the LogicalSpace. In the LogicalSpace, pixels have a normalized size. In order to avoid the visual discontinuity shown in Figure 4.8, the Mapper computes a new transformation that combines the projection of the surface in the LogicalSpace with (possibly) rescaling. The transformation is sent to the SurfaceProxy of surface 2 so that surface 2 can render the effective interactors appropriately.

4.6 THE INTERACTOR LEVEL: IAMINTERACTOR PACKAGE

An IAMInteractor provides the programmer with the conventional programming paradigm. As a result, an IAMInteractor can be created, destroyed, moved, etc. in the LogicalSpace. It has a position in the LogicalSpace, it has a height and width expressed in terms of normalized pixels, etc. It hides away the facts that it can migrate seamlessly between surfaces at the pixel level.

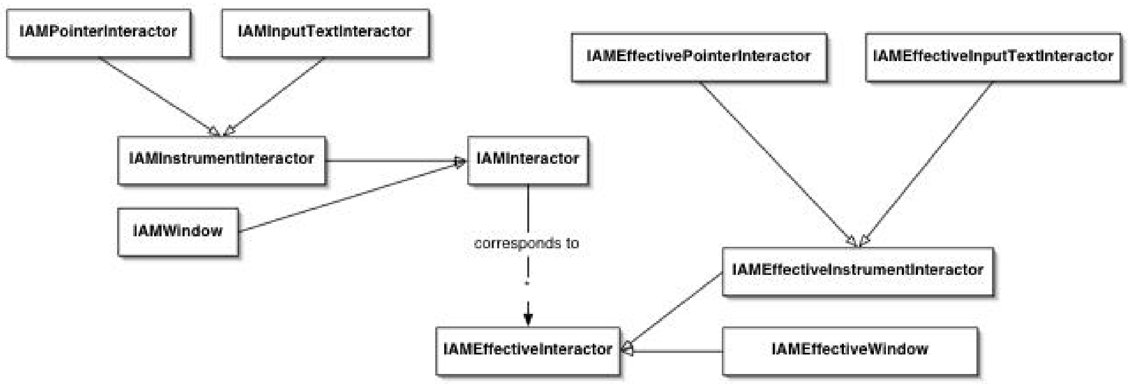


Figure 4.11. UML class diagram of the IAM Interactor Level.

To do so, an IAMInteractor I of an IAMApp A is mapped into several EffectiveInteractors, with one EffectiveInteractor E per surface S used by A. In other word, E results from the mapping process applied to I on S. It sends to I the input events it receives from the instruments for example "PointerInstrument M has entered", "PointerInstrument M has been clicked", etc. Conversely, any change of I by A is notified to all of its associated EffectiveInteractors.

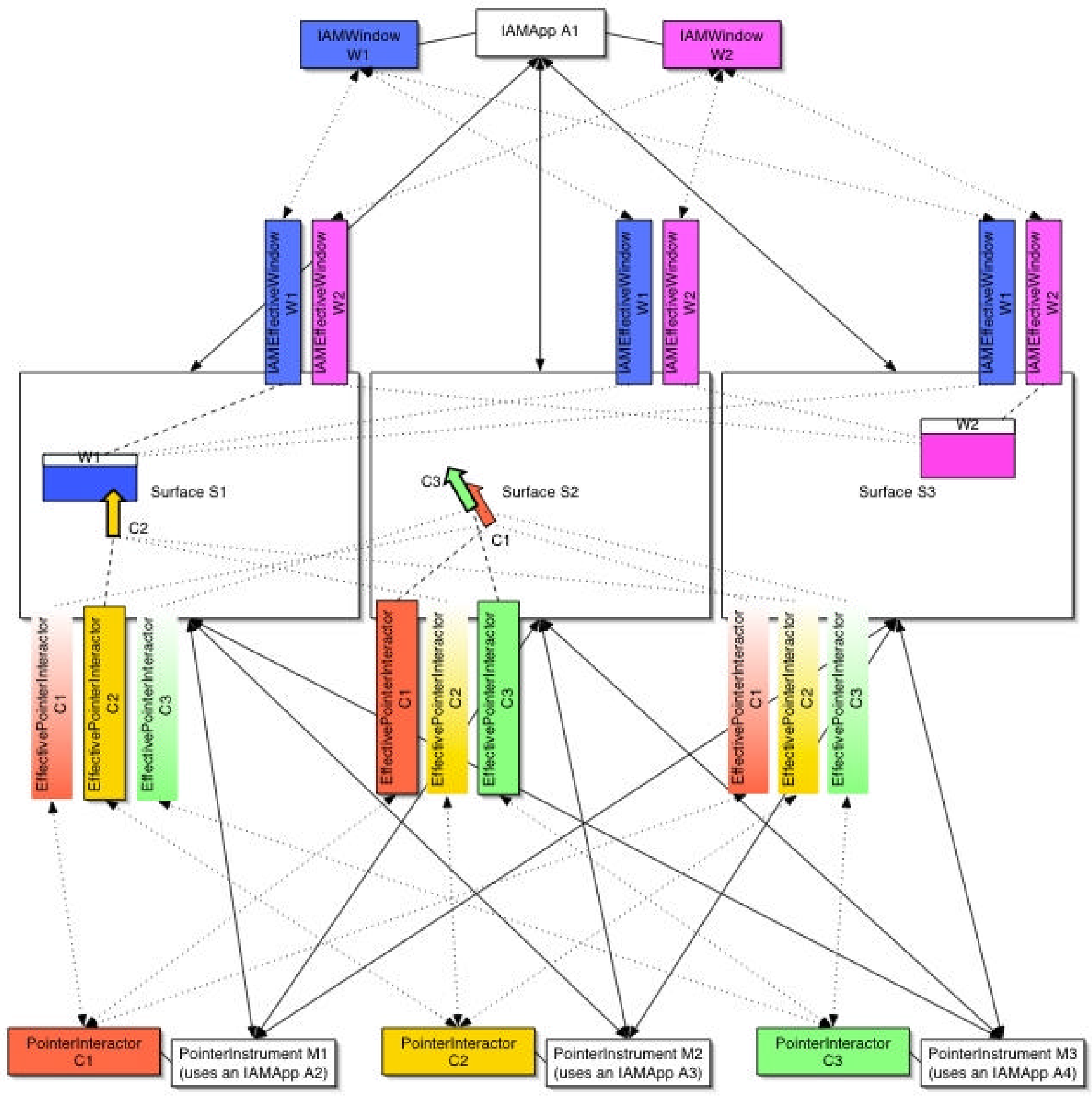

IAMWindow W1
IAMApp A1
IAMWindow W2
IAMEffectiveWindow W1
IAMEffectiveWindow W2
IAMEffectiveWindow W1
IAMEffectiveWindow W2
IAMEffectiveWindow W1
IAMEffectiveWindow W2
W1
Surface S1
C2
C3
Surface S2
C1
W2
Surface S3
EffectivePointerInteractor C1
EffectivePointerInteractor C2
EffectivePointerInteractor C3
EffectivePointerInteractor C1
EffectivePointerInteractor C2
EffectivePointerInteractor C3
EffectivePointerInteractor C1
EffectivePointerInteractor C2
EffectivePointerInteractor C3
PointerInteractor C1
PointerInstrument M1 (uses an IAMApp A2)
PointerInteractor C2
PointerInstrument M2 (uses an IAMApp A3)
PointerInteractor C3
PointerInstrument M3 (uses an IAMApp A4)

Figure 4.12. Example of relationships between an IAMApp A, its IAMInteractors, EffectiveInteractors, surfaces, and instruments.

Figure 4.11 shows the UML class diagram for IAMInterators. For now, we have implemented three subclasses of IAMInteractors: IAMwindow interactors as graphics containers, and IAMInstrumentInteractors that permit to graphically render the state of input text and pointer instruments. Similarly, we have defined the classes for corresponding EffectiveInteractors. In the current implementation, EffectiveInteractors are implemented with Swing. Another graphical toolkit, such as OpenGL is possible.

Figure 4.12 shows an example of the relationships between an IAMApp A, its IAMInteractors, EffectiveInteractors, surfaces, and instruments. In this example, IAMApp A has created two IAMWindows W1 and W2 in its LogicalSpace. It is currently using three mappable surfaces S1, S2, S3, each one handled by a distinct elementary platform. In addition, each one of the platforms handles a mouse instrument

M1, M2, M3. As discussed in Section 4.4.3, instruments are modeled as IAMInteractors owned by specific kinds of IAM Applications dedicated to instruments.

As shown in Figure 4.12, an IAMInteractor owns a corresponding EffectiveInteractor on every surface. Therefore, IAMWindow W1 owns an EffectiveWindow interactor W1 for S1, for S2, and for S3. Depending on the location of W1 in the LogicalSpace, either one of these EffectiveInteractors is visible, or, given the current topology of the surfaces, two of them at most are simultaneously visible (Cf. Description in 4.5.3).

The same mechanism applies to instruments. Every IAMPointerInstrument has one EffectiveInteractor per surface. Therefore, the user can take any mouse, move it across surfaces boundaries seamlessly to manipulate any effective interactor. For example, the user can take Mouse M1 to manipulate Window W2 currently visible on Surface S3 and bring it in S2. The user can observe the movement of M1 through the movement of the currently visible EffectivePointerInteractor C1.

As Figure 4.12 shows, IAM offers the possibility to show as many mouse cursors as there are pointing instruments. In the example, the cursors of M1 and M3 are in S2 hile that of M2 is in S1. This facility opens the way for new opportunities in multi-user interaction. However, from the user's perspective, the presence of multiple cursors, which may engender confusion, needs to be addressed carefully.

So far, we have presented the structure, services and mechanisms offered by I-AM. The next section briefly presents I-AM from the developer's perspective.

4.7 A PROGRAM EXAMPLE

The program shown in Figure 4.13 shows how to create a window that can be migrated seamlessly across surfaces possibly managed by a cluster of heterogeneous operating systems and machines.

```
//1.To exploit I-AM services, I need to be an IAMApp
    IAMApp myiamapp = new IAMApp ();

//2.I create a window mywindow whose barycentre is at point (300,300)
in the LogicalSpace, width and height (300, 200), and rotated by a
factor of Pi.
    IAMWindow mywindow = new IAMWindow ();
    mywindow.setCenterLocation (300, 300);
    mywindow.setSize (300,200);
    mywindow.setRotation (3.1416);

//3. I register mywindow so that it can be managed by I-AM.
    myiamapp.addInteractor (mywindow);
```

Figure 4.13. Creation of a migratable window across multiple surfaces.

Once step 3 of the program is executed, all of the mappables surfaces S used by *myiamapp* are notified of the existence of the window in the following way. For every

S, its IAMEffectiveInteractorManager receives the following XML messages (where the <action> tag denotes a method call and <id> denotes the identification of interactor[10]):

```
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=constructor></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setCenterLocation x=300 y=300></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setSize width=300 height=200></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setRotation rotation=3.1416></action>
```

When receiving the message with the attribute `method=constructor`, the IAMEffectiveInteractorManager of S creates an IAMEffectiveWindow that processes the rest of the message to produce the appropriate feedback on the surface, as well as any future message related to *mywindow*.

If, later on, the developer decides to change the size of the window:

```
mywindow.setSize (400,300);
```

then all of the mappable surfaces of *myiamapp* will receive the following message:

```
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setSize width=400 height=300></action>
```

and the corresponding IAMEffectiveInteractors will be updated.

If a new mappable surface arrives, its IAMAppEffectiveInteractorManager receives the following message which will create an IAMEffectiveInteractor for *mywindow* (note that the obsolete method calls have been discarded):

```
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=constructor></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setCenterLocation x=300 y=300></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setSize width=400 height=300></action>
<action id=129.87.31.43_4 class=IAMInteractor.IAMWindow
method=setRotation rotation=3.1416></action>
```

Conversely, if a surface disappears, its IAMAppEffectiveInteractorManager will be required by *myiamapp* to destroy its IAMEffectiveInteractors.

[10] id=129.87.31.43_4 means that *mywindow* has been created on the elementary platform whose ID is its IP address (i.e., 129.87.31.43) and 4 is a unique ID for this machine generated by the IDManager of I-AM.

So far, we have described how I-AM manages the interaction resources of an interactive space. This is one aspect of the general problem of plastic user interfaces. In the next section, we show how I-AM fits within an infrastructure that supports the run time adaptation of UI's.

5 I-AM WITHIN AN INFRASTRUCTURE FOR PLASTIC UI

Figure 5.1 shows the run-time infrastructure that UJF has designed for the CAMELEON R&D IST-2000-30104 European project. Details of this infrastructure can be found in [Coutaz 03a].

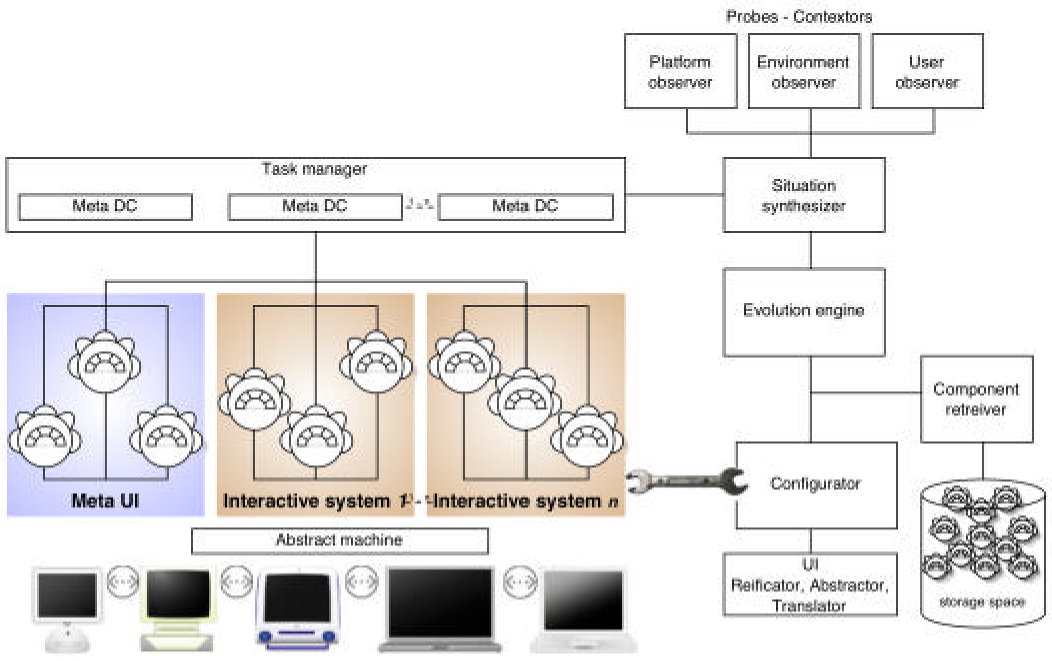


Figure 5.1. I-AM within the functional decomposition of the CAMELEON run time infrastructure.

The CAMELEON run time infrastructure includes three types of run time components: 1) the Interaction Abstract Machine that abstracts away the heterogeneity and the management of dynamic system resources, 2) close-adaptive components embedded in the interactive systems that currently run on the platform, and 3) specialised components to insure open-adaptiveness.

5.1 CLOSE-ADAPTIVE COMPONENTS

Close-adaptive components either are constituents of the interactive systems that currently run on the platform, or items saved in a data base for future use when adaptation is required. Interactive systems may be conventional applications (for example, a document editor, a video-viewer) or a meta-user interface.

A *meta-user interface* is to clusters and distributed UIs what the desktop is to centralised UIs on conventional PC's. Like the desktop, it serves as an interactive glue between the application-oriented interactive systems. It includes:

- Interaction techniques that allow users and the system to control the interaction resources of the platform. For example, coupling interactive surfaces by proximity as in Rekimoto's Pick and Drop [Rekimoto 97], by alignment [Tandler 01b], by bumping as in [Henckley 03], by synchronous clicks on dedicated keyboard keys [Rekimoto 03], or by any new technique yet to be invented;

- Interaction techniques, including negotiation, that allow users and the system to express the redistribution, migration, adaptation of all (or part of) the user interface of an interactive system, including the meta-user interface.

5.2 COMPONENTS FOR OPEN-ADAPTIVENESS

Open-adaptivity is ensured by a set of dedicated components that, by definition, are external to the interactive systems that currently run o the cluster. They include observers, a situation synthesizer, an evolution engine, a component retriever, and a configurator assisted by reificators, abstractors, and translators.

- *Observers* detect the causes for potential adaptation. Some observers detect changes in the context of use: these are the *platform*, *environment* and *user observers*. Our contextors infrastructure offers a way to implement these observers. The Aura environment manager and context observer correspond, respectively, to our platform and environment observers. A particular observer, the *task manager,* probes and maintains users' evolution within the task space. The task space is composed of the task spaces supported by each one of the interactive systems that currently run on the cluster. The Aura Prism component is an instance of a task manager.
- The *situation synthesizer* computes the current situation and context from information provided by the observers. As discussed in [Crowley 02], a new situation may be recognized, for example a change within the surfaces topology (e.g., a new PDA has arrived).
- The *evolution engine* computes a reaction in response to the new situation or new context. The response may require one or more of the current running interactive systems to adapt. If the adaptation required by the new situation falls within the domain of plasticity of the interactive system, then the interactive system is able to self-adapt. On the other hand, if the required adaptation does not fall within the domain of plasticity of the interactive system, then the evolution engine retrieves the appropriate components from the components data base and produces a description of a new configuration of the interactive system using an ADL (Architecture Description Language).
- The *configurator* creates a new executable interactive system from a configuration description. The components referred to in the description do not necessarily correspond to executable code. They may instead be high-level descriptions such as task models. If so, the configurator relies on *reificators* such as Teresa [Paternò 02] and ARTStudio [Calvary 02] to produce executable code. Conversely, a component may need to be reversed engineered through *abstractors*, and then transformed by *translators* and reified again into executable code as in Vaquita [Vanderdonckt 01].

This infrastructure has been implemented and experimented with the development of an exemplar application: CamNote.

5.3 AN EXAMPLE: CAMNOTE

CamNote (for CAMELEON Note) is a Slides viewer that can run on a dynamic cluster composed of a single PC or of a PC and a PocketPC. It is composed of three components:

- A slides viewer. On the PC, this viewer allows the insertion of translucent videos provided by cameras (also known as pixels mirrors [Morikawa 98] [Vernier 99]). This component is available for PC's only.
- A personal notes editor/viewer. Comments can be entered for each slide. This component exists for PC's only.
- A remote controller that allows the user to navigate from slides to slides and to control the level of transparency of the pixels mirror. A remote controller component is available for both PC's and for PocketPC's.

The user interface of CamNote can be rotated at the workspace level. In other words, windows do not need to be parallel to the borders of the display screens. Figures 5.2 and 5.3 show screen dumps of the CamNote user interface.

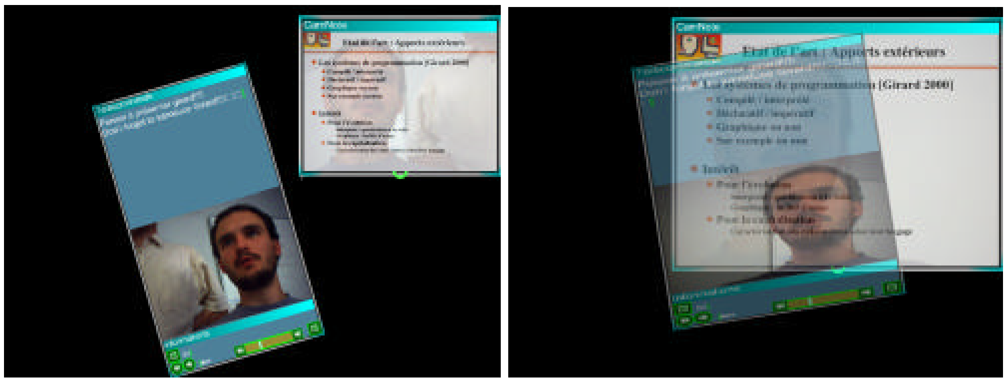


Figure 5.2. On the left, CamNote when the user interface is centralised on a PC screen. At the top right, the slides viewer window. In the middle of the screen a rotative window groups together a remote controller for navigating between the slides and controlling the transparency level of the pixels mirror, a personal notes viewer, and the video image of the speaker. Rotation may be needed when the window is migrated to an horizontal surface. On the right, the PC screen during the adaptation process. In this configuration, the user is enlarging the size of the slides viewer using a mouse. The evolution model expresses the following adaptation: "If the screen real estate gets too small to show the remote controller, then replace the graphic mouse-driven remote controller with a keyboard-driven remote controller and, if there is a PDA, migrate the remote controller to the PDA". Therefore, migration is partial and is performed at the workspace level (Cf. Section 2). As shown in the picture, during the adaptation process, the mouse-driven remote controller dynamically weaves itself into the slides viewer window and them disappears. This movement is a kind of Meta-UI. Figure 5.3 shows the UI of CamNote that results from the adaptation process.

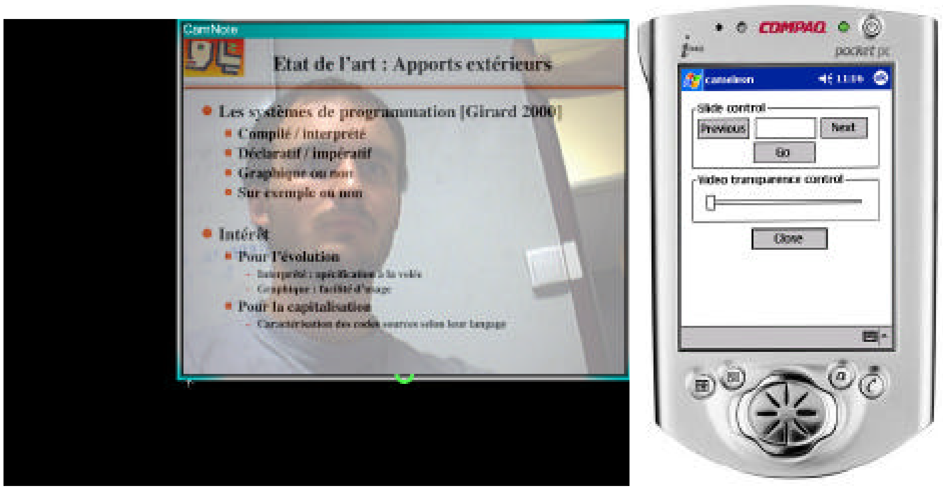


Figure 5.3. The user interface of CamNote when distributed on a PC screen and a PocketPC screen. This distribution results from the migration of the remote controller from the PC to the PDA. The original remote controller available on a PC has been replaced with a new remote controller whose buttons and layout are best suited to the PocketPC. The UI of CamNote is therefore distributable, migratable and plastic. This is made possible in a transparent way for the user and the developer by the underlying run time infrastructure.

6 CONCLUSION

In summary, we have designed a reference framework that helps understanding, reasoning and implementing user interfaces for global smart spaces. This framework includes an ontology that makes explicit the concepts of multi-surface interaction and, based on this ontology, I-AM, a software infrastructure that supports the dynamic composition of heterogeneous interaction resources connected into a unified space. In this space, users can distribute and migrate whole or parts of user interfaces as if these components were handled by a unique computer. I-AM provides users with the illusion of a unified space at no extra cost for the developer. We have shown how I-AM can be integrated in a wider infrastructure designed to support the run time adaptation of plastic user interfaces. I-AM is implemented in Java. It currently runs on clusters composed of MacOS X, Windows NT and Windows XP.

From a pure technical perspective, I-AM advances the state of the art by addressing all of the following problems:

1. Platforms heterogeneity (e.g., clusters of machines running a mix of MacOs X, Windows NT and Windows XP),
2. Interaction resources heterogeneity (e.g., screens with different sizes and resolutions),
3. Platforms and interaction resources discovery based on a fabric of contextors,
4. Multi-surface interaction grounded on the dynamic composition of hinged display surfaces whose spatial relationships are automatically modeled and maintained,
5. Multi-keyboard, multi-pointer capabilities (so that a user can use the mouse of a PC to manipulate a window displayed on a MacOS screen and drag the window across screens boundaries as if there were a single screen).

Although our ontology and I-AM push the state of the art one step forward, a number of limitations need to be addressed. In particular, we need to measure the performance of I-AM in a formal way (e.g., latency) and consider legacy applications more carefully. We need to integrate advanced sensory technologies and invent the Meta-UI that goes beyond the desktop.

We plan to integrate sensory technologies for two primary purposes: 1) to maintain a 3D topology of the interaction resources and 2) to capture additional physical characteristics of surfaces such as sophisticated shapes, textures, weight, etc. in order to infer the properties we have elicited in our ontology: for example, is the surface traversable? Can it be carried around?

We plan to invent elements of the Meta-UI. As discussed briefly in section 5, a Meta-UI covers many aspects of human computer interaction. Let's look at one example: for traditional workstations, the state of the interaction resources is observable in a simple manner. In a ubiquitous world, users need to know which resources are in reach and how they can be composed, borrowed and lent. Early work on this problem such as that of Hinckley [Hinckley 03] and Rekimoto [Rekimoto 03] addresses simple cases (e.g., connect two workstations or tablets). Although, we have not yet devised any Meta-UI except for the migration of user interface components (Cf. CamNote), I-AM offers the technical foundations for exploring this new problem.

7 REFERENCES

[Ballagas 03] Ballagas, R., Ringel, M., Stone, M., Borchers, I. IStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environment. *In Proc. CHI 2003*, Fort Lauderdale, April 5-10, 2003, ACM Publ., 5(1), pp. 537-544

[Bederson 00] Bederson, B., Meyer, J. Good, L. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *In Proceedings of UIST 2000*. May 2000. p171-180.

[Bier 92] Bier E., Freeman, S., Pier, K., The Multi-Device Multi-User Multi-Editor. *In Proc. of the ACM conf. On Human Factors in Computer Human Interaction (CHI92)*, (1992), pp. 645-646.

[Borkowski 03] Borkowski, S., Riff, O., Crowley, J.L. Projecting rectified images in an augmented environment. *In Proceedings of ProCams Workshop*. International Conference on Computer Vision (ICCV 2003) , IEEE Computer Society Press, October 2003, Nice, France.

[Brumitt 01] Brumitt, B., Shafer, S. Better Living Through Geometry. *Personal and Ubiquitous Computing 2001*. Vol 5.1 Springer.

[Calvary 01] Calvary, G., Coutaz, J. Thevenin. D. A Unifying Reference Framework for the Development of Plastic User Interfaces. *IFIP WG2.7 (13.2) Working Conference, EHCI01*,Toronto, May 2001, Springer Verlag Publ., LNCS 2254, M. Reed Little, L. Nigay Eds, pp.173-192.

[Coutaz 02] Coutaz, J., Rey, G. Foundations for a theory of Contextors. Proc. of *Computer-Aided Design of User Interfaces III*, J. Vanderdonckt, C. Kolski Eds., Kluver Academic Publ., 2002, pp. 13-32.

[Coutaz 03] Coutaz, J., Lachenal, C., Dupuy-Chessa, S. Ontology for Multi-surface Interaction. Proc. *Interact 2003*, M. Rauterberg et al. Eds, IOS Press Publ., IFIP, 2003, pp.447-454.

[Coutaz 03a] Coutaz, J., Balme, L., Barralon, N., Calvary, G., Demeure, A., Lachenal, C., Rey, G., Bandelloni, R., Paternò, F. *Initial Version of the CAMELEON Run Time Infrastructure for User Interface Adaptation*, Deliverable D2.2 V1.1, October 2003, Cameleon project, IST 2000-30104, http://giove.cnuce.cnr.it/cameleon.html

[Crowley 02] Crowley, J., Coutaz, J., Rey, G., Reignier, P. Perceptual Components for Context-Aware Computing, *UbiComp 2002:Ubiquitous Computing*, 4th International Conference, Göteburg, Sweden, Sept./Oct. 2002, G. Borriello, L.E. Holmquist Eds., LNCS, Springer Publ., 2002, pp. 117-134.

[Garlan 01] Garlan, D., Schmerl, B., Chang, J. Using Gauges for Architectural-Based Monitoring and Adaptation. *Working Conf. on Complex and Dynamic Systems Architecture*, Australia, Dec. 2001.

[Gelernter 92] Gelernter, D., and Carriero, N., Coordination Languages and their Significance, *Communications of the ACM*, Vol.32, Number 2, February, 1992.

[Gregor 97] Gregor, K. et al. *Aspect-Oriented Programming. In proceedings of ECOOP'97*, LNCS 1241, Springer-Verlag, pp. 220-242, Juin 97.

[Guttman 01] Guttman, E. Autoconfiguration for IP Networking : Enable Local Communication. *IEEE Internet Computing*, May-June 2001, pp. 81-86.

[Hinckley 03] Hinckley, K. Synchronous gestures for multiple persons and computers. *Proc. UIST 2003*, ACM, 2003.

[Hourcade 99] Hourcade J., Bederson, B. Architecture and Implementation of a Java Package for Multiple Input Devices (MID). 1999 http://www.cs.umd.edu/hcil/mid/

[Hubinette 02] Hubinette, F. x2vnc 1.31 (home page)

[Johanson 02a] Johanson, B., Hutchins, G., Winograd, T. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. *In Proc. of User Interface Software and Technology (UIST 2002)*, ACM Publ., pp. 227-234.

[Johanson 02b] Johanson, B., Fox, A. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. *In Proc. of the 4th IEEE Workshop on Mobile Computer Systems and Applications (WMCSA 2002)*, June, 2002.

[Luyten 2002] Luyten, K., Vandervelpen, C., Coninx, K. Migratable user interfaces Descriptions in Component-Based Development. *DSV-IS 2002*, Rostock, Springer Verlag Publ., 2002.

[Microsoft 00] Understanding Universal Plug and Play, White Paper, *Windows Millenium edition*, Microsoft 2000.

[Morikawa 98] Morikawa, O., Maesako, T. HyperMirror : Toward Pleasant-to-use Video Mediated Communication System. In *proceedings of CSCW'98*, ACM Publ., Seattle, Washington USA. pp. 149-158

[Myers 98] Myers, B., Stiel, H., Gargiulo, R. Collaboration Using Multiple PDAs Connected to a PC. *In Proceedings CSCW'98: ACM Conference on Computer-Supported Cooperative Work*, 1998, Seattle, WA. pp. 285-294.

[Oreizy 99] Oreizy, P., Taylor, R., et al. An Architecture-Based Approach to Self-Adaptive Software. *In IEEE Intelligent Systems*, May-June, 1999, pp. 54-62.

[Paternò 02] Paternò, F., Santoro, C. One model, many interfaces. *In Proc. Computer-Aided Design of User Interfaces III (CADUI)*, J. Vanderdonckt, C. Kolski Eds., Kluver Academic Publ., 2002.

[Rekimoto 97] Rekimoto, J. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments. *In Proceedings of UIST'97*, ACM Publ., 1997, pp. 31-39.

[Rekimoto 99] Rekimoto, J., Masanori, S. Augmented Surfaces : A Spatially Continous Workspace for Hybrid Computing Environments. *Proceedings of CHI'99*, ACM publ., 1999.

[Rekimoto 03] Rekimoto, J., Ayatsuka, Y., Kohno, M. SyncTap: an Interaction Technique for Mobile Networking. In *Proc. Mobile HCI 2003*, L. Chittaro Ed., Springer Publ., LNCS 2795, pp. 104-115.

[Richardson 98] Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A. Virtual Network Computing, *IEEE Internet Computing*, Vol 2, No 1, Jan/Feb 1998, pp. 33-38.

[Sousa 02] de Sousa, J., Garlan, D. Aura : an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *IEEE-IFIP Conf. on Software Architecture*, Montreal, 2002.

[Streitz 99] Streitz, et al. i-LAND: An interactive Landscape for Creativity and Innovation. *In Proceedings of CHI'99*, ACM publ.

[Tandler 01a] Tandler, P. Software Infrastructure for Ubiquitous computing Environments : Supporting synchronous Collaboration with Heterogenous devices. *In Proceedings of UbiComp 2001*, Springer Publ.

[Tandler 01b] Tandler, P., Prante, T., Müller-Tomfelde, C., Streitz, N., Steinmetz, R. ConnecTables: Dynamic Coupling of Displays for the Flexible Creation of Shared Workspaces. *In Proc. UIST 2001*, ACM publ., 2001, pp. 11-20.

[Thevenin 99] Thevenin, D., Coutaz, J. Plasticity of User Interfaces: Framework and Research Agenda. *In Proc. Interact99*, Edinburgh, , A. Sasse & C. Johnson Eds, IFIP IOS Press Publ. , 1999, pp.110-117.

[Vanderdonckt 01] Vanderdonckt, J., Bouillon, L., and Souchon, N., Flexible Reverse Engineering of Web Pages with Vaquita. *In Proc. WCRE'200: IEEE 8th Working Conference on Reverse Engineering*. Stuttgart, October 2001. IEEE Press.

[Vernier 99] Vernier, F., Lachenal,C., Nigay, L., Coutaz, J. Interface Augmentée Par Effet Miroir, in *Proc. IHM'99*. (AFIHM conference on Human-Machine Interface, 22-26 November 1999 Montpellier, France), Cepadues Publ., pp. 158-165.

[Viswanathan 01] Viswanathan, P., Gill, B., Campbell, R. *Security Architecture in Gaia*. University of Illinois, report, UIUCDCS-R-2001-2215, May, 2001.

[Want 01] Want, et al. The Personnal Server : The Center of Your Ubiquitous World. *Intel Research White Paper*, May 2001.

ANNEX1: FORMAL DEFINITION OF A PLATFORM

Let

- C be the set of core configurations
- E be the set of extension resources
- C', C'' $\neq$ {}: C' $\subset$ C and C'' = C – C'
- E', E'': E' $\subset$ E and E'' = E – E'
- $c_1, \ldots, c_n \in C$, $e_1, \ldots, e_m \in E$ for $n \in N^*$, $m \in N$
- Operational be a predicate over a set of resources that returns true when this set forms a working computational artefact whose state can be observed and/or modified by a human user.

A platform is composed of a set of core and extension resources which, connected together, form a working computational artefact whose state can be observed and/or modified by a human user:

$P = \{ c_1, \ldots, c_n \} \cup \{ e_1, \ldots, e_m \}$ and Operational (P)

P is an elementary platform if and only if:

$\neg\exists$ C', E', C'', E'': Operational (C' $\cup$ E') and Operational (C'' $\cup$ E'').

In other words, P is an elementary platform if it not possible to build two platforms from the set of resources that constitute P.

P is a cluster if it is possible to compose multiple platforms from the set of resources that constitute P:

$\exists$ C', E', C'', E'': Operational (C' $\cup$ E') and Operational (C'' $\cup$ E'').

Note that:

A core configuration is not necessarily Operational. For example, the Intel Personal Server, a Bluetooth-enabled micro-drive with no interaction device, is a core configuration but not an elementary platform: its state cannot be observed nor modified until it wirelessly connects to extension resources such as a display and/or a keyboard [Want 01]. On the other hand, a laptop is an elementary platform even when augmented with extension resources such as a second mouse and sensors. Similarly, Rekimoto's data tiles form an elementary platform that can be dynamically extended by placing physical transparent tiles on a tray composed of an LCD flat screen display [Rekimoto 01].