

Supporting Virtual Interaction Objects with Polymorphic Platform Bindings in a User Interface Programming Language

Anthony Savidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,
Vassilika Vouton, GR-71110, Heraklion, Crete, Greece
as@ics.forth.gr

Abstract. Today, there are numerous software patterns for the software engineering of User Interfaces through interaction object classes that can be automatically retargeted to different graphical environments. Such methods are usually deployed in implementing multi-platform User Interface libraries, delivering Application Programming Interfaces (APIs) typically split in two layers: (a) the top layer, encompassing the platform independent programming elements available to client programmers; and (b) the bottom layer, delivering the platform specific bindings, implemented differently for each distinct graphical environment. While multi-platform interaction objects primarily constitute programming generalizations of graphical interaction elements, virtual interaction objects play the role of abstractions defined above any particular physical realization or dialogue metaphor. In this context, a sub-set of a User Interface programming language is presented, providing programming facilities for: (a) the definition of virtual interaction object classes; and (b) the specification of the mapping-logic to physically bind virtual object classes across different target platforms.

1 Introduction

The notion of abstraction has gained much attention in software engineering as a solution towards recurring development problems. The basic idea has been the establishment of software frameworks clearly separating those implementation layers relevant only to the nature of the problem, from the engineering issues, which emerge when the problem class is instantiated in practice in various different forms. The same philosophy, when applied to developing interactive systems, means employing abstractions for building dialogues so that a dialogue structure composed of abstract objects can be re-targeted to various alternative physical forms, through an automatic process controlled by the developer. In the context of User Interface development, interaction objects play a key role for implementing the constructional and behavioural aspects of interaction. In this context, numerous software libraries exist, such as MFC, JFC, GTK+, etc., offering comprehensive collections of object classes whose

instantiation by the running program effectively results in the interactive delivery of graphical interaction elements; such libraries are commonly known as interface toolkits. Currently, there are no similar libraries for abstract interaction objects, practically implying that their implementation and programming linkage to concrete interface toolkits has to be manually crafted by client programmers.

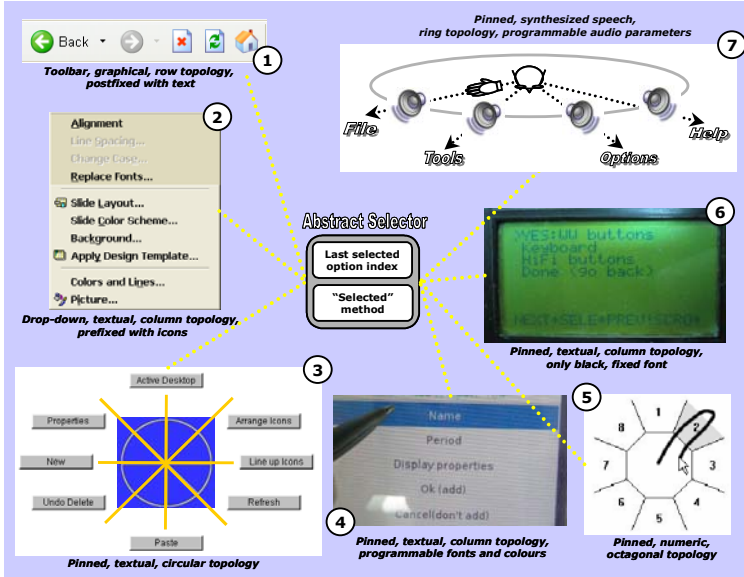


Fig. 1. Alternative incarnations of an abstract Selector varying with respect to topology, display medium, content of options, input devices, and appearance attributes. (1) and (2) are from WNT, (3) is from [3], (4) and (6) are from the 2WEAR Project [1], (5) is from [7], and (7) is from [8]

However, there are a few 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 [2,4,5,6,10] reflects 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 interaction objects, the latter merely forming generalisations of similar graphical interaction objects met in different toolkits, through standardised APIs. The software API of generalised objects is easily designed in a way offering fine-grained control to the physical aspects of interaction objects, since the target object classes exhibit very similar graphical properties. For instance, a multi-platform “push button” offers attributes like position, colour, label, border width, etc., since all graphical interface toolkits normally offer programming control of such “push button” object attributes.

In Figure 1, the large diversity of interaction objects supporting “selection from an explicit list of options” is demonstrated. Such alternative instantiations differ so radically with respect to physical appearance and behaviour, practically turning the quest for a common multi-platform generalised API, offering fine-grained programming control over the physical characteristics, to a technically unachievable task. However, as it is also depicted in Figure 1, from a programming point of view there is a common denominator among these different physical forms, which can constitute an initial design towards an abstract selector object. This abstract entity is concerned only with: (a) the index of the last option selected by the user; and (b) the logical notification that an option has been selected (i.e., “Selected” method). Since this entity has no physical attributes, there is still an open issue regarding the way fine-grained programming control of physical views is to be allowed when coding with abstract selectors.

Abstraction for interaction objects gains more practical value in domains where the implemented interfaces should have dynamically varying physical forms, depending on the interaction facilities offered by the particular host environment. Such an issue becomes of primary importance in domains like ubiquitous computing and universal access. Today there are no actual recipes for implementing and linking abstract interaction objects to different interface toolkits. In this context, we have implemented a User Interface programming language, i.e. a domain-specific language, encapsulating and implementing a specific software programming pattern for abstract interaction objects, while offering to the programmer declarative methods for the easy definition and deployment of abstract interaction objects. The next sections provide an overview of the programming facilities offered in the context of the I-GET User Interface programming language for the definition and deployment of abstract interaction objects - see [11], chapter 10, pp 120-151.

2 Definition of Virtual Interaction Object Classes

Abstract interaction objects are explicitly supported in the I-GET language through the keyword `virtual`, subject to specific deployment regulations. Therefore, abstract interaction objects are another category of domain-specific classes supporting compile-time type-safety. Definitions of key virtual object classes are provided in Figure 2. The source code for the complete definition of typical virtual object classes is surprisingly very small, while some of the supported features, such as the provision of constructor and destructor blocks are not practically needed. At the header of each virtual object class, the identifiers of the imported toolkits for which it actually constitutes an abstraction need to be explicitly enumerated, separated with commas. In Figure 2, the defined virtual classes are applicable to four toolkits, i.e., Xaw, MFC, Hawk (Savidis et al., 1997) and JFC.

In Figure 3, the physical mapping schemes of the State virtual object class are defined for the MFC and Hawk imported toolkits. The keyword `instantiation` issues the beginning of a specification block encompassing the logic to physically instantiate a virtual class to concrete toolkit interaction objects, through alternative mapping schemes. For each distinct toolkit, a separate instantiation definition has to be provided. At the top of Figure 3 two macros are defined. The first, named `EQUALITY`,

employs monitors to establish non-exclusive additive equality constraints between two variables – see [11], chapter 5, Monitors. This is a more general approach than built-in constraints – see [11], chapter 6, Constraints, which upon activation supersede the previously active constraint on a variable. The second, named TERNARY, is used for syntactic convenience to simulate the ternary operator, not supported in the I-GET language. In each instantiation definition, there are arbitrary mapping schemes as subsequent distinct blocks (labels 1 in Figure 3), where each such block starts with a header engaging two identifiers separated by a colon (shaded lines in Figure 3).

```
#define ALL MFC, Xaw, Hawk, JFC

virtual Selector (ALL) [
    public:
    method Selected;
    word UserChoice = 0;
    constructor []
    destructor []
]

virtual Container (ALL) [
    public:
    constructor []
    destructor []
]

virtual State (ALL) [
    public:
    bool State = true;
    method Changed;
    constructor []
    destructor []
]

virtual Button (ALL) [
    public:
    method Pressed;
    constructor []
    destructor []
]

virtual Message (ALL) [
    public:
    string label = "";
    constructor []
    destructor []
]

virtual Textfield (ALL) [
    public:
    string Text = "";
    method Changed;
    constructor []
    destructor []
]
```

Fig. 2. The complete definition of the most representative virtual interaction object classes (no code has been omitted)

The first identifier is a programmer-decided descriptive scheme name, e.g. ToggleButton, which has to be unique inside the context of the container instantiation definition, while the second identifier is the name of the toolkit class, e.g. ToggleButton, to which the virtual class is mapped (this need not be unique inside an instantiation). Even though the scheme name and the toolkit class name need not be the same, we have chosen to follow a naming policy in which the scheme identifier is the same as its associated toolkit class.

In each scheme, the programmer supplies the code to maintain a consistent state mapping between the virtual instance, syntactically accessible through {me}, and the particular physical instance, syntactically accessible through {me}Toolkit, e.g., {me}MFC or {me}Hawk. In this context, state consistency is implemented by: (a) the equality of the virtual instance attributes with the corresponding physical instance attributes, which is implemented through the monitor-embedding EQUALITY macro (see label 2 in Figure 3), or through explicit monitors when no direct type conversions are possible (see labels 3 in Figure 3); and (b) the artificial method notification for the virtual instance, when the corresponding method of the physical instance is triggered (see labels 4 in Figure 3). After all scheme blocks are supplied, a default scheme

name is assigned, e.g., default `ToggleButton` or default `RadioButton`. Such a scheme is activated automatically by default upon virtual class instantiation, if no other scheme name is explicitly chosen.

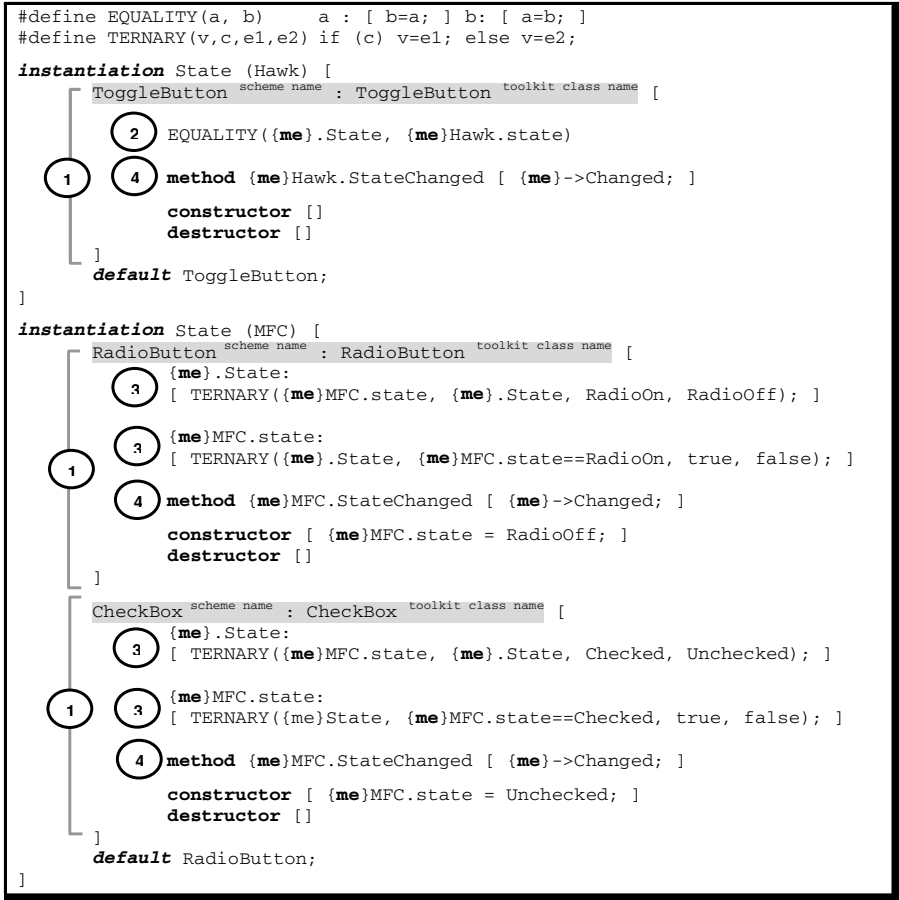


Fig. 3. The logic for polymorphic mapping of the *State* virtual object class for the MFC and Hawk imported toolkits. The `#include` directives for the virtual class definition file, and the toolkit interface specification files, are omitted for clarity

3 Declaration and Deployment of Virtual Object Instances

The instantiation definitions for each different imported toolkit can be defined in separately compiled files, while being optionally linked as distinct software libraries, called instantiation libraries. During User Interface development, programmers have to link the necessary instantiation libraries with the overall User Interface compiled code. At runtime, when a virtual instance is created, it requests the realisation of its physical instantiation from every linked instantiation library (see Figure 4, step 1).

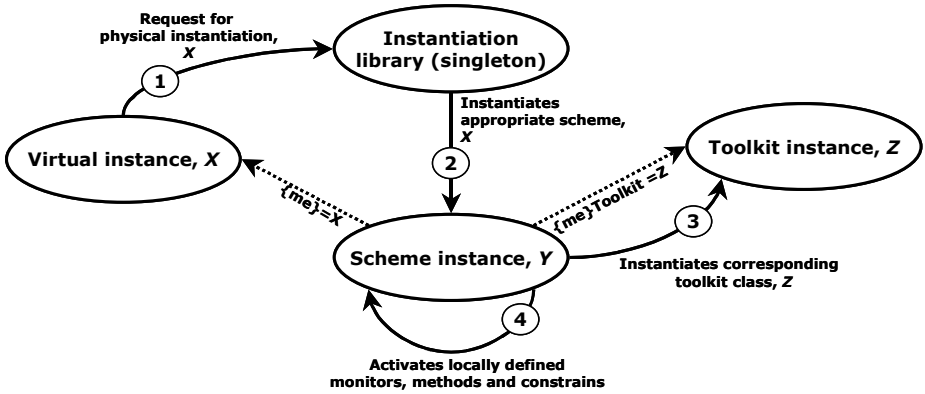


Fig. 4. The automatic runtime steps to physically instantiate a virtual class, for each target toolkit

As a result, each instantiation class, i.e., a singleton class, will create an instance of the appropriate mapping scheme (see Figure 4, step 2), i.e., either the default, or a scheme explicitly chosen upon virtual instance declaration. Then, the newly created scheme instance automatically produces an instance of its associated toolkit class (see Figure 4, step 3). Next, the scheme activates any locally defined monitors, constraints or method implementations (see Figure 4, step 4), which actually establish the runtime state mapping between the virtual instance and the newly created toolkit instance.

The resolution of the `{me}` and `{me}Toolkit` language constructs, in the context of scheme classes generated by the compiler, is also illustrated in Figure 4: `{me}` is mapped to the `X` virtual instance, while `{me}Toolkit` is mapped to the `Z` toolkit instance. As it can be observed from the above description of physical instantiation, a single virtual instance is always mapped to a number of concurrently available toolkit instances, this number being equal to the total instantiation libraries actually linked. In practice, this implies that, during runtime, virtual instances can be delivered with plural physical instantiations. For instance, if one links together the MFC and Xaw instantiation libraries, all virtual instances have dual physical instantiations for both Xaw and MFC. If the running User Interface is connected upon start-up with the respective toolkit servers of those imported toolkits, then the User Interface will be consistently replicated in two forms, at each toolkit server machine. A scenario of such runtime interface replication with two concurrent instances for windowing toolkit servers may not be considered to be particularly beneficial for end-users. However, if the interface is replicated for toolkit servers of toolkits offering complementary modalities to interactions objects, such as, for instance, the MFC-Hawk or the Xaw-Hawk toolkit pairs, then the resulting interface provides an augmented physical realisation of the application dialogue in complementary interoperable physical forms. Such interfaces can be effectively targeted to a broad audience including user groups with different interaction requirements, similarly to Dual User Interfaces [10], which offer a concurrent graphical, auditory and tactile dialogue delivery for both blind and sighted users.

```

#define PARENT(o) \
    parent (MFC)={o}MFC :parent (Xaw)={o}Xaw :parent (Hawk)={o}Hawk

agent ConfirmQuit (string text) [
    virtual Container    cont    : scheme (MFC) = FrameWindow
                                : scheme (Xaw) = PUPUPWINDOW;
    virtual Message     msg     : PARENT(quit);
    virtual Button      yes     : PARENT(quit);
    virtual Button      no     : PARENT(quit);

    method {yes}.Pressed [ terminate; ]
    method {no}.Pressed [ destroy {myagent}; ]

    method {yes}Hawk.Pressed [ printstr("Hawk YES."); ]
    method {no}MFC.Pressed  [ printstr("MFC NO."); ]

    constructor [
        {msg}.label = text;
        {yes}Xaw.label = {yes}MFC.text = {yes}Hawk.msg = "Yes";
        {no}Xaw.label = {no}MFC.text = {no}Hawk.msg = "No";
        {msg}Xaw.borderWidth = 4;
        {msg}Xaw.bgColor = "green";
        ...
    ]

    destructor [ ... ]
]

```

Fig. 5. An example of a unified implementation of a confirmation dialogue, engaging virtual object instances, retargeted automatically to the MFC, Xaw and Hawk toolkits

In Figure 5 an example is provided showing the implementation of a simple confirmation dialogue through virtual object instances. When declaring virtual object instances, programmers may optionally choose for each toolkit any of the named schemes supplied in the corresponding instantiation definition.

Additionally, since virtual instances are delivered with multiple physical instantiations, the corresponding physical parent instances have to be explicitly supplied per toolkit. For instance, the definition `ok: parent(MFC) = {cont}MFC` denotes that the physical parent for the MFC physical instantiation for the `ok` virtual instance is the MFC physical instantiation of the `cont` virtual instance. The I-GET language provides syntactic access to each of the alternative physical instantiations of a virtual instance through explicit toolkit qualification. For example, the expression `{ok}MFC` provides syntactic visibility to the MFC instantiation of the `ok` virtual instance, which, as it is reflected in the default scheme of the MFC instantiation, is actually an MFC Button instance. As it is depicted in Figure 5, virtual objects enable singular object declarations and method implementations, while also providing fine-grained control to physical aspects through physical scheme selection, physical-instance hierarchy control, and syntactic visibility of toolkit specific instances. The latter, apart from appearance control, allows specialised toolkit-specific method implementations to be supplied: see Figure 5, the implementation of methods `{yes}Hawk.Pressed` and `{no}MFC.Pressed`.

4 Code Generation

The support of virtual object classes having polymorphic plural instantiations, through facilitating scheme selection upon virtual instance declarations, is one of the most demanding and complex code generation patterns.

The runtime organization to accomplish this functional behavior has been illustrated earlier in Figure 4, introducing also the logical differentiation of classes among virtual objects, instantiation definitions and mapping schemes. Such logical distinction of roles is also reflected in code production, leading to the generation of appropriate classes from those three key categories. We will present the code generated for the State virtual object class of Figure 2, and its respective instantiation definitions of Figure 3. In Figure 6 and 7, the code generation from the compilation of the virtual class definition is provided. Overall, the code generation for virtual classes provides the ground for multiple concurrent physical instantiations, by delivering the placeholder as well as the activation mechanism for the toolkit-specific instantiation definitions. This capability to automatically activate any instantiation definitions of virtual classes, once their respective instantiation library is linked with the User Interface code, is the most demanding feature.

Following Figure 6, the produced header file encompasses firstly the forward declarations of the instantiation classes for each target toolkit, e.g., class `_INSTStateHawk` for the Hawk toolkit. The generated virtual class, e.g., `_VICState`, encapsulates pointer variable declarations for all potential instantiation classes (see label 2), like `_INSTStateMFC*_instMFC` for MFC. Additionally, all instantiation classes are defined as friends for the generated virtual class. The code generation for any local definitions made inside virtual classes is collected in one fragment (label 3).

Our approach towards automatic support for multiple instantiation, according to the particular instantiation libraries linked, is based on explicit instantiation and destruction functions pairs, e.g., `instantiateMFC` and `deleteMFC` (see label 1), one for each target toolkit, e.g., MFC. The implementation idea is that, initially, those functions will be supplied with a default empty implementation by the generated virtual class (see labels 6 and 7). Then, during runtime, each linked instantiation class sets, upon global data initialisation, its specific pair of fully implemented instantiation and destruction functions. To ensure that the pair set by such instantiation classes cannot be overwritten during initialisation by the default pair of the virtual class, we use a technique introduced in (Schwarz, 1996) for safely initializing static variables in C++ libraries. This technique uses a static flag per toolkit, e.g., `flagMFC` for MFC, which is to be unconditionally set by the respective instantiation classes after its specific pair of instantiation and destruction functions of the virtual class is set (this will be discussed later in the context of code generation of instantiation classes).

At the virtual class side, the default functions per toolkit are set only if the corresponding flag is not set (see label 8); consequently, irrespective of the order of initialization, the initializations made by instantiation classes can never be overwritten. Within the produced virtual class, those functions are called to actually perform the physical instantiation (in the constructor) or destruction (in the destructor) for each target toolkit (see label 4 for instantiation, and label 5 for destruction).

As depicted in Figure 8, instantiation definitions are generated as instantiation classes, e.g., `_INSTStateMFC`, while each embedded mapping scheme is produced as a distinct scheme class, e.g., `_SCHStateMFCRadioButton`. Scheme classes encompass two key member pointers, holding their runtime associated virtual instance and physical instance respectively (see label 1), e.g., `myvirtual` of type `_VICState*` and `myphysical` of type `_PICRadioButtonMFC*`. In the code generation of the scheme blocks, the compiler always resolves the `{me}` expression as `myvirtual`, and the

{me}Toolkit expression as myphysical. As it can be observed from Figure 8, instantiation classes like `_INSTStateMFC` are very simple upon code generation, encompassing a super-class `Scheme*` scheme pointer (see label 2), which holds the particular runtime active scheme. Finally, the generated header file employs the technique for static data initialisation (see label 3), so as to ensure that schemes are safely initialized prior any runtime use (see also Figure 9). In Figure 9, key fragments of the generated implementation file for the instantiation definition of the State virtual class are supplied. As it has been previously discussed, scheme classes are responsible for the automatic creation of physical instances for their associated lexical class. Following this need, as reflected in Figure 9, the constructor of the `_SCHStateMFCRadioButton` scheme class (see label 1) firstly stores in the myvirtual member the caller virtual instance, and then performs the creation of a `_PICRadioButtonMFC` lexical object instance, stored in myphysical. The instantiation of the appropriate scheme classes is performed inside the constructor of the container instantiation class, through a switch statement (see label 2) over a parameter (e.g., the `i`) that provides the order of appearance of the desirable scheme within the instantiation definition. This number is easily defined by the compiler upon virtual instance declaration, being the order of either the programmer supplied scheme or the default scheme.

```

class _SCHStateMFCRadioButton : public Scheme {
public:
  _VICState*          myvirtual;
  _PICRadioButtonMFC* myphysical;
  Code generation here for constructs defined in the scheme block
  _SCHStateMFCRadioButton (Agent*, _State*, LexicalClass*);
  ~_SCHStateMFCRadioButton();
};

class _SCHStateMFCCheckBox : public Scheme { ... };

class _INSTStateMFC {
public:
  Scheme* scheme;
  _INSTStateMFC (Agent*, _VICState*, unsigned, LexicalClass*);
  ~_INSTStateMFC () { assert(myphysical); delete myphysical; }
};

class StateMFC_Initializer {
public:
  static unsigned flag;
  StateMFC_Initializer (void);
};
static StateMFC_Initializer stateMFC_Initializer;

```

Fig. 8. The code generation of the header file for the instantiation definition of State virtual class for the MFC toolkit

Additionally, the compiler produces the key pair of functions for the instantiation (see label 3) and destruction (see label 4) of the generated instantiation class. Those functions are appropriately assigned upon initialization to the corresponding members of the virtual class (see label 5). After the assignment is performed, the corresponding flag is set, e.g., `flagMFC`, thus disabling overwriting of those functions with the default implementations due to virtual class initialization.

```

_SCHStateMFCRadioButton::_SCHStateMFCRadioButton (
    Agent* a, _VICState* v, LexicalClass* p
) {
    myvirtual = v;
    myphysical = new _PICRadioButtonMFC(p); } 1
    Code generation here for initializations of constructs defined in the scheme block
}

_INSTStateMFC::_INSTStateMFC (
    Agent* a, _VICState* v, unsigned i, LexicalClass* p
) {
    switch (i) {
    case 1: scheme = new _SCHStateMFCRadioButton(a, v, p); break;
    case 2: scheme = new _SCHStateMFCCheckBox(a, v, p); break;
    } } 2

static _INSTStateMFC* InstantiateMFC (
    Agent* a, _VICState* v, unsigned i, LexicalClass* p ) 3
) { return new _INSTStateMFC(a, v, i, p); }

static void DeleteMFC (_INSTStateMFC* inst) 4
    { assert(inst); delete inst; }

unsigned StateMFC_Initializer::flag;
StateMFC_Initializer::StateMFC_Initializer (void) {
    if (!flag) {
        _VICState::instantiateMFC = InstantiateMFC;
        _VICState::destroyMFC     = DestroyMFC;
        flag = State_Initializer::flagMFC = 1;
    } } 5
}

```

Fig. 9. Key fragments of the generated implementation file for the instantiation definition of State virtual class for the MFC toolkit

5 Discussion and Conclusions

Though the presented language constructs, the developer is enabled to define and instantiate abstract object classes, while having control on the physical mapping schemes that will be active for each abstract object instance at runtime; mapping schemes explicitly define the alternative candidate physical classes to physically realise abstract object class.

Polymorphism in virtual interaction objects	Polymorphism in typical OOP languages
<ul style="list-style-type: none"> Aims to support alternative morphological realisations (i.e. polymorphism with its direct physical meaning). 	<ul style="list-style-type: none"> Aims to support re-use and implementation independence from different toolkits (i.e. polymorphism with its metaphoric meaning).
<ul style="list-style-type: none"> Instantiation is applied directly on abstract object classes. 	<ul style="list-style-type: none"> Instantiation is always applied on derived non-abstract classes.
<ul style="list-style-type: none"> Multiple physical instances, manipulated via the same abstract object instance, may be active in parallel at a time. 	<ul style="list-style-type: none"> References to an abstract always refer to a single derived object instance at a time.

Fig. 10. Key differences, with respect to polymorphism and abstract objects between general purpose OO programming languages and polymorphism as supported through virtual object classes

The need for having multiple physical instances active, all attached to the same abstract object instance (i.e. plural instantiation) can be exploited in case that the alternative physical forms are compatible, while their co-presence results in added-value interactions. For instance, in the context of Dual interface development [10], two concurrently active instances are always required (i.e. a visual and a non-visual) for each abstract object instance. The notion of polymorphic physical mapping and plural instantiation, have fundamentally different functional requirements, with respect to polymorphism of super-classes in OOP languages. The key differences are outlined in Figure 10.

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

References

1. 2WEAR project (2003). Web site, <http://2wear.ics.forth.gr>, see Demonstrator pictures.
2. Blattner, M.M.; Glinert, J.A. & Ormsby, G.R. (1992). "Metawidgets: towards a theory of multimodal interface design". In Proceedings of COMPSAC '92 (pp. 115-120). Los Alamitos, CA: IEEE Computer Society Press.
3. Bronevetsky, G. (2003). Circle Menus. Demo implemented in Java, available electronically from: <http://www.cs.cornell.edu/boom/2001sp/Bronevetsky/Circle%20Menu%20Documentation.htm>
4. Duke, D., Harrison, M. (1993). Abstract Interaction Objects. Computer Graphics Forum, 12 (3), 1993, 25-36.
5. Duke, D., Faconti, G., Harrison, M., Paterno, F. (1994). Unifying view of interactors. Amodeus Project Document: SM/WP18, 1994.
6. Foley, J., Van Dam, A. (1983). Fundamentals fo interactive computer graphics. Addison-Wesley Publishing, 1983 (1st edition), 137-179.
7. McGuffin, M., Burtnyk, N., Kurtenbach, G. (2001). FaST Sliders: Integrating Marking Menus and the Adjustment of Continuous Values. Graphics Interface 2001, paper available online from: <http://www.graphicsinterface.org/cgi-bin/DownloadPaper?name=2002/174/paper174.pdf>.
8. Savidis, A., Stephanidis, C., Korte, A., Crispian, K., Fellbaum, K. (1996). A Generic Direct-Manipulation 3D-Auditory Environment for Hierarchical Navigation in Non-visual Interaction. In proceedings of the ACM ASSETS'96 conference, Vancouver, Canada, April 11-12, 1996, 117-123.
9. Savidis, A., Stergiou, A., & Stephanidis, C. (1997). Generic Containers for Metaphor Fusion in Non-Visual Interaction: The HAWK Interface Toolkit. In Proceedings of the 6th International Conference on Man-Machine Interaction Intelligent Systems in Business (INTERFACES '97), Montpellier, France, 28-30 May (pp. 194-196).
10. Savidis, A., & Stephanidis, C. (1998). The HOMER UIMS for Dual User Interface Development: Fusing Visual and Non-visual Interactions. International Journal of Interacting with Computers, 11 (2), 173-209.
11. Savidis, A. (2004). The I-GET User Interface Programming Language: User's Guide, Technical Report 332, ICS-FORTH, January 2004, available electronically from: ftp://ftp.ics.forth.gr/tech-reports/2004/2004.TR332.I-ET_User_Interface_Programming_Language.pdf
12. Schwarz, J. (1996). *Initialising static variables in C++ libraries*. In C++ Gems, Lippman, S. (Ed), SIGS Books, New York, pp 237-241.