Inclusive development: Software engineering requirements for universally accessible interactions

Anthony Savidis\textsuperscript{a}, Constantine Stephanidis\textsuperscript{a,b,*}

\textsuperscript{a}Foundation for Research and Technology—Hellas (FORTH), Institute of Computer Science, Vassilika Vouton, GR-71300, Heraklion, Crete, Greece
\textsuperscript{b}University of Crete, Department of Computer Science, Greece

Received 22 June 2004; revised 17 June 2005; accepted 18 June 2005
Available online 11 August 2005

Abstract

The notion of ‘universal access’ reflects the concept of an Information Society in which potentially anyone (i.e. any user) will interact with computing machines, at anytime and anyplace (i.e. in any context of use) and for virtually anything (i.e. for any task). Towards reaching a successful and cost effective realization of this vision, it is critical to ensure that the future interface development tools provide all the necessary instrumentation to support inclusive design, i.e. facilitate \textit{inclusive development}. In the meantime, it is crucial that both tool developers and interface developers acquire awareness regarding the key development features they should pursue when investigating for the most appropriate software engineering support in addressing such a largely demanding development goal (i.e. universally accessible interactions). This paper discusses a corpus of key development requirements for building universally accessible interactions that has been consolidated from real practice, in the course of six medium-to-large scale research projects, all completed, within a 10 years timeframe.

© 2005 Elsevier B.V. All rights reserved.

* Corresponding author. Tel.: +302810391741; fax: +302810391740.
E-mail address: cs@ics.forth.gr (C. Stephanidis).

0953-5438/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.intcom.2005.06.005
1. Introduction

In the context of the Information Society, the notion of ‘computing platform’ concerns a wide range of devices, apart from traditional desktop computers, such as public-use terminals, phones, TVs, car consoles, and home appliances. Today, such computing platforms are mainly delivered with embedded operating systems (such as Windows Mobile, Java Micro Edition, or Symbian), while the various operational capabilities (e.g. interaction) and supplied services are controlled through software. As a result, it is expected that virtually anyone may potentially use interactive software applications, from any context of use, to carry out any particular task. In this context, interactive software products should be developed to address the demands of ‘running anywhere’ and be ‘used by anyone’. It is argued that, due to the above, existing software development processes will have to be revisited, as they suffer from two fundamental problems:

- They are designed for the so-called ‘average’ user. However, in the context of an Information Society, where potentially anyone becomes a computer-user, it is not useful to define an ‘average’ user case.
- They are typically developed for desktop computer systems (high-resolution display, mouse and keyboard), usually running windowing interactive environments. Instead, in the Information Society context, the following are expected: (a) a wide range of computing platforms, with large variations of interactive peripheral equipment; and (b) new ways of metaphoric interaction maximally designed for particular usage-contexts and/or individual user characteristics, departing from the old traditional windows metaphor (Smith et al., 1982).

This new situation has been already identified by the introduction of new concepts, which have been intensively supported through related research and development efforts, such as: user interfaces for all (Stephanidis, 2001a), ubiquitous computing (Weiser (1991), Abowd and Mynatt, 2000), migratory interfaces (Bharat and Cardelli, 1995), wearable interfaces (Bass et al., 1997), ambient intelligence (ERCIM NEWS (2001)), disappearing computer (IST/FET, 2001), and plastic user interfaces (Calvary et al., 2001). The above concepts share many common objectives, principles and semantics, consolidated altogether in a situation depicted within Fig. 1, showing the three key layers of diversity in the context of universally accessible interactions. Clearly, there are still numerous categories of software systems that are not targeted for potentially anyone, anywhere at anytime. For instance, mission critical systems, like aircraft cockpits, air traffic control systems, and space shuttle management systems, are all designed to be optimally usable for specific, and in many cases restricted, groups of people. In this context, the reported work on universally accessible interactions is concentrated on software systems that are primarily targeted to broad user populations, used under largely varying situations and environments.

This paper will report a consolidated account of an in depth technical analysis, originated from real-life development experience in the course of six medium-to-large scale research projects, spanning across a timeframe of about 10 years, by presenting
an appropriate definition and classification of the key software engineering requirements
and the necessary functionality to produce universally accessible interactive applications
and services. Those research projects have been partially funded by the European
Commission—(TP1001—ACCESS\textsuperscript{1}; IST-1999-20656-PALIO\textsuperscript{2}; ACTS AC042-
AVANTI\textsuperscript{3}; IST-1999-14101-IS4ALL\textsuperscript{4}; IST-2000-25286-2WEAR\textsuperscript{5}), or by national
funding agencies (EPET-II: NAUTILUS\textsuperscript{6}).

The key objective of this retrospective analysis has been the identification of commonly
occurring development requirements, genuinely emerging from the primary need to
support universally accessible interactions—anywhere, for anyone and anytime—by
offering an appropriate classification scheme denoting the generic and representative
categories in the form of software engineering requirements.

Hence, irrespective of the adopted software engineering method (i.e. ‘approach to
development’, following the IEEE-90 definition), there are specific requirements
emerging from the pursuit of inclusive interactions. Such requirements are very critical

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{The three key layers of diversity in the context of universally accessible interactions.}
\end{figure}

\footnotesize
\begin{itemize}
\item Personalized Access to Local Information and Services for Tourists, 2000–2003.
\item Information Society for All, 2001–2003.
\end{itemize}
to software developers as they enable to foresee the concrete development challenges during the overall engineering process.

In this context, the focal point of the discussion is shifted away from a particular development tool-type, such as interface toolkits. Instead, it is targeted towards a concrete set of unavoidable top-level implementation issues, while outlining the profile of the necessary development instrumentation to practically address those, drawing from real life experience. Some of those requirements concern necessary enhancements of the development process, some relate to required implementation mechanisms of the host tool, while other emphasize the need for special-purpose software engineering strategies. The results of this analysis are summarized in four general categories of development requirements, each representing a particular dimension of universally accessible interactions:

- **Need for new interaction metaphors.** In view of the wide variety of computing platforms, situations of use, individual end-users, and tasks to be carried out through interactive software products, it is critical to provide the most appropriate metaphoric designs to ensure the highest possible interaction quality, supporting intuitive interaction, ease-of-use, as well as efficiency, effectiveness and user-satisfaction. As it will be discussed, the development of new metaphors, different from the desktop graphical style traditions, may either be necessary in some situations, or may constitute the foundation for experimenting with future interaction paradigms.

- **Need for manipulating diverse collections of interaction objects.** The development of universally accessible interactions encompasses diverse interaction elements, which, on the one hand, may be originated from different interaction metaphors (e.g. windowing and Rooms—Savidis and Stephanidis, 1995a), while, on the other hand, can be realized into alternative physical forms (e.g. 2D/3D visual graphical, auditory, tactile). As a result, it is crucial that interface tools supply to interface developers all the necessary implementation mechanisms for the manipulation of such diverse categories of interaction elements.

- **Need for automatic interface adaptation.** In order to maximize the quality of the delivered interfaces, it is imperative to support user- and usage-context- awareness, while enabling the interface to adapt itself ‘on-the-fly’ to the particular end-user and usage-context. Automatic interface adaptation implies that the software encompasses and organizes appropriately alternative dialogue patterns in an implementation form, inherently requiring appropriate software engineering for the run-time activation and control of dynamic dialogue components.

- **Need for ambient interactions.** In order to support user mobility in an open computational environment, it is necessary to support typical scenarios in which the ‘the environment is the interface’. The latter requires facilities for dynamic discovery, control and utilization of remote dynamically exposed User-Interface micro-services embedded in environment devices.

It should be noted that the methods and practices reported in this paper concern the specific domain of User Interface development, with a particular focus on universal access (i.e. interfaces for anyone, anywhere, anytime). In this context, the focal point of
the discussion is not on introducing updates on general-purpose software engineering practices, like agile development, extreme programming or software patterns, as those are actually orthogonal to domain-specific practices. For example, although compiler-construction embodies engineering strategies that do not originate (and cannot originate) from the general software engineering field, they can be clearly combined with general-purpose software engineering methods. Similarly, in this paper, a corpus of domain-specific software-engineering requirements is defined, applicable in the context of universally accessible interactions, while the particular methods and techniques presented do not constitute propositions for the general software engineering field.

2. Identification of requirements

Following the outline of the most prominent requirements towards universally accessible interactions, a detailed account of the development needs to accommodate those requirements in real practice follows. In this context, driven from implementation experience in the course of real-life projects, appropriate methods to address those requirements are introduced, followed by an in-depth analysis of the inherent lower-level software engineering requirements that need to be met by the employed development tools. In this context, the presented domain-specific code of practice is not to be considered as the only possible solution to effectively cope with the specific formulated requirements. However, it constitutes a repeatedly tested and validated recipe, that developers may well consider as an appropriate starting point.

2.1. The need for new interaction metaphors

The proliferation of advanced interaction technologies has enabled the construction of new interactive experiences in popular application domains. Thus, for example, educational software titles provide new interactive computer embodiments based on metaphoric themes that children are familiar with, such as the playground, or interactive books. Interaction metaphors concern the realisation of real world objects in the interaction context through computing artifacts that directly reflect the representation and behaviour of their real world counterparts. The key role of metaphors is to provide users natural means to attain a wider range of tasks in a manner that is effective, efficient and satisfactory, by providing a better cognitive fit between the interactive embodiment of the computer and real-world analogies with which users are already familiar. It is expected that new metaphors will depart significantly from graphical window-based interaction, which, inspired from the Star interface (Smith et al., 1982), was intended to meet the demands of able-bodied users working in an office environment.

The foreseen transition from the current massive use of WIMP interfaces to the post-WIMP era has been identified earlier in (Van Dam, 1997), with primary emphasis on new forms and metaphors of graphical interaction. The need for sensing environment events and supporting context-specific input in developing future interactive systems has been identified in the Context toolkit (Salber et al., 1999). Additionally, in the white paper of the recently formed Accessibility Group of the Independent Game Developers Association,
(IGDA Accessibility, 2004), the effort towards marrying universal access and video games clearly poses new design challenges for effective and accessible metaphoric interactions. Currently, the design and realization of interaction metaphors is not supported by appropriate reusable software-libraries and toolkits. Rather, it is internally hard-coded within the User Interface software of interactive applications and services. Existing multimedia-application construction libraries are too low-level, requiring that developers undertake the complex task of building all metaphoric interaction features from primitive interaction elements. For instance, even though various educational applications provide virtual worlds familiar to children as an interaction context, the required ‘world’ construction kits for such specialized domains and metaphors are lacking. This is analogous to the early period of Graphical User Interfaces (GUIs) where developers used a basic graphics package for building window-based interactive applications. However, the evolution of GUIs into a de facto industry standard did not take place until tools for developing such User Interfaces became widely available. Similarly, it is argued that the evolution of new metaphors to facilitate the commercial development of novel applications and services targeted to the population at large will require the provision of the necessary implementation support within interface tools. This paper (see Section 3.1) provides a metaphor development methodology which may be employed to pursue the design and implementation of new interaction metaphors, exposing the key demands for crafting metaphor-specific interface toolkit libraries, to practically assist in building new forms and styles of interactions. In this context, the methodology itself becomes a necessary technical ingredient that developers should possess, so as to effectively attack the challenge of building new interaction metaphors. Additionally, two metaphor development cases will be also presented: a metaphor-flexible toolkit for non-visual interaction, and a collection of new graphics-intensive based interaction methods.

2.2. The need for manipulating diverse collections of interaction objects

As discussed in the previous section, the effective deployment of new interaction metaphors is dependent on the practical availability of the necessary tools to support the implementation of the dialogue patterns and artifacts embodied in those metaphors (e.g. visual windowing-, 3D auditory-, tactile-, or switch-based scanning- dialogues). As in the case of GUIs, the primary means to construct metaphoric interactions are likely to be in the form of implemented reusable interaction elements provided by software libraries commonly known as toolkits (e.g. OSF/Motif, MFC, InterViews, Xaw/Athena widget set, JFC). Such tools provide programming facilities to mainly: (i) manipulate interaction objects and construct object hierarchies; (ii) handle incoming input-device events; and (iii) display lower-level graphic primitives.

Traditionally, interaction objects (e.g. windows, buttons, scrollbars, check-boxes, etc.) have been treated as the most important category of interaction elements, since the largest part of existing interface development toolkits is exclusively devoted to providing rich sets of interaction objects, appropriately linking to the underlying functionality. Moreover, interaction objects constitute a common vocabulary for both designers and user interface programmers, even though the type of knowledge typically possessed by each group is rather different. Thus, designers usually have more detailed knowledge regarding
the appropriateness of different interaction objects 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 specific physical entity being represented. Interface toolkits have traditionally provided a means for bridging the gap between lexical-level design and implementation. In other words, they provide a vehicle for the direct mapping of common lexical-level user tasks, such as selection, command or text-entry, revealed through a design activity, to their respective implementation constructs available in a target toolkit, like a menu, a push button or a text-field.

In the course of developing applications targeted to radically different user groups, engaging largely diverse styles of interactions, the necessity has been experienced of introducing new, functionally extended forms of interaction objects (as compared to traditional graphical toolkit elements). Those extended functional needs have emerged in the context of real-life special-purpose application developments, such as a dual hypermedia electronic book (Petrie et al., 1997), an advanced interpersonal communicator (Kouroupetroglou et al., 1996), and a full-fledge user-adapted Web browser (Stephanidis et al., 2001). Such applications were targeted to diverse user groups, differing with respect to physical, sensory and motor abilities, preferences, domain-specific knowledge, and role in organizational context. In this context, various interaction technologies and interaction metaphors had to be employed, including graphical windowing environments, auditory/tactile interaction, Rooms-based interaction metaphors, etc. In the course of the above developments, the following set of commonly recurring functional requirements has been consolidated, regarding the required implementation facilities to manipulate interaction objects:

- **Integration** of third-party collections of interaction objects, the latter offered as independent toolkit libraries, while delivering novel interaction facilities possibly complying to alternative interaction metaphors;
- **Augmentation**, i.e. provision of additional interaction techniques on top of existing interaction objects offered by the currently available toolkits, in cases where the original dialogue for such objects is considered inadequate for specific combinations of target user groups and contexts of use characteristics;
- **Expansion** of the original set of interaction objects by introducing new custom-made interaction objects, when particular newly-designed interactive behaviors are not implemented by the utilized toolkits;
- **Abstraction** applied on interaction objects even when those comply to different interaction metaphors, through the delivery of abstract interaction objects supporting the construction of logical dialogue kernels that are completely relieved from physical interaction characteristics, while appropriately supporting polymorphic physical-level binding.

2.3. The need for automatic interface adaptation

In the context of anyone, anywhere, anytime access to Information Society Technologies, due to the large diversity of potential target user groups and contexts of
use, it is unrealistic to expect that a single interface design instance will ensure high-quality interaction. Instead, it is most likely that alternative design decisions will have to be taken. Therefore, the ability of the interface to automatically adapt to the individual end-user, as well as to the particular context of use, is a required key property of universally accessible interactions. From the development point of view, such a facility reveals the need for explicit design and implementation of alternative interactive ways to enable different end-users carry out the same task, i.e. alternative dialogue patterns even for the same sub-tasks. During run-time, the software interface relying upon computable user-oriented knowledge (i.e. internally stored profiles), is responsible to assemble the eventual interface ‘on-the-fly’, by collecting and gluing together all the various end-user specific constituent dialogues components. This type of initial best-fit automatic interface configuration, originally introduced in the context of the Unified User Interfaces Development framework (Stephandis, 2001b), has been called adaptability. Additionally, the dynamic interface behaviour to enable a continuous enhancement of the dialogue with the user by ‘changing’ the original interface, based on interaction monitoring, has been commonly referred to as adaptivity or adaptive interaction. In this context, the software engineering of the user interface to accommodate such run-time behaviour requires sophisticated techniques for software architecture and organisation (Stephanidis et al., 2001; Savidis and Stephanidis, 2001b).

2.4. The need for ambient interactions

The concept of ubiquitous computing reflects an infrastructure where users are typically engaged in mobile interaction sessions, within environments constituted of dynamically varying computational resources. In ambient interaction scenarios, the user carries a very small processing unit, e.g. the size of a credit card, with an embedded operating system and wireless networking, including short-range radio networking like BlueTooth. Additionally, the user may optionally collect any number of wearable wireless gadgets. Once the ‘on’ button of the processing unit is pressed, the system boots, and then seeks for in-range devices capable of hosting interaction. When such devices are detected, they are appropriately employed to support interaction. At some point, some devices get out-of-range (i.e. they are ‘lost’), and the system tries to use some other available devices to maintain interaction. If the available devices do not suffice for the current interaction purposes, dialogue is considered as ‘stalled’. When new devices become in-range (i.e. they are ‘discovered’), the system tries to engage those devices in interaction, either to resume dialogue from a stalled state, or to further optimize it by offering a better interaction alternative. This notion of mobile ambient interactions is illustrated in Fig. 2, which depicts two layers of dynamically engaged interaction-capable devices: (a) Inner layer wearable devices, which the user may or may not carry, depending on the situation, and which are not anticipated to vary ‘on-the-move’ as frequently as environment devices. (b) Outer layer ambient devices, i.e. the particular set of devices falling inside a wireless communication range with the mobile processing unit, which will normally vary according to the particular user location.
3. Analysis of requirements

The identified software engineering requirements reflect two key levels of implementation functionality, that can be either supported as it is by interface development tools or need to be manually crafted by interface developers: (a) required functionality, being mandatory to address the identified software requirements; and (b) recommended functionality, offered to optimally address the identified requirements.

3.1. Metaphor development

In the methodology proposed in this paper, the metaphor development process is split in three distinct phases (see Fig. 3): (i) design of the required metaphoric representation, which entails both the selection of suitable metaphoric entities and the definition of their computer equivalents in terms of presentation properties, interactive behaviors and relationships; (ii) realization of the interactive embodiment of the metaphor through the selection of media and modalities, interaction object classes and associated attributes; and (iii) implementation of a metaphor realization, through the provision of interface development software libraries, which comprise dialogue elements that comply with that particular metaphor realization.

It should be noted that we distinguish between metaphor realization and metaphor implementation to account for the fact that there may be many realizations of a real world metaphor and many implementations for a particular realization of a metaphor. For instance, a room can be realized visually as a graphical entity, as in (Card and Henderson, 1987), but also non-visual as in COMONKIT (Savidis and Stephanidis, 1995a) and AudioRooms (Mynatt and Edwards, 1995). Additionally, various implementations can be built for a particular metaphor realization. This is the case with the various existing window-based toolkits, e.g. OSF/Motif, MFC, InterViews, etc. which may differ with
respect to their software implementation and programming model. However, all windowing-toolkits implement a common set of dialogue techniques corresponding to the visual realization of the desktop metaphor. It follows, therefore, that such a distinction between metaphor realization and metaphor implementation is important, because it allows modifications to be introduced at a particular level without necessarily affecting the higher levels.

3.1.1. User-oriented design of metaphoric interaction

During the metaphor design and realisation stages, specific user attribute values need to be considered. Hence, the resulting metaphor design(-s) and realisation(-s) are directly associated to those user attribute values. One such representative example concerns the design and realisation of the desktop metaphor, which is currently reflected in all windowing interactive environments. The original design had considered the needs of an ‘average’ person working in an office and performing tasks primarily engaging information conveyed on paper. The resulting realization has been targeted towards sighted users, and has been based on the effective exploitation of the human-visual information processing capability. It is argued that both accessibility and usability problems may arise when trying to deploy interaction metaphors across user populations other than those originally considered during the metaphor design and realization stages. The following are two examples of cases that can be characterized as less-than-perfect metaphor use:

- **Windowing interaction for blind users.** This scenario is typically reflected in existing screen readers, aiming to provide access to windowing applications by blind users. In this case, visual realizations of the desktop metaphor are reproduced in a non-visual form. However, the metaphor realization is even closer to sighted user needs, than the metaphor design itself, since specific visual interaction means are considered. In conclusion, fundamental entities (e.g. windows, icons, visual cues) and relationships (e.g. overlapping, spatial arrangement) in the desktop metaphor require considerable further investigation in order to verify whether their reproduction in a non-visual form is meaningful.
• **Windowing interaction for children (preschool, early school).** Various software products for educational or entertainment purposes have been produced, targeted to children of the preschool or early school age, many of them working under popular windowing environments. Hence, the desktop metaphor is directly employed for interaction. However, some of the common properties of windowing environments, such as concurrency of input actions, multitasking (e.g. many applications), intuitive data exchange among applications (e.g. copy/paste), direct manipulation and direct activation (e.g. drag and drop), etc. are mainly directed towards business/office tasks in a working environment. This problem has been recognized at an early point, leading to a new generation of *edutainment* software products, demonstrating a large amount of custom-made metaphoric interaction strategies like cartoons and animation, story telling, and live characters.

3.1.2. **The key role of top-level containers**

Containers are those classes of interaction objects which may physically enclose arbitrary instances of interaction objects. In running interactive applications, those container object instances which are not enclosed within other containers are called top-level containers. For instance, windows providing interactive management facilities, which are not included within other windows, are called top-level windows. When designing metaphoric interaction, there can be many real-world analogies which are transferred in the interaction domain. Hence, practically, multiple distinct metaphors may be combined. For example, in windowing applications, the following interaction object classes are typically met, each representing a specific real-world analogy:

- **Windows**—*sheets of paper*.
- **Push buttons, sliders, potentiometers and gauges**—*electric devices*.
- **Check boxes**—*form filling*.
- **Menus**—*restaurant*.
- **Icons**—*visual signs*.

Naturally, the original real-world physical regulations are effectively broken when containment relationships are designed (e.g. none would expect to see an electric button on a sheet of paper in the real world, while push buttons are normally embedded within windows). The effect of such containment relationships is that interaction metaphors are embedded at various levels in interactive applications. Related work in the past has investigated the design aspects of embedded interaction metaphors (Carroll et al., 1988).

In the context of the universally accessible interactions, the focus has been on the identification of those interactive entities, which play a key role in providing the overall metaphoric nature of an interaction environment. It is likely that different interaction metaphors will have to be provided to diverse users in order to achieve accessible and high-quality interaction. If we are able to detect those entities the largely affect the overall metaphoric ‘look and feel’ of the interactive environment, then we may only need to provide different metaphoric representations for those entities, in order to derive alternative metaphoric environments. This potentially alleviates the overhead of designing from scratch.
alternative metaphoric artifacts. The resulting methodological framework to address the above issue is based on the following principle:

The overall interaction metaphor is characterized and primarily conveyed by the metaphoric properties of top-level containers, while all embedded interaction objects are physically projected within the interaction space offered by the top-level containers.

This principle is depicted in Fig. 4, where it clearly appears that embedded objects cannot alter the original characterization of the overall interaction metaphor. Finally, as the key property of metaphoric interaction is enabling end-users to quickly familiarize with interface artifacts, as the later are interactive realizations of carefully selected objects from the real world, more focus is needed in the selection of those objects for universal access. More specifically, it may be practically impossible to identify universally usable metaphoric entities. For example, it is likely that cultural or age differences may imply radically different real-life experiences, which may cause variant interpretations even for a single real-world artefact. Moreover, it is possible that some interaction artifacts may not be recognized by particular groups of people due to the lack of considerable real-life experience with their real-world counterparts.

3.1.3. A metaphor development case for accessibility

The practical applicability of this principle has been demonstrated within two specific research efforts: (a) the development of a non-visual toolkit called COMONKIT (Savidis and Stephanidis, 1995b), providing a single top-level container with Rooms-based interaction, and many standard interaction object classes like ‘menu’, push button’, etc.; and (b) the development of a non-visual toolkit called HAWK (Savidis et al., 1997a), providing a generic container object (capable of realising various metaphoric representations), as well as various conventional interaction objects classes (like in COMONKIT).

Testing the above principle in Rooms/COMONKIT quickly led to the need of: (i) providing further variations on non-visual presentation and feedback; and (ii) suppling alternative top-level metaphoric entities, like ‘books’, ‘desk’, and ‘library’, with genuine

---

Fig. 4. Some representative design scenarios of metaphoric elements, demonstrating how top-level containers largely affect the overall interactive experience of the resulting metaphoric environment.
non-visual realisations. This gave rise to the design and implementation of the HAWK toolkit, providing a generic container which may realise alternative metaphoric representations.

The generic container class in the HAWK toolkit provides various presentation and dialogue attributes through which alternative metaphoric non-visual styles can be derived, by appropriately combining messages and sound feedback: (a) synthesized speech message (speech message to be given when the user ‘focuses’ on the object); (b) Braille message (message displayed on Braille device when the user ‘focuses’ on object); (c) ‘on-entry’ digitized audio file (to be played when the user focuses the object); and (d) ‘on-exit’ digitized audio file (to be played when the user ‘leaves’ the object). Practically, an interaction object class is considered to allow alternative metaphoric styles, if it enables its different instantiations within the User Interface to be perceived by the end-user as realizations of particular different real-world objects. In the container object class of the HAWK toolkit, this is accomplished by supplying different values to the supported presentation attributes.

For instance, Fig. 5 depicts the assignment of specific values to the container presentation attributes in order to derive alternative metaphoric representations. Three container instances, realising ‘books’, ‘desk-top’ and ‘rooms’ metaphors respectively, are defined. In the case of non-visual interaction, it has been relatively easy to design such parameterised metaphoric representations, due to the simplicity of the output channels (i.e. audio, speech and Braille). This approach has been validated in the context of the ACCESS Project (ACCESS Project, 1996), both with respect to its usability as an engineering method, as well as with respect to the usability of the produced interfaces (Savidis et al., 1997a), while the HAWK toolkit has been used for the implementation of a non-visual electronic book (Petrie et al., 1997), and the non-visual component of a user-adaptable Web-browser (Stephanidis et al., 2001).

3.1.4. Advanced graphical metaphors

Apart from metaphors specifically designed for blind users, the introduction of radically new graphical interaction metaphors for sighted users has also been investigated. In this context, the experimentation target has been primarily twofold: (a) to study the usability of new highly-interactive and visually dynamic artifacts, going beyond the traditional windowing style; and (b) to analyze the development barriers inherent in the implementation of those demanding artifacts, as well as the potential to directly combine them with existing windowing implementation toolkits. In this context, two parallel design and implementation efforts have been carried out:

- The development of direct-manipulation immersive hierarchical information spaces. In this context, a 3D-space efficient method to render hierarchical structures has been designed, named *inverted umbrella trees*, as opposed to typical cone trees. The resulting interaction toolkit has been employed to construct a real-time animated 3D interactive file manager (see Fig. 6).
- The development of dynamic real-time animation-based effects for windows. In this framework, specific categories of real-life phenomena, e.g. fire, smoke, icing, etc. (see Fig. 7), have been simulated with an animation engine relying upon heuristic particle
Fig. 5. Conveying alternative metaphoric representations using digital audio effects and synthesized speech messages, through container instances for non-visual interaction object hierarchies using the HAWK development toolkit.
systems (i.e. computing simulations of particle systems like fire or smoke, via techniques that instead of employing the precise mathematical modeling, use simpler computation models to accomplish fast visualizations, with satisfactory results). Those effects have been designed to provide metaphoric feedback methods for application-specific events such as:

- Resource demanding computation (net, processor, disc);
- Illegal data access (potential security leak);

Fig. 6. The implementation of a 3D direct-manipulation navigator in hierarchical information spaces, reflecting the rendering of two key metaphoric representations: (a) the newly designed inverted umbrella trees (first four snapshots); and (b) traditional cone trees (last two snapshots).
Failure to complete a requested operation (compile errors, failed search, could not save file);
Application system crash;
Resource access errors (hard disc, net);
Virus infection;
Application that could not terminate (zombies);

Fig. 7. The implementation of real-time animation-based feedback-effects for windowing applications, reflecting the dynamic rendering using heuristic particle systems of three key metaphoric events: fire, smoke and progressive icing.
• Application halts (hanged) due to an error;
• Application put in stand-by mode by user control;
• Application idle for too long.

From the usability point of view, the early evaluation experiments indicated a high-degree of acceptance by end-users, especially from young people, who expressed particular satisfaction due to the presence of such appealing graphical artifacts within traditionally 2D rectangular windows. Additionally, within informal interviews, it became clear that the expectations of young people for interaction quality are today raised at a surprising level, mainly due to the tremendous progress on rendering and simulation quality of video games being widely popular to children, teenagers and young adults. Moreover, most of those users expressed the opinion that they considered the static rectangular appearance of windowing applications to be rather boring and obsolete, while they expressed the desire to see dynamic effects, like those met in typical video games, in typical computer applications.

Although the small-scale usability evaluation gave positive evidence for the introduction of more dynamic metaphoric phenomena within traditional interactive spaces, the software engineering conclusions are clearly less enthusiastic and encouraging. To implement the 3D rendering and navigation of hierarchical structures, OpenGL, a very popular and powerful portable software library for 3D graphics, has been employed. In this context, there was no way to inter-operate or mix with typical windowing interaction elements, in order to support the initial target of implementing the real-time navigation in 3D, while providing details and parameter editing with form-based interfaces projected in the 2D plane. In other words, while programming 3D worlds, one should either completely forget the 2D interaction environment, or pay the overhead of implementing it from scratch. Although it is possible to instantiate 2D windowing elements in 3D scenes (e.g. via OpenGL) or in 2D direct-screen access mode (e.g. via GDI API for Windows direct mode), those are, unfortunately, entirely separated from the normal-mode windowing environment and the rest of the running applications. Regarding the second experiment, although it concerned the implementation of 2D real-time animation effects, the display regulations and constraints of the windowing environment practically forbid the real-time implementation of dynamic effects within the normal windowing environment, especially when those effects cross the window display boundaries. In this case, it was only possible to implement dynamic animation effects with direct screen access mode, a mode that is not interoperable with normal windowing mode, treating windows merely as colored bitmaps. The overall implementation-oriented conclusion from this comprehensive experiment with radically new metaphors is that the present implementation regulations of windowing-style toolkits pose severe restrictions for the injection of graphics-intensive interactive metaphoric entities when the latter depart from the desktop metaphor. It is clear that the existing implementations are still very constraining, and do not yet consider openness and interoperability with potentially novel emerging interaction styles. Developers may only implement such artifacts in segregation, i.e. as specific closed-world applications, or otherwise pay the huge overhead of implementing from scratch a complete development framework for their metaphors, e.g. the analogous of a windowing toolkit.
3.2. Toolkit integration

Development tools are considered to support toolkit integration if they allow importing any particular toolkit, so that all interaction elements of the imported toolkit effectively become implementationally available. For instance, if an interface builder providing interactive graphical construction techniques supports toolkit integration, then it should be possible to use the original construction techniques to also manipulate the imported object classes. Toolkit integration does not assume any particular interface building method, and may be supported by tools with various methods for interface implementation, such as programming-based, interactive constructions, state-based, event-based, demonstration-based, fourth generation languages, etc.

An explicit distinction needs to be made between the toolkit integration requirement and the multi-platform capability of certain toolkits. In the latter case, a single toolkit is provided with multiple (hard-coded) implementations across different operating systems, available when the toolkit product is released, i.e. multi-platform toolkits such as XVT, ILOG Views, Qt, Zinc, ZooLib, Open Amulet, and JFC. In the former case, a tool is made open so that its users can take advantage of a well-documented functionality for connecting to arbitrary toolkits. The need for importing toolkits is evident in cases where the interaction elements originally supported by a 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 and Stephanidis, 1995b), where interfaces concurrently accessible by sighted and blind users are built, non-visual interaction techniques are required together with typical 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 and Stephanidis, 1995a), or HAWK (Savidis et al., 1997a) is necessary. In the more general case of universally accessible interactions, it can be argued that scenarios necessitating toolkit integration are likely to emerge, not only as a result of user diversity, but also as a consequence of proliferating interaction technologies and the requirement for portable and platform-independent user interface software.

When toolkit integration is supported, interface tools supply mechanisms to developers for importing third-party development toolkits. Currently, a very small number of interface tools supports this notion of platform connectivity, i.e. the capability to implementationally ‘connect’ to any particular target toolkit platform (e.g. connecting to OSF/Motif, MFC, Xt/Athena, GTK+, etc.), thus enabling developers to manipulate its interaction elements as if they were an integral part of the host interface tool. The first tool known to provide comprehensive support for toolkit integration was SERPENT (Bass et al., 1990), a User Interface Management System (UIMS, Myers, 1995), where the toolkit layer was termed lexical technology layer. The architectural approach developed in the SERPENT UIMS revealed key issues related to the programmatic interfacing of toolkits. Toolkit integration has also been supported by the HOMER UIMS (Savidis and Stephanidis, 1998), developed in order to facilitate the construction of Dual User Interfaces. The HOMER UIMS provided a powerful integration model, which is general enough to enable the integration of non-visual interaction libraries and of traditional visual windowing toolkits. More recently, the I-GET UIMS (Savidis and Stephanidis, 2001a), provided more
comprehensive toolkit integration facilities, enabling multiple toolkits to be imported and deployed concurrently through the I-GET programming language (Savidis, 2004). The MFC-toolkit integration and deployment process are detailed in (Hatziantoniou et al., 2003).

From the early introduction of toolkit integration facilities by the SERPENT UIMS, to more recent tools providing enhanced integration support like the I-GET UIMS, there has been little adoption and encapsulation of such facilities in commercially available instruments. Popular multi-platform toolkits like JFC (the derivative of Abstract Windowing Toolkit—AWT), available for nearly a decade, emphasized platform neutrality of a basic GUI toolkit. The single-platform toolkit MFC has been originally introduced 15 years ago, as the Windows object library, subject to numerous enhancements and re-implementations following severe OS updates and enhancements. Overall, the market has put little emphasis on universal access, while critical issues like accessibility have been poorly addressed through accessibility add-ons, rather than through the introduction of novel accessibility-oriented interface toolkits that could turn integration facilities to a necessary tool ingredient. In this context, in the industrial arena, the investments made on competing software toolkits like JFC and MFC were primarily targeted on mainstream capabilities, like component-ware support, multimedia elements, web embedding, etc. rather than on universal-access oriented facilities, such as toolkit integration.

3.2.1. Required functionality for toolkit integration

The required development tool properties for toolkit integration are intended to characterize a particular tool with respect to whether it supports some degree of openness, so that interaction elements from external (to the given interface tool) toolkits can be utilized (subject to some particular implementation restrictions). For a user interface tool to support toolkit integration, the required properties are twofold:

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

If the required properties are present, it is possible for the developer to import and combine software modules that utilize interaction elements from different toolkits. In particular, the properties are satisfied in those cases where the imported toolkit provides new styles of interaction elements. This is for instance the case when the interface tool is a programming-based library of windowing interaction elements and the target toolkit offers audio-processing functionality for auditory interaction. In such cases, potential conflicts can be easily resolved and elements from the two toolkits can be combined. In contrast to the above scenario, the required tool properties are currently not met when the imported toolkit supplies similar categories of interaction elements with the interface tool being used. For example, when trying to combine various libraries of graphical interaction elements delivered for the same programming language, various conflicts may arise, such as:
• Link conflicts at the binary library level, due to commonly named functions or global objects;
• Compiling conflicts, due to commonly named data structures, classes or namespaces;
• Execution conflicts, due to system-level conflicting configurations, conflicting access to shared resources or inability to combine in parallel the toolkit main loops.

3.2.2. Recommended functionality for toolkit integration

In this context, a development tool should offer additional means to effectively support toolkit integration. These features, constituting the comprehensive set of recommended tool properties for toolkit integration, are:

• Support for well-behaved (i.e. functionally robust and reliable), and well documented (i.e. developers should be able to predict functional behavior from comprehensive documentation resources) compilation and linking cycles for interfaces utilizing the imported toolkits; this applies to all types of interface building methods, not only to methods supported by programming-oriented tools;
• Possibility to adopt a single implementation model for all imported toolkits. Thus, when the development tool provides visual construction methods, the same facilities should allow manipulation of interface elements from all the integrated toolkits;
• Possibility to change aspects of the programming interface (i.e. the programmable ‘view’) of any of the imported toolkits. This would minimize the effort required for programmers to become familiar with the programming style of the newly imported toolkits;
• Effective resolution of the typical conflicts, e.g. linking, compilation and execution, arising from the combination of multiple style-similar toolkits.
• Importability of any type of interface toolkit, irrespective of the style of interaction supported (e.g. windowing toolkits, auditory/tactile toolkits, VR-based interaction toolkits);
• Toolkit interoperability, i.e. programmers should be enabled to combine toolkits for creating cross-toolkit object hierarchies.

Today, no single interface tool is reported in the literature that exhibits the above comprehensive set of recommended properties. Regarding the notion of a single programming interface, multi-platform toolkits already provide adequate support, by means of a single predefined programming layer. Only UIMS tools like SERPENT (Bass et al., 1990), HOMER (Savidis et al., 1995b) and I-GET (Savidis and Stephanidis, 2001a) supply adequate support for toolkit integration, by enabling the establishment of developer-defined programming interfaces on top of software toolkits. When a single programming interface is supported for all platforms, one important question concerns the ‘look and feel’ of supported interaction objects across those platforms. There are three alternative approaches in this case:

• Employing the native interaction controls of the platform, i.e. making direct use of the platform’s controls, along with their presentational and behavioral attributes (e.g. as in
Java’s early version of the AWT library, or multi-platform toolkits like XVT, Qt, Zinc, etc.;
- Mimicking the native controls of the platform, i.e. providing controls that are capable of altering their presentation and behavior to match the respective attributes of the target platform’s native controls (e.g. as in Java’s JFC Swing library);
- Providing custom interaction elements across all platforms, i.e. providing custom controls that ‘look’ and behave the same across platforms, independently of the platforms’ native controls (e.g. Tcl/Tk (Ousterhout, 1994)).

Regarding toolkit interoperability and mixed toolkit hierarchies (see Fig. 8), only the Fresco User Interface System (X Consortium, 1994) is known to support cross-toolkit hierarchies. In particular, Fresco facilitates the mixing of InterViews-originated objects with Motif-like widgets, relying upon the CORBA implementation of User Interface elements as distributed replicated objects. The latter is technically distinguished from multiple ‘look and feel’ support offered by a single toolkit, like JFC (by Sun Soft Corp.) or WDS (by Infinite Monkeys Inc.), as it allows runtime ‘look and feel’ mixing.

It should be noted that, even though elements from different toolkits may be combined, possibly employing a different ‘look and feel’, consistency is not necessarily compromised. Fig. 9 outlines a scenario depicting the combination of a windowing toolkit with a toolkit implementing the ‘book’ metaphor, exemplifying support for cross-toolkit object hierarchies at the implementation level. The advantages of mixing multiple toolkits are more evident in those cases where combined toolkits offer container objects with different metaphoric representations (like the example of Fig. 9), thus practically leading to the fusion of alternative metaphors. In this context, toolkit interoperability plays a critical software engineering role, defined technically as follows:

- To enable the construction of interfaces by combining together interaction objects from different toolkits, possibly complying with different metaphors that may impose diverse physical realizations. To enable embedded objects display and function inside the physical context of containers, through generic embedding and interoperation protocols.

Fig. 8. A hypothetical implementation scenario for cross-toolkit mixed interaction-object hierarchies, where object instances from WINDOWS, Motif, Xt/Athena and Xview are engaged.
The support for toolkit interoperability is a very demanding topic, not yet supported by User Interface toolkits, requiring development methods and techniques for the following issues:

- **Space negotiation protocols**, to allow embedded objects request and negotiate physical space provided by container objects, e.g.:
  
  `request space: 2D, graphic, rectangle[120×45]`
  
  `provide space: 2D, graphic, color[16 bits], rectangle[12, 45, 120, 90]`

- **Space representation geometry**, to allow embedded objects query the type of physical space and its relevant parameters offered by container objects, e.g.:
  
  `class: graphic, dimensions: 2, axis: yes[X: integer, Y: integer]`
  
  `class: auditory, dimensions: 0, axis: no, attributes: yes[volume: integer]`

- **Projection logic and constraints**, to allow embedded objects to define the way they can be projected in the context of container objects, e.g.:
  
  `transform: auditory→graphic`
  
  `transform: o1.id→o2.label`
  
  `transform: if (o1.audiofile ≠ empty) then [o2=button, o2.notify = [play(o1.audiofile)]]`

- **Semantic annotation and mapping regulations**, enabling embedded objects to encapsulate application-oriented semantic data, so that the transformation to a projected entity may reflect the encapsulated semantics, e.g.:
  
  `if (o1.semantic.role =(emphatic)) then o2.font = (typeface.Bold.12)`

The above issues are only representative, while the supplied examples outline roughly the need for protocols and micro-languages to allow specifying and implementing on-the-fly interoperation logic, independently from the particular representation method used in the examples. It is believed that interoperability is an issue deserving further attention, as it can

---

![Fig. 9. A hypothetical dialogue artifact for mixing container interaction objects of different software toolkits (i.e. ‘books’ and ‘windowing’) complying with different interaction metaphors.](image-url)
significantly boost the capability to implement new categories of artifacts through the open concurrent deployment of multiple toolkits.

3.3. Toolkit augmentation

Augmentation is defined as the design and implementation process through which additional interaction techniques are ‘injected’ into the original (native) interaction elements supplied by a particular toolkit, thus leading to improved accessibility or enhanced interaction quality for specific user categories. Newly introduced interaction techniques become an integral part of the existing interaction elements, while existing applications that make use of the toolkit ‘inherit’ the extra dialogue features, without requiring revisiting of their implementation (e.g. through re-compilation, or re-linking, in the case of programming-based implementation approaches).

The need for toolkit augmentation arises mainly from shortcomings or design deficiencies regarding the supplied interaction entities of existing user interface development toolkits. Since the majority of interactive software products are built by utilizing such toolkits, these shortcomings and deficiencies are propagated to the resulting applications. For instance, graphical interaction libraries do not support voice-controlled interaction. Thus, non-visual access to the interface, in a situation where direct visual attention is not possible (e.g. while driving) cannot be supported. Another typical example where augmentation is required is the case of accessibility of window-based interaction by motor-impaired users. In this case, additional interaction techniques are needed in order to enable a motor-impaired user to access the user interface through specialized input devices (e.g. binary switches). In both cases, augmentation implies the development of new interaction techniques, as well as the integration of (support for) special-purpose I/O devices (e.g. voice I/O hardware, binary switches).

In (Savidis et al., 1997b), the augmentation of the basic Windows object library to provide switch-based access, through automatic/manual hierarchical scanning of graphical elements for motor-impaired access, is discussed. One of the most important enhancements in this work has been the decomposition and augmentation of the user-dialogue for performing window management with just two switches. 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 Fig. 10); notice the alphabetic ordering of the letters, as opposed to the standard QWERTY keyboard, as hand-motor impaired users may have no experience with using the QWERTY layout (alternative predefined/custom layouts are also supported). Apart from top-level windows (e.g. FrameWindow class in the windows object library), augmented dialogues for the remaining object categories (e.g. button categories, container classes, composite objects, and text-entry objects) have been also designed and implemented. In (Dragicevic and Fekete, 2004), a flexible input configuration toolkit (ICon) is discussed which enables the visual programming of alternative input bindings for interactive elements, a system which can be considered as an appropriate input-augmentation instrument.
3.3.1. Required functionality for toolkit augmentation

The required user interface development tool properties for toolkit augmentation depict the set of functional capabilities that enable augmented object classes to be introduced into the original software libraries of a toolkit. The set of these properties is not bound to any particular programming language or category of programming languages (e.g. procedural, object-oriented, scripting). However, it is assumed that typical development functionality, such as introducing interaction object classes, accessing or modifying object attributes, and implementing callbacks or event handlers, is supported. The required facilities are:

- Provision of support for device integration. For example, when one considers input devices, this may require low-level software to be written and implemented either through a polling-based scheme (i.e. continuously checking device status), or through a notification-based scheme (i.e. device-level software may asynchronously send...
notifications when device input is detected). In either case, the newly introduced input device will have to be accompanied with the necessary application programming extensions, i.e. event class and low-level event generation;

- Provision of methods to manipulate the focus object (i.e. the object to which input from devices is redirected). During interactive episodes, different interaction objects normally gain and lose input focus via user control (e.g. through mouse or keyboard actions in windowing environments). In order to augment interaction, it is necessary to have programmatic control of the focus object, since any device input originating from the additional peripheral devices will need to be communicated to the current focus object.
- Programmable manipulation of the object hierarchy. When implementing augmented interaction, it is necessary to provide augmented analogies of the user’s control actions, as the user must be enabled to ‘navigate’ within the interface. Hence, the hierarchical structure of the interface objects must be accessible to the programmer when implementing the augmented navigation dialogue.

The relationship between the native object classes and the augmented object classes is a typical inheritance relationship, since augmented toolkit object classes inherit all the features of the respective original classes (see Fig. 11). This scheme can be implemented either via sub-classing, if the toolkit is provided in an object-oriented programming (OOP) framework, or via composition, in the case of non-OOP frameworks. In the latter case, composition is achieved through the definition of an augmented object structure that comprises the original object features (by directly encompassing an instance of the original object class), as well as the newly introduced augmented properties. Such an explicit instantiation is necessary to physically realize the toolkit object, which, in the case of sub-classing, would be automatically carried out. Additionally, the interface tool should provide facilities for specifying the mapping between the attributes of the toolkit object instance and its corresponding features defined as part of the new object structure (e.g. procedural programming, constraints, monitors). Currently, most programming toolkits available through an OOP language (like MFC, JFC, GKT+, Flash Action Script, etc) provide all the required support for the basic augmentation functionality. In contrast, scripting languages and fast interface

![Fig. 11. Relationship between original and augmented toolkit classes, in the context of the required functionality to support toolkit augmentation.](image-url)
prototyping instruments do not provide the necessary programming functionality to optimally cope with the implementation requirements of the augmented dialogue. We have also applied toolkit augmentation over Flash Action Script for the development of a web-based universally accessible chess game, which supports game session among blind, hand-motor impaired users and able users (available on-line: http://www.ics.forth.gr/uachess). In this game, the graphical interactive chessboard has been augmented with concurrent dynamic graphical and auditory output to support hierarchical scanning and non-visual interaction, which cannot be implemented through the simple static facilities of languages like X+V (the later being merely XHTML with voice output, see http://www-106.ibm.com/developerworks/library/wi-xvlanguage/).

3.3.2. Recommended functionality for toolkit augmentation
The comprehensive set of recommended tool properties for full support of toolkit augmentation introduces additional functional properties that need to be present in the development tool. These properties, primarily targeted towards enabling an easier and more modular implementation of the augmented object classes, are (see also Fig. 12):

- Support for the extension of object attributes and methods ‘on top’ of the original toolkit classes, i.e. without affecting their original programming interface. This alleviates the problem of defining new object classes, while enabling already existing applications to directly make use of the augmented dialogue features without modifications at the source code level;
- Provision of a syntactically visible (i.e. accessible to programming entities outside the scope of the object class) extensible constructor, where additional interactive behavior can be added. This allows installing new event-handlers and performing all the necessary initializations directly at the original class level. The notion of a constructor (in its object-oriented sense) may also be supported by non-programming oriented interface tools, by means of user-defined initialization scripts;
- Provision of a modular device installation/integration layer, so that new device input can be attached to the toolkit input-event level. This facilitates the management of additional peripheral devices, through the original event-management layer of the given interface development tool.

![Fig. 12. Toolkit augmentation to support the recommended functionality, reflecting the required capabilities for introducing dynamic object extensions.](image-url)
The most important advantage of a tool exhibiting the above mentioned comprehensive set of recommended properties for toolkit augmentation is the elimination of the need to introduce new specialized object classes encompassing the augmented interaction features. As a result, it is not necessary to make changes to existing applications. In comparison, when only the required properties are supported, newly defined object classes, not originally deployed within previously developed applications, convey the augmented capabilities. If developers desire to introduce some additional dialogue control logic within existing applications, so as to take advantage of the augmented object attributes and methods, modifications, recompilation and re-linking are evidently required in all cases. Even if no changes are needed in existing applications, recompilation or re-linking may still be necessary (especially in the case of purely compiled languages and static linkage to the toolkit libraries).

3.4. Toolkit expansion

Expansion over a particular toolkit is defined as the process through which toolkit users (i.e. user interface developers) introduce new interaction objects, not originally supported by that toolkit. An important requirement of toolkit expansion is that all newly introduced interaction objects are made available, in terms of the manipulation facilities offered, in exactly the same manner as original interaction objects—in effect rendering ‘add-on’ objects indistinguishable from original objects from the developer’s point of view.

A typical case of toolkit expansion is the introduction of new interaction object classes, built ‘on-top’ of existing interaction facilities, to embody an alternative interactive metaphor, such as the case of a book-like metaphor in (Moll-Carrillo et al., 1995). Toolkit expansion may also be necessitated by domain-specific or application-specific functionality that needs to be presented as a separate self-contained interactive entity (for example, a temperature-pressure graphical interaction object, to be employed in the implementation of a factory process control system, used for temperature and pressure visualization and control).

One of the early toolkits providing expansion support was the generic Xt toolkit, built on top of the Xlib library for the 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 complex enough to turn expansion to an expert’s programming task. Other approaches to expansion concern toolkit frameworks supported by 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 a straightforward sub-classing task. This is the typical case with OOP toolkits like the MS Windows object library or InterViews.

Apart from user interface programming toolkits, the expansion mechanism is also supported in some higher-level development tools, such as UIMSs. In Peridot (Myers, 1988), the demonstration-based method for defining interactive behaviors, leads to the introduction of interaction objects which can be subsequently recalled and employed in interface construction. This capability can be viewed as expansion functionality. The Microsoft Visual Basic development environment for interface construction and behavior scripting is currently supported with various peripheral tools from third-party vendors.
One such tool, called VBXpress, introduces expansion capabilities by supporting the interactive construction of new VBX interaction controls.

Finally, a more advanced approach to toolkit expansion concerns distributed object technologies for interoperability and component-based development. New interaction objects can be introduced through the utilization of a particular tool, while being employed 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. Microsoft has been the first to allow ActiveX controls to be embedded in JavaBeans containers. JavaSoft’s Migration Assistant accomplishes exactly the opposite (or symmetric) link, thus enabling JavaBeans to work inside ActiveX containers. The result is that today there is software enabling the inter-operation of ActiveX and JavaBeans components in both directions. For programmers using one of these component categories, this capability is an expansion of the set of available interaction controls. For instance, ActiveX programmers are enabled to use directly JavaBeans objects within ActiveX containers.

3.4.1. Required tool properties for toolkit expansion

The required development tool properties relate to the presence of an appropriate object expansion framework supported by the user interface development tool. The most typical forms of such an expandable object framework are:

- **Super class.** Expansion is achieved by taking advantage of the inheritance mechanism in OOP languages, while expanded objects are defined as classes derived from existing interaction object classes. Examples of such an approach are the MS Windows object library and the InterViews toolkit.

- **Template structure.** In this case, an object implementation framework is provided, requiring developers to fill-in appropriate implementation ‘gaps’ (i.e. supply code), mainly relevant to dialogue properties, such as visual attributes, display structure and event handling. The most representative example of this approach is the Xlib/Xt widget expansion model of the X Windowing System. The JavaBeans approach is a more advanced version of an object implementation framework.

- **Application Programming Interface (API).** In this case, resource manipulation and event propagation corresponds to services and event notifications, realizing object management APIs that are built on top of standardized communication protocols. This approach is usually blended with an object-oriented implementation framework, providing a way to combine objects irrespective of their binary format, thus achieving open, component-based development. The ActiveX model is the most typical example. The OMG CORBA model, though not providing a standardized API, allows customized APIs to be built, e.g. the Fresco User Interface System (X Consortium, 1994).

- **Physical pattern.** In this case, newly introduced object classes are built via interactive construction methods. For example construction could start from basic physical structures (e.g. rectangular regions), adding various physical attributes (e.g. textual items, colors, borders, icons), defining logical event categories (e.g. ‘selected’), and implementing behavior via event handlers (e.g. highlighting on gaining focus, returning to normal state upon losing focus). The way in which physical patterns are supported
varies depending on the tool. For instance, Microsoft Visual Basic provides an ‘exhaustive’ definition and scripting approach, while Peridot (Myers, 1988) offers a demonstration-based approach.

- **4GL model.** Fourth-generation interface development languages support the combination of their interaction object model with the dialogue construction methods, allowing new object classes to be built. These dialogue implementation methods are to be utilized for implementing the interactive behavior of the new objects. The I-GET UIMS is a typical example of an interface tool supporting a 4GL expansion model (Savidis et al., 2001a).

### 3.4.2. Recommended functionality for toolkit expansion

The comprehensive set of development tool properties for toolkit expansion includes one additional recommended functional property, and namely:

- **Closure:** if an interface tool is to fully support object expansion, then it should allow developers to implement the dialogue for new interaction objects via its native dialogue construction facilities (see Fig. 13).

In other words, developers should be allowed to define dialogues for new interaction objects via the facilities they have already been using for implementing conventional interfaces. For instance, in an interface builder, the full functionality for expansion is available only when interactive object design and implementation is facilitated.

### 3.5. Toolkit abstraction

Toolkit abstraction is defined as the ability of the interface tool to support manipulation of interaction objects, which are entirely decoupled from physical interaction properties. Abstract interaction objects are high-level interactive entities reflecting generic behavioral properties with no input syntax, interaction dialogue, and physical structure. However,

![Diagram](image)

Fig. 13. **Closure** property in maximally supported expansion: the resulting expanded objects are constructed through the original dialogue implementation facilities, made indistinguishable, under a development perspective, from the native object classes.
during execution, abstract interaction objects are automatically mapped to physical interaction objects of the employed toolkit. An example of an abstract interaction object (namely a selector) is provided in Fig. 14. Such an abstract interaction object has only two properties: the number of options and the selection (as an index) made by the user. Additionally, it may encompass various other programming attributes, such as a callback list (i.e. reflecting the select method), and a Boolean variable to distinguish among multiple-choice and single-choice logical behaviors.

As illustrated in Fig. 14, multiple physical interaction styles, possibly corresponding to different interaction metaphors, may be defined as physical instantiations of the abstract selector object class. When designing and implementing interfaces for diverse user groups, even though considerable structural and behavioral differences are naturally expected, it is still possible to capture various commonalities in interaction syntax, by analyzing 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 at various levels of abstraction, in order to support modification only at a single level, i.e. the abstract level. Such a scenario requires implementation support for: (a) organizing interaction objects at various levels of abstraction; (b) enabling developers to define the way in which abstract objects may be mapped (i.e. physically instantiated) to appropriate physical artifacts; and, (c) providing the means to construct dialogues composed of abstract interface objects. Abstract interaction objects can be employed for the design and implementation of generic reusable dialogue components that do not reflect physical interaction properties at development-time. In this sense, such dialogue patterns are not restricted to any particular user group or interaction style. The introduction of the intermediate physical instantiation levels is also required, so that abstract forms can be mapped to concrete physical structures. By automating such an instantiation mechanism, development for diverse target user group is facilitated at an abstract layer, while the physical realization is automated on the basis of an appropriate object instantiation mechanism.

The notion of abstraction has gained increasing interest in software engineering as a solution towards recurring development problems. The basic idea behind abstraction is the establishment of software frameworks that clearly separate the implementation layers relevant only to the general problem class, from the specific software engineering issues that emerge when the problem class is met with alternative instantiations. The same approach applies to the development of interactive systems, in order to allow a dialogue structure composed of abstract objects to be re-targeted to various alternative physical forms through an automation process configured and controlled by the developer.

Currently, there are various design models, in certain cases accompanied with incomplete suggested design patterns, as to what actually constitutes abstract interaction objects and their particular software properties. Past work in the context of abstract interaction objects relates to abstract interaction elements and model-based interaction design (Blattner et al., 1992; Foley et al., 1988; Duke et al., 1993; Wise and Glinert, 1995; Puerta, 1997; Savidis and Stephanidis, 1998) reflecting the need to define appropriate programming versions relieved from physical interaction properties such as colour, font size, border, or audio feedback, and only reflecting an abstract behavioural role, i.e. why an object is needed. This definition makes a clear distinction of abstract interaction objects from multi-platform
Fig. 14. Alternative instantiations of an abstract Selector varying with respect to topology, display medium, content of options, input devices, and appearance attributes.
interaction objects, the latter merely forming generalisations of similar graphical interaction objects met in different toolkits, through standardised APIs.

3.5.1. Required functionality for toolkit abstraction

The required development functionality for toolkit abstraction is target towards facilitating interface construction based on abstract objects. Additionally, some high-level implementation issues reveal the complexity of explicitly programming abstract objects if the interface development tool does not support them inherently. The required functionality for toolkit abstraction is:

- **Closed set of abstractions**, i.e. a predefined collection of abstract interaction object classes is provided;
- **Bounded polymorphism**, i.e. for each abstract object class \( C \), a predefined mapping scheme \( S_C \) is supported, for the runtime binding of abstract instances to physical instances, the latter belonging to a predefined list of alternative physical object classes \( P_1, \ldots, P_n(C) \);
- **Controllable instantiation**, for each abstract object instance \( I \) of a class \( C \), it is possible to select at development-time the specific physical class \( P_j \in S_C \) to which instance \( I \) will be bind at runtime;

The above properties enable the developer to instantiate abstract objects while having control over the physical mapping schemes that will be active for each abstract object instance. Mapping schemes define the candidate classes for physically realizing an abstract object class.

An approach to implement the software structure accommodating the required functionality for abstraction is provided in Fig. 15. As it is shown, abstract interaction objects reflect concrete program classes, which delegate their physical instantiation as concrete physical object classes to a respective mapping scheme class. The key point to this design is the mapping scheme class, which bridges classes of abstract interface objects with classes of physical interface objects, while also preserving the independence among the abstract and physical interface object classes. Abstract objects upon instantiation never directly instantiate physical object classes, but instead request their mapping scheme instance object to perform physical instantiation (through the \textit{Create} function). The interface programmer may extract or even modify the mapping scheme instance of an abstract object, and may alter the physical instance of its associated abstract object (i.e. by calling \textit{Destroy} followed by a \textit{Create} with the desirable physical object class name).

3.5.2. Recommended functionality for toolkit abstraction

The recommended functionality introduced below can be used to judge whether interface tools provide powerful methods for manipulating abstractions, such as defining, instantiating, polymorphosing, and extending abstract interaction object classes. Support for such facilities entails the following:

- **Open abstraction set**, i.e. facilities to define new abstract interaction object classes;
- **Open polymorphism**, i.e. methods to define alternative schemes for mapping abstract
Fig. 15. The software programming structure to implement the required functionality for abstraction, in an OOP language, enabling abstract objects to on-the-fly retarget to alternative mapping schemes as well as to alternative physical object instances.
object classes to physical object classes, so that, for example, an abstract ‘selector’ may be mapped to a visual ‘column menu’ and a non-visual ‘list-box’;

- **Physical mapping logic**, i.e. facilities for defining run-time relationships between an abstract instance and its various concurrent physical instances. This may require the definition of attribute dependencies among the physicals and abstract instances, together with propagation of call-back notifications, e.g. if a logical event occurs in the context of a physical instance its associated abstract instance must be appropriately notified;

- **Physical instance resolution**: when an abstract instance $I$ of class $C$ is employed in interface implementation, syntactic access to all plausible physical instances of classes $P_j \in S_C$ should be facilitated.

Currently, the recommended functionality can be normally accommodated in general-purpose object-oriented programming (OOP) language like C++ or Java, requiring demanding software patterns to be manually programmed by interface developers. Additionally, the I-GET language, (see Savidis, 2004, chapter 10), provides genuine support for the specification of abstract interaction objects, with polymorphic instantiation relationships and multiple physical mapping schemes, while facilitating controllable instantiation and syntactical resolution of the physical instance.

### 3.6. Automatic interface adaptation

In order to accomplish the runtime delivery of user- and usage-context- adapted User Interfaces, developers need to implement ways of manipulating during runtime alternative dialogue components. In this context, the proposed functionality is not distinguished into required or recommended, as with previously discussed software engineering requirements for handling interaction objects. Instead, a comprehensive set of functional requirements is defined, to support the adaptation-oriented manipulation of dialogue components. These requirements are described below.

#### 3.6.1. Dialogue component model and dynamic interface assembly

This requirement reflects the necessity to provide interface developers with a genuine component-mode, so as to support a straightforward mapping from the design domain of dialogue design patterns to the implementation domain of fully working dialogue components. Additionally, the effective run-time manipulation of dialogue components requires facilities for dynamic component instantiation and destruction, in an imperative or declarative manner. In this context, *imperative* means that developers add instantiation or destruction statements as part of a typical program control flow (i.e. via statements or calling conventions). *Declarative* means that the instantiation or destruction events are associated to declarative constructs, such as precondition-based activations or notification handlers. Normally, instantiation or destruction of components will be ‘coded’ by developers in those points within the implementation that certain conditions dictating those events are satisfied. For this purpose, the declarative approach offers the significant advantage of relieving developers from the burden of algorithmically and continuously testing those conditions during execution, for each component class. Normally, in general-purpose programming-based toolkits the delivery of those facilities is mostly trivial, however, in specialized
interface development instruments (e.g. task-based development or model-based development) the software engineering methods offered for component manipulation are less powerful.

The software organization of components should reflect the hierarchical task-oriented discipline of the interface design. This implies that some components may be dependent on the presence of other, hierarchically higher, dialogue components. This reflects the need to make the interface context for particular sub-tasks available (to end-users of the interface), if and only if the interface context for ancestor tasks is already available. For instance, the ‘Save file as’ dialogue-box may appear only if the ‘Editing file’ interface is already available to the user.

Additionally, it is critical to support for orthogonal expansion of interface components. More specifically, when adding new dialogue components, or even interaction monitoring components, the overall implementation structure should encapsulate the appropriate placeholders to accommodate such component extensions. Finally, the activation of components should be orchestrated to take place on the fly, reflecting the genuinely runtime decision for the end-user bets-fit dialogue components. In this context, the organization structure of the User Interface should not reflect a particular hard-coded interface instance, but has to effectively accommodate the dynamic process of hierarchical interface assembly and delivery from runtime chosen components. An appropriate way to address such implementation requirements is parametric polymorphic containment hierarchies, i.e. container hierarchies in which: (a) alternative possible decompositions may be defined for a single container object instance; and (b) in each such decomposition, every constituent component may be potentially met with different alternative incarnations.

3.6.2. Parametric polymorphic containment hierarchies

In the context of the AVANTI user-adapted web browser (Stephanidis et al., 2001), it has been necessary to support physical dialogue components for which the contained items could vary ‘on-the-fly’, since alternative designed and implemented versions of such embedded components had to be supported (see Fig. 16). This functional feature required the software engineering of container components to support effectively such dynamic containment, through methods of parametrization and abstract Application Programming Interfaces (APIs), i.e. polymorphism. Some similarities with dynamic interface assembly can be found in typical web-based applications delivering dynamic content. The software engineering methods employed in such cases are based on the construction of application templates (technologies such as Active Server Pages by Microsoft—ASP or Java Server Pages—JSP by JavaSoft, are usually employed), with embedded queries for dynamic information retrieval, delivering to the user a web-page assembled on-the-fly. In this case, there are no alternative embedded components, just content to be dynamically retrieved, while the web-page assembly technique is mandatory when HTML-based web pages are to be delivered to the end-user (in HTML, each time the content changes, a different HTML page has to be written). However, in case a full-fledged embedded component is developed (e.g. as an ActiveX object or Java Applet), no run-time assembly is required, since the embedded application internally manages content extraction and display, as a common desktop information retrieval application.
Fig. 16. Parametric polymorphic containment with variant constituent components in the AVANTI browser. The indication ‘Empty’ signifies components whose presence may have to be omitted upon dynamic interface delivery for certain user categories.
The software implementation is organised in hierarchically structured software templates, in which the key place-holders are parameterised container components. This hierarchical organisation mirrors the fundamentally hierarchical constructional nature of interfaces. The ability to diversify and support alternatives in this hierarchy is due to containment parameterisation, while the adapted assembly process is realised by selective activation, engaging remote decision making on the basis of end-user and usage-context information.

In Fig. 17, the concept of parametric container hierarchies is illustrated. Container classes expose their containment capabilities and the type of supported contained objects by defining abstract interfaces (i.e. abstract OOP classes) for all the contained component classes. These interfaces, defined by container class developers, constitute the programming contract between the container and the contained classes. In this manner, alternative derived contained-component classes may be instantiated at run-time as constituent elements of a container. Following the definition of polymorphic factor $PL$, which provides a practical metric of the number of possible alternative run-time configurations of a component, the $PL$ of the top level application component gives the number of all possible alternative dynamically assembled interface instances (see also Fig. 17). Notably, this does not reflect the total number of legal interface instances, as the combination of alternatives is not freely supported, but provides significant evidence of the potential polymorphic factor of such a hierarchical adaptation-oriented interface structure.

3.6.3. Dynamically controlled interaction monitoring

To support adaptive interface behavior, the run-time collection and analysis of interaction monitoring information is required. This approach, which has been traditionally employed in adaptive interface research (Dieterich et al., 1993), has been also implemented in the context of the AVANTI browser. To achieve dynamically controlled interaction monitoring, all dialogue components need to expose (i.e. implement) a common programming interface (i.e. abstract class), for installing or un-installing monitoring functionality (mainly event handlers). From a software engineering point of view, the effective management and control of interaction monitoring requires the careful design of standard programming APIs, for all dialogue components, as well as the separation of the interaction monitoring logic, from the typical dialogue control logic of each component. This will enable the runtime orchestration of interaction monitoring, so as to collect interaction events and collate an interaction history, the latter constituting the basis to draw inferences regarding the particular end-user.

3.6.4. User profiles and decision making

In automatically adapted interactions, the storage of a user-profile is mandatory, necessitating the employment of appropriate user-model representation methods. Additionally, the runtime necessity for adaptation-oriented decision-making, i.e. deciding on the fly when and how adaptation is to be performed, requires appropriate decision-logic representation methods. Various relevant technical approaches are discussed in (Kobsa and Pohl, 1995), (Vergara, 1994), (Savidis and Stephanidis, 2001b).
Fig. 17. The notion of dynamic polymorphic hierarchical containment in automatically adapted interactions, to cater for the runtime interface assembly process.

$PL(A) = PL(A1) * PL(A2) * PL(A3)$

$PL(A1) = 3$

$PL(A2) = ...$

$PL(A3) = ...$

$PL(A2) = PL(A21) * PL(A22)$

$PL(A21) = 4$

$PL(A22) = ...$

$PL(A3) = PL(A31) * PL(A32)$

$PL(A31) = 2$

$PL(A32) = 5$
3.7. Ambient interactions

The primary motivation for ambient interactions is based on the idea that future computing platforms will not constitute monolithic ‘all-power-in-one’ and ‘all interface in one’ devices, but will likely support open interconnectivity enabling users to combine on the fly the facilities offered by distinct devices. Physically distributed devices may be either wearable or available within the ambient infrastructure (either stationary or mobile), and may be connected via a wireless communication link for easier deployment. Operationally, each such device will play a specific role by exposing different processing capabilities or functions, such as character display, pointing, graphics, audio playback, speech synthesis, storage, network access, etc. From a hardware point of view, such devices may be wristwatches, earphones, public displays, home appliances, office equipment, car electronics, sunglasses, ATMs, etc. To allow the dynamic engagement and coordination of such computing devices, a central management and control point is needed, which should be reconfigured, adapted and fine-tuned by the end-user. Small portable processing units with embedded client applications, supporting on the fly device employment, are a particularly promising infrastructure for such dynamically formed ambient computing clusters.

The applications running on such portable machines should be state safe regarding failures or disconnections of externally utilized devices, while simultaneously offering comprehensive facilities to the end-user for the management of alternative device-composition configurations. The technical challenges for service-oriented composition depend on whether it concerns internal processing services or User Interface elements. In this context, the reported work addresses the issue of dynamic composition from User Interface micro-services hosted by dynamically engaged devices. Some of the foreseen key application domains, which would largely benefit from this approach, are infomobility and navigation, intelligent office environments, smart homes, and mobile entertainment. The specific functional requirements for ambient interactions, relying upon the experience acquired in the development of the Voyager development framework for ambient interactions (Savidis and Stephanidis, 2003a,b), in the context of the 2WEAR Project (see acknowledgements), are:

- **Device discovery and wireless networking.** Even though this requirement might seem as mostly related to core systems’ developments, it is imperative that interface developers manage the ‘on-the-fly’ detection of any in-range environment I/O devices that can be used for interaction purposes, while at the same time they should also be supplied with all the necessary instrumentation for handling wireless short-range dynamic communication links (e.g. the Bluetooth L2CAP library).
- **Device-embedded User Interface micro-services.** It is necessary to implement the runtime query of interaction-specific device capabilities (e.g. text display support, supported number of text lines, presence of a software cursor, etc.). This feature implies the provision of well-documented standardized service models for dynamically available remote UI devices, along with the definition and implementation of the concrete protocols for run-time control and coordination.
Automatic and on-demand dialogue reconfiguration. To cope with the dynamic presence or disengagement of remote I/O devices, the detection of loss of connection through typical network programming libraries is needed. Additionally, it is important to optionally allow end-users to dynamically re-configure the ambient interface, in case they have the knowledge and skills to do so effectively, offering the on-demand deployment of alternative interaction-capable devices from the local environment infrastructure. Finally, the support for predefined re-configuration scenarios for the automatic retargeting of the devices exploited by the interface is critical to allow automatic dialogue reconfiguration when, during interaction, particular I/O resources get out of wireless communication range or fail.

State persistence and abstract interaction objects. The key characteristic of ambient interactions is the inherent remote distribution of User Interface I/O micro-services within the surrounding computational environment. Such I/O resources may support a range of facilities, such as character input, text display, picture display, audio output, hardware push buttons, or on/off switches, etc. Since failure and loss of connection may take place at any time, it is important to ensure that the dialogue state is centrally maintained within the mobile interface application kernel. Arguably, the most appropriate way to program such a behavior is via abstract interaction objects (Desoi et al., 1989; Duke and Harisson, 1993; Savidis and Stephanidis, 1995b; Wise and Glinert, 1995).

An example of an application with an ambient interface is the Break Out ambient game (Savidis and Stephanidis, 2004), shown in Fig. 18. An in-depth technical analysis of the previously mentioned functional requirements, together with detailed design propositions for software library API and runtime architecture may be found in (Savidis and Stephanidis, 2002b), while the software-design evaluation process and results are reported in (Savidis and Stephanidis, 2003b). Additional information may also be found at the 2WEAR Project web site http://2wear.ics.forth.gr/.

![iPAQ™ Linux GTK+ Text Display](image1.png) ![MASC Wristwatch Text Display](image2.png)

Fig. 18. Two of the alternative output configurations of the pervasive Break Out game-board display; on the left it displays on a palm device, while on the right it is displayed on a h/w prototype of an ‘I/O open’ wristwatch device (all communication takes place over Bluetooth).
4. Discussion and conclusions

Interaction objects play a central role in interface development. Consequently, a large part of the software for implementing commercially available interface tools is dedicated to the provision of comprehensive collections of graphical interaction objects. The basic layer providing the implementation of interaction objects is the toolkit layer, while interface tools typically provide additional layers on top of that. We have studied interaction-object based interface development under the perspective of universally accessible interactions, identifying four key categories of software functionality to effectively manipulate interaction objects, namely integration, augmentation, expansion and abstraction. Currently, there are variable degrees of support for the investigated basic mechanisms. Regarding toolkit integration, the vast majority of commercial tools offer 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 overhead required, as well as the inherent implementation complexity, turns the expansion task into an activity primarily targeted to expert programmers. Regarding higher-level development tools, there is an increasing number of commercially available graphical construction tools supporting expansion, while there are currently only two 4GL-based interface tools supporting expansion. Finally, toolkit abstraction, although it is the most important mechanism in terms to cope with universal access, is also the least supported mechanism in existing interface tools, requiring primarily hand-made solutions with demanding software structures. The I-GET 4GL-based UIMS (Savidis and Stephanidis, 2001a) offering the I-GET pattern-reflecting language for User Interface programming with explicit support for virtual interaction objects (Savidis, 2004, chapter 10, available on-line), is a development tool which has been specifically developed to exhibit the full set of properties required for supporting abstraction.

Software toolkits, as collections of implemented User Interface elements, typically reflect the design of particular interaction metaphors, such as the desktop/windowing metaphor. In this context, top-level container objects play a primary role since they largely affect the overall metaphoric experience. In this sense, the pursuit of new styles and forms of metaphoric interaction can be mainly focused on the design of appropriate novel container objects, as it has been demonstrated in previous work, such as the generic containers of the Hawk toolkit (see Fig. 5), and the inverted umbrella trees of the immersive file manager (see Fig. 6). The ability to hierarchically mix interaction object instances from different toolkits opens new opportunities for deployment and experimentation with mixed-toolkit artifacts. However, the technical ground for such open interoperability is not yet prepared, requiring software-intensive research efforts to design, implement, test and deploy open toolkit interoperability infrastructures.

The need for User Interface adaptation in universally accessible interactions requires methods to accommodate dynamic user-adapted interface behavior. To implement automatic adaptation, interactive software applications should encompass the capability to appropriately deliver ‘on-the-fly’ adapted interface instances, performing appropriate runtime processing that engages the selection and activation of distinct implemented dialogue
components. The runtime assembly process of an interface requires an appropriate structure, which accommodates variability of contained components within container components, exhibiting the properties of dynamic polymorphic containment hierarchies. Additionally, interaction monitoring is necessary in order to collect and analyze interaction history so that user-oriented preferences can be drawn.

Finally, the recently introduced notion of ambient interaction reflects various alternative scenarios and technical solutions, all complementarily contributing towards the goal of smoothly and transparently integrating computing artifacts within the physical environment. Ambient interaction refers to interaction hosted by environment devices during user mobility. More specifically, in ambient interactions users carry miniaturized pocketsize processing unit (PPU), without any I/O modules, encompassing various user-chosen embedded software applications. Those applications should be capable of dynamically detecting the presence of devices within the surrounding environment, and of using them on the fly so as to realize meaningful interaction with the user. To effectively implement such a scenario, a number of key technical prerequisites emerge: (a) presence of wireless short-range networking technology (like BlueTooth); (b) ability to discover the presence of ambient devices on the move; (c) standard communication protocols for querying and controlling the services of ambient devices; (d) programmer’s APIs for dynamic User-Interface micro-service utilization; and (e) ability for dynamic interface re-configuration to the changing computational environment, while ensuring interface state persistence.

The development of universally accessible interactions entails software engineering requirements so as to address the grand development challenges inherent in the following four fundamental layers (see Fig. 19), directly relating to the three key layers of diversity in universally accessible interactions (see Fig. 1, interaction technologies concern both metaphors and platforms):

- **Metaphor layer**, reflecting the primary metaphoric styles and forms of the overall interactive experience, having as key challenge the development of new metaphors.

![Fig. 19. The four fundamental layers in universally accessible interactions, with their associated development challenges, constituting the basis of the software engineering requirements analysis.](image-url)
• **Platform layer**, reflecting the delivery of metaphors as software libraries encompassing collections of implemented interaction object classes, having as key challenge the effective manipulation of interaction objects.

• **User layer**, reflecting the individual requirements, characteristics, abilities and preferences, having as key challenge the support for user-adapted interface delivery.

• **Ambient layer**, reflecting the surrounding environment as an infrastructure of dynamically engaged computational resources, having as key challenge the provision of ambient interactions.

In this context, this paper has presented the key software engineering requirements, reflecting hands-on development experience in the course of six projects, in a timeframe of fifteen years. Overall, the development of universally accessible interactions is a complex task, since it engages a highly demanding interface design process, together with a programming-intensive software implementation process. Surprisingly, at present all the identified software engineering requirements can be better addressed through lower-level programming instruments, since commercially available higher-level interface construction tools are mostly optimized for typical windowing User Interfaces. Currently, there are various development propositions or software engineering recipes for many perspectives of ambient interactions or user-adapted interfaces, which may help in the reduction of the overall development complexity. However, in order to support the wide proliferation of universally accessible interactions as a common interface paradigm for future applications and services, further work is needed in the domain of User Interface development tools, in order to encompass automation mechanisms optimally suited for inclusive development.

**Acknowledgements**

Part of this work has been carried out in the context of the:

• ACCESS TP1001 (Development platform for unified ACCESS to enabling environments) project, partially funded by the TIDE Programme of the European Commission, and lasted 36 months (January the 1st, 1994 to December the 31, 1996). The partners of the ACCESS consortium are: CNR-IROE (Italy)—Prime contractor; ICS-FORTH (Greece); University of Hertfordshire (United Kingdom); University of Athens (Greece); NAWH (Finland); VTT (Finland); Hereward College (United Kingdom); RNIB (United Kingdom); Seleco (Italy); MA Systems and Control (United Kingdom); PIKOMED (Finland).

• AVANTI AC042 (Adaptable and Adaptive Interaction in Multimedia Telecommunications Applications) project, partially funded by the ACTS Program of the European Commission, and lasted 36 months (September the 1st, 1995 to August the 31, 1998). The partners of the AVANTI consortium are: ALCATEL Italia, Siete division (Italy)—Prime Contractor; IROE-CNR (Italy); ICS-FORTH (Greece); GMD (Germany), VTT (Finland); University of Siena (Italy), MA Systems and Control (UK); ECG (Italy); MATHHEMA (Italy); University of Linz (Austria); EUROGICIEL (France); TELECOM (Italy); TECO (Italy); ADR Study (Italy).
• 2WEAR - IST-2000-25286 (A Runtime for Adaptive and Extensible Wireless Wearables) project, partially funded by the Disappearing Computing Initiative (DCI) of the IST programme of the European Commission, and lasted 36 months (January the 1st, 2001 to December the 31, 2003). The partners of the 2WEAR consortium are: Foundation for Research and Technology—Hellas, Institute of Computer Science (ICS-FORTH), Greece (Prime Contractor); Nokia Research Centre (NRC), Finland; Swiss Federal Institute of Technology—Zurich Institute for Computer Systems (ETHZ), Switzerland; MA Systems and Control Limited, UK.

References


