EHCI-DSVIS'2004

Towards a new generation of widgets for supporting software plasticity: the "comet"

Gaëlle Calvary, Joëlle Coutaz, Olfa Dâassi, Lionel Balme, Alexandre Demeure

CLIPS-IMAG,
BP 53, 38041 Grenoble Cedex 9, France
{Gaelle.Calvary, Joelle.Coutaz}@imag.fr

Abstract. This paper addresses software adaptation to context of use. It goes one step further than our early work on plasticity [5]. Here, we propose a revision of the notion of software plasticity that we apply at the widget level in terms of comets. Plasticity is defined as the ability of an interactive system to withstand variations of context of use while preserving quality in use where quality in use refers to the ISO definition. Plasticity is not limited to the UI components of an interactive system, nor to a single platform: adaptation to context of use may also impact the functional core, it may have an effect on the nature of the connectors, and it may draw upon the existence of multiple platforms in the vicinity to migrate all or portions of the interactive system. A new reference framework that structures the development process of plastic interactive systems is presented to cover these issues. The framework is then applied at the granularity of widgets to provide the notion of a comet. A comet is an introspective widget that is able to self-adapt to some context of use, or that can be adapted by a tier-component to the context of use, or that can be dynamically discarded (versus recruited) when it is unable (versus able) to cover the current context of use. To do so, a comet publishes the quality in use it guarantees, the user tasks and the domain concepts that it is able to support, as well as the extent to which it supports adaptation.

1 Introduction

Mobility coupled with the development of a wide variety of access devices has engendered new requirements for HCI such as the ability of interactive systems to run in different contexts of use. By context of use we mean a triple <user, platform, environment> where the user denotes the archetypal person who is intended to use the interactive system; the platform refers to the hardware and software devices available for sustaining the user interaction; the environment describes the physical and social conditions where the interaction takes place. To master the diversity of contexts of use in an economical and ergonomic way, the *plasticity* property has been introduced [31]. Basically, plasticity refers to the adaptation to context of use that preserves the user's needs and abilities. For example, FlexClock [15] is a clock that expands or shrinks its user interface (UI) when the user resizes the window (Fig. 1). The time remains readable during and after the adaptation.

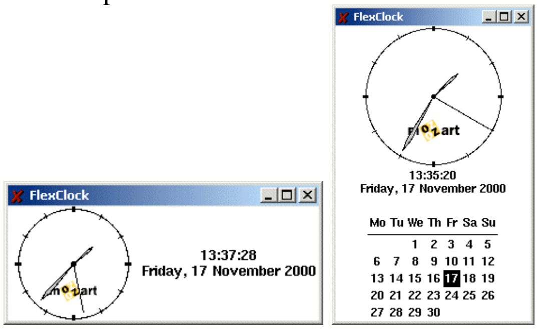

Fig. 1. FlexClock, an example of adaptation to the platform.

When applied at the widget level, the plasticity property gives rise to a new generation of widgets: the *comets* (COntext of use Mouldable widgETs). As a simple example, a set of radio buttons that shrinks into a combo box is a comet (Fig. 2).

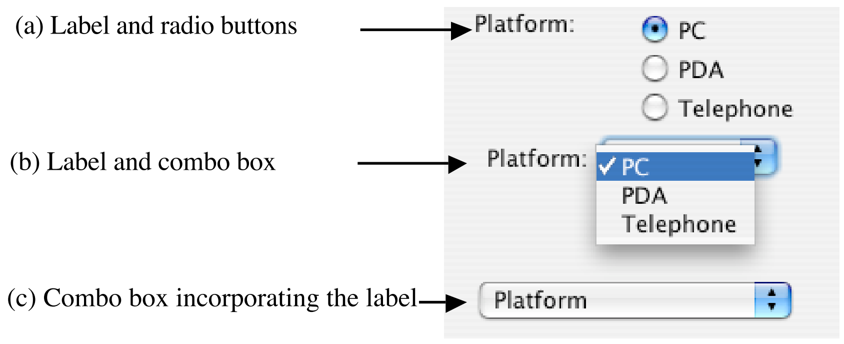

Fig. 2. Three graphical mockups supporting the same task “selecting one option among a set of options” through a) a label and radio buttons; b) a label and a combo box; c) a combo box incorporating the label. The example concerns the specification of the target platform (PC, PDA, telephone) for a centralized UI.

This paper presents our notion of comets. First we present new advances in plasticity to provide sound foundations for their elaboration. Then we focus on the comets per se considering both the design and run time perspective.

2 Foundations for comets: advances in plasticity

This section focuses on the lessons learned from experience that directly underpin the notion of comets. First, we propose a new definition for plasticity, then we examine the property from both a user and a system centered perspective.

2.1 A new definition of plasticity

Plasticity was previously defined as “the capacity of a user interface to withstand variations of context of use while preserving usability” [31]. Based on our experience, we have identified three reasons for revising the definition:

- In reality, plasticity is not limited to the UI components but may also impact the functional core. This occurs typically with services discovery. For example, because Bob has moved and is now in a place that makes a new service available, this service now appears on his PDA. The desktop is reshuffled (or tuned) to incorporate this new service and support an opportunistic interaction. Thus, the scope of the definition must be enlarged: plasticity must refer to the capacity of an *interactive system*, and not only to its UI, to adapt to the context of use;
- The current definition focuses on the preservation of usability only. As a result, utility is implicit. To make explicit the possibility to specify requirements concerning the preservation of functional (and not only non functional) properties (e.g., task accomplishment), the scope of the definition must be enlarged. To do so, we refer to *quality in use* instead of just usability. As defined by ISO [18], quality in use is based on internal and external properties (Fig. 3) including usability (Fig. 4);
- The definition is not operational enough. Due to ISO, the definition is now reinforced by a set of reference *characteristics* (factors), *sub-characteristics* (criteria) (Fig. 4) and metrics [19]. The framework QUIM (Quality in Use Integrated Map) [29] also contributes in this area by relating data, metrics, criteria and factors. A sound basis exists in HCI for usability ([1] [17] or more specifically [32] for dialog models).

Based on this new definition, an interactive system is said to be “plastic for a set of properties and a set of contexts of use” if it is able to guarantee these properties whilst adapting to cover another context of use. The properties are selected during the specification phase among the set of characteristics and sub-characteristics elicited by ISO (Fig. 4). Thus, plasticity is not an absolute property: it is specified and evaluated against a set of relevant properties (e.g., the latency and stability of the interactive system with regard to the “efficiency” characteristic, “time behavior” sub-characteristic).

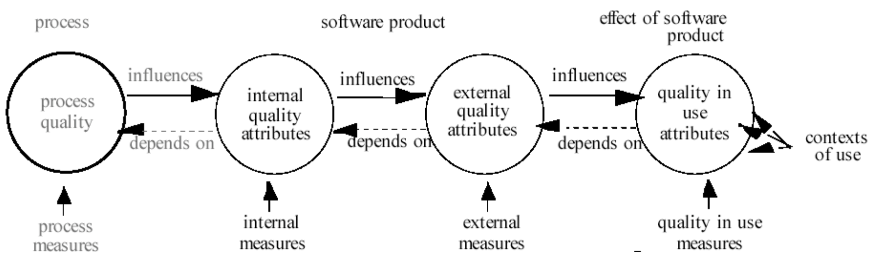

Fig. 3. Relationships between quality in use and internal and external qualities. Extracted from [18].

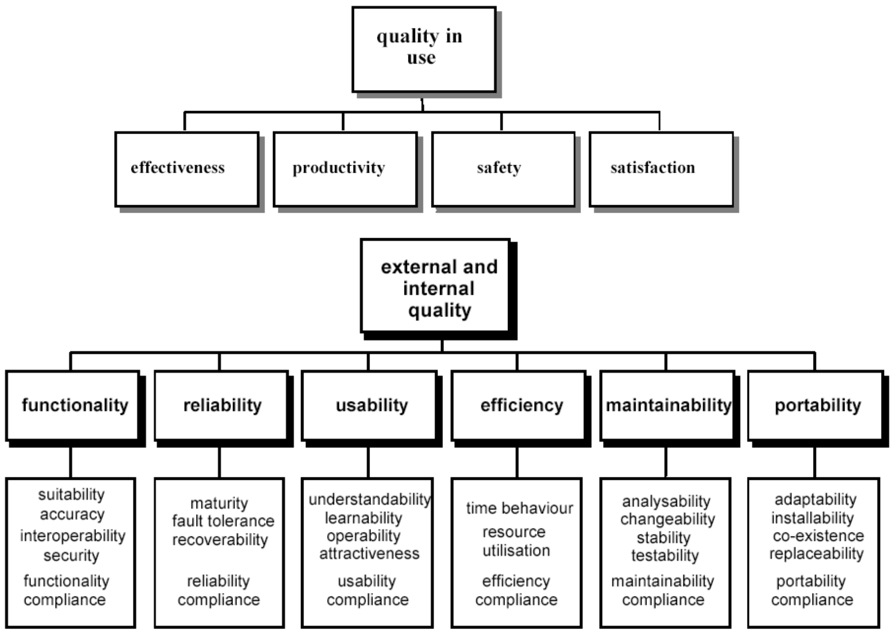

Fig. 4. Quality models for quality in use and internal and external qualities. These ISO models provide a sound basis for specifying and evaluating the extent to which an interactive system is supposed to be plastic. Extracted from [18].

The next section presents how to *plastify* an interactive system from a user centered perspective.

2.2 Plasticity from a user centered perspective

Whilst plasticity has always been addressed from a centralized perspective [5] (the UI was locally tuned as in FlexClock [15]), it is now obvious that ubiquitous computing favors the distribution of the interactive system among a set of platforms. As a result, two means are now available for adapting:

- Recasting the interactive system: this consists in reshuffling the UI, the functional core or the connector between both of these parts locally without modifying its distribution across the different platforms. Figure 1 provides an example of recasting;
- Redistributing the interactive system: it consists in migrating all (total migration) or part of (partial migration) the interactive system across the different platforms. Partial migration has been introduced by Rekimoto's painter metaphor [27] [4] and is now a major issue in HCI.

In ubiquitous computing, the notion of platform is no longer limited to an *elementary platform*, i.e., a set of physical and software resources that function together to form a working computational unit [7]. The notion of platform must definitely be seen as a *cluster*, i.e., a composition of elementary platforms that appear and disappear dynamically. For example, when Alice arrives in Bob's vicinity, her laptop extends the existing cluster composed of Bob's laptop, the PDA and the mobile phone. Bob's current interactive system can partially or fully migrate to Alice's laptop. Typically, to obtain a larger screen, it could be a good option to "bump" [16] the two laptops and split the interactive system between both of them (partial migration) (the *bumping* is illustrated in Figure 5 with two desktops). But when Bob's laptop battery is getting low, a full migration to Alice's laptop seems to be the best option as the screens of the PDA and mobile phone are too small to support a comfortable interaction.

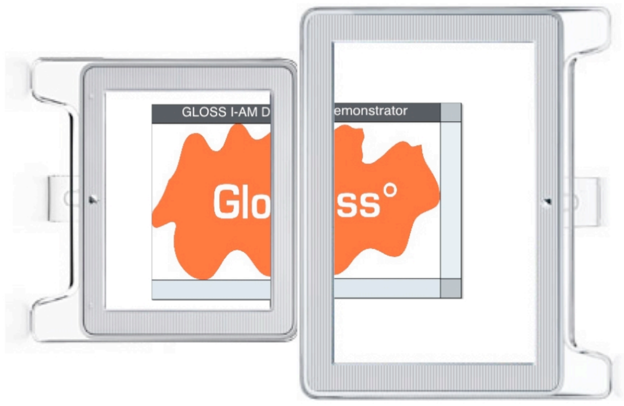

Fig. 5. A partial migration enabled by a top-to-top composition of the screens. Extracted from [9].

The granularity for distribution may vary from the application level to the pixel level [7]:

- 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 migrate between platforms are workspaces. A workspace is an interaction space. It 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 windows and panels. The painter metaphor presented in Rekimoto's pick and drop [27] [4] is an example of a distribution at the workspace level: the palettes of tools are presented on a PDA whereas the drawing area is mapped onto an electronic white board. Going one-step further, the tools palette (possibly the drawing area) can migrate at run time between the PDA and the electronic board;
- At the *domain concept level*, the user interface components that can be distributed between platforms are physical interactors. Here, physical interactors allow users to manipulate domain concepts. In Rekimoto's augmented surfaces, domain concepts, such as tables and chairs, can be distributed between laptops and horizontal and vertical surfaces. As for Built-IT [26], the topology of the rendering surfaces matters: objects are represented as 3D graphic interactors on laptops, whereas 2D rendering is used for objects placed on a horizontal surface;
- At the *pixel level*, any user interface component can be partitioned across multiple platforms. For example, in I-LAND [30], a window may simultaneously lie over two contiguous white boards (it is the same case in Figure 5 with two desktops). When the cluster is heterogeneous, designers need to consider multiple sources of disruption. For example, how to represent a window whose content lies across a white board and a PDA? From a user's perspective, is this desirable?

Migration may happen on the fly at run time or between sessions:

- *On the fly migration* requires that the state of the functional core is saved as well as that of the user interface. The state of the user interface may be saved at multiple levels of granularity: with regard to the functional decomposition promoted by Arch [3], when saved at the Dialogue Component level, the user can pursue the job from the beginning of the current task; when saved at the Logical Presentation or at the Physical Presentation levels, the user is able to carry on the current task at the physical action level, that is, at the exact point within the current task. There is no discontinuity;
- *Migration between sessions* implies that the user has to quit, then restart the application from the saved state of the functional core. In this case, the interaction process is heavily interrupted.

Recasting and redistribution are two means for adaptation. They may be processed in a complementary way. A full migration between heterogeneous platforms will typically require a recasting for fitting to a smaller screen. Conversely, when the user enlarges a window, a partial migration may be a good option to get a larger interaction surface by using a nearby platform. The next section addresses plasticity from a system's perspective.

2.3 Plasticity from a system centered perspective

The CAMELEON reference framework for plasticity [7] provides a general tool for reasoning about adaptation. It covers both recasting and redistribution. It is intended to serve as a reference instrument to help designers and developers to structure the development process of plastic interactive systems covering both the design time and run time.

The design phase follows a model-based approach [25] (Fig. 6). A UI is produced for a set of *initial models* according to a *reification process*:

- The initial models are specified manually by the developer. They set the applicative domain of the interactive system (concepts, tasks), the predicted contexts of use (user, platform, environment), the expected quality of service (a set of requirements related to quality in use and external/internal qualities) and the adaptation to be applied within as well as outside the current context of use (evolution, transition). The domain models are taken from the literature. Emerging works initiated by [12] [28] deal with the definition and modeling of context of use. The *Quality Models* can be expressed with regard to the ISO models presented in section 2.1. The *Evolution Model* specifies the reaction to be performed when the context of use changes. The *Transition Model* denotes the particular *Transition User Interface* to be used during the adaptation process. A transition UI allows the user to evaluate the evolution of the adaptation process. In Pick and Drop [27], the virtual yellow lines projected on the tables are examples of transition UIs. All of these initial models may be referenced along the development process from the domain specification level to the running interactive system;
- The design process is a three-step process that successively reifies the initial models into the final running UI. It starts at the concepts and tasks level to produce the *Abstract User Interface* (Abstract UI). An abstract UI is a collection of related workspaces called *interaction spaces*. The relations between the interaction spaces are inferred from the task relations expressed in the task model. Similarly, connectedness between concepts and tasks is inferred from the concepts and tasks model. An abstract UI is reified into a *Concrete User Interface* (Concrete UI). A concrete UI turns an abstract UI into an interactor-dependent expression. Although a concrete UI makes explicit the final look and feel of the *Final User Interface* (Final UI), it is still a mockup that runs only within the development environment. The Final UI generated from a concrete UI is expressed in source code, such as Java and HTML. It can then be interpreted or compiled as a pre-computed user interface and plugged into a run-time infrastructure that supports dynamic adaptation to multiple targets.

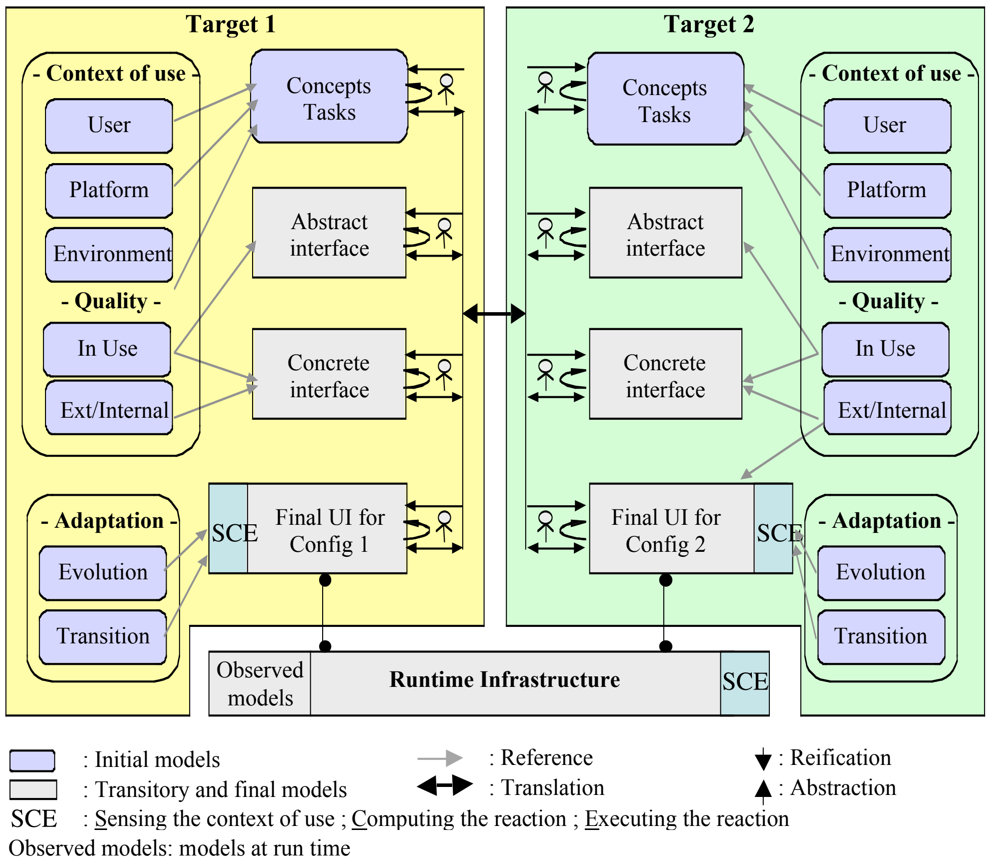

Fig. 6. The Reference Framework for supporting plastic user interfaces. The picture shows the process when applied to two distinct targets. This version is adapted from [7] where the quality models defined in 2.1 are now made explicit. Whilst reifications abstractions and translations are exhaustively made explicit, only examples of references are provided. In the example, the reference to the evolution and transition models is made at the latest stage (the final UIs).

At any level of reification:

– References can be made to the context of use. We identify four degrees of dependencies: whether a model makes hypothesis about the context of use; a modality; the availability of interactors; or the renderer used for the final UI. From a software engineering perspective, delaying the dependencies until the later stages of the reification process, results in a wider domain for multi-targeting. Ideally, dependencies to the context of use, to modalities and to interactors are associated with the concrete UI level (Fig. 7 a). In practice, the task model is very often context of use and modality dependent (Fig. 7b). As figure 7 shows, a set of four sliders (or stickers) can be used to locate the dependencies in the reification process. The movement of the stickers is limited by the closeness of their neighbour (e.g., in Figure 7b, the interactor sticker has a wide scope for movement between the concepts and tasks level and the final UI level, respectively corresponding to the position of the modality and renderer stickers);

- References can be made to the quality properties that have guided the design of the UI at this level of reification (cf. arrows denoted as "reference" in Figure 6);
- A series of abstractions and/or reifications can be performed to target another level of reification;
- A series of translations can be performed to target another context of use.

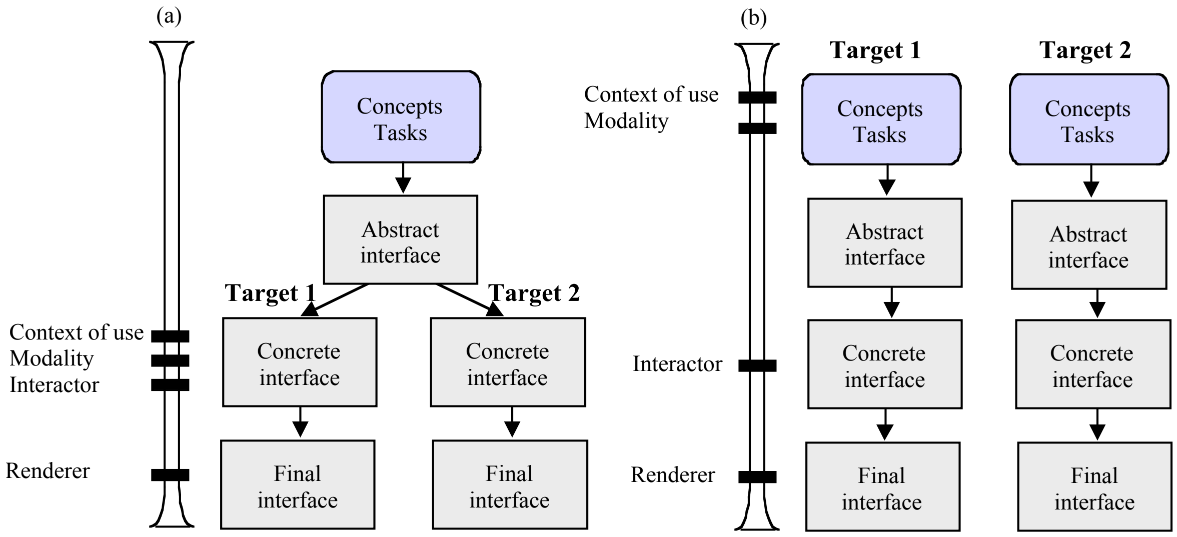

Fig. 7. Two instanciations of the design reference framework. The dependencies to the context of use, modalities, interactors and renderer are localized through stickers that constraint each other in their movement.

Reifications and translations may be performed automatically from specifications, or manually by human experts. Because the automatic generation of user interfaces has not found wide acceptance in the past [23], the reference framework makes possible manual reifications, abstractions and translations (Fig. 6).

As for any evolutive phenomenon, the adaptation at run time is structured as a three-step process: sensing the context of use (S), computing a reaction (C), and executing the reaction (E) [6]. Any of these steps may be undertaken by the final UIs and/or an underlying run time infrastructure (Fig. 6). In the case of distributed UIs, communication between components may be embedded in the components themselves and/or supplied by the runtime infrastructure. As discussed in [24], when the system includes all of the mechanisms and data to perform adaptation on its own (sensing the context of use, computing and executing the reaction), it is said to be *close-adaptive*, i.e., self-contained (autonomous). FlexClock is an example of close-adaptive UI. *Open-adaptiveness* implies that adaptation is performed by mechanisms and data that are totally or partially external to the system. FlexClock would have been open-adaptive if the mechanisms for sensing the context of use, computing the reaction or executing the reaction had been gathered in an external component providing general adaptation services not devoted to FlexClock.

Whether it is close-adaptive or open-adaptive, dynamic reconfiguration is best supported by a component-connector approach [24] [11] [14]. Components that are capable of reflection (i.e., components that can analyze their own behavior and adapt) support close-adaptiveness [21]. Components that are capable of introspection (i.e., components that can describe their behavior to other components) support open-adaptiveness.

The next section applies these advances to the design and run time of comets.

3 The notion of comet

This section relies on the hypothesis that adaptation makes sense at the granularity of a widget. The validity of this hypothesis has not been proven yet, but is grounded in practice: refining an abstract UI into a concrete UI is an experimental composition of widgets with regard to their implicit functional (versus non functional) equivalence or complementarity. Basically, no toolkit makes explicit the functional equivalence of widgets (e.g., the fact that the three versions of Figure 2 are functionally but not non functionally equivalent: they support the same task of selecting one option among a set of options, but differ in many ways, in particular, in their pixels cost). Based on these statement and hypothesis, this paper introduces the notion of comet. It is first defined then examined from both a design and run time perspective. It is finally compared to the state of the art.

3.1 Definition

A comet is an introspective interactor that publishes the quality in use it guarantees for a set of contexts of use. It is able to either self-adapt to the current context of use, or be adapted by a tier-component. It can be dynamically discarded (versus recruited) when it is unable (versus able) to cover the current context of use.

The next section presents a taxonomy and a model of comets from a design perspective.

3.2 The comet from the design perspective

Based on the definition of comets and the advances in plasticity (section 2.3), we identify three types of comets (Fig. 8):

– *Introspective comets* refer to the most basic kind of comets, i.e. interactors that publish their functional and non functional properties (Fig. 9). The functional properties can include adaptation abilities (e.g., sensing the context of use, computing and/or executing the reaction), or be limited to the applicative domain (e.g., selecting one option among a set of options). For instance, the "combo box" comet (Figure 2) does not have to include the adaptation mechanisms for switching from one *form* to another one. It just has to export what it is able to do (i.e., single selection, the task it supports) and at which cost (e.g., footprint, interaction trajectory) to be called a comet;

– *Polymorphic comets* are introspective comets that embed (and publish because of their introspection) multiple versions of at least one of their components. The polymorphism may rise at the functional core level (i.e., the comet embeds a set of algorithms for performing the user task; the algorithms may vary in terms of precision, CPU cost, etc.), at the connector level between the functional core and the UI components (e.g., file sharing versus sockets), or at the UI level (e.g., functional core adaptor, dialog controller, logical or physical presentations with regard to Arch [3]). A comet incorporating the three versions of Figure 2 for selecting one option among a set of options would illustrate the polymorphism at the physical level. Polymorphism provides potential alternatives in case of a change in the context of use. For instance, Figure 2c is more appropriate than Figure 2a for small windows. The mechanism for switching from one *form* to another one may be embedded in the comet itself and/or supplied by a tier-component (e.g. the runtime infrastructure – see section 2.3);

– *Self-adaptive* (or *close-adaptive) comets* are comets that are able to self-adapt to the context of use in a full autonomous way. They embed mechanisms for sensing the context of use, computing and executing the reaction. The reaction may be based on polymorphism in case of polymorphic comets.

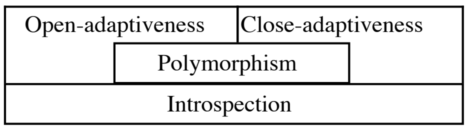

Fig. 8. A taxonomy of comets.

Introspection is the keystone capability of the comet. The properties that are published can be ranked against two criteria (Fig. 9): the type of the property (functional versus non functional) and the type of the service (domain versus adaptation). Examples of properties are provided in Figure 9. Recent research focuses on the notion of *continuity of interaction* [13]. The granularity of distribution and state recovery presented in section 2.2 belong to this area.

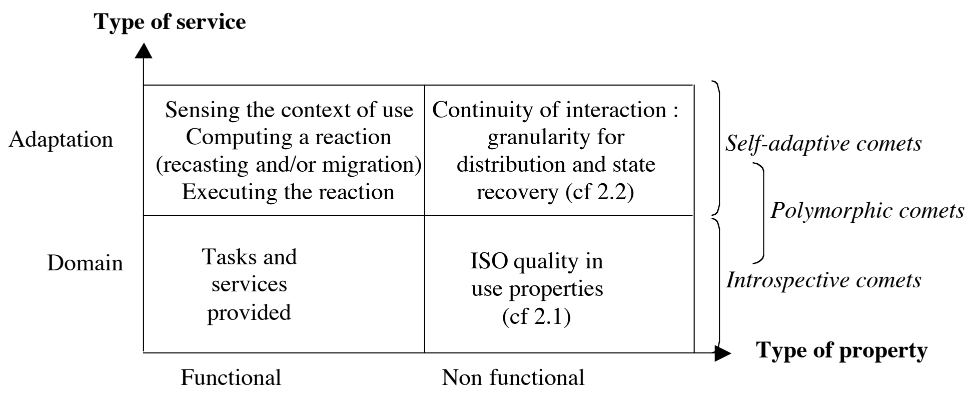

Fig. 9. A taxonomy of properties for structuring introspection.

Based on the nature of the domain task, a difference can be made between general comets that support basic tasks (i.e., those that are supported by classical widgets such as radio buttons, labels, input fields or sliders) and specific comets that support specific tasks. For instance, *PlasticClock* may be seen as a specific comet that simultaneously makes observable the time at two locations, Paris and New York (Figure 10). PlasticClock is polymorphic and self-adaptive. Its adaptation relies on two kinds of polymorphism, thus extending FlexClock:

- Polymorphism of abstraction: PlasticClock is able to compute the times in both an absolute and a relative way. The absolute version consists in getting the two times on web sites. Conversely, the relative way requests one time only and computes the second one according to the delay;
- Polymorphism of presentation: as shown in Figure 10, PlasticClock is able to switch from a large presentation format putting the two times side by side, to a more compact one gathering the two times on a same clock. Two hands (hours and minutes) are devoted to Paris. The third one points out the hours in New York (the minutes are the same). Allen's relations [2] provide an interesting framework for comparing these two presentations from a non functional perspective.

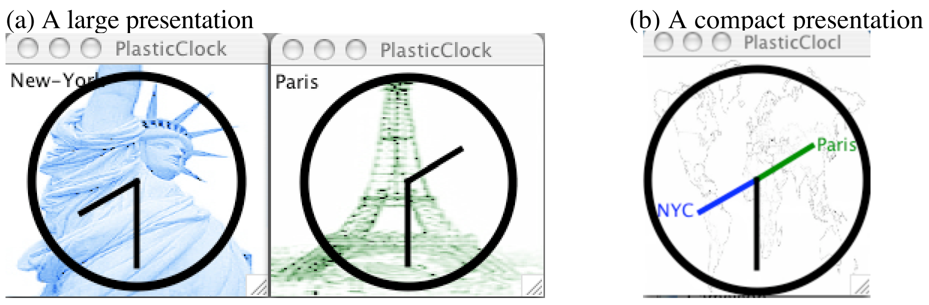

Fig. 10. PlasticClock.

The specific comets raise the question of the threshold between a comet and an interactive system. Should PlasticClock be considered as a comet or an interactive system? To our understanding, the response is grounded in software engineering: it depends on the expected level of reusability. As a result, comets can be designed as interactive systems. Figure 11 provides an UML class diagram obtained by applying both the reference framework and the taxonomy of comets for modeling a comet:

- A comet may be defined at four levels of abstraction. The most abstract one, called abstraction, is mandatory. This level may serve as starting point for producing abstract, concrete and final interaction objects (AIO, CIO, FIO) through a series of reifications and/or abstractions;
- At any level of reification, comets are introspective, i.e., aware of and capable of publishing their dependencies and quality of service (QoS). The dependencies are expressed in terms of context of use, modality, interactor and renderer. The quality of service denotes the quality in use the comet guarantees on a set of contexts of use. It is expressed according to a reference framework (e.g. ISO) by a set of properties. In a more general way, introspective components publish their API;
- Specific information and/or services are provided at each level of reification. At the abstraction level, they are related to the concepts and task the comet supports; at the AIO level, the structure of the comet in terms of interaction spaces; at the CIO level, the style of the comet (e.g., the style "button") and its typicality for the given purpose (e.g., whether it is or not typical to use radio buttons for specifying the platform – Figure 2a); at the final level, the effective context of use and the interaction state of the comet. Managing the interaction state (i.e., maintaining, saving and restoring the state of the comet) is necessary for performing adaptation in a continuous way;
- The comets may embed an evolution and a transition model for driving adaptation. The comet publishes its polymorphism and self-adaptiveness capabilities for a set of contexts of use. Going one step further, it directly publishes its plasticity property for a set of properties $\boldsymbol{P}$ and a set of contexts of use $\boldsymbol{C}$. It is plastic if any property of $\boldsymbol{P}$ is preserved for any context of $\boldsymbol{C}$.

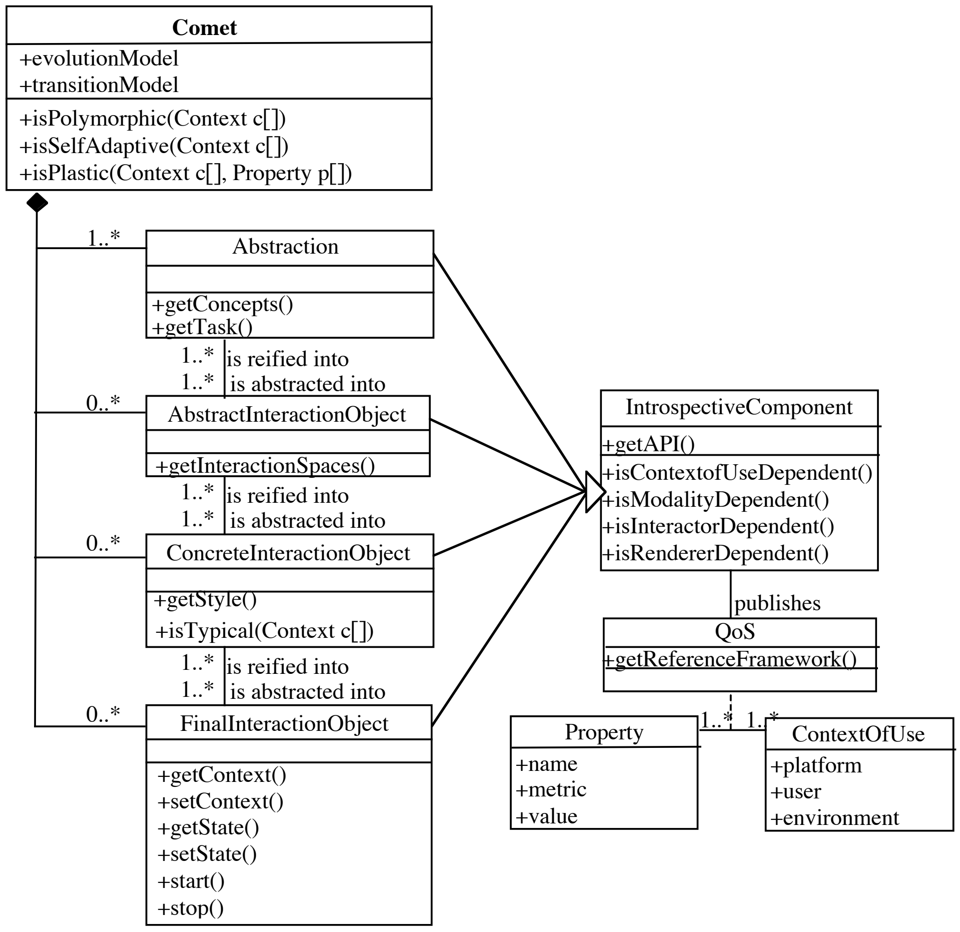

Fig. 11. A comet modeling taking benefit from both the reference framework and the taxonomy of comets.

The next section deals with the comets at run time.

3.3 The comet from the run time perspective

This section addresses the execution of comets. It elicits a set of strategies and policies for deploying plasticity. It proposes a software architecture model for supporting adaptation.

We identify four classes of strategies:

– Adaptation by *polymorphism*. This strategy preserves the comet but changes its *form*. The change may be performed at any level of reification according to the three following cardinalities, 1-1, 1-N, N-1 depending on the fact that the original form is replaced by another one (cardinality 1-1), by N forms (cardinality 1-N) or that N forms, including the original form, are aggregated into an unique one (cardinality N-1). For instance, in Figure 2, when the comet switches from a to b, it performs a 1-1 polymorphism: the radio buttons are replaced with a combo box. When it switches from b to c, it performs a 2-1 polymorphism (respectively switching from c to b is a 1-2 polymorphism);

- Adaptation by *substitution*. Conversely to the adaptation by polymorphism, this strategy does not preserve the comet. Rather, it is replaced by another one (cardinality 1-1) or N comets (cardinality 1-N) or is aggregated with neighbor comets (cardinality N-1);
- Adaptation by *recruiting* consists in adding comets to the interactive system. This strategy supports, for instance, a temporary need for redundancy [1];
- Adaptation by *discarding* is the opposite strategy to the recruiting strategy. Comets may be suppressed because the tasks they support no longer make sense.

At run time, the strategies may be chosen according to the evolution model of the comet. The selected strategy is performed according to a policy. The policies depend on the autonomy of the comets for processing adaptation. We identify three types of policies:

- An *external non-concerted policy* consists in fully subcontracting the adaptation. Everything is performed externally by a tier-component (e.g. another comet or the runtime infrastructure) without any contribution of the comet. This policy is suitable for comets which are unable to deal with adaptation. In practice, this is an easy way for guarantying the global ergonomic consistency of the interactive system. In this case, adaptation may be centralized in a dedicated agent (the tier-component);
- Conversely, the *internal non-concerted policy* consists in achieving adaptation in a fully autonomous way. Everything is performed inside the comet, without cooperating with the rest of the interactive system. The open issue is how to maintain the global ergonomic consistency of the interactive system;
- Intermediary policies, said *concerted policies*, depend on an agreement between the comet and tier-components. An *optimistic* version consists in applying the decision before it is validated by peers, whilst in a *pessimistic* version the comet waits for an authorization before applying its decision. The optimistic version is less time consuming but requires an undo procedure to cancel a finally rejected decision.

In practice, the policy decision will be chosen against criteria such as performance (c.f. the efficiency characteristic, time behavior sub-characteristic in section 2.1). The software architecture model Compact (COntext of use Mouldable PAC for plasticity) has been designed to take into account such an issue.

Compact is a specialization of the PAC (Presentation Abstraction Control) [8] model for plasticity. PAC is an agent-based software architecture model that identifies three recurrent facets in any component of an interactive system: an abstraction, a presentation and a control that assures the coherence and communication between the abstraction and the presentation facets. According to the "separation of concerns" principle promoted by software engineering, Compact splits up each facet of the PAC model in two slices, thus isolating a logical part from physical implementations in each facet (Fig. 12):

_ Abstraction: as with the functional core adaptor in Arch, the logical abstraction acts as an API for the physical abstraction. It provides a framework for implementing the mechanisms to switch between physical abstractions (i.e., the functional core(s) of the comet; they may be multiple in case of polymorphism at this level). It is in charge of maintaining the current state of the comet;

_ Presentation: in a symmetric way, as with the presentation component in Arch, the logical presentation acts as an API for the physical presentation part. It provides a framework for implementing the mechanisms to switch between presentations (they are multiple in case of polymorphism at this level);

_ Control: the logical part of the control assumes its typical role of coherence and communication between the logical abstraction and the logical presentation. The physical part, called "Plastic" (Fig. 12), is responsible for (a) receiving and/or sensing and/or transmitting the context of use whether the comet embeds or not any sensors (i.e., the Sensing step of the Reference Framework), (b) receiving and/or computing and/or transmitting the reaction to apply in case of changes of context of use (i.e., the Computation step of the Reference Framework), and (c) eventually performing the reaction (i.e., the Execution step of the Reference Framework). The reaction may consist of switching between physical abstractions and/or presentations. The computation is based on a set of pairs composed of compatible physical abstractions and presentations. At any point in time, one or many physical abstractions and/or presentations may be executed. Conversely, logical parts are only instanciated once per comet.

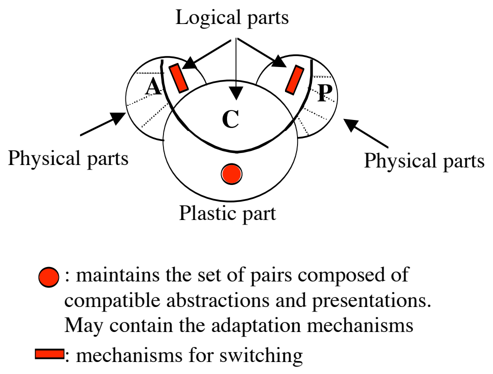
Logical parts
A
C
P
Physical parts
Physical parts
Plastic part
● : maintains the set of pairs composed of compatible abstractions and presentations. May contain the adaptation mechanisms
▬: mechanisms for switching

Fig. 12. The Compact software architecture model, a version of the PAC model (Presentation, Abstraction, Control) specifically mold for plasticity.

As in PAC, an interactive system is a collection of Compact agents. Specific canals of communication can be established between the plastic parts of the controls to propagate information in a more efficient way and/or to control ergonomic consistency in a more centralized way. Compact is currently under implementation as discussed in the conclusion. The next section analyses the notion of comet with regard to the state of the art.

3.4 Comets and the state of the art

Plasticity is a recent property that has mostly been addressed at the granularity of interactive systems. The widget level has rarely been considered. We note that most of these works focus on the software architecture modeling. Based on the identification of two levels of abstraction (AIOs and CIOs) [33], they propose conceptual and implementational frameworks for supporting adaptation [22] [20] [10]. But adaptation is limited to the presentation level [20] [10]. They do not cover adaptations ranging from the dialog controller to the functional core.

We now have to go further in the implementation. We keep in mind the issue of legacy systems [20] and the need for integrating multimodality as a means for adaptation [10].

4 Conclusion and perspectives

Based on a set of recent advances in plasticity, this paper introduces a new generation of widgets: the notion of comets. A comet is an interactor mold for adaptation: it can self-adapt to some context of use, or be adapted by a tier-component, or be dynamically discarded (versus recruited) when it is unable (versus able) to cover the current context of use. To do so, a comet publishes the quality in use it guarantees, the user tasks and domain concepts it is able to support, as well as the extent to which it supports adaptation. The reasoning relies on a scientific hypothesis which is as yet unvalidated: the fact that adaptation makes sense at the widget level. The idea is to promote task-driven toolkits where widgets that support the same tasks and concepts are aggregated into a unique polymorphic comet. Such a toolkit, called "Plasturgy studio" is currently under implementation. For the moment, it focuses on the basic graphical tasks: specification (free specification through text fields, specification by selection of one or many elements such as radio buttons, lists, spinners, sliders, check boxes, menus, combo boxes), activation (button, menu, list) and navigation (button, link, scroll). This first toolkit will provide feedback about both the hypothesis and the appropriate granularity for widgets. If successful, the toolkit will be extended to take into account multimodality as a means for adaptation.

Acknowledment

This work is being supported by the European commission funded CAMELEON R&D project IST-2000-30104. The authors would like particularly to thank Jean

Vanderdonckt and Quentin Limbourg, members of the project. Many thanks to Julie Dugdale for checking the paper.

References

1. Abowd, G.D., Coutaz, J., Nigay, L.: Structuring the Space of Interactive System Properties, Engineering for Human-Computer Interaction, Larson J. & Unger C. (eds), Elsevier Science Publishers B.V. (North-Holland), IFIP (1992) 113-126
2. Allen, J.: Maintaining Knowledge about Temporal Intervals, Journal Communication of the ACM 26(11), November (1983). 832-843
3. Arch: “A Metamodel for the Runtime Architecture of An Interactive System”, The UIMS Developers Workshop, SIGCHI Bulletin, 24(1), ACM Press (1992)
4. Ayatsuka, Y., Matsushita, N. Rekimoto, J.: Hyperpalette: a hybrid Computing Environment for Small Computing Devices. In: CHI2000 Extended Abstracts, ACM Publ. (2000) 53–53
5. Calvary, G., Coutaz, J., Thevenin, D.: A Unifying Reference Framework for the Development of Plastic User Interfaces, Proceedings of 8th IFIP International Conference on Engineering for Human-Computer Interaction EHCI’2001 (Toronto, 11-13 May 2001), R. Little and L. Nigay (eds.), Lecture Notes in Computer Science, Vol. 2254, Springer-Verlag, Berlin (2001) 173-192
6. Calvary, G., Coutaz, J., Thevenin, D.: Supporting Context Changes for Plastic User Interfaces : a Process and a Mechanism, in “People and Computers XV – Interaction without Frontiers”, Joint Proceedings of AFIHM-BCS Conference on Human-Computer Interaction IHM-HCI’2001 (Lille, 10-14 September 2001), A. Blandford, J. Vanderdonckt, and Ph. Gray (eds.), Vol. I, Springer-Verlag, London (2001) 349-363
7. Calvary, G., Coutaz, J., Thevenin, D., Bouillon, L., Florins, M., Limbourg, Q., Souchon, N., Vanderdonckt, J., Marucci, L., Paternò, F., Santoro, C.: The CAMELEON Reference Framework, Deliverable D1.1, September 3th (2002)
8. Coutaz, J.: PAC, an Object Oriented Model for Dialog Design, In Interact’87, (1987) 431-436
9. Coutaz, J. Lachenal, C., Barralon, N., Rey, G.: Initial Design of Interaction Techniques Using Multiple Interaction Surfaces, Deliverable D18 of the European GLOSS (Global Smart Spaces) project, 27/10/2003
10. Crease, M., Gray, P.D. & Brewster, S.A.: A Toolkit of Mechanism and Context Independent Widgets. In procs of the Design, Specification, and Verification of Interactive Systems workshop, DSVIS’00, (2000) 121-133
11. De Palma, N., Bellisard, L., Riveill, M. : Dynamic Reconfiguration of Agent-Based Applications . Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island (Portugal), (1999)
12. Dey, A.K., Abowd, G.D.: Towards a Better Understanding of Context and Context-Awareness, Proceedings of the CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness, The Hague, Netherlands, April 1-6, (2000)
13. Florins, M., Vanderdonckt, J.: Graceful degradation of User Interfaces as a Design Method for Multiplatform Systems, In IUI’94, 2004 International Conference on Intelligent User Interfaces, Funchal, Madeira, Portugal, January 13-16, (2004) 140-147

14. 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)
15. Grolaux, D., Van Roy, P., Vanderdonckt, J.: QTk: An Integrated Model-Based Approach to Designing Executable User Interfaces, in PreProc. of 8th Int. Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'2001 (Glasgow, June 13-15, 2001), Ch. Johnson (ed.), GIST Tech. Report G-2001-1, Dept. of Comp. Sci., Univ. of Glasgow, Scotland, (2001) 77-91. Accessible at http:// www.dcs.gla.ac.uk/~johnson/papers/dsvis_2001/grolaux
16. Hinckleyss, K.: Distributed and Local Sensing Techniques for Face-to-Face Collaboration, In ICMI'03, Fifth International Conference on Multimodal Interfaces, Vancouver, British Columbia, Canada, November 5-7, (2003) 81-84
17. IFIP BOOK: Design Principles for Interactive Software, Gram C. and Cockton G. (eds), Chapman & Hall, (1996)
18. ISO/IEC CD 25000.2 Software and Systems Engineering – Software product quality requirements and evaluation (SquaRE) – Guide to SquaRE, 2003-01-13 (2003)
19. ISO/IEC 25021 Software and System Engineering – Software Product Quality Requirements and Evaluation (SquaRE) – Measurement, 2003-02-03
20. Jabarin, B., Graham, T.C.N.: Architectures for Widget-Level Plasticity, in Proceedings of DSV-IS (2003) 124-138
21. Marangozova, V., Boyer, F.: Using reflective features to support mobile users. Workshop on Reflection and meta-level architectures, Nice, Juin, (2002)
22. Markopulos, P.: A compositional model for the formal specification of user interface software. Submitted for the degree of Doctor of Philosophy, March (1997)
23. Myers, B., Hudson, S., Pausch, R.: Past, Present, Future of User Interface Tools. Transactions on Computer-Human Interaction, ACM, 7(1), March (2000), 3–28
24. Oreizy, P., Tay lor, R., et al.: An Architecture-Based Approach to Self-Adaptive Software. In IEEE Intelligent Systems, May-June, (1999) 54-62
25. Pinheiro da Silva, P.: User Interface Declarative Models and Development Environments: A Survey, in Proc. of 7th Int. Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'2000 (Limerick, June 5-6, 2000), F. Paternò & Ph. Palanque (éds.), Lecture Notes in Comp. Sci., Vol. 1946, Springer-Verlag, Berlin, (2000) 207-226
26. Rauterberg, M. et al.: BUILT-IT: A Planning Tool for Consruction and Design. In Proc. Of the ACM Conf. In Human Factors in Computing Systems (CHI98) Conference Companion, (1998) 177-178
27. Rekimoto, J.: Pick and Drop: A Direct Manipulation Technique for Multiple Computer Environments. In Proc. of UIST97, ACM Press, (1997) 31-39
28. Salber, D., Abowd, Gregory D.: The Design and Use of a Generic Context Server, In the Proceedings of the Perceptual User Interfaces Workshop (PUI '98), San Francisco, CA, November 5-6, (1998) 63-66
29. Seffah, A., Kececi, N., Donyaee, M.: QUIM: A Framework for Quantifying Usability Metrics in Software Quality Models, APAQS Second Asia-Pacific Conference on Quality Software, December, Hong-Kong (2001) 10-11
30. Streitz, N. et al.: I-LAND: An interactive landscape for creativity and innovation. In Proc. of the ACM Conf. On Human Factors in Computing Systems (CHI99), Pittsburgh, May 15-20, (1999) 120-127

31. 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) 110–117
32. Van Welie, M., van der Veer, G.C., Eliëns, A.: Usability Properties in Dialog Models: In: 6th International Eurographics Workshop on Design Specification and Verification of Interactive Systems DSV-IS99, Braga, Portugal, 2-4 June (1999) 238-253
33. Vanderdonckt, J., Bodart, F.: Encapsulating knowledge for intelligent automatic interaction objects selection, In Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E., White, T. (Eds), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93, Amsterdam, ACM Press, New-York, 24-29 April, (1993) 424-429