

Unified manipulation of interaction objects: integration, augmentation, expansion and abstraction

Anthony Savidis and Constantine Stephanidis

Institute of Computer Science (ICS),
Foundation for Research and Technology - Hellas (FORTH),
Science & Technology Park of Crete (STEP-C),
Heraklion, Crete, GR-71110, GREECE
E-mail: {as, cs}@ics.forth.gr
Fax: +30-81-391741, Tel: +30-81-391741

Abstract. Interface developers combine interaction elements in order to implement the User Interface of interactive software applications, using the development facilities available by a given development tool. The functional capabilities of the interface tool may significantly affect the quality of the resulting interactive software product, as well as the resources needed for further maintenance, upgrade, porting and expansion. Interaction objects play a key role in interface tools, irrespective of the nature of the interface construction technique (e.g. graphical construction, programming language, declarative specification, task notation). We have identified four fundamental categories of mechanisms for manipulating interaction objects in interface tools. Their merits in the context of interface development tools are identified, particularly in the context of developments for diverse user groups and openness for different interaction technologies. We will also show that these two functional requirements play a key role towards meeting the objectives of User Interfaces for All.

1. INTRODUCTION

Interface development tools play a key role in the User Interface development life cycle, while their functional capabilities may affect considerably the eventual quality of interactive software products. The vast majority of commercially available tools is targeted towards the implementation phase, while some of them also provide support for design-oriented development activities. In all cases, interface tools provide development facilities in which the notion of interaction elements, as the basic interaction building blocks, is directly mapped to explicit constructs. The most typical categories of such interaction elements are: (i) interaction objects / interaction techniques / object hierarchies; (ii) input events; (iii) graphic primitives; and (iv) call-backs / methods / notifications.

Usually, instances of such interaction element categories are provided in an implementation form by software libraries called *toolkits* (e.g. OSF / Motif, WINDOWS Object Library, InterViews, Xaw / Athena widget set). Interaction objects (e.g. windows, buttons, check-boxes) are the most important interaction element category, since the largest part of existing interface development toolkits is devoted to providing rich sets of interaction objects, accompanied with all the necessary functionality. Interaction objects are communicated at both the design as well as the implementation domain; in other words, interface designers are well aware of the various interaction objects classes and their respective “look & feel”, while programmers also share this type of knowledge. Naturally, designers have more detailed knowledge regarding their appropriateness for particular user tasks, while programmers have primarily implementation-oriented knowledge. In any case, a “button”, or a “window” has the same meaning for both designers and programmers, when it comes to the physical entity being represented. This is a distinctive property, which is, unfortunately, hardly met in other

types of design entities. For instance, user tasks do not directly map to available implementation constructs, hence, programmers would need an explicit mapping of a task model to an appropriate structure being closer to their implementation world. This mapping, unless automated by a tool, introduces an extra overhead; furthermore, the resources spent for such an activity are not meant to increase the eventual interface quality.

In recent development paradigms, such as the Web infrastructure for distributed interactive hypermedia documents, the importance of interaction objects has been particularly demonstrated. In HTML 3.0 (and included in all subsequent versions), form elements support a comprehensive collection of interaction objects; also, in the JAVA language, the AWT library (Abstract Window Toolkit) provides a rich set of interaction objects supporting *retargetability* (i.e. mapping to multiple graphical platforms). In conclusion, interaction objects, as re-usable interaction building blocks, are currently considered as a widely accepted and applied paradigm, for the design and implementation phases.

We have studied the functional needs for manipulating interaction objects, in the context of real-life application development ([Petrie et al., 1996], [Savidis et al., 1995a], [Savidis et al., 1995b], [Stephanidis et al., 1997b]), targeted towards diverse user groups, differing with respect to various parameters such as: physical / mental / sensory abilities, preferences, domain-oriented knowledge, role in organisational context, etc. In this context, various interaction technologies and interaction metaphors had to be employed, like: windowing graphical environments, auditory / tactile interaction, Rooms-based interaction metaphors, etc. We summarise below our key remarks from this study, which concern the manipulation of interaction objects in interface tools.

- Need another additional toolkit (i.e. *integration / importing* of a particular software library is needed);
- Dialogue for objects, as provided by the toolkit(-s) being used, is not adequate (i.e. *augmentation* with extra interaction techniques is imposed);
- Some necessary interaction objects are not provided from the toolkit(-s) being utilised, and / or new custom-made interaction objects are designed (i.e. *expansion* of the particular toolkit is required);
- Require manipulation of objects at a level “higher” than the typical implementation layer of toolkits, in order to make the dialogue design applicable to multiple user groups and target toolkits (i.e. need *abstraction*, applied on interaction objects).

From the above remarks, the need of the following four mechanisms for manipulating interaction objects emerges: (i) integration; (ii) augmentation; (iii) expansion; and (iv) abstraction. We consider those mechanisms as fundamental, for handling interaction objects within interface tools in a unified manner. We will incrementally analyse each of the mechanisms as follows: firstly, we will provide its basic definition, and then draw its importance in the context of a “User Interfaces for All” perspective. Secondly, we will provide representative examples of applying each mechanism in practice. Then, we will identify the minimal functional requirements for interface tools, so that the mechanisms can be considered as practically supported. Finally, we will discuss the full range of functional requirements in order to maximally support each particular mechanism.

2. TOOLKIT INTEGRATION

2.1 Definition and importance in the context of User Interfaces for All

We consider as toolkits all sorts of software libraries providing implemented interaction elements. A given interface development tool is considered to support toolkit integration, if it allows importing of *any* particular toolkit, so that *all* interaction elements of the imported toolkit(-s) are subsequently “exposed” (i.e. made available) within the original interaction building techniques of that particular given interface tool. For instance, if an interface builder providing graphical construction techniques supports toolkit integration, then, by integrating a particular toolkit (which supplies interaction objects with a specific “look and feel”) the original interactive graphical design facilities should directly enable manipulation of those interaction object classes. In this definition, we do not assume any particular interface construction method for interface tools. Hence, toolkit integration could be supported by: programming-based tools, interactive interface builders, state-based tools, event-based tools, demonstration-based tools, interface 4GLs (4th Generation Languages), etc.

The need for importing toolkits is evident in cases that the interaction elements originally supported by the particular interface development tool do not suffice. This is a possible scenario if interface development for diverse user groups needs to be addressed. For instance, in the context of Dual Interface development [Savidis et al., 1995a], where interfaces concurrently accessible by sighted and blind users need to be constructed, non-visual interaction techniques are required, together with typical visual graphical interaction elements. Existing windowing toolkits do not supply such interaction techniques, hence, integration of special-purpose non-visual interaction toolkits, such as COMONKIT [Savidis et al., 1995b], or HAWK [Savidis et al., 1997c] is necessitated. In the context of the User Interfaces for All objective, such scenarios are likely to emerge, since developing for diverse user groups is of primary importance.

2.2 Examples where toolkit integration has been supported

The toolkit integration capability implies that interface tools supply mechanisms which are made available to developers (i.e. to be utilised *after* the tool-product is launched). Currently, a very small number of interface tools supports this notion of platform connectivity. The first interface tool which provided comprehensive support for toolkit integration has been the SERPENT UIMS [Bass et al., 1990], where the toolkit layer has been called lexical technology layer; the architectural approach developed in the SERPENT UIMS revealed key issues in toolkit interfacing. The HOMER UIMS [Savidis et al., 1995a], which has been developed to facilitate the construction of Dual User Interfaces, also supported toolkit integration, by providing a powerful integration model, general enough to enable integration of non-visual interaction libraries, apart from traditional visual windowing toolkits.

We should make an explicit distinction among toolkit integration requirements and the multi-platform capability of certain toolkits. In the latter case, a single toolkit is met with multiple fixed implementations across different OS platforms, already made available when the toolkit product is released (i.e. multi-platform toolkits like YACL, Amulet, XVT, and JAVA AWT library). In the former case, a tool is made open, expecting that tool users will take advantage of the well documented functionality for connecting to arbitrary toolkits.

2.3 Minimalistic implementation requirements

The minimalistic requirements define if a particular tool supports some degree of openness, so that interaction elements from external (to the interface tool) toolkits can be utilised, still subject to some implementation restrictions. These requirements are the following:

- Ability to *link / mix* code at the software library level (i.e. combining object files, linking libraries together).
- Support for documented *hooks*, in order to mix at the source code level (i.e. calling conventions, type conversions, common errors and compile conflicts, linking barriers).

If the minimalistic requirements are satisfied, it will be made possible to combine together software modules which utilise interaction elements from different toolkits. The question is whether existing interface tools support the minimalistic requirements. The answer to this question is both yes, and no:

- *Yes*, if the toolkit to be integrated provides different categories of interaction elements with respect to the interface tool being used. For instance, assume that the interface tool being used is a programming-based toolkit of windowing interaction elements. Then, if audio-processing functionality for auditory interaction is to be imported and combined with this particular tool, possible conflicts are expected to be easily resolved.
- *No*, in the majority of cases, if the imported toolkit supplies similar categories of interaction elements with respect to the interface tool being used. For example, trying to combine various libraries of windowing interaction elements, from different vendors, leads to unrecoverable conflicts (e.g. WINDOWS object libraries from different vendors, Xt-based toolkits like OSF/Motif and Athena widget set). The primary reason for such conflicts is usually name collision at the binary library level, due to commonly implemented constructs within such toolkits like: “*object*”, “*window*”, “*button*”, “*initialize_toolkit*”, “*close_toolkit*”, “*event_handler*”, etc.

2.4 Maximalistic implementation requirements

The maximalistic requirements reveal the full range of functional capabilities regarding toolkit integration. If those requirements are met, toolkit integration becomes a practically simple task, while various parameters are also supported, concerning the way integration may be carried out by developers. The maximalistic requirements are:

- Well behaved and well documented compilation and linking cycles for interfaces utilising the integrated toolkit(-s).
- Single programming interface made possible for all integrated toolkits.
- Ability to change aspects of the programming interface of each imported toolkit.
- Resolve problems which appear when trying to combine multiple toolkits together, like: language conflicts, compile conflicts, linking conflicts and execution conflicts (more information can be found in [Savidis et al., 1997a]).
- Ability to import any type of interface toolkit, irrespective of the style of interaction supported (e.g. windowing toolkits, auditory / tactile toolkits, VR-based interaction toolkits).
- Ability to combine toolkits together for creating cross-toolkit object hierarchies; this is known as the *toolkit interoperability* requirement.

Currently, there is no single interface tool known which meets all the maximilastic requirements listed above. Regarding the notion of single programming interface, multi-platform toolkits already support this by means of a fixed programming layer. UIMS tools like SERPENT [Bass et al., 1990], HOMER [Savidis et al., 1995a], and I-GET [AT-HCI, 1997], all supporting toolkit integration, enable the establishment of developer-defined programming interfaces for toolkits; the same holds for PIM [Savidis et al., 1997a], a tool particularly developed for providing toolkit integration services.

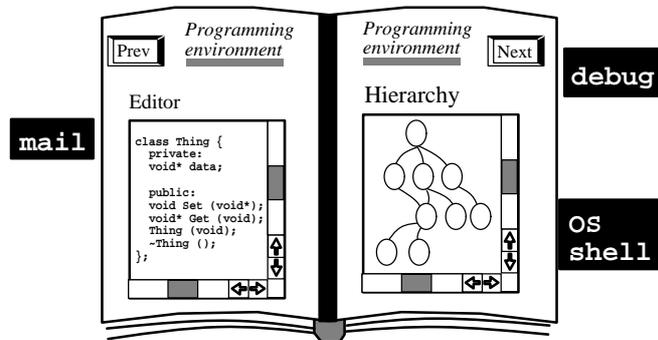
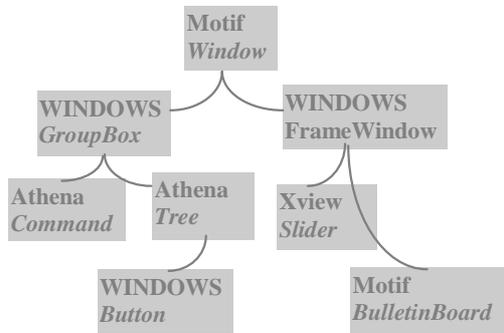


Figure 2.4-1: Example of a cross-toolkit hierarchy. **Figure 2.4-2:** A dialogue artefact for mixing toolkits.

When a single programming interface is supported for all platforms, one important question concerns the “look and feel” across those platforms. There are three alternative ways to cope with the “look and feel” in this case: (i) employ the native interaction controls as they are provided by each underlying platform; (ii) mimic the native controls of each platform; and (iii) provide custom-made interaction elements for all target platforms. Regarding toolkit interoperability (see Figure 2.4-1), only the Fresco User Interface System [X Consortium, 1994] is known to support cross-toolkit hierarchies, where mixing of InterViews-originated objects with Motif-ish widgets is facilitated. It should be noted that even though elements from different toolkits may be combined, possibly employing a different “look & feel”, consistency is not always damaged. In Figure 2.4-2, a scenario of mixing a windowing toolkit with a Books-toolkit is outlined, requiring a cross-toolkit object hierarchy at the implementation level. The advantages of mixing multiple toolkits are more evident in case that the combined toolkits offer container objects with different metaphoric representations (like the example of Figure 2.4-2). Various innovative dialogue artefacts may emerge, leading to the potential enhancement of the interface quality, an effect contributing towards our User Interfaces for All objective.

3. TOOLKIT AUGMENTATION

3.1 Definition and importance in the context of User Interfaces for All

Augmentation is defined as the design and implementation process through which additional interaction techniques are “injected” within the original interaction elements supplied by a particular toolkit, aiming towards enhancing accessibility and interaction quality for specific user categories. Newly introduced interaction techniques become an integral part of the original interaction elements, while old applications, if re-compiled and / or re-linked with the augmented toolkit version, inherit the extra dialogue features.

Currently, most interactive software products are built by utilising commercially available toolkits, such as the WINDOWS object library. As a result, those software products inherit the advantages, as well as the restrictions, of those interaction elements. For instance, voice control of interaction is not supported in WINDOWS object library, thus access in a situation where direct visual attention is not possible (e.g. while driving) cannot be supported. Another typical example where augmentation is required concerns accessibility of windowing interaction by motor-impaired users; we will address in more detail this example within the next Section. In both example cases above, augmentation implies the development of new software interaction techniques, as well as the installation of special purpose I/O devices (e.g. voice I/O hardware, binary switches). It is evident that augmentation promotes both accessibility and interaction quality, which are the two objectives in the context of “User Interfaces for All”.

3.2 Examples where toolkit augmentation has been supported

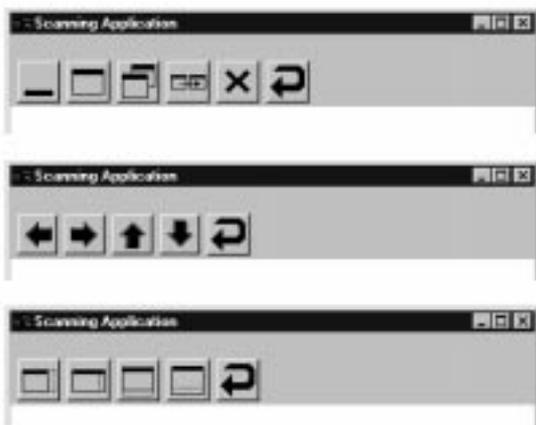


Figure 3.2-1: The additional accessible window management toolbar.

We will briefly discuss a real-life project which addressed the problem of augmenting the WINDOWS object library with embedded scanning techniques, in order to enable motor-impaired user access [Savidis et al., 1997b]. Scanning is a technique where the dialogue is decomposed on the basis of only two fundamental actions: *next* and *select*. No spatial location devices like the mouse, or multiple-key devices like the keyboard can be used by the target user group, due to severe motor disability.

One of the most important enhancements in this work [Savidis et al., 1997b], has been the decomposition and augmentation of the user-dialogue for performing window management. All top-level window interaction objects have been augmented with an additional accessible toolbar, supporting scanning interaction, thus providing all window-management operations in an accessible form (see Figure 3.2-1). Apart from top-level windows (i.e. *FrameWindow* class in the WINDOWS object library), augmented dialogues for the rest object categories (e.g. button categories, container classes, composite objects, and text-entry objects) has been also designed and implemented. In our implementation approach, augmented object classes inherit (in Object Oriented Programming - OOP - terms) from their corresponding original WINDOWS object classes, thus maintaining the original behaviour and programming interface (i.e. object attributes and methods). The augmented dialogue, as well as particular new object attributes and methods, are all defined as part of the sub-class, without affecting the native WINDOWS classes.

3.3 Minimalistic implementation requirements

The purpose of the minimalistic requirements is to define a set of functional capabilities through which augmented object classes can be realistically implemented. The set of these requirements which will follow, is not tight to any particular programming language, nor the

type of language (e.g. procedural, object-oriented, scripting). We already assume that the typical minimal toolkit functionality, such as creating object hierarchies, reading or writing object attributes, defining call-backs and implementing event handlers is already present.

- Device installation can be supported. This may require low-level software to be written, and it will work either via a polling-based scheme (i.e. continuously checking device status), or a notification-based scheme (i.e. device-level software may asynchronously sent notifications when device input is detected).
- Manipulation of focus object. During interaction, different interaction objects will normally gain and loose the focus, via user control (e.g. through mouse or tab-keys in WINDOWS) . In order to augment interaction, it is needed to have programming control of the focus object, since any device input originated from the extra peripherals will always concern the particular object having the dialogue focus.

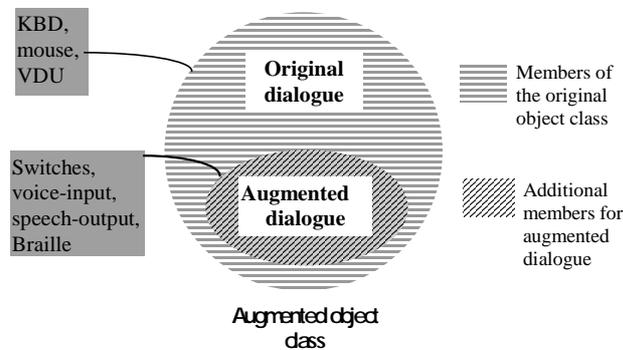


Figure 3.3-1: Relationship between original and augmented object classes.

- Manipulation of the object hierarchy. When implementing augmented interaction, it is necessary to provide augmented analogies of the user-focus control actions; this is required since the user must be enabled to “navigate” within the interface. Hence, the hierarchical structure of the interface objects must be accessed in a programming manner, as part of the navigation dialogue implementation.

The relationship between the native object classes and the augmented object classes is a typical *ISA* relationship, in the sense that augmented classes inherit all the features of the original toolkit object classes (see Figure 3.3-1). This scheme can be implemented either via sub-classing, if the toolkit is provided in an OOP framework, or via composition, in case of non-OOP classes. The composition is realised by defining an augmented object structure which collects the original as well as the augmented object features, including also an instance declaration for the original object class; this explicit instantiation is necessary to physically realise the toolkit object, which in the case of sub-classing would be automatically carried out. Code should be written to maintain a consistent mapping between the attributes of the toolkit object instance and its corresponding features defined as part of the new object structure.

3.4 Maximalistic implementation requirements

The maximalistic requirements include the minimalistic requirements, as well as some new functional criteria, primarily targeted towards enabling an easier and more modular implementation of the augmented object classes. The extra maximalistic requirements are listed below, while their application is illustrated in Figure 3.4-1.

- Expanding object attributes and methods directly supported by the original toolkit classes. This will alleviate the problem of defining new object classes.

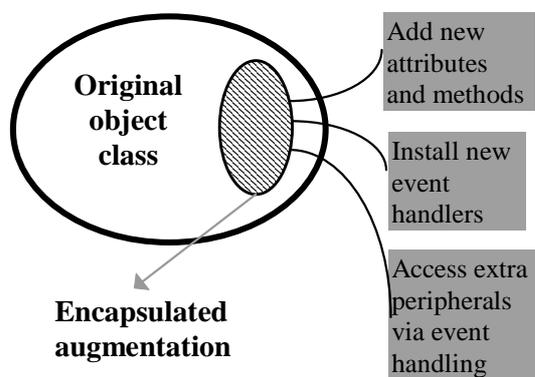


Figure 3.4-1: Augmentation in case of maximilastic functionality.

- Visible and extensible constructor, where additional behaviour can be added. This will allow to install new event-handlers and perform necessary initializations directly at the original class level.
- Clean device installation layer, so that new device input can be attached to toolkit input-event level. This will facilitate the management of extra peripherals through the original event-management toolkit layer.

4. TOOLKIT EXPANSION

4.1 Definition and importance in the context of User Interfaces for All

Expansion, over a particular toolkit, is defined as the process through which toolkit users introduce new interaction objects, not originally supported by that toolkit. An important requirement of toolkit expansion is that all newly introduced interaction objects should be made available, in terms of programming, in exactly the same manner as with original interaction objects; hence, for toolkit users not aware of the original toolkit classes, it should be indistinguishable which object classes have been implemented as add-ons.

In the context of diverse user requirements and varying situations and contexts of use, it is likely that new interaction paradigms, metaphors and dialogue techniques will be dictated. There are already existing cases clearly demonstrating this need: (a) multi-media CD-ROMs (i.e. information systems), targeted to a broad user population, employ custom-made interaction controls and provide graphical design paradigms clearly beyond the windowing-based traditions; and (b) educational software products, primarily those targeted for young children, employ new metaphors of interaction and practically reject the windowing-based interaction metaphor. However, all these software products still run on top of mainstream graphical toolkits, such as WINDOWS object library. As a result, developers are faced with the problem of expanding the basic interaction facilities with new ones, serving better the needs of their target user groups.

4.2 Examples where toolkit expansion has been supported

Currently, toolkit expansion mechanisms are supplied within various categories of commercial interface tools, while there is a considerable variety with respect to: the way expansion is supported, how difficult is for developers to define new interaction objects, and the method for employing newly introduced objects within normal interface development. We put particular emphasis on the easiness of the approach, since even though powerful expansion features may be supported, the inherent complexity of the mechanism may pose practical barriers. One of the early toolkits providing expansion support has been the generic Xt toolkit interface, standing on top of the Xlib library for X Windowing System. The Xt

mechanism provides a template widget structure where the developer has to provide some implemented constructs. The mechanism of Xt is complicated enough to turn expansion to an expert's programming task. Other approaches to expansion concern toolkit frameworks supported for OOP languages, such as C++ or JAVA. If key super-classes are distinguished, with well documented members, providing the basic interaction object functionality, then expansion becomes straightforward sub-classing. This is the typical case with OOP toolkits like WINDOWS object library or InterViews. Apart from programming toolkits, the expansion mechanism is also met in higher-level development tools, like User Interface Management Systems (UIMS). The I-GET UIMS [AT-HCI, 1997] provides a 4GL-based mechanism for object expansion, which is particularly easy to employ. The Visual Basic tool, for interactive interface construction and scripting, is currently supported with various peripheral tools from third-party vendors; one such tool, called VBXpress, supports the interactive construction of new VBX interaction controls (i.e. expansion).

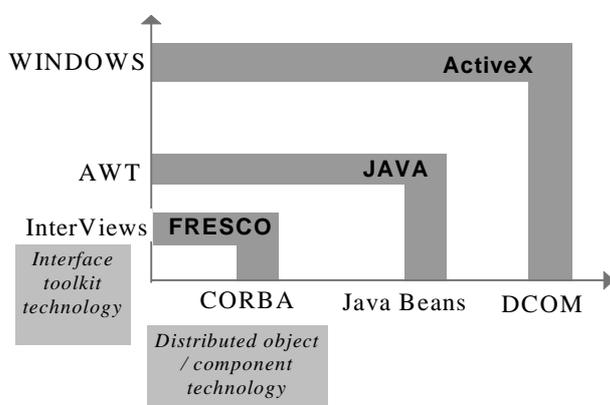


Figure 44.2-1: Combining toolkit technologies with distributed object / component technologies.

Finally, the last and more advanced approach to toolkit expansion concerns distributed object technologies for interoperability and component-based development. New interaction objects can be introduced through the utilisation of a particular tool, while being utilised by another. This functionality is accomplished on the basis of generic protocols for remote access to various software resources, supporting distribution, sharing, functionality exposure, embedding, and dynamic invocation.

Typical examples of such technologies are CORBA and DCOM, while interaction-object technologies derived from mixing conventional toolkits with such technologies are the FRESKO User Interface System (from the InterViews toolkit) and ActiveX controls (from the WINDOWS object library) respectively (see Figure 4.2-1).

4.3 Minimalistic implementation requirements

In order to minimally support expansion of interaction objects, the interface tool should provide an expandable interaction object framework. The most typical forms of such expandable object frameworks are:

- Super-class (via derived classes);
- Template structure (by filling implementation gaps);
- API (implementing multiple services complying to a particular API for external clients);
- 4GL object model (interaction objects are defined via dedicated mechanisms in specialised 4GLs);
- Physical pattern (building physically an interaction object around pre-defined physical patterns).

4.4 Maximalistic implementation requirements

Maximalistic requirements impose one additional need over the minimalistic ones: if an interface tool supports object expansion, then it should be allowed to implement the dialogue for new interaction objects via the native dialogue construction facilities of the interface tool. In other words, developers should be allowed to define dialogues for new interaction objects via the facilities they have been already using for implementing conventional interfaces. For instance, in an interactive construction tool, maximal functionality for expansion is considered to be available only if interactive object design and implementation is enabled.

5. TOOLKIT ABSTRACTION

5.1 Definition and importance in the context of User Interfaces for All

Toolkit abstraction is defined as the ability of the interface tool to support manipulation of interaction objects which are thoroughly relieved from physical interaction properties. Abstract interaction objects are high-level interactive entities reflecting generic behavioural properties, having no input syntax, interaction dialogue, and physical structure. An example of an abstract interaction object is provided in Figure 5.1-1, where an abstract *selector* object is illustrated. Such an abstract interaction object has only two properties: the number of options and the selection (as a number) made by the user.

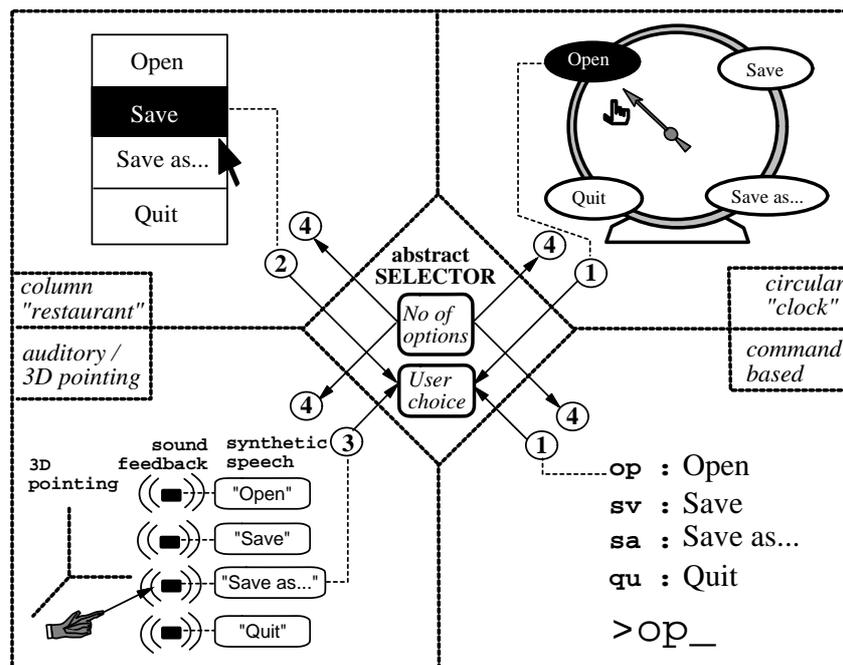


Figure 5.1-1: An abstract *selector* interaction object.

As it is indicated in Figure 5.1-1, multiple alternative physical styles, possibly corresponding to different metaphors of interaction, can be defined as physical instantiations of the abstract *selector* class. When designing and implementing interfaces for diverse user groups, even though considerable structural and behavioural differences are naturally expected, it is still possible to capture various syntactic commonalities, by analysing the structure of sub-dialogues at various levels of the task hierarchy. In order to promote effective and efficient design, implementation, and refinement cycles, it is crucial to express such shared patterns into various levels of abstractions, in order to support modification only at a single abstract level, by automating the propagation to the various alternative physical artefacts.

Abstract interaction objects can be employed for designing and implementing dialogue patterns which need to have no physical interaction properties. In this sense, such dialogue patterns are not restricted to any particular user group or style of interaction. The introduction of the intermediate physical instantiation levels, so that abstract forms can be mapped to concrete physical structures is also required. By automating such an instantiation mechanism, development for diverse users is facilitated in a unified fashion (at an abstract layer), while the physical realisation is automated on the basis of an appropriate object instantiation mechanism.

5.2 Examples where toolkit abstraction has been supported

Behaviour generalization. Approaches which fall in this category aim to support interaction objects at a level above toolkits, however, they fail to provide pure abstract interaction entities, due to explicit involvement of various physical properties. *Interaction tasks* [Foley et al., 1984] and *interactors* [Myers, 1990] are the most representative paradigms of generic highly parameterized primitives, derived from an in depth analysis of the various behavioural aspects of interaction in direct manipulation graphical interfaces. The notion of cross-platform objects indicates interactive entities being re-targetable to different platforms; this concept has been already discussed under toolkit integration.

Behaviour abstraction. The concept of *meta-widgets* [Wise et al., 1994] has been introduced to designate interaction entities above platforms, exhibiting no physical interaction properties. Current support for meta-widgets is provided by means of non-expandable implementation [Wise et al., 1994], with respect to the classes of meta-widgets available and their physical instantiation schemes. The notion of *virtual objects*, originally referring to multi-platform objects [Myers, 1995], has been revisited and enhanced in [Savidis et al., 1995a] to indicate abstraction on top of toolkits and interaction metaphors. Open mechanisms for defining abstract interaction objects and mapping to physical interaction classes, in a polymorphic fashion, have been supported in the HOMER UIMS [Savidis et al., 1995a] and within the I-GET UIMS [AT-HCI, 1997].

5.3 Minimalistic implementation requirements

In the discussion of the minimalistic requirements which follows, we define a basic set of functional criteria judging if interface development tools facilitate development based on abstract objects. Additionally, we also discuss some high-level implementation issues, revealing the complexity of explicitly programming abstract objects, if the interface development tool does not support them explicitly.

- The interface tool supplies a pre-defined collection of abstract interaction object classes.
- For each abstract object class, there is a pre-defined mapping scheme to various alternative physical object classes - *bounded polymorphism*;
- For each abstract object instance, the developer may select which of the alternative physical classes (in the mapping scheme) will be instantiated - *controllable instantiation*;
- More than a single physical instance may need to be active for a particular abstract object instance - *plural instantiation*.

The above requirements imply that the developer is enabled to instantiate abstract objects, while having control on which physical mapping schemes will be active for each abstract object instance; mapping schemes define the candidate physical classes for physically

realising an abstract object class. The need for having multiple physical instances active, all attached to the same abstract object instance (i.e. plural instantiation), is imposed in case that a running interactive application maps to multiple concurrent physical interface instances.

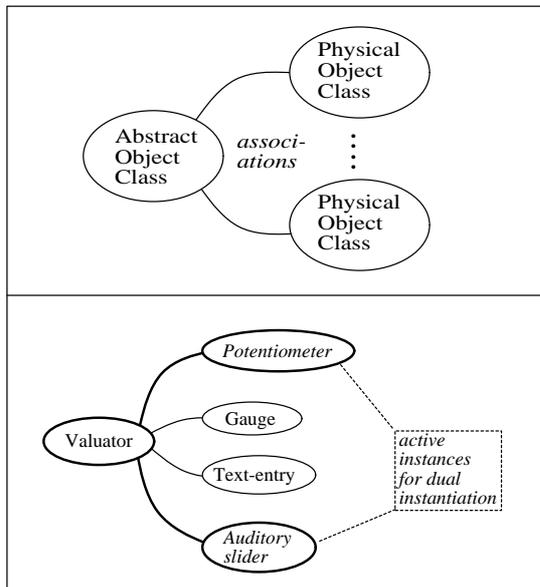


Figure 5.3-1: Polymorphic physical mapping for abstract objects (top), and plural instantiation in dual interaction (bottom).

This is a typical case for CSCW applications, and it is also needed in the context of Dual interface development [Savidis et al., 1995a], where two concurrently active instances are required (i.e. a visual and a non-visual) for each abstract object instance.

The notion of polymorphic physical mapping, as well as plural instantiation for dual interaction is outlined in Figure 5.3-1. This type of polymorphism has different functional requirements, with respect to polymorphism of super-classes in OOP languages.

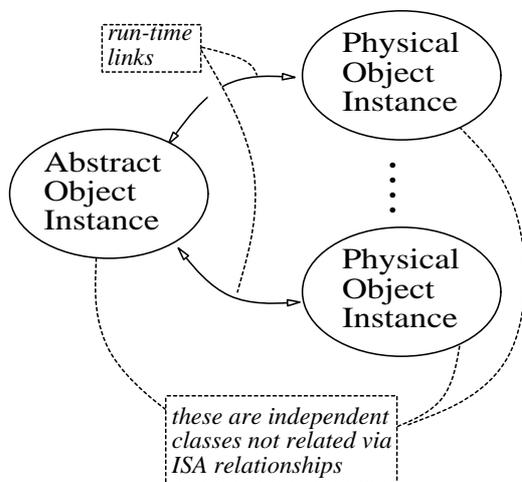


Figure 5.3-2: Necessary run-time links between abstract objects and their corresponding physical instances.

- In user interface development, polymorphism aims to support various dialogue faces (i.e. polymorphism with its direct physical meaning), while instantiation must be applied on abstract classes; multiple physical instances, manipulated via the same abstract object instance, may be active in parallel.
- In OOP, abstractions aim to primarily support re-use and implementation independence (i.e. polymorphism with its metaphoric meaning), while instantiation is always applied on derived classes; use of reference variables to the abstract class, for manipulating multiple types of physical instance is commonly applied.

Clearly, the traditional schema of abstract / physical class separation in OOP languages by means of class hierarchies and ISA relationships cannot be directly applied for implementing the abstract / physical class schema as needed in interface development. An explicit run-time architecture is required, where connections among abstract and physical instances are explicit programming references, beyond the typical *instance-of* run-time links from ISA hierarchies (see Figure 5.3-2).

5.4 Maximalistic implementation requirements

If the maximalistic requirements are met, an interface tool provides powerful methods for manipulating abstractions, such as: defining, instantiating, polymorphosing, and extending abstract interaction object classes.

- Facilities to define new abstract interaction object classes;
- Methods to define schemes for mapping abstract classes to arbitrary sets of physical interaction object classes;
- Mechanisms for defining run-time associations and dependencies between an abstract instance and its various concurrent physical instances; this may require the definition of attribute dependencies and propagation of call-back notifications (i.e. if a particular physical instance is manipulated by the user, the abstract instance must be appropriately notified);
- Enabling direct programming access, through the abstract object instance, to all associated (concurrently) physical instances.

6. DISCUSSION AND CONCLUSIONS

Interaction objects play a central role in interface development; a large part of the software for implementing commercially available interface tools concerns vast amounts of graphical interaction controls. The basic layer providing the implementation of interaction objects is defined to be the *toolkit layer*, while interface tools typically provide additional layers on top of the toolkit layer. We have studied object-based interface development under the perspective of “User Interfaces for All”, and we have identified four key mechanisms for manipulating interaction objects: integration, augmentation, expansion and abstraction. The relationships among these fundamental mechanisms are illustrated in Figure 6-1.

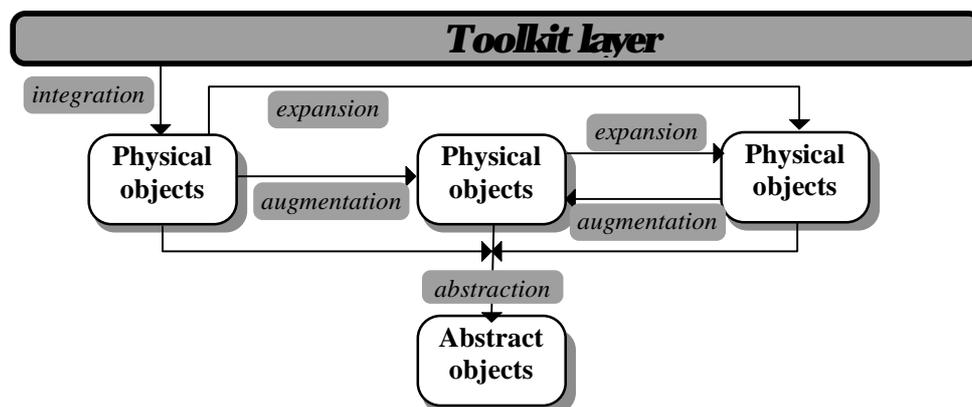


Figure 6-1: Relationships among the four fundamental mechanisms for object manipulation.

The support for each of the basic mechanisms varies today. Regarding toolkit integration, the vast majority of commercial tools is targeted towards multi-platform support in a hard-coded manner, rather than providing open mechanisms for connecting to arbitrary toolkits. Toolkit augmentation is supported in most programming-based interface tools, while higher-level development tools are very weak in this perspective. Toolkit expansion is also supported in most programming-oriented interface tools, but the considerable required overhead, as well as

the inherent implementation complexity, turns the expansion task to an activity primarily targeted to expert programmers; regarding higher-level development tools, there is an increasing number of interactive construction tool-products supporting expansion, while 4GL-based interface tools supporting expansion currently count to a number of one or two. Finally, toolkit abstraction, being the most promising mechanism towards the User Interfaces for All objective [Stephanidis et al, 1997a], is the least supported mechanism in existing interface tools; the I-GET 4GL-based UIMS [AT-CHI, 1997] is the only tool currently known to fulfil the maximal abstraction requirements.

Studying the mechanisms from a global perspective, it is observed that no commercial tool provides support for all of the mechanisms. Since distinct tools provide functionality for these mechanisms to a different degree, it becomes necessary to allow developers to employ multiple tools, by taking advantage of the distinctive features of each interface tool. This paradigm shift from monolithic tools / environments, towards multiple servers and tools, requires specific advancements in the area of interface development tools. In particular, interoperability, distribution and component-based development are the key properties which are required. In the past few years, the commercial support in this context has exploded, leading to currently mature component technologies and multi-server based interface development paradigms.

The Web paradigm, starting from the early support for interactive distributed hypertext documents, is evolving to an application development layer, supporting remote downloading of application components for highly interactive, assembled on the fly, interactive software products. The fundamental software layers to accomplish this target falls in the domain of distributed software technologies such as CORBA (by OMG) and DCOM (by Microsoft), and software component technologies like Java Beans (by JavaSoft) and ActiveX (by Microsoft). By blending such technologies with typical toolkit architectures, and by enhancing the present facilities regarding the identified four fundamental mechanisms, the manipulation capabilities for objects are considerably enhanced, closing more the gap between the present software interface technology and the objective towards "User Interfaces for All".

ACKNOWLEDGEMENTS

Part of this work has been carried out in the context of the ACCESS Project (TP1001) funded by the TIDE Programme of the European Commission. Partners in this consortium are: CNR-IROE (Prime Contractor), Italy; ICS-FORTH, Greece; University of Athens, Greece; RNIB, U.K.; SELECO, Italy; MA Systems Ltd., U.K.; Hereward College, U.K.; National Research and Development Centre for Welfare and Health, Finland; VTT, Finland; PIKO Systems, Finland; University of Herfordshire, U.K.

REFERENCES

- [Bass et al., 1990] Bass, L., Hardy, E., Little, R., Seacord, R. Incremental development of User Interfaces. In *Engineering for Human-Computer Interaction*. G. Cockton, Ed. North-Holland, 1990, 155-173.
- [Foley et al., 1984] Foley, J. D., Wallace, V. L., Chan, P.. The human factors of computer graphics interaction techniques. *IEEE Computer Gr. & Appl*, 4, 11 (November 1984), 13-48.

- [X Consortium, 1994] FRESCO™ Sample Implementation Reference Manual. X Consortium Working Group Draft, Version 0.7, April 9, 1994.
- [Guinan, 1997] Guinan, J. Platform-Independent C++ GUI Toolkits. In *C/C++ Users Journal (CUJ)*, January 1997, 19-26.
- [AT&HCI, 1997] The I-GET UIMS tool. http://www.ics.forth.gr/proj/at-hci/html/tools__i-get.html
- [Myers, 1990] Myers, B. A. A New Model for Handling Input. *ACM Trans. Inform. Syst.* 8, 3 (July 1990), 289-320.
- [Myers, 1995] Myers, B. User Interface Software Tools. In *ACM Transactions on Human-Computer Interaction, Vol 2, No. 1*, March 1995, 64-103.
- [Petrie et al., 1996] Petrie, H., Morley, S., McNally, P., Graziani, P., Stephanidis, C., Savidis, A., Majoe, D. An interface to hypermedia systems for blind people. In the proceedings of the *ACM Hypertext'96* (demonstration).
- [Savidis et al., 1995a] Savidis, A., Stephanidis, C. Developing Dual Interfaces for Integrating Blind and Sighted Users: the HOMER UIMS. In proceedings of the ACM CHI'95 conference in Human Factors in Computing Systems, Denver, Colorado, May 7-11, 106-113.
- [Savidis et al., 1995b] Savidis, A., Stephanidis, C. Building non-visual interaction through the development of the Rooms metaphor. In companion of the *CHI'95 conference in Human Factors in Computing Systems*, Denver, Colorado, May 7-11, 244-245.
- [Savidis et al., 1997a] Savidis, A., Stephanidis, C., Akoumianakis, D. Unifying Toolkit Programming Layers: a Multi-purpose Toolkit Integration Module. In proceedings of the Eurographics DSV-IS'97 workshop in Design, Specification and Verification of Interactive Systems, Granada, Spain, Springer-Verlag, 1997.
- [Savidis et al., 1997b] Savidis, A., Vernardos, G., Stephanidis, A. Embedding scanning techniques accessible to motor-impaired users in the WINDOWS object library. In *Design of Computing Systems: Cognitive Considerations*, 21A, Elsevier, 1997, 429-432.
- [Savidis et al., 1997c] Savidis, A., Stergiou, A., Stephanidis, C. Metaphor Fusion in Non-visual Interaction. In poster sessions (abridged proceedings) of the HCI International'97 conference, San Francisco, August 26-29, pp. 61.
- [Stephanidis et al., 1997a] Stephanidis, C. , Savidis, A., Akoumianakis, C. Unified Interface Development: Tools for Constructing Accessible and Usable User Interfaces. All day tutorial, August 26, HCI International'97, San Francisco.
- [Stephanidis et al., 1997b] Stephanidis, C., Paramythis, A., Savidis, A., Sfyraakis, M., Stergiou, A., Leventis, A., Maou, N., Paparoulis, G., Karagiannidis, C. *Developing Web Browsers Accessible to All: Supporting User-Adapted Interaction*, to appear in the 4 th European Conference for the Advancement of Assistive Technology (AAATE '97), Thessaloniki, Greece, 29 September - 2 October 1997.
- [Wise at al., 1995] Wise, G. B., Glinert, E. P. Metawidgets for multimodal applications. In proceedings of the RESNA'95 conference, Vancouver, Canada, June 9-14, 455-457.