Jessie Kennedy and Peter Barclay (Eds)

Interfaces to Databases (IDS-3)

Proceedings of the 3rd International Workshop on Interfaces to Databases, Napier University, Edinburgh, 8-10 July 1996

Paper:

DRIVE: An Environment for the Organised Construction of User-Interfaces to Databases

K. Mitchell and J. Kennedy

Springer

DRIVE : An Environment for the Organised Construction of User-Interfaces to Databases

Kenneth J Mitchell

Jessie B Kennedy

Computer Studies Department, Napier University,
Edinburgh, Scotland

Abstract

This paper describes a runtime user-interface development environment (UIDE) for the novel capability of interactively using and specifying user-interfaces to object-oriented databases (IDSs). A framework provides the foundation for IDSs constructed. This concerns interpreting an IDS, specified in a conceptual object-oriented data language, using a persistent meta model. A generic environment model facilitates dynamic integration of existing user-interface widgets into the meta model. This achieves the goal of providing a database representation independent visual environment (DRIVE). The architecture and use of DRIVE are described and the benefits of this approach are discussed.

1   Introduction

As the issue of providing user-interfaces to databases (IDSs) becomes the focus of increasing numbers of research groups, there is a call for tools that act as sketch pads for new interaction and presentation ideas. Unless such tools follow a well-defined organised approach, results will be harder to reproduce in commercial systems and remain limited to the original tool of conception.

Creating such tools raises critical problems in terms of their flexibility and performance. Supporting diverse interactive components and coordinating them uniformly with respect to database components [14] presents a challenge to all IDS developers. In addition, the interactive specification of IDS elements through direct manipulation [17] with immediate feedback on modifications [7] requires a particular solution when the data resides in a database. Typically this problem is exacerbated by an impedance mismatch between the user-interface programming language and the database's data model.

In regard of these issues, a database representation independent visual environment (DRIVE) is presented which serves as a tool for using and specifying existing and prototype IDSs in a runtime user-interface development environment (UIDE).

To ensure well-defined organised IDS construction, DRIVE enforces the relationships and dependencies identified in our framework for user-interfaces to databases [11] using a conceptual object-oriented data language. The architecture presented provides a means for coordinating disparate interface components using a persistent meta model of the IDS framework together with an environment model. This scheme has the following distinct advantages:

- Interactive support for IDS schema and object definition and manipulation without requiring the support of schema evolution in the object oriented database system used.
- Explicit consideration for users of IDSs as modelled within the IDS framework, allowing user-specific configurations incorporating security measures.
- Dynamic integration of existing and novel user-interface objects, such as widgets and controls, in an environment model.

The following section briefly describes our IDS framework with particular emphasis on the mapping to the conceptual language. Section 3 shows an IDS prototyped in the DRIVE UIDE, which is used as an exemplar for the following technical content. Section 4 presents DRIVE's architecture based on runtime interpretation of a persistent meta model of the IDS framework. Section 5 details the environment model, which dynamically binds visualisation classes of the framework meta model to external user-interface objects. Section 6 describes the
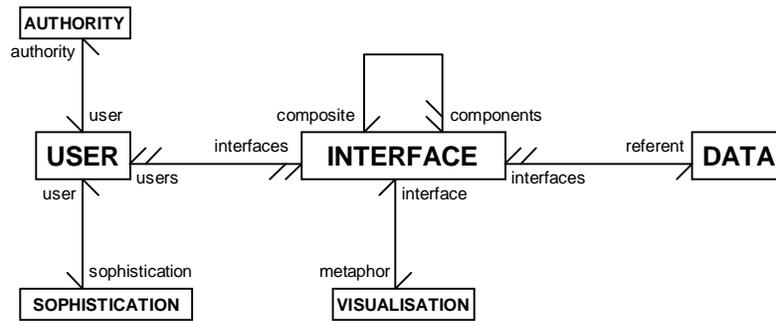
Figure 1 A NOODL meta-model of the IDS framework

process of IDS construction, for interactive manipulation of conceptual language specifications. Finally, conclusions and some further work are discussed.

2   Background

The framework that provides the foundation for IDS construction in DRIVE is based on Abowd and Beale's [1] interaction framework, which identifies four major components of an interactive system, ie. *system, input, output* and *user*. In applying this framework specifically to IDSs, modifications and extensions to these components have been made. We have identified an IDS to be the composition of *database, interaction, visualisation* and *user* components. In addition, the common features of each of these IDS components have been identified, eg. a visualisation component has a *referent, metaphor* and *layout*.

We have shown how such a detailed classification of components may be mapped to a conceptual language that embodies the relationships and dependencies among the components of an IDS. For this purpose, we have chosen the data modelling language NOODL (Napier's object-oriented data language). NOODL is a combined data definition, manipulation and query language [4] based on the modelling approach described in [2]; it has been used to model and to support the implementation of novel database applications, and also for the investigation of specific modelling issues such as declarative integrity constraints and activeness in object oriented data models.

A NOODL schema contains a list of class definitions detailing its name, the ancestors of each class, and the names, sorts, and, optionally, definitions of the properties of each class. It may also contain operations, constraints, and triggers. Full details may be found in [3] and an example class definition is given below.

```
class Artefact
properties
   name              : Text ;
   description       : Text ;
   catalog_id        : Number
operation
   update ( Text n, Text d, Number c ) is
       self.name(n), self.description(d), self.catalog_id(c)
constraint
   self.name <> bottom
trigger
   self.description > 2000 =>
       error("Description Too Long"), self.description(bottom)
```

Figure 1 depicts the classes and properties of a meta-model of the IDS framework using NOODL constructs. In mapping the framework to this data language, certain components are depicted as classes, some as properties, while others are modelled using operations, constraints or triggers. An example schema based on the above template is given in the appendix, which shows the specification for a prototype user-interface to a museum's database.

In common with Rumbaugh's user interface modelling approach [15] multiple interface objects are associated with each data object. This permits database updates to be broadcast to each relevant interface object, thus realising the facility for multiple-coordinated views. Much work in the field of HCI reflects this conceptual

organisation. Conceptual architectures for user interface management systems (UIMS) typically involve the identification of the system (data) in separation from input and output (interface) components. This is evident in the archetypal Seeheim workshop model with application interface, dialogue control and presentation components; Smalltalk's model, view, and controller (MVC) paradigm; and the presentation, abstraction and control model (PAC)[14].

The NOODL meta model of the IDS framework provides a number of features, including,

- Multiple interface objects associated with each data object
- Interface object composition for specification of layouts
- Explicit user modelling

If an architecture exists which can interpret the constructs of the NOODL data model, then concise IDS specifications, based on the above IDS framework meta model, may be interpreted to automatically realise functional IDSs. DRIVE supports interactive specification and run-time interpretation of NOODL constructs.
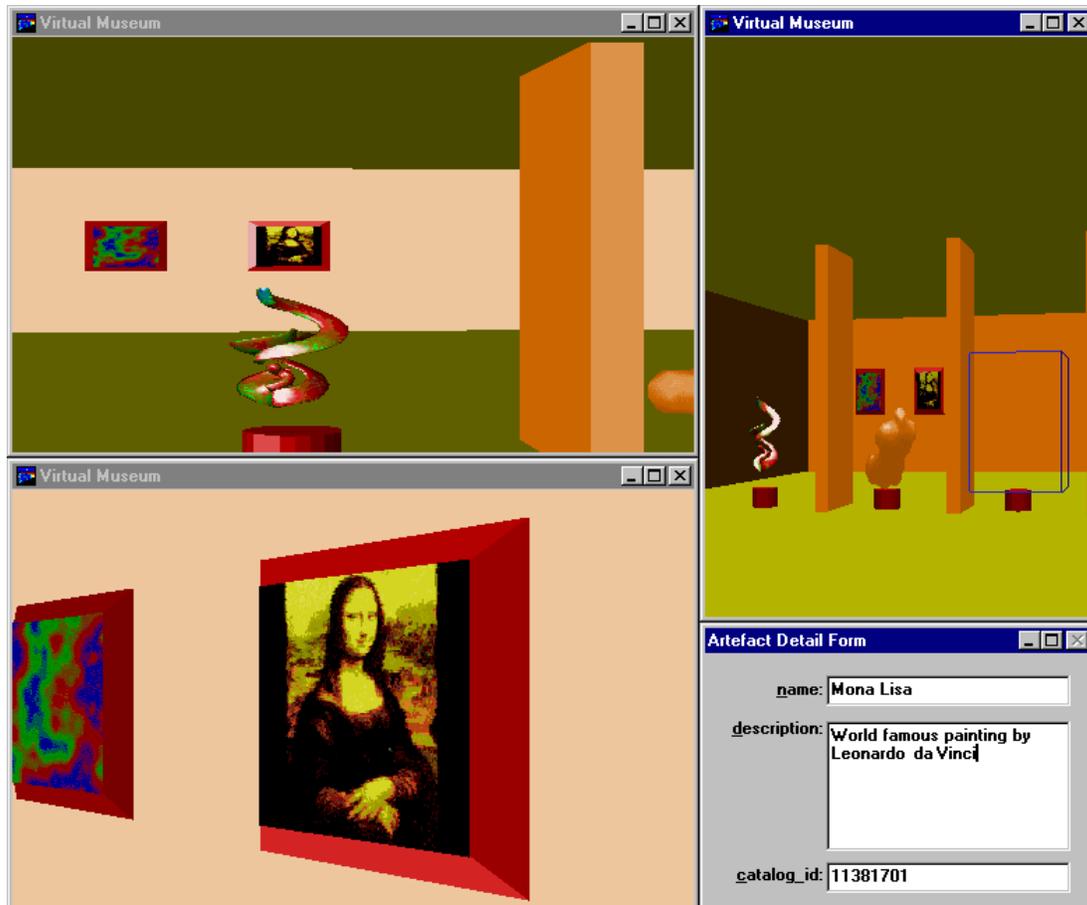

Figure 2 Integrated 2D and 3D widgets in the virtual museum IDS

3   Example User-Interface to a Database

Figure 2 shows a display from the prototype IDS specified using the DRIVE UIDE (see appendix for the specification). The display shows various views of the museum, which may be edited interactively with the mouse and keyboard in *designer* mode. If an object is moved or altered in one view, then the change will be updated in the other views. The dialog box in the lower right of the display shows the details of the currently selected artefact. This shows the effect of the browse operation of the Artefact Interface class and serves as an example of the novel applications possible with integration of 3D graphics technology with traditional 2D graphics. This class is detailed below.

```
class Artefact_Interface                                    (* Interface Class*)
properties
   referent          : Artefact ref interface ;                  (* referent *)
   metaphor          : Shape3d ref interface ;                   (* metaphor *)
   museum            : Museum_Interface ref artefacts          (* composite *)
operation
   browse is self.metaphor.select                        (* interface action *)
constraint
   self.metaphor.position.is_inside(self.museum.metaphor.extent)
trigger
   browse =>                                           (* interface response *)
      self.museum.name_interface.metaphor.name(self.referent.name),
      self.museum.desc_interface.metaphor.name(self.referent.description),
      self.museum.cata_interface.metaphor.name(self.referent.catalog_id)
```

Figure 3 shows all the classes of the virtual museum's NOODL specification. Each IDS class is grouped according to the respective framework component modelled. From the diagram we can see that an Artefact Interface class has three linking properties, ie. a referent, metaphor and composite, corresponding directly to the above schema text. Upon selecting a 3D shape in the virtual museum, the browse trigger responds by setting the text of each field in the artefact detail form to the properties of the referent of the selected shape.
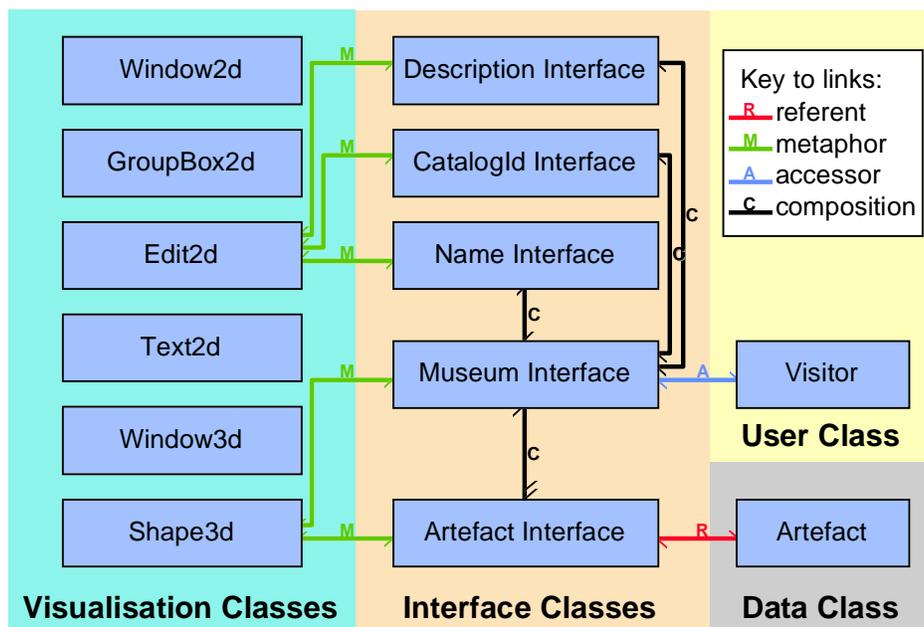

Figure 3 Classes and links in the example IDS under the NOODL Data Model supported by DRIVE

4   Architecture

Fundamental to the provision of such tools is their architecture. Figure 4 shows three alternative architectures where database (DB) and user-interface (UI) components are linked in alternative ways. Gray and Cooper [9] have identified a need for architectures of *integration*, where relationships amongst IDS components are managed using some integrating model. This is necessary both to maintain multiple coordinated views of data [15] and provide the composition of interactive components from disparate origins. In opposition to this are architectures of *seclusion*, where specific IDS components are defined inside and in terms of the user-interface development environment's (UIDE) model. However, if such UIDE's incorporate an integrating environment model (EM) supporting
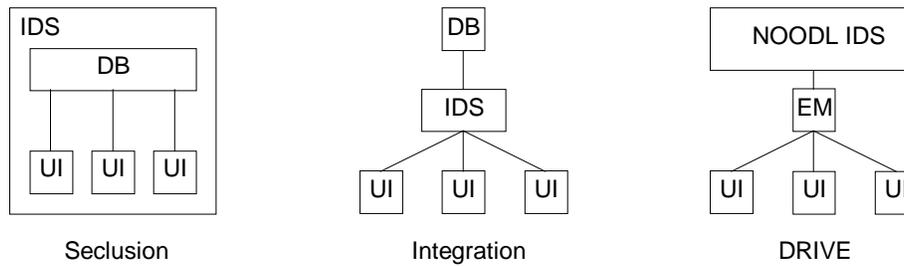
Figure 4 A comparison of alternative IDS architectures

coordinated management of disparate interactive components, then the 'closed world' criticism of such tools is no longer appropriate. In DRIVE there exists such an environment model, which coordinates existing user-interface components with IDSs specified using the NOODL data model.

Essential to the IDS design process is immediate feedback on modifications [7]. Consequently, an architecture for runtime interpretation of IDS designs is desirable. Mogetto [16] is a good example of a runtime architecture. Embedded interface (MOG[8]) objects are manipulated interactively by means of an event switch, which channels inputs to the interface object's editing behaviour. However, as recognised by Sawyer et al. [16], changes are restricted at runtime to those which only locally affect display configurations and default query callbacks. Deeper changes such as editing application functionality require (often time consuming) recompilation. If such changes are to be permitted at runtime then responses to events originating from the database must be considered in addition to providing an interpreted computationally complete data language.

Given a conceptual level specification, a computational level interpretation is required to execute the functional IDS. As DRIVE supports the interpretation of NOODL constructs the structure and behaviour specified can be automatically realised. However, in order that novel user-interface widgets may be utilised, the interpretation of NOODL visualisation class definitions must be configurable. This is achieved by mapping visualisation classes to computational level user-interface widget code using the environment model.

DRIVE consists of three major components, an *IDS component*, an *environment model*, and a *design environment*. The IDS component uses a persistent NOODL meta model to instantiate IDS framework specifications in terms of NOODL constructs. The environment model manages the integration of a set of user-interface widgets with the NOODL visualisation classes of the IDS component. The design environment consists of tools for editing and executing multiple textual and graphical IDS specifications.

Data, user, interface and visualisation classes combine to make up the IDS component. The environment model contains a scheme for handling interaction events and mapping them to the appropriate IDS objects, via instances of visualisation classes. A visualisation class may be derived in conjunction with existing user-interface widgets, such as OWL[13] controls, and advanced interface widgets, such as the 3D widget set's (TDW)[6] access widget.

The NOODL classes of an IDS are built upon a persistent data model layer. This layer, (similar to ObjectStore's Meta Object Protocol [12]) implemented with the POET [5] persistent C++ extension, provides a means of dynamically creating persistent IDS schemata together with their data. It is implemented as a vanilla C++ meta model, using only single inheritance, complex objects and polymorphic behaviour, in order to facilitate porting to other object oriented database systems (OODBs).

5   Environment Model

Figure 5 shows the visualisation classes of the IDS specified in the appendix together with corresponding environment and user-interface widget classes managed by the environment model. The headings underneath identify the language used to define each particular class, with NOODL classes, C/C++ environment classes and (for this example) C++ user-interface classes. The lines between classes represent object linkage, using either the linking properties of the IDS framework (for the NOODL classes) or normal C pointer references. With this linkage strategy the vast majority of existing user-interface widgets may be utilised.
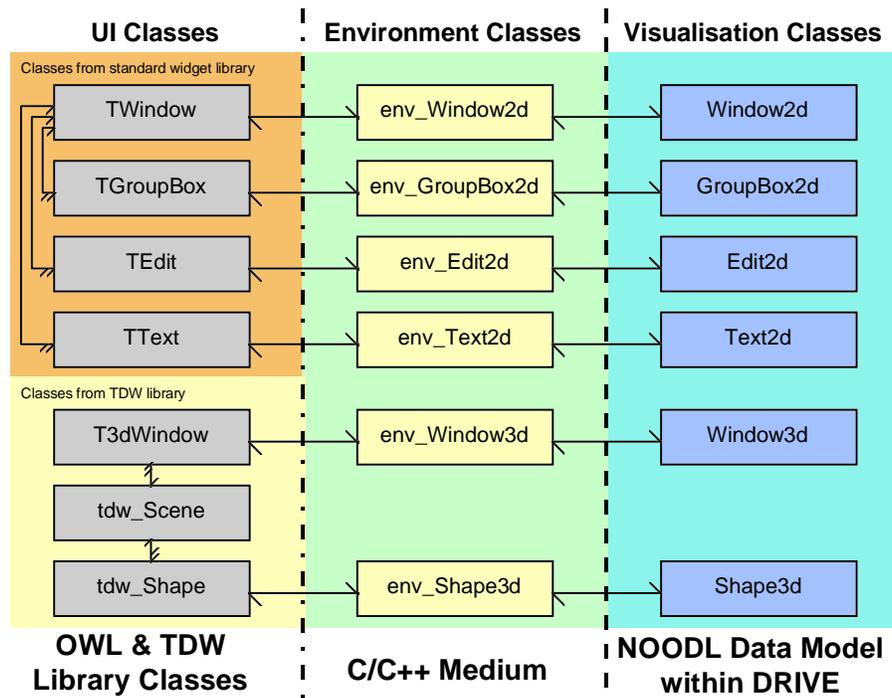
Figure 5 Mapping NOODL visualisation classes to C++ user-interface widget classes

To achieve linkage between NOODL visualisation classes and C/C++ environment classes the NOODL meta model allows access to the properties of visualisation classes via the *environment manager*. In tandem with this, visualisation classes can access the features of environment classes with the environment manager, described next.
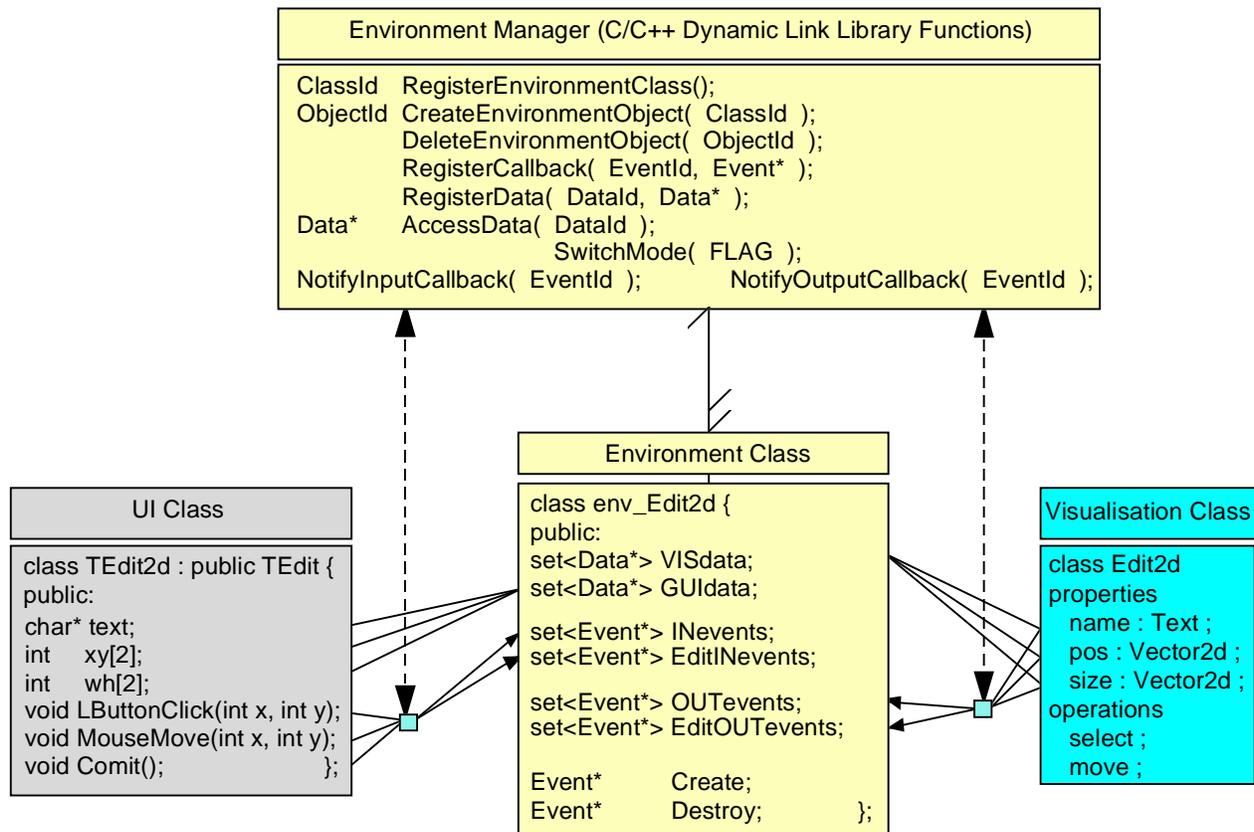
5.1 Environment Manager

Figure 6 details the operation of the environment manager in relation to user-interface widget, environment and visualisation classes, from left to right respectively. The environment manager contains a set of environment classes. It performs three functions,

- provides dynamic registration of environment classes for user-interface widgets.
- maps input and output events between corresponding user-interface and visualisation classes
- channels events according to the mode of operation (*designer* or *execution* mode)

Dynamic registration of environment classes is achieved by placing the environment manager in a dynamic link library (DLL), which is initially linked to DRIVE. Whenever a new type of widget is required, the code module (normally also a DLL) containing the widget code links with the environment manager and calls the appropriate C functions to register the environment class. These calls must define a template for the corresponding visualisation class as well as register the properties and callbacks of the environment class.

Consider when a user clicks the left mouse button on an artefact in the virtual museum example. Normally, this event is handled by the widget and the LButtonClick callback function is called within the encapsulated widget to respond to this event. However, in order to map this user-interface event to a NOODL select event, it must be forwarded to the environment manager with a message identifier using the *NotifyInputCallback*. This function matches the LButtonClick event with the select input callback of the registered env_Shape environment class and calls it.

In the virtual museum specification, the browse trigger of the artefact's interface object is fired when the select event occurs. This trigger assigns the text of each associated edit object in the artefact detail form to the properties of the referent of this object. Such a change will result in an *output* event, which then must be forwarded to the environment manager using the *NotifyOutputCallback*. The output callback accesses the text fields of the Form visualisation object and updates the TEdit user-interface widgets according to the newly selected artefact.

Figure 6 Generic runtime environment manger for integrating disparate user-interface widgets

If many interface objects share the same referent, a change in the linked data object will trigger updates in all associated visualisation objects. In this way, the mechanism for multiple coordinated user-interface widgets may be realised.

With all events channelled through the environment manager, a simple switch is sufficient to realise a change in mode of operation. If designer mode is set then all events will be forwarded to the appropriate environment class' edit callbacks. If execution mode is set, then they will be sent to normal callbacks. Although, this technique has been employed in MOG objects [8], the editing behaviour is internal to the encapsulated MOG object. Here, output callbacks allow editing behaviour and application functionality to be defined in the NOODL IDS.

5.2 Environment Classes

It is possible to categorise the features of environment classes according to the input/output direction of events. Each input callback may contain code to modify a visualisation class' active property from its *VISdata* set or trigger a visualisation class' event operation. These may read information about the widget's state using its *GUIdata* set. For example, in figure 5, the commit input callback would read TEdit2d's text string and pass it to the Edit2d visualisation class' text. Conversely, each output callback may modify a user-interface widget class' property via the GUIdata set and read information from the VISdata set.

Two other callbacks are required for creating and deleting environment objects at runtime. The *create* callback is called after a visualisation object is created or retrieved from the database. It must initialise the user-interface widget's state using the environment manager's *CreateEnvObj* and assign the GUIdata set using the VISdata set to read information from the derived visualisation object. The *destroy* callback is be called after visualisation object is deleted or unloaded from memory. It must remove the user-interface widget from memory and then call the environment object's destructor using the *DeleteEnvObj* function.

5.3 Integrating Advanced User-Interface Widgets

The environment model is general enough to make use of the majority of existing user-interface widget sets. The term *environment* is purposely chosen to indicate that an IDS can interact with potentially much more than standard widgets. Through environment objects an IDS may communicate data to and from novel environments (eg. dynamic data streams, special I/O peripherals, other databases, etc.). This flexibility has been exploited in the task of designing three dimensional database environments using the 3D Widget set (TDW) [6]. Figure 2 shows the use of TDWs to construct a 3D environment representing the artefact of a museum. Each artefact uses a *tdw_Shape* to represent itself. These shapes are registered in a 3D scene (*tdw_Scene*) and rendered in a window (*T3DWindow*). This derived OWL window handles keyboard and mouse events (such as navigation and selection), which are passed to the shape that currently has the user's focus. In this way the user can directly manipulate the shapes in the 3D environment and change the state of the database through the environment model.

6   NOODL Data Model Interpreter

In order to manage the components of an IDS defined in the NOODL data model under the IDS framework, each class must identify the component to which it belongs, eg. a *Visitor* class must belong to the *user* component of the IDS framework. In this way the properties, operations and triggers of the framework meta model are constrained by the interpreter to appear only in their appropriate classes. The mechanism for enforcing this rule marks every property, operation and trigger with its identity in terms of the framework. These are detailed below.

6.1 Framework Class Templates

All classes may act as the referent of one or more interface classes and so may have one or more *interface* properties, eg. from the Artefact data class,

```
name_interface : Name_Interface ref referent
```

which defines the link to an interface class or classes. *Data* classes are constrained to this type of framework property only. The remaining features are particular to the type of framework component the class belongs.

 User classes have *authority*, *sophistication*, and *accessors* properties defining the links to *Authority*, *Sophistication* and their accessible *Interface* classes. User classes may have operations defined as *tasks*, which are used to model the purpose of the user. Authority and sophistication classes have *user* properties refering back to their user classes. In the simplified virtual museum example, a Visitor is a user class with one accessor property linking it to a set of Window Interfaces, specifying that visitors interact via a number of windows.

 Interface classes are permitted to have *referent, components, composite* and *metaphor* properties. The referent property defines the link to the subject of the interface component, eg. the referent of Artefact Interface is the Artefact itself. Components and composite properties allow hierarchies of interface objects to be built and managed, including support for layouts. The Museum Interface class contains both a set of Artefact Interface objects and single Name Interface, Description Interface and CatalogId Interface objects. The metaphor property links an interface class to a *Visualisation* class. In addition, interface classes have interface *actions* [14] and *responses* for operations and triggers, respectively. In terms of the framework, the definition of interface action operations specifies the *medium* of the interaction component and the definition of interface response triggers specifies the *effect* of the interaction. The browse trigger mentioned in section 5.1 defines a response to the browse interface action operation, which permits a visitor to select an artefact to view its details.

 Visualisation classes may have a set of *active* properties and *event* operations, which enables the environment model to respond to events in the external user-interface widgets. For example, the properties and operations of the Edit2d class in figure 6 are all active properties and event operations, respectively.

 In addition to these framework features, each class may use standard NOODL constructs to model an IDS. However, with these features certain classes will depend on the existence of other classes, ie. authority and sophistication classes may not exist without user classes, and interface classes may not exist without a referent, metaphor and user. Indeed, the default IDS schema must contain at least one user class, in order to permit access to the IDS.

 This provides a general for scheme the organised construction of advanced IDSs. The next section covers their interactive specification.
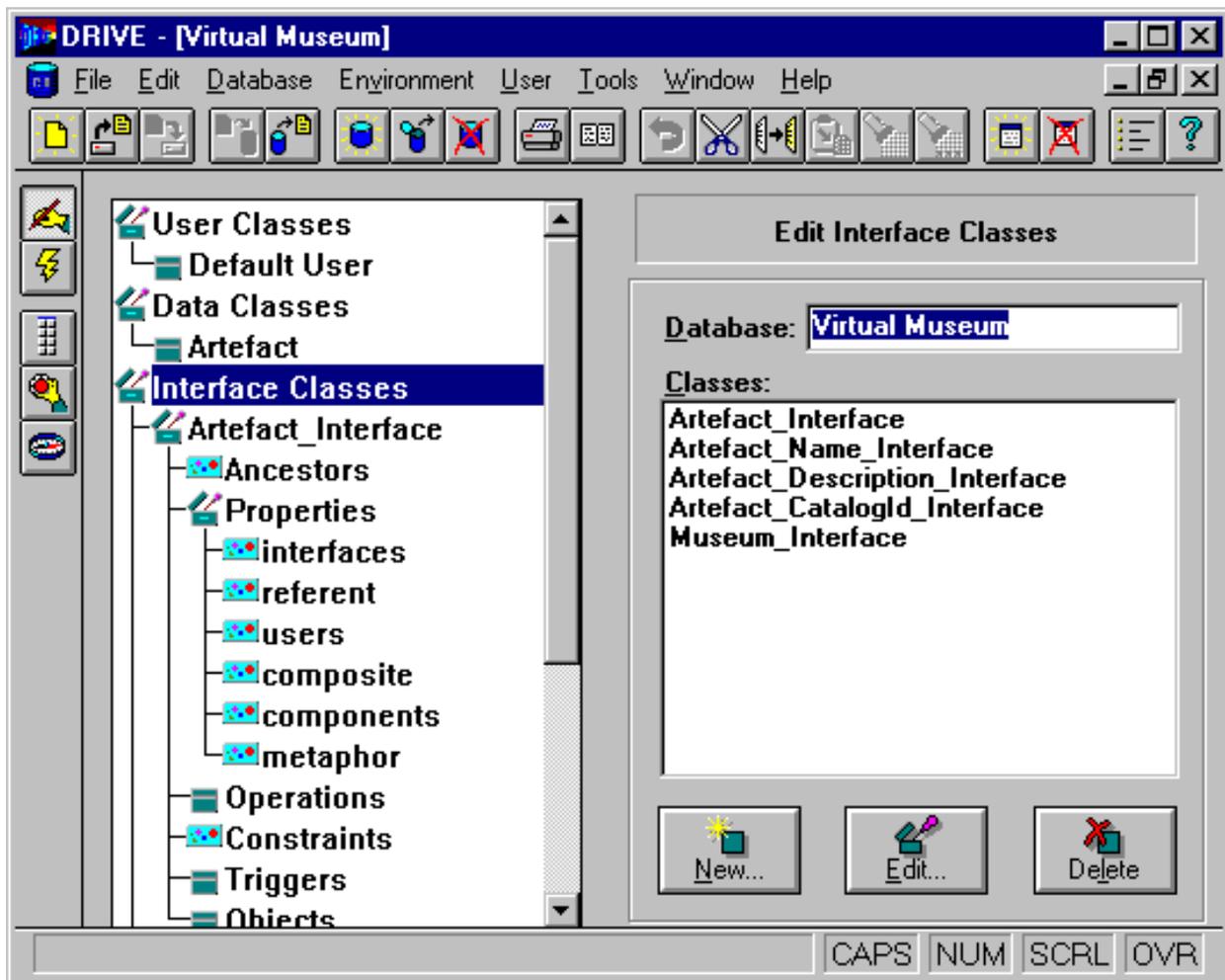
Figure 7 IDS editor using persistent object resource key tree.

7   IDS Construction

Given an architecture supporting the runtime interpretation of NOODL IDS specifications the facility exists within DRIVE to compile textual NOODL script files into the IDS framework meta model and generate working IDSs. In addition, with a runtime NOODL interpreter it is possible to entirely construct IDSs within DRIVE. Once a designer is satisfied, concise NOODL script specifications may be written out and documented.

7.1 IDS Editor

Interactive specification of IDSs is carried out through the IDS editor. This combines a hierarchical list view of the IDS schema with a node specific dialog. When the user selects an element in the schema list, the view on the right changes to display the controls required to edit that element. Hierarchical list views are becoming common place in windows applications. Sub-lists may be expanded or collapsed by double clicking the mouse button on the composite list node. This has been achieved with respect to an OODB by means of a persistent object resource key tree (PorkTree).

Figure 7 shows the virtual museum specification with the PorkTree expanded to show the default properties of an interface class. Classes are strictly grouped according to their categorisation in the IDS framework.

Each node in the PorkTree contains an object of the IDS framework meta model and a dialog object. With this information a node acts as a unique key to the desired object. When a node is selected in the list view, the dialog object is passed the persistent meta model object for editing. The PorkTree contains nodes for editing the

entire schema, from lists of class ancestors to individual properties. In addition to the schema information, IDS objects and their values also form part of the PorkTree.

This method works well for a few levels of sub-division, but if the tree becomes over-complicated access to deep sub-nodes is sluggish and impractical. For this reason, the definitions of derived properties, operations, constraints and triggers are specified in an additional definition editor.

7.2 Context Sensitive Definition Editor

Given that DRIVE's design environment has access to the IDS component, and therefore all the constructs of the NOODL data model, it is possible at any point within the definition of a NOODL expression to determine the set of valid operators and operands. This is exploited in the definition editor where the context of the expression is represented by the elements in the operator and operand lists. With these lists the user can define complete operation definitions and constraint expressions, simply by selecting the available element and pressing the *add* button. Figure 8 represents the definition editor after the browse operation of the Artefact Interface class has been partially specified. The operand box is filled with the active properties and event operations of the artefact interface's associated Shape metaphor. The operator box contains complete, settor and delete elements to signify the completion of the current expression and the metaphor's settor and delete operators, respectively.

As with the PorkTree dialogs, the context sensitivity of the editors gives the designer support for constructing IDSs within the framework without deviating from the semantics of the data model. This is particularly useful for designers unfamiliar with NOODL where no knowledge of syntax is necessary. Consideration for advanced designers has been acknowledge by the provision of keyboard and mouse shortcuts. Typically, this method requires significantly less keyboard and mouse interactions than using a traditional text editor. Further, because the number of valid operators and operands is frequently less than 20, the performance of this editor does not hinder the specification process.
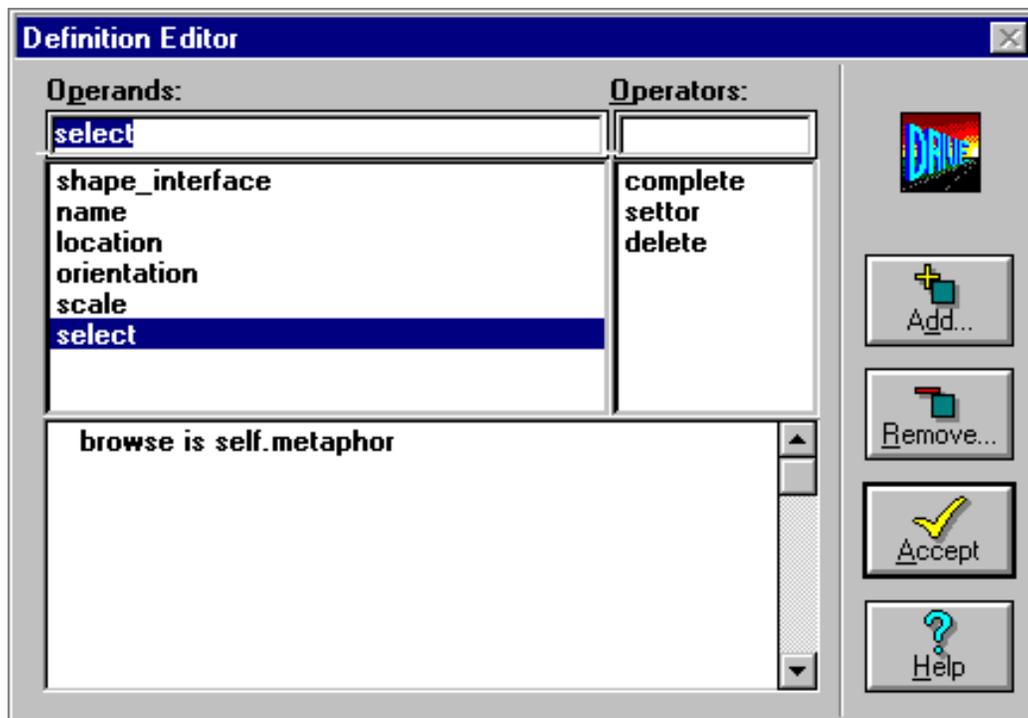

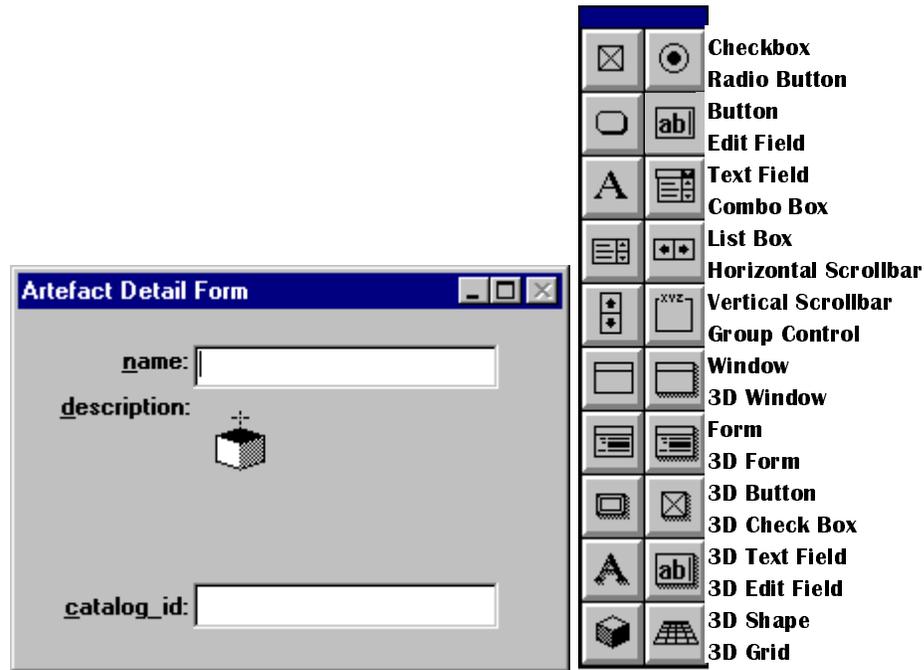**Figure 8 Context sensitive NOODL definition editor**

Figure 9 Drag'n'drop placement of an edit field

7.3 Direct Manipulation of User Interface Widgets for Editing

With each environment class registered, a corresponding visualisation class is added to the NOODL IDS specification and the design environment manages an associated palette of widgets that may be added to the IDS. This is activated when a developer enters designer mode. Each widget may be dragged and dropped into an appropriate user-interface window. This includes the creation of new windows and dialog boxes (by dragging onto a non-composite area). Figure 9 depicts the creation of an EditField in the artefact detail form. The edit field button on the palette has been pressed and the shaded box with cross hairs above it is the cursor used for placement. Once created, the widget's defined editing behaviour allows the designer to modify (move, size, etc.) it directly.

8   Conclusions

In this paper we have shown a database representation independent visual environment (DRIVE), which serves as a tool for using and specifying existing and prototype IDSs in a UIDE. A general philosophy of widespread mutability of IDS elements has been achieved in a dynamic interactive environment. An IDS component has been detailed, which ensures well-defined organised construction within our framework for user-interfaces to databases. User security is safe-guarded by defining the set of accessible interface objects within the IDS framework.

Tight coupling in DRIVE's design environment with NOODL data model constructs enable context sensitive editors to provide highly specialised support for novice designers ensuring the validity of designs under the IDS framework. The encapsulated editing behaviour of environment objects achieves direct manipulation of user-interface widgets.

The environment model permits dynamic integration of existing and novel user-interface objects, through independent environment objects. Such objects are linked to the application at run-time using an environment manager. If an existing widget is required which is unable to link with the environment manager, then it should be possible to provide a messaging interface, which uses the operating system's message protocol (eg. Windows messages) or a shared file messaging protocol. Indeed, dynamic reconfiguration or replacement of the IDS framework component is possible using the same technique in the reverse direction.

Currently, DRIVE supports NOODL IDS specifications integrating a range of simple 2D, 3D and video widgets. We intend investigating techniques for composing and mapping between such widgets, which may be reproduced in commercial applications. Empirical evaluations of such applications will doubtless follow.

9 References

1. G.D. Abowd & R. Beale (1991) Users, systems and interfaces: A unifying framework for interaction, HCI'91: People and Computers, 4, 73-87.

2. P.J. Barclay & J. Kennedy (1991) Regaining the conceptual level in object oriented data modelling. In: Proceedings of BNCOD (Jackson and Robinson, eds). Wolverhampton: Butterworths. 9, 269-305.

3. P.J. Barclay (1993) Object oriented modelling of complex data with automatic generation of a persistent representation. Phd Thesis. Edinburgh: Napier University.

4. P.J. Barclay & J.B. Kennedy (1994) A conceptual language for querying object-oriented data, British National Conference on Databases, 12:13, 187-204.

5. B.K.S. Software (1994) POET (Version 2.1) - Programmer's & Reference Guide. B.K.S. Software.

6. J. Boyle & K. Mitchell (1996) Embedding three dimensional graphics inside a user interface development framework, Technical Report submitted for publication. Robert-Gordon University, Aberdeen.

7. Cardelli, L. (1988) Building User Interfaces by Direct Manipulation, in proceedings of ACM SIGGRAPH Symposium on User Interface Software.

8. A. Colebourne, P. Sawyer & I. Sommerville (1993) MOG user interface builder: a mechanism for integrating application and user interface, Interacting with Computers, 5:3.

9. P. Gray & R. Cooper (1995) Thoughts on the Requirements for 3D Visualisation Systems, 2nd International FADIVA Workshop, Glasgow University. Glasgow.

10. K.J. Mitchell, J.B. Kennedy & P.J. Barclay (1995) Using a Conceptual Language to Describe a Database and its Interface, British National Conference on Databases, 13:7, 101-119.

11. K.J. Mitchell, J.B. Kennedy & P.J. Barclay (1996) A Framework for User-Interfaces to Databases, in proceedings of the International Workshop on Advanced Visual Interfaces'96.

12. M.O.P. (1994) ObjectStore : Meta Object Protocol. Object Design Ltd.

13. O.W.L. (1994) ObjectWindows (Version 2.0) for C++ - Programmer's Guide. Borland International Inc.

14. N.W. Paton, R.L.Cooper, D. England, G. al-Qaumari & A.C. Kilgour (1994) Integrated architectures for database interface development, in IEE proceedings of Computers & Digital Technology, 141:2, 73-78.

15. J.Rumbaugh (1995) Modelling models and viewing views: A look at the model-view-controller framework, Journal of Object Oriented Programming, , 14-22.

16. P. Sawyer, A. Colebourne, J.A. Mariani & I Sommerville (1995) Database object display definition and management with Moggetto, 3rd Working Conference on Visual Database Systems, Lausanne, Switzerland.

17. B. Shneiderman (1983) Direct Manipulation: a Step Beyond Programming Languages, IEEE Computer, 16, 57-69.

Appendix - Example User-Interface to Database Specification

This schema specifies the prototype IDS shown under the DRIVE UIDE in figure 2. The specification concerns a museum's database which is interacted with through a desktop virtual reality user interface. The data of the database is specified by the *Artefact* class, which holds its name, description and catalog_id. This information is displayed through the linked interface classes, *Artefact Interface, Museum Interface, Name Interface, Description Interface* and *CatalogId Interface*.

Users of this system are modelled by the class *Visitor*. Sophistication and authority properties have been omitted, because no particular sophistication or authority is appropriate to this example. The user interacts with a number of *Museum Interface* instances, through which s/he may browse the artefacts of the museum.

Each 3D window is a *Window3d* visualisation class, which is interpreted to provide a virtual environment for the visitor to navigate. *Museum Interface* object uses a *Shape* metaphor to represent the museum's building.

The museum interface is composed of a collection of *Artefact Interfaces* and single *Name Interface, Description Interface* and *CatalogId Interface* objects. Each *Artefact Interface* also uses a *Shape* metaphor and the position of its shape must lie within the bounds of the museum's shape. If the user's intention is to browse a particular artefact, then the text of the other interface objects' metaphors will be set to the selected relevant properties of the artefact interface's referent. This has the effect of showing a form describing the details of the currently selected artefact.

```
schema Virtual_Museum                               (* Framework Identification *)

class Artefact                                              (* Data Class *)
properties
   shape_interface    : Artefact_Interface ref referent      (* Interfaces *)
   name_interface     : Name_Interface ref referent          (* Interfaces *)
   desc_interface     : Description_Interface ref referent    (* Interfaces *)
   cata_interface     : CatalogId_Interface ref referent     (* Interfaces *)
   name               : Text ;
   description        : Text ;
   catalog_id         : Number


class Visitor                                              (* User Class *)
property
   accessors          : #Window_Interface ref users         (* accessor *)
operation
   browse is self.accessors.museum.artefacts.browse              (* task *)


class Artefact_Interface                                  (* Interface Class*)
properties
   referent           : Artefact ref interface ;            (* referent *)
   metaphor           : Shape3d ref shape_interface ;        (* metaphor *)
   museum             : Museum_Interface ref artefacts       (* composite *)
operation
   browse is self.metaphor.select                    (* interface action *)
constraint
   self.metaphor.position.is_inside(self.museum.metaphor.extent)
trigger
   browse =>                                      (* interface response *)
      self.museum.name_interface.metaphor.name(self.referent.name),
      self.museum.desc_interface.metaphor.name(self.referent.description),
      self.museum.cata_interface.metaphor.name(self.referent.catalog_id)

class Name_Interface                                      (* Interface Class *)
properties
   referent           : Artefact ref name_interface ;        (* Referent *)
   composite          : Museum_Interface ref name_component ;  (* Composite *)
   metaphor           : Edit2d ref name_interface             (* Metaphor *)

class Description_Interface                               (* Interface Class *)
properties
   referent           : Artefact ref desc_interface ;        (* Referent *)
   composite          : Museum_Interface ref desc_component ;  (* Composite *)
   metaphor           : Edit2d ref desc_interface             (* Metaphor *)
```

```
class CatalogId_Interface                                    (* Interface Class *)
properties
   referent           : Artefact ref cata_interface ;              (* Referent *)
   composite          : Museum_Interface ref cata_component ;      (* Composite *)
   metaphor           : Edit2d ref cata_interface                  (* Metaphor *)


class Museum_Interface                                       (* Interface Class *)
properties
   name_component     : Name_Interface ref composite ;            (* Component *)
   desc_component     : Description_Interface ref composite ;     (* Component *)
   cata_component     : CatalogId_Interface ref composite ;       (* Component *)
   artefacts          : #Artefact_Interface ref composite        (* Components *)



class Window2d                                               (* Visualisation Class *)
properties
   name               : Text ;                                       (* Active *)
   position           : Vector2d ;                                   (* Active *)
   size               : Vector2d ;                                   (* Active *)
   style              : Number ;                                     (* Active *)
   type               : Number ;                                     (* Active *)
   mouse              : Vector2d                                     (* Active *)
operations
   select             ;                                               (* Event *)
   move               ;                                               (* Event *)
   size                                                               (* Event *)


class GroupBox2d                                             (* Visualisation Class *)
properties
   name               : Text ;                                       (* Active *)
   position           : Vector2d ;                                   (* Active *)
   size               : Vector2d                                     (* Active *)
operations
   select             ;                                               (* Event *)
   move               ;                                               (* Event *)
   size                                                               (* Event *)


class Edit2d                                                 (* Visualisation Class *)
properties
   name_interface     : Name_Interface ref metaphor ;            (* Interfaces *)
   desc_interface     : Description_Interface ref metaphor ;     (* Interfaces *)
   cata_interface     : CatalogId_Interface ref metaphor ;       (* Interfaces *)
   name               : Text ;                                       (* Active *)
   position           : Vector2d ;                                   (* Active *)
   size               : Vector2d                                     (* Active *)
operations
   select             ;                                               (* Event *)
   move               ;                                               (* Event *)
   size               ;                                               (* Event *)
   commit                                                             (* Event *)
```

```
class Text2d                                              (* Visualisation Class *)
properties
   text              : Text ;                                        (* Active *)
   position          : Vector2d ;                                    (* Active *)
   size              : Vector2d                                      (* Active *)
operations
   select            ;                                                (* Event *)
   move              ;                                                (* Event *)
   size                                                              (* Event *)


class Window3d                                            (* Visualisation Class *)
properties
   name              : Text ;                                        (* Active *)
   position          : Vector2d ;                                    (* Active *)
   size              : Vector2d ;                                     (* Active *)
   location          : Vector3d ;                                    (* Active *)
   orientation       : Vector3d                                      (* Active *)
operations
   select            ;                                                (* Event *)
   move              ;                                                (* Event *)
   size              ;                                                (* Event *)
   navigate                                                          (* Event *)


class Shape3d                                             (* Visualisation Class *)
properties
   shape_interface   : Artefact_Interface ref metaphor ;        (* Interfaces *)
   name              : Text ;                                        (* Active *)
   location          : Vector3d ;                                    (* Active *)
   orientation       : Vector3d ;                                    (* Active *)
   scale             : Vector3d                                      (* Active *)
operations
   select            ;                                                (* Event *)
   move              ;                                                (* Event *)
   size                                                              (* Event *)

end
```