Towards a new Generation of MBUI Engineering Methods: Supporting Polymorphic Instantiation in Synchronous Collaborative and Ubiquitous Environments

George Vellis[1], Dimitrios Kotsalis[1], Demosthenes Akoumianakis[1], Jean Vanderdonckt[2]

[1]Department of Applied Information Technology & Multimedia,
Technological Education Institution of Crete
Stavromenos 71004, Heraklion, Crete, Hellas
{g.vellis, kotsalis, da}@epp.teicrete.gr
[2]Louvain School of Management, Université catholique de Louvain
Place des Doyens, 1 – B-1348, Louvain-la-Neueve, Belgium
jean.vanderdonckt@uclouvain.be

ABSTRACT

This paper extends model based UI engineering so as to address polymorphic UI development in the light of distributed synchronous collaborative ubiquitous environments. Our current effort concentrates on extensions to a popular model-based user interface description language, namely: "UsiXML" and proposes a suitable development methodology and a dedicated runtime infrastructure.

Author Keywords

Polymorphic instantiation, Multi-user interfaces, Social awareness, UIDL, UsiXML, Model-based UI Engineering.

General Terms

Design, Experimentation, Human Factors, Verification.

ACM Classification Keywords

D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – Rapid Prototyping; reusable software. H5.2 [**Information interfaces and presentation**]: User Interfaces – *Prototyping*.

THE PROBLEM

The remarkable progress which we have witnessed over the last years, in fields relevant to information technology, has resulted in a wide range of novel computational appliances, having deeply penetrated a broad range of our daily practices. Increasingly, users come across of a sharply growing number of interactive applications enabling them to engage in various sorts of collaborative and social endeavors in a wide variety of contexts. In such cases one important aspect to consider relates to the UI design to support group work appropriately. It turns out that such development is quite challenging since prominent engineering methods and supporting tools fail to account for the requirements imposed by diverging user- and usage-context parameters (i.e.: platform, environment, user-stereotype). 'Polymorphic' instantiation, conceived of as a capability to manage concurrently alternative instances of the same interaction element at runtime, constitutes key provision in the direction of coping with this diversity (through extensive and intuitive UI adaptations). Specifically, it separates the role of the object (i.e., why an object is used) from its physical / presentation details in a designated context. It also hides technical features of an object's implementation from its core pattern that is independent of physical realization, thus common to potentially more than one implementation. In this account, polymorphic instantiation functions as 'idiom' in user interface engineering. Nevertheless, prevalent UI toolkits (java/swing, MAUI, QT, etc.) and languages do not make sufficient inroads towards supporting such polymorphic UI arrangements. Instead, they remain committed to supporting provisions for cross-platform generalizations (i.e.: addressing portability) rather than 'true' abstractions. Supporting a stronger notion of abstraction (with corresponding toolkit-level provisions) would enable, among other things, richer specifications of interactive behaviors, enumeration of alternative widget incarnation (i.e.: via encapsulation) and finer-grained and dynamic assembly of polymorphic UIs.

A particular scenario where polymorphic instantiation reveals its power as an engineering pattern is in the case of synchronous collaborative sessions supporting real-time distributed team work. Arguably, the state of the art offers a a rich insight into the desired groupware functionality and the features to facilitate such functionality (i.e., session management, object sharing, floor control, etc.). However, maintaining synchronized versions of potentially inconsistent multi-user UI instances, due to context-related parameters i.e., platforms or roles, continuous to pose substantial challenges. Toolkit-level sharing as exploited by the state of the art in groupware toolkits (i.e.: MAUI [1]) fails to account for the diversity of target devices involved in emerging ubiquitous environments, since it requires that the same groupware toolkit is available in every target-device. Nevertheless, this turns out to be an impractical assumption since dedicated runtime environments required for operating such toolkits might even not be supported (i.e.: JRE). Furthermore, even if alternative groupware

toolkits, accommodating constraints imposed by target-platforms, were used for assembling each target multiuser interface and that interoperability could be granted in some way, another major obstacle to overcome would be that of supporting awareness across engaged participants. Since different groupware toolkits make use of alternative mechanisms to facilitate awareness (i.e.: telepointers, radar-views, etc.), it would be rather impossible to achieve successful collaboration among users since awareness support would be foredoomed to failure. Finally, it is worth pointing out that groupware toolkits inherit the unimorphic style of programming imposed by their single-user counterparts; therefore they dismiss support for advanced and intuitive adaptation (i.e.: polymorphic instantiation in synchronous sessions). In this paper we focus on addressing these challenges to support complex interaction in synchronous collaborative and ubiquitous scenarios. By this account, our aim is to layout the foundations of a UI engineering methods that combine MBUI methods and toolkit programming to support polymorphic UI instantiation patterns.

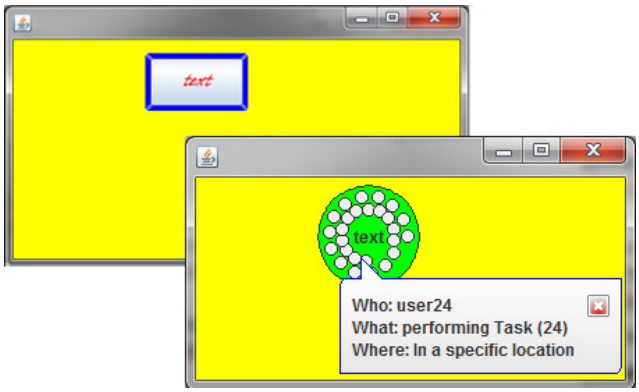

Figure 1. Alternative instantiations of 'abstractButton'.

To this effect, we propose to briefly elaborate on a very simplistic but demanding scenario. Figure 1 depicts two versions of a simple UI comprising one container object and one button. Each version of the UI is intended to serve a different user role. The left-hand side depicts the instance for participants (role A) where the button is manifested using a conventional rectangular pattern with two states (as commonly encountered in various toolkits). The right-hand side represents the moderator's (role B) view of exactly the same button rendered in a synchronous collaborative and distributed setting (i.e.: multiuser). This time the instance of of button is augmented (at both physical and syntactic levels) so as to possess visual affordances that convey awareness.In the specific example this is supported by using a dedicated nested radial layout indicating how many participants (i.e., users with role A) are currently engaged in the session or have access to this replica. The button's label remains the same in both cases just to indicate that we are working with the same abstraction that is capable of polymorphic instantiation.

In light of the above, the present research is motivated by some key questions;

- Can the above UIs be designed so as to be generated from the same specification?
- Can this be supported in a manner that allows designers to designate complex interactive behaviors as means to facilitate affordances such as awareness?
- What kind of widget attributes need to be manipulated (i.e., layout, topology, access policies) at client and server sides?

Responding to these questions is expected to enlighten our views on prominent architectural limitations of current UI engineering methods. Most importantly, however, it is likely to lead to new insights into affordance-based design of UIs and how they may be associated with MBUI engineering.

RELATED WORK

One approach to address the problem is to craft UIs by combining popular UI toolkit-based programming and model-based UI engineering techniques. Toolkit programming is grounded on the philosophy of building user interfaces as hierarchies of reusable widgets by registering event event handlers. This allows for complex interactive behaviors and customized dialogues [1].

On the other hand, MBUI engineering makes use of abstract notations and mark-up languages to facilitate mapping of abstract components to platform-specific toolkit libraries by delegating the display to a platform-specific renderer [2]. At first site, this seems to provide a better frame of reference to addressing the problem as it offers greater flexibility and provisions with regards to potential heterogeneities in contexts of use (i.e.: cell phones, web, etc.). Nevertheless, the key issue in the problem presented earlier is not portability and the degree to which it is supported for a set of pre-determined widgets.

Rather, it is a matter of utilizing widgets (either native or custom) so as to facilitate polymorphic instantiation with variations at all levels including physical, syntactic and semantics. This is an issue which is only partly and loosely addressed by model-based approaches which are still limited to crafting rather simplistic form-based UIs for conventional presentation vocabularies and contexts of use. This limitation is attributed to lack of provisions a) for enhancing expressive capacity of interactions by allowing augmentation and/or expansion of UI vocabularies and b) to allow alternative instantiations to be encapsulated in the context of supported unimorphic widgets.

Polymorphic instantiation

Polymorphic instantiation is a demanding notion for UIs which has not been adequately addressed in the recent literature. It was initially introduced in toolkit based systems such as the HOMER user interface management suite [3], and the Platform Integration Module [4]) in unified user in-

terface development. These studies explored the challenge for 4GLs to realize polymorphic UI objects. At the time, the concept did not implicate any kind of design considerations regarding the specification of polymorphic dialogues. Rather, it was conceived as a property of the language, not a desirable affordance to be designed.

Model-based UI approaches promised to alleviate this shortcoming by offering new models and abstractions to reconsider affordance-based UI design. In practice, this never turned out to be the case, as most of the available scholarship concentrates on methodologies, engineering techniques and tools to describe native interaction components and their transformation and mapping from one dialect to another. For instance, COMETS [5] provide support for multi-level UI adaptations at runtime driven by support for semantic networks (no WSL, no advanced widgets). It is worth noticing that none of these systems integrates novel widget specification languages (WSL) or advanced widgets.

Collaborative aspects

Another key issue in reconsidering affordance-based design of UIs is collaboration and in particular synchronous collaboration. Again, here progress has been slow in both toolkit-based systems and model-based UI engineering. Groupware toolkits are around for more than two decades now. Their focus has been on managing technical properties of collaboration such as session management, floor control, object sharing and replication. Awareness was conceived of at multiple levels (task, activity, social awareness) but it was never fully supported.

Notable exceptions include research prototypes such as the MAUI toolkit which supports group awareness, is java-based and facilitates multi-user UI design via a dedicated IDE provided (JBuilder). Other systems such as BEACH [6] concentrate on aspects of colocation in ubiquitous collaborative systems but offer no support for awareness,

Model-based UI engineering, once again, has brought about modest but notable improvements. TOUCHE [7] provides multiuser functionality using adhoc mappings to a custom underlying groupware toolkit. Support for awareness is fixed during design phase. In the same vein, indicative examples include approaches such as CIAM (Collaborative Interactive Applications Methodology) [8] or AMENITIES (A Methodology for aNalysis and desIgn of cooperaTIve systEmS) [9].

These efforts primarily concentrate on devising notations and tools to model cooperative behavior and workflows. In effect, their primary contribution is that they make explicit different elements of collaboration (i.e., roles, responsibilities and tasks) using dedicated notations (i.e.: CIAN [10]). As a result they provide no support for designing the UI. No support for session modeling.

APPROACH

First principles

The present work seeks to improve on the state of the art by addressing several of the challenges and the limitations introduced earlier. Specifically, it focuses on issues related to polymorphic instantiation and collaborative management of UIs in ubiquitous distributed contexts. The approach builds on the Model-based UI engineering paradigm aiming to inject new elements so as to facilitate a more accountable affordance-based UI perspective. To this end, our current effort is grounded on a very popular UIDL, namely UsiXML [11] and inherits a single design process for all supported contexts of use. In turn, this leads to significant reductions in (re-) engineering costs and programming complexity while resulting to more reliable and coherent UIs. UsiXML makes use of three distinct levels of abstraction (i.e.: CTT, AUI, CUI) for incrementally specifying a UI in order to promoting separation of concerns. Specifically, the CTT model captures a UI specification in a computation independent manner while adopting a user centered perspective. Additionally, the Abstract User Interface (AUI) enables the definition and derivation of both modality-independent and multimodal interactive object hierarchies by providing support for the CARE properties. Finally, the CUI-model focuses on a platform-independent (or better toolkit-independent) UI specification. At all times, transition between supported models is enabled via dedicated transformations. A salient feature of UsiMXL is support for plasticity [12], which is addressed via the 'context-model' enabling context - sensitive transformations to be performed so as to accommodate diverse requirements posed by different contexts of use.

Extensions

Despite the relative ease in designing for multiple environments, UsiXML lacks support for taking advantage of advanced interactive capabilities offered by target platforms since its support is limited only to a reduced set of rather simplistic natively supported form-based elements (i.e.: buttons, labels, etc.). Moreover, as expected, no widget specification language is provided since the widget range is a priori known and hardcoded within the transformation logic via direct calls to platform-specific presentation vocabularies. Besides, support for polymorphic assembly of UIs is completely dismissed, since no dedicated mechanisms or widget specification language exists so as to allow widgets to encapsulate alternative instantiations. In addition, no provisions are made either for modeling or runtime support of distributed multi-user interactions (i.e.: session modeling, multi-user artifacts, awareness support, etc.).

In order to address these shortcomings, our efforts concentrated on a series of modifications in language-level which resulted in either new models (i.e.: language expansions), or enhancements of already existing (i.e.: language augmentations).

	WSL	UsiXML: UI Model											Architecture – Runtime Environment		
		Existing Models					New Models								
		CTT	AUI	CUI++	Context	Mapping++	Squad	Widget Resource	Abstraction	Consistency	Behavior	Session	Collaboration Plugin	Platform Server	Web Services
Non-native Widgets Support	√			√				√			√			√	
Polymorphic Instantiation	√			√	√	√	√	√	√	√	√			√	
Abstraction	√			√		√		√	√	√	√			√	
Replication									√			√	√	√	√
Collaboration				√				√	√	√	√	√	√	√	
Social Awareness	√			√		√		√				√	√	√ social proxy	√

Table 1. Enhancements in UsiXML.

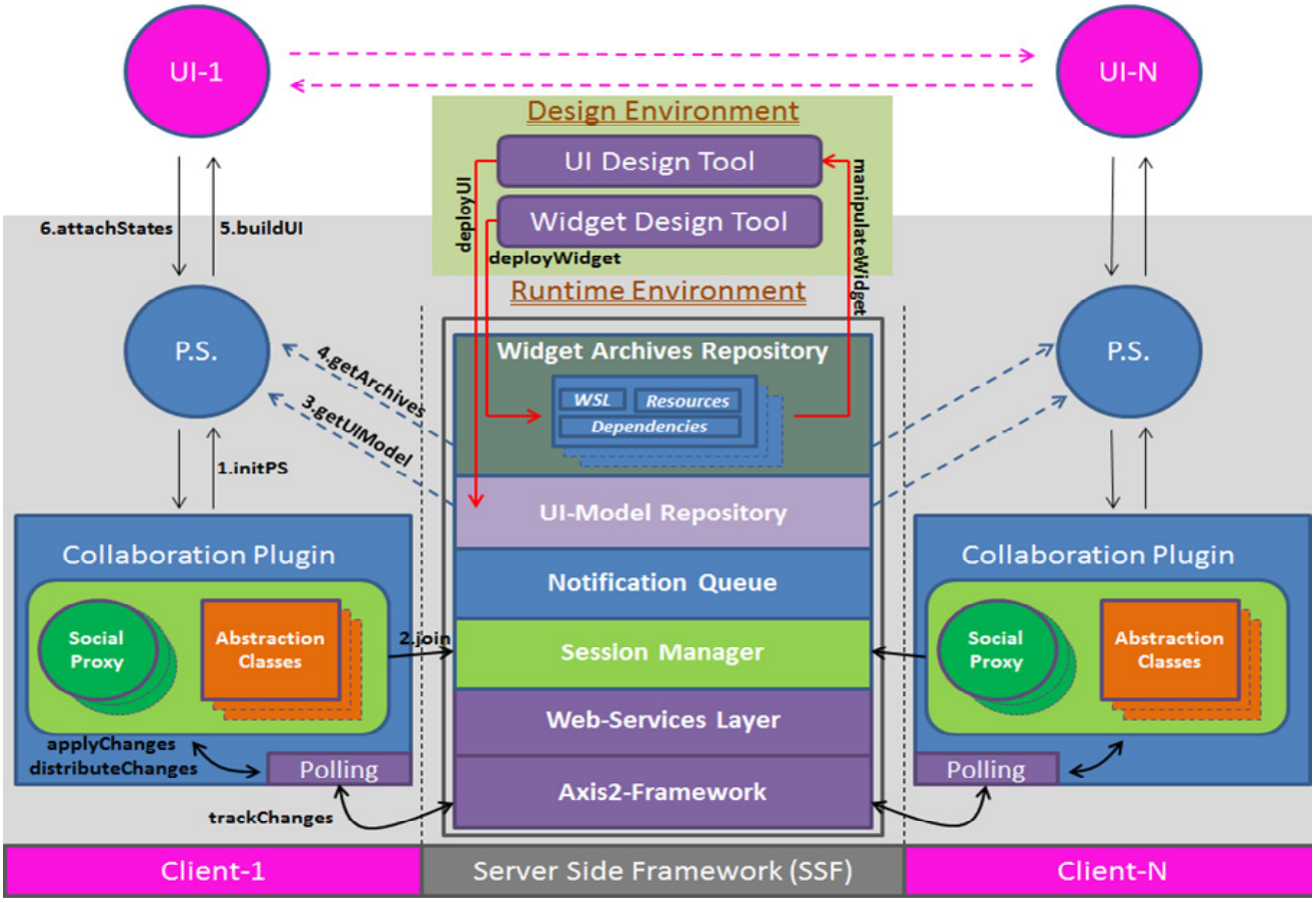

Figure 2. Architectural Abstraction.

Table 1 summarizes the injections introduced into the latest version of UsiXML models reflecting the state of our current research. As shown the focus is on key affordances (left column) and how they are implicated through revisions in or extensions of UsiXML models. In turn, this gives rise to a general purpose architectural abstraction which is depicted in Figure 2.

ARCHITECTURAL MODEL AND COMPONENTS

Our architecture makes use of several novel components that extend model-based UI engineering and make new ground in specification-based engineering

Widget specifications (for native and custom widgets)

In order to fully exploit the management of truly diverse interaction elements and vocabularies in target platforms (i.e.: natively and/or non-natively supported widgets) we crafted a formal (i.e.: compliant with a dedicated xml schema) specification mechanism referred to as Widget Specification Language (WSL). In order for a widget to be properly integrated and fully deployed by language's constructs, each must be separately introduced in a compliant to the provided WSL schema manner.

A WSL is responsible for capturing all required details for widget's proper utilization, parameterization and smooth

operation. Such details include widget's name, unique id, description, etc. Moreover, specific constraints can be defined, such as: platform availability, runtime environments required (i.e.: JRE), or even constraints on specific versions of external libraries required for widget's proper instantiation. In addition, widget's api (i.e.: accessor and mutator methods, constructors, etc.) is properly codified along with proper mappings to a widget resource model instance that it incrementally manifested exclusively at design time for capturing widget's detailed configuration (i.e., pre-instantiation state) in terms of simple or compound property-value pairs.

Polymorphism, abstraction and collaboration

Support for polymorphic instantiation is primarily granted via provisions in the WSL permitting multiple alternative instantiations to be defined (i.e.: encapsulation) for a specific widget type (i.e.: 'abstract widget'). To this end, also support for abstraction is granted since beyond polymorphic properties enumeration, also abstract properties must be defined, which constitute the common bond across all alternative instantiations. Moreover, it worth noting that widget resource model is also responsible (at design time) for capturing the range of adaptations each polymorphic widget may be permitted to run. Decision logic upon which alternative incarnations are utilized is specified in the light of the context and/or squad models. Each polymorphic instance is expected to define all supported behaviors in terms of properly codified finite state machines. At this level only supported states and permitted transitions among these can be defined and not at any case application-specific logic which is captured at design time by the 'Behavior' model. Abstraction (and the abstraction model) is not only used to facilitate polymorphic instantiation in collocated settings, but also to provide the mechanism to support distributed synchronous collaborative interactions in the context of the 'abstraction' model. Specifically, Abstraction model's prime aim is to establish an abstraction layer between multiuser widgets in the light of thorough or partial commonalities in regards to their models (in terms of Model View Controller architecture) while keeping their corresponding views synchronized.

Abstraction model comprises classes defined using class diagrams (at design time), in an attempt to facilitate model-level sharing which is completely relieved (in contrast to toolkit-level sharing) from physical-level properties, thus providing higher flexibility with regards to potential variations between alternative to be synchronized views. The reason for not selecting to synchronize directly widget corresponding models is mainly due to the need for design and implementation simplicity which is best served by centralizing concerns (i.e.: shared model).

Specifically, it would be much more complex to implement in a peer to peer manner intertwined relations between models each adhering to different widget than defining a single external model and providing appropriate hooks to synchronize every widget with that single model. Moreover it would be rather infeasible to inject collaboration-aware code and logic (i.e.: consistency and/or concurrency control) to collaboration unaware widget-models. Abstract classes alleviate this issue since collaboration-aware code is properly injected during their translation to platform specific code so as to automatically broadcast, receive and apply changes made by remote users. There are also dedicated provisions for properly managing consistency or concurrency related issues. Support for feed through, i.e.: apply changes received as an input of another distributed user, is implemented via dedicated mappings between shared properties of classes (i.e.: 'abstraction classes') defined in the context of abstraction model and direct calls to instance-specific API.

Replication and awareness

Moreover, it is worth noting that abstraction classes are distributed by replication in the context of particular synchronous collaborative sessions modeled via the 'session model'. In this context, support for social awareness is natively provided for widgets capable to visualize social scent using mixed dialogues. Specifically, social awareness is enabled inside widgets' specifications at polymorphic-instance level in order to indicate capacity for that specific instance to properly visualize social awareness. Nevertheless, the final decision for engaging social awareness for 'socially-aware' polymorphic instance is determined via a corresponding to that instance entry inside the WidgetResource model instance.

SUPPORTING THE DESIGN PHASE

Having described briefly architectural elements of our approach, an attempt is made to explain how these concepts are implicated during the design phase. We have developed a prototype system that takes advantage of the NetBeans platform, introducing on top of it a number of custom modules to support the development of either single user or distributed collaborative projects. The main differences between these two project types are in the way they employee to 'compile', distribute and execute the produced UI specifications, as well as in the number of available plugins engaged by default. For instance in case of collaborative application, a pre-requisite is the registration of a compatible server side environment dedicated to managing special purpose collaborative aspects (i.e.: synchronization, session management, etc.). Moreover, in distributed collaborative applications where UI models need to be accessible by several users over the network the pre-requisite is a centralized repository for depositing shared resources (i.e.: common models, widget archives, etc.). Furthermore, additional provisions are required for distributing a reference to all users may engage in a particular session (i.e.: 'distributed shortcuts'). Nevertheless, in all cases the development process remains quite common in terms of tool support, since most of custom plugins devised, remain the same (i.e.: editors for manipulating: CTT, CUI, Squad, etc.).

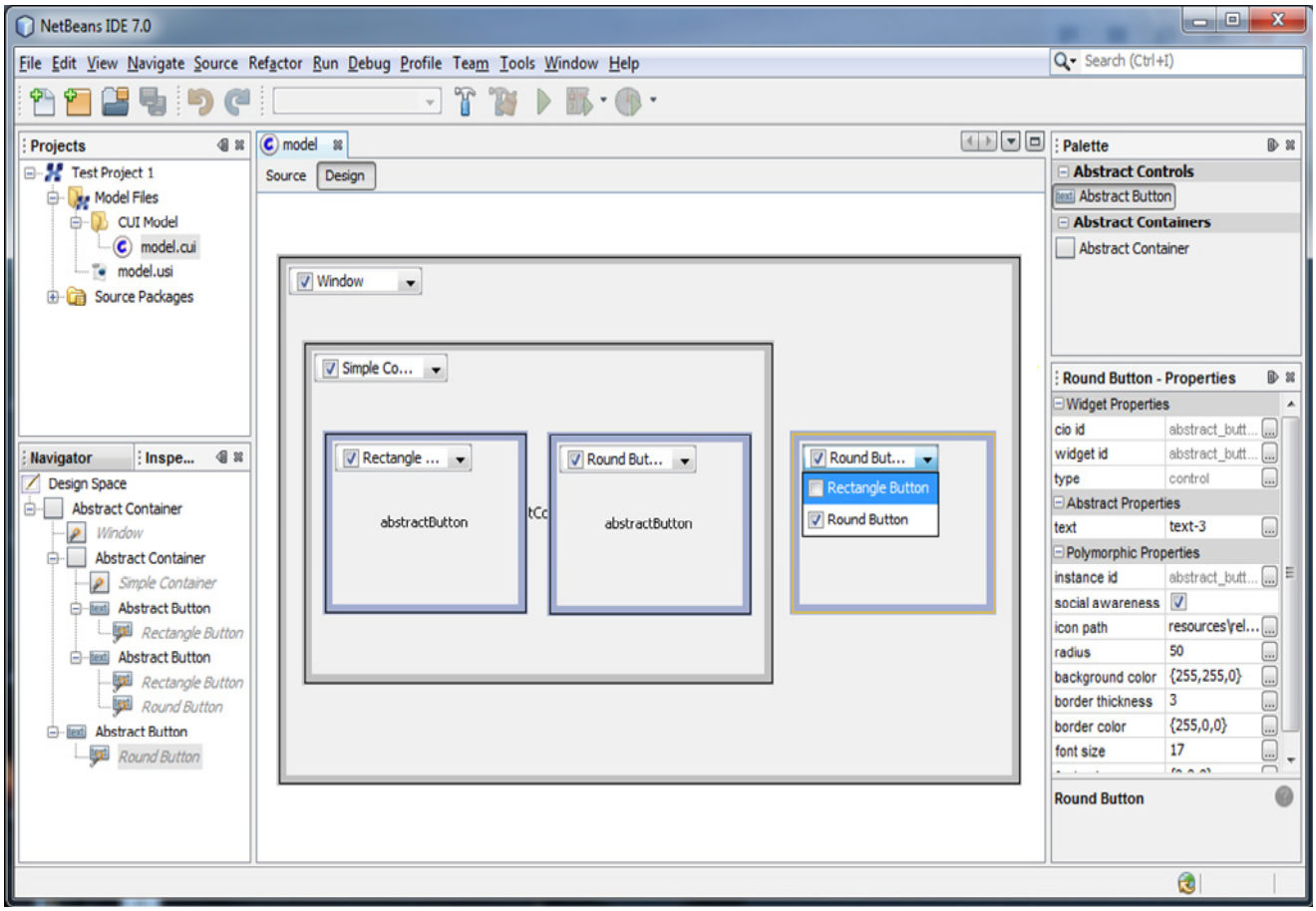

Figure 3. Design Environment for Polymorphic UIs.

A representative instance of our system depicting the design of the UI in Figure 1 is presented in Figure 2. At any time, a design project has a dual view – the graphical editor's visual depiction of abstract interaction components and the source (XML) view of the respective model.

Typically, designers utilize the palette to introduce components in a direct manipulation fashion and specify their properties. Design updates are immediately manifested as XML model changes. The properties of each widget fall in three categories, namely widget-specific properties, abstract properties and polymorphic properties. The way in which these properties are manipulated is dependent on specifications in the widget archive.

Widget archives

Widget archives provide the means for introducing advanced customized widgets. They may be non-native interaction components, developed by third-parties and shared to the design community. We have designed several custom widgets, some of them quite complex, to test the concept of a widget archive and the way in which it is articulated using our system. For purposes of illustration, in our example we discuss one such component which is referred to as 'Round button'. In order for a widget archive such as that encapsulating the 'Round button' incarnation to be deployed into our tool, it must be firstly installed by a series of steps depicted in Figure . Due to space limitations and the focus of the present work, we will not provide further details on this process.

Visual manipulation of abstract interaction elements

Each uploaded widget appears in the 'Abstract Control' section of the palette and can be utilized in a direct manipulation fashion. Thus, it can be attached to a container object, resized, relocated and specified. For widget archives with polymorphic instantiation, the visual depiction of the abstract component indicates the options available (see right-hand side round button in 3). Further specification of the widget can be attained by manipulating properties in the lower part of the palette leading to a concrete instance. Figure summarizes the mapping for the right-hand side button which exploits polymorphic instantiation capability. We refer to this model as the 'WidgetResource' model which conveys the range of possible polymorphic adaptations. Thus, Figure depicts how such customizations are defined (i.e.: triplet comprised of WSL instance id and instances of the WR & CUI model) for the specific widget in Figure 3.

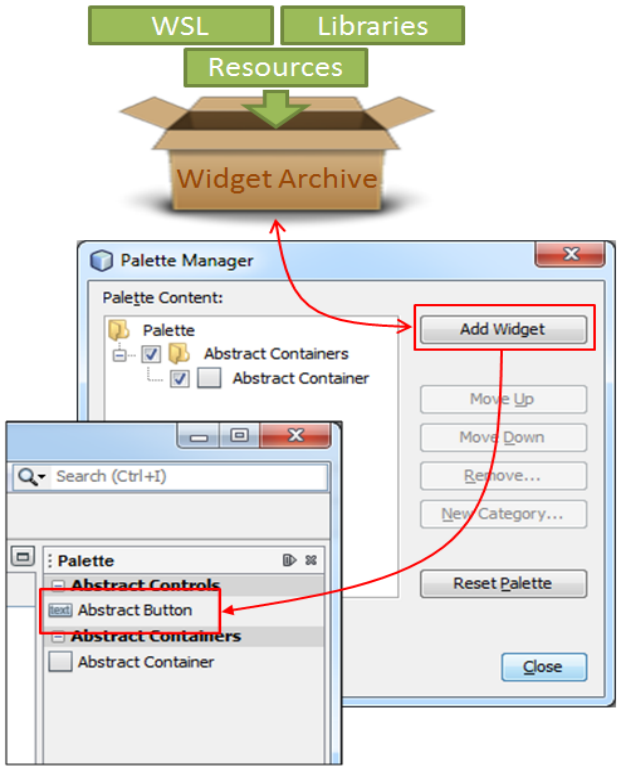

Figure 4. Widget Deployment Workflow.

Figure 5. Widget's Pre-instantiation Configuration Mapping.

RUNTIME ENVIRONMENT

In order to support the novel features introduced in the previous sections, advanced software components have been crafted both at the client as well as at the server side (see Figure 2).

Client-side components

At the client side of particular interest is a runtime infrastructure developed namely: 'Platform Server' (P.S.), denoting a multifunctional software component which guarantees language's smooth and consistent operation. Dedicated to a particular platform, the P.S. constitutes a virtual software layer between UsiXML models and the underlying system with its role being limited to: distributed class loading (in case of managing non-native interactive elements), event management (as part of facilitating collocated and/or distributed synchronization), as well as compilation (see: Domain model, Abstraction Model) and interpretation of UsiXML models at runtime. In addition, another very important function in the context of collaborative sessions, either synchronous or synchronous, relates to client-side support for session management. To this end, P.S. handles both grabbing and distribution of shared actions via triggering and managing inter-client (i.e.: inter-P.S.) message exchanges in the context of a particular session. To support this functionality P.S. interoperates with a custom dedicated server-side general purpose framework, built on top of the apache axis2 framework, to support session management. Moreover P.S. is responsible for handling replication process via managing (i.e.: generation of replicaIds, distributed registration, etc.) and maintaining a dedicated client-side replication list with replicaIds associated to corresponding object-references. In case of detected variations (via 'context-sniffer' daemon thread) regarding the context of use, P.S. is responsible for engaging a re-adaptation process driven-instructed from the server-side developed framework. Furthermore, another important function assigned to the P.S. relates to the process of overall handling non-native widgets (based on a Widget Specification Language devised on our side) part of which relates to 'custom events management' and 'widget data model' handling briefly discussed in previous section.

Server-side components

On the other hand at the server-side there is a generic-purpose server-side framework (SSF). The role of the SSF in the context of distributed settings focus on maintaining a repository of accessible at runtime UsiXML models, for initial or re-adaptation process, correlated to a particular session (either synchronous or asynchronous). The SSF also handles low-level session management (in both modes, i.e.: synchronous and asynchronous), build on top of apache axis2 framework, by performing several functions such as creation, registration, etc., while also maintains a list with all running sessions. Regarding non-natively provided widgets, in the context of a particular UI description specification, the SSF contributes by maintaining a shared repository with platform-specific widget libraries, in respect to widget's platform availability, facilitating P.S. via distributed class loading to handling non-natively supported interactive hierarchies. Finally, it keeps a per-session notification Queue, accessed by performing polling on the client side based on dynamically determined intervals, facilitating synchronization of distributed multi-user components.

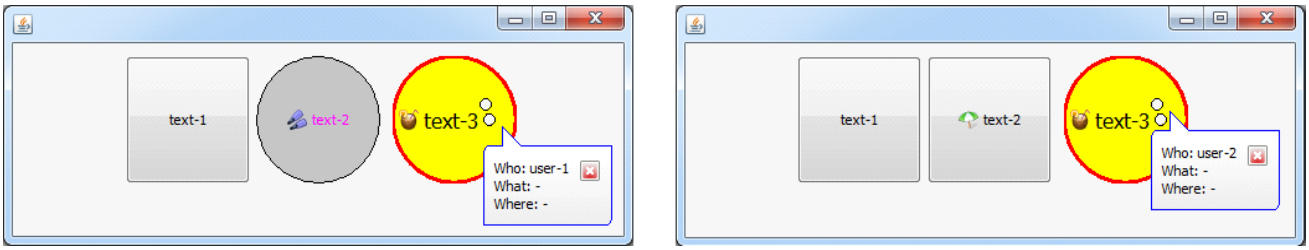

Figure 6. Runtime Instance of Final UIs.

AN EXAMPLE DESIGN CASE - UI GENERATION

In the present section we will elaborate on an indicative scenario in an attempt to provide a more detailed insight into the role of the supported models, the tools comprising the the overall environment and the development practices. More specifically, let's assume that support was required so so as to facilitate synchronous collaborative interactions in the light of heterogeneous contexts of use. In such cases a convenient point for engaging the design process would be that of CUI. Supposing that the UI design properly addressing our scenario is that displayed in Figure 3 then, in the course of alternative UI - execution flows, two potential instantiations (i.e.: assemblies) for two random users could be be these depicted in the Figure 6. Henceforth the view sided on the left in Figure 6 would be referred as "user's 1" while the other as "user's 2". Access to these UIs is granted after a user-request is made, triggered by clicking on a dedicated 'shortcut' available in both target platforms (i.e.: user1 and user2), for engaging to the appropriate collaborative session with which the corresponding UI specification has been associated with (at design time). Upon successful engagement to a collaborative session, P.S. automatically gains access and downloads all data (i.e.: UI models, dependencies, etc.) required in order to proceed to assembling the UI. Following this, UI generation begins by interpreting the retrieved CUI specification. In order for the P.S. to facilitate polymorphic instantiation, it requires decision logic (D.L.) which will allow it to decide which instance is to be delivered. For the purposes of our example, this decision is to be made on the grounds of socially-aware criteria (i.e.: community membership) which are properly codified inside the three supported models, i.e.: the squad model, the WRM and the CUI model. Specifically, we defined the middle 'abstractButton' to incarnate as a non-native circular button in case the user belongs to 'community-1' while on the other hand, it should be instantiated as a two-state rectangular button in case the user is a member of 'community-2'. Figure 7 depicts how these relations are codified in the supporting models so as to deliver the desired effect.

Runtime instantiation of RoundButtons (i.e.: non-native interactive element), is instructed by widget's specification language which facilitates P.S. to fully exploit instance-specific apis properly codified so as to standardize among other dynamic linking to external libraries required for its instantiation (i.e.: dependencies) as well as allow direct calls to be made so as to alter its state which could be useful in order to applying instance-specific configurations available from design phase (i.e.: icons, border color, etc., see: bottom right hand side in Figure 3). Each time a polymorphic interactive incarnation is instantiated in collaborative settings, P.S. seeks to determine (i.e.: WSL) whether or or not that instance type provides native support for social awareness (SA, i.e.: social awareness enabled).

Figure 7. Rule for Polymorphic Decision Logic.

Native support implies that toolkit-level provisions have been made in the course of widgets' development so as to: a) enable social scent be properly visualized as well as b) give access for its manipulation by appropriating a dedicated standardized API (i.e.:, 'addIndicator()', 'removeIndicator()', etc.) ready to be used by the runtime framework. In the present case, Round Button constitutes the only incarnation that has been properly manifested to provide support for social awareness. To this end, we had to apply toolkit-augmentation in many ways so as to create a circular bounded widget (note that non-rectangular widgets are not directly supported in java/swing) as well as a special purpose topology-policy to radially layout social indicators. In addition to physical-level enhancements, for SA to be properly managed, we had also to semantically augment

Round Button so that it can be appropriated by the runtime framework. Notably, such technical details are transparent to designers (see: Figure 3) whose role is limited to enabling or disabling this feature in case it is supported. Figure 8 attempts to clarify the allocation of provisions to handle SA in the current implementation.

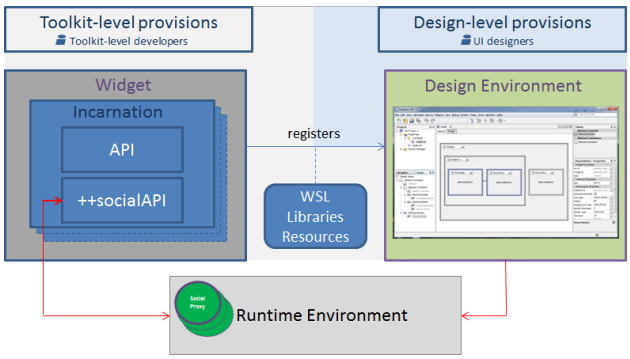

Figure 8. Implications in supporting Social Awareness in the course of the Design & Implementation process.

Moreover, in case of native support, a final check is also performed in order to determine whether or not that affordance has been enabled (WRM) for that polymorphic instance. In case both checks are successful, P.S. engages SA (via polymorphic method binding to that widget) and performs a direct call to the Collaboration plugin which then assigns a social proxy keeping social scents up to date.

```
<widget id="abstract_button" name="abstractButton" type="control">
    ...
    <instances>
        <instance id="inst_1" name="rectangleButton" isSocialAware="false"... >
            ...
        </instance>
        <instance id="inst_2" name="roundButton" isSocialAware="true"... >
            ...
        </instance>
    </instances>
</widget>
<widgetResourceModel>
    <widgetResource id="wdg_rsc_4" cioId="abstract_button_1" widgetId="abstract_button">
        ...
        <instance id="inst_1" ... >...</instance>
    </widgetResource>
    <widgetResource id="wdg_rsc_5" cioId="abstract_button_2" widgetId="abstract_button">
        ...
        <instance id="inst_1" ... >...</instance>
        <instance id="inst_2" socialAwarenessEnabled="false"... >...</instance>
    </widgetResource>
    <widgetResource id="wdg_rsc_6" cioId="abstract_button_3" widgetId="abstract_button">
        ...
        <instance id="inst_2" socialAwarenessEnabled="true"... >...</instance>
    </widgetResource>
    ...
</widgetResourceModel>
```

Figure 9. Enabling/Disabling Social Awareness Support.

Figure 9 shows how SA is disabled for the round button in the middle of the UI, and respectively enabled for the round round button, in the right hand side. Finally, widgets' behaviors (i.e.: Finite State Machines, Behavior model) and abstraction classes are compiled on the fly to machine code. code. Once behaviors get compiled they are automatically attached, by a dedicated P.S. module instructed by widget's specification language, to the proper instances of the UI. In the case of our example press state was synchronized (i.e.: shared property of an Abstraction Class).

CONCLUSIONS AND FUTURE WORK

Currently, we have a fully implemented version of all the components (models, plug-ins, architectural components, etc.) introduced earlier and working prototypes of several UIs that exhibit the required properties. In fact, all examples presented in the paper are realized using our research prototype. Ongoing work concentrates on several research lines. One is aiming to extend the widget archiving method to handle more complex and customized widgets intuitively. Another related line integrates the required revisions so as to further enhance the framework's capabilities to cope with more advanced affordances and quality attributes such as social translucence and UI plasticity in collaborative ubiquitous settings.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of iSTLab (www.istl.teicrete.gr) and the ITEA2 UsiXML project. Jean Vanderdonckt would like to acknowledge of the ITEA2-Call3-2008026 USIXML (User Interface extensible Markup Language) European project and its support by Région Wallonne DGO6.

REFERENCES

1. Hill, J. and Gutwin, C. The MAUI toolkit: Groupware widgets for group awareness. In *Computer Supported Cooperative Work*, 2004, vol. 13, pp. 539-571.
2. Lee, C., Helal, S., and Lee, W. Universal Interactions with Smart Spaces. *IEEE Pervasive Computing 5* (Jan.-Mar. 2006), pp. 16-21.
3. Savidis, A. and Stephanidis, C. Developing Dual User Interfaces for Integrating Blind and Sighted Users: the HOMER UIMS. In *Proc. of ACM Conf. on Human Factors in Computing Systems CHI'95* (Denver, 1995). ACM Press, New York (1995), pp. 106-113.
4. Savidis, A., Stephanidis, C., and Akoumianakis, D. Unifying toolkit programming layers: a multi-purpose toolkit integration module. In *Proc. of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems DSV-IS'1997* (Granada, 1997), pp. 177–192.
5. Demeure, A., Calvary, G., Coutaz, J., and Vanderdonckt, J. The Comets Inspector: Towards Run Time Plasticity Control based on a Semantic Network. In *Proc. of 5th Int. Workshop on Task Models and Diagrams for User Interface Design TAMODIA'2006* (Hasselt, 23-24 October 2006). K. Coninx, K. Luyten, K. Schneider (Eds.). Lecture Notes in Computer Science, vol. 4385, Springer-Verlag, Berlin (2007), pp. 324-338.
6. Tandler, P. The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments. *Journal of Systems and Software 29*, 3 (2004), pp. 267-296.

7. Penichet, V., Lozano, M., Gallud, J., and Tesoriero, R. User Interface Analysis for Groupware Applications in the TOUCHE Process Model. *International Journal Advances in Engineering Software* (2009).
8. Molina, A.I., Redondo, M.A., Ortega, M., and Hoppe, H.U. CIAM: A methodology for the development of groupware user interfaces. *Journal of Universal Computer Science* (2007).
9. Garrido, J.L., Gea, M., and Rodríguez, M.L. Requirements enginnering in cooperative systems, In *Requirements Enginnering for Sociotechnical Systems*. Idea Group, Inc., (2005), pp. 226–244.
10. Molina, A.I., Redondo, M.A., and Ortega, M. A conceptual and methodological framework for modeling interactive groupware applications. In *Proc. of 12th International Workshop on Groupware CRIWG'2006* (Valladolid, 2006). Lecture Notes in Computer Science. Springer-Verlag, Berlin (2006).
11. Limbourg, Q. and Vanderdonckt, J. UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence. In *Engineering Advanced Web Applications*, M. Matera, S. Comai, S. (Eds.). Rinton Press, Paramus (2004), pp. 325-338.
12. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers 15*, 3 (2003), pp. 289–308.
13. Souchon, N. and Vanderdonckt, J. A Review of XML-Compliant User Interface Description Languages. In *Proc. of 10th Int. Conf. on Design, Specification, and Verification of Interactive Systems DSV-IS'2003* (Madeira, 4-6 June 2003). J. Jorge, N.J. Nunes, J. Cunha (Eds.). Lecture Notes in Computer Science, vol. 2844, Springer-Verlag, Berlin (2003), pp. 377–391.