The YUIO Language: Supporting Evaluation and Implementation of Virtual Windows

Jacob Winther Jespersen

IT University Technical Report Series TR-2006-79

ISSN 1600-6100 January 2006

Copyright © 2006, Jacob Winther Jespersen

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN 1600-6100

ISBN 87-7949-117-0

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web: www.itu.dk

The YUIO Language: Supporting Evaluation and Implementation of Virtual Windows

Jacob W. Jespersen
IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 S
Denmark
Tel: +45 72185000
jwj@itu.dk

ABSTRACT
The *Virtual Windows* method is a systematic approach to the design of user interfaces to information systems that support users' tasks efficiently. This paper introduces *Yuio*, a complementary declarative user interface description language (UIDL) that allows a formal specification of virtual windows, and of their behavior during interaction with the user. We find that Yuio is well suited to express the essential design choices made during design and implementation of user interfaces based on the approach, and that it provides a gentle slope of complexity when refining designs.

Our work includes a tool that makes it possible to quickly evaluate virtual windows designs in practice by creating a virtual prototype. Among future work is a designer-guided transformation of a virtual prototype to a final system taking into account characteristics of a target platform.

ACM Classification: H.5.2. [Information Interfaces and Presentation]: User Interfaces; D.2.2. [Software Engineering]: Design Tools and Techniques

General terms: Design, Experimentation, Languages.

Keywords: Virtual Windows, User Interface Design, User Interface Modeling, Object Orientation.

INTRODUCTION
Design and construction of interactive systems has been a research topic for a long time with steady progress in terms of methods, models, and tools. User interface software tools [1] have evolved from toolkits to UIMSs to the model-based approaches and environments (MB-UIDE). A recurring theme in this evolution is the relationship between a system's data and the user's tasks. In particular since second-generation MB-UIDE's, such as Adept [2], the task and data perspectives have been the dominant abstractions in user interface software tools for information system development.

The Virtual Windows (VW) method [3] is a method to user interface design that has proven to be successful in practice dealing with complex systems design. It involves domain modeling (data and task) in a traditional way, e.g. data as an E/R-diagram and task descriptions in textual form. An apparent strength in the VW-method seems to stem from its focus on early graphical design of the logical parts (called *virtual windows*) that make up final screen contents. This explicit focus is in opposition to keeping screen contents in some abstract form until late in the design process before choosing the concrete graphical form.

Yuio is a declarative language designed to express the relationship between data and its articulation in the context of using an interactive system. As we will show, the design of Yuio folds several aspects of user interface specification into a single pattern that through composition can describe also complex user interfaces. Yuio fundamentally builds on the notion of types and dynamic type inheritance. We find that virtual windows are essentially type-specific articulations of data that support the user's understanding of data in a particular context, e.g. while carrying out a task.

Currently, there is no special tool support for the Virtual Windows method. Data and task models are produced on paper, or in a general-purpose tool (word processor, drawing program, etc.) as convenient. What comes out of following the VW method – the design product – is a paper-based specification of the set of virtual windows (and their composition into final graphical screens) that are required to efficiently support users in the given domain (of task and data). On this basis the designer creates the necessary dialog design.

Language as tool
We have worked to create tool support for developing user interfaces based on the VW approach. This paper introduces the user interface description language Yuio that embraces and extends the notion of virtual windows in order to support the designer's steps towards implementation. The extension primarily concerns the behavior associated with interaction, e.g. feedback from selections, and variations of the user interface based on the run-time context.

The activities supporting the design of virtual windows, specifically the domain analysis and its integration into the design process, are unaffected by our current work.

This paper is organized as follows: First, we detail the Virtual Windows method and why we find it interesting to introduce tools to support it. Then we explain the architecture of a run-time environment that allows running *virtual*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'05, October 23–27, 2005, Seattle, Washington, USA.
Copyright ACM 1-59593-023-X/05/0010...$5.00.

prototypes based on Yuio specifications. (By 'virtual prototype' we mean a functional prototype running on a pseudo platform.)

In the sections following we introduce the language and give several examples, and end with related work, which is primarily the model-based UI development approaches [4].

THE VIRTUAL WINDOWS APPROACH

According to its authors, the important insight supporting virtual windows is that user interface design must consider task and data at the same time. Giving a single perspective the dominant position is likely to cause undesirable effects. A too task-oriented design results in the 'soda-straw effect': it is hard to get an overview if each window shows only what is strictly needed for a task and thus hides the task's context. A too data-oriented design mechanically aligns windows with the data model's entity boundaries, which may require the user to inefficiently navigate back and forth to get to the data needed for a task. Good user interface designs balance the task and data perspectives properly; the VW method aims to make it easier to reach the right balance.

In Lauesen[3] is this description:

"A virtual window is a picture on an idealized screen. With a PC application, think of it as a GUI window on a large physical screen. This window shows data but has no buttons, menus, or other functions. For devices with specialized displays, think of the virtual window as a picture on a large display; for Web systems, think of it as a page or frame. A complex application needs several virtual windows."

As said here, the graphical expression of data intentionally takes precedence over functions. Which functions should be available to users in each window, and the choice of syntax for function activation, is part of a later dialog design step.

A virtual window articulates data from one or more data entities in order to support one or more tasks. Likewise, each task may require one or more virtual windows on screen simultaneously. The most difficult part of the design method is identifying the required virtual windows. A handful of guidelines steer this high-level design: minimize the number of window types, minimize the needed number of window instances (per type) for each task, avoid showing the same data in more than one window, and keep the size of the virtual windows within the limits of the likely target platform(s). See Lauesen [3] for a discussion of the human factors in support of the guidelines.

Designers make compromises

It is seldom possible to respect all of the guidelines when doing a design; conflicting demands almost always require deliberate compromises. Consequently, a designer needs to revise the virtual windows several times during the design phase as he discovers the consequences of a compromise. The iterative approach to refine the design by continually testing if it deals well with realistic (and extreme) data from the domain is important to achieve a good design.

Advanced graphical display of data is not prohibited by the virtual windows method. On the contrary, designers are encouraged to experiment with graphical representations of domain data and go beyond form fill-in. *Understandability tests* are the means by which the designer checks if his graphical virtual windows (with an advanced display or not) fulfill their mission to convey domain data. The designer tests understandability by showing the virtual windows one by one to a relevant user and asking her/him to explain what they show. Windows must show realistic and coherent data to give the user a fair chance of interpretation.

Design steps

All in all, the Virtual Windows method prescribes five overall steps. The following section gives a quick overview by the example given in [3]: supporting a hotel receptionist managing guests, rooms and services. Illustrations are reused with permission from the author.

Make a task list

In this step the designer describe session tasks and subtasks with variants. In a hotel the receptionist must accomplish these tasks:

- Book guest
- Check-in guest
- Change a guest's room
- Check-out guest
- Enter breakfast list

Subtasks and variants are omitted for brevity.

Make a data model

A data model expresses what data the hotel system stores in terms of entities, their attributes and relationships.

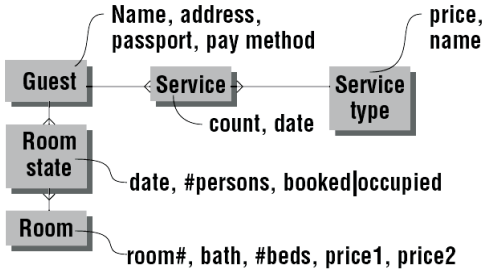

Figure 1. Data model as E/R diagram

Outline virtual windows

A virtual window *plan* shows the relation between tasks and virtual windows in an outline. In this case, *outline* means an abstract form without concrete data presentation.

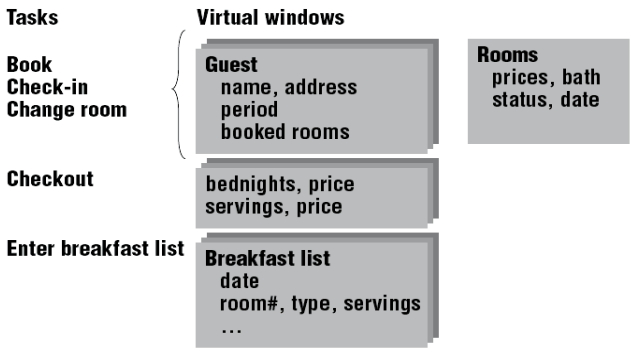

Figure 2. Plan that links tasks to virtual windows.

Detail the graphics and populate windows

A concrete presentation is the goal of this next step. The designer chooses widgets and shows realistic data for each virtual window. Notice that the Guest and Breakfast windows have a 'multiple sheet' signature, which means that several instances of these windows may be visible to the user when the system runs. But there can only be a single Rooms and Service Charges window shown at any time.

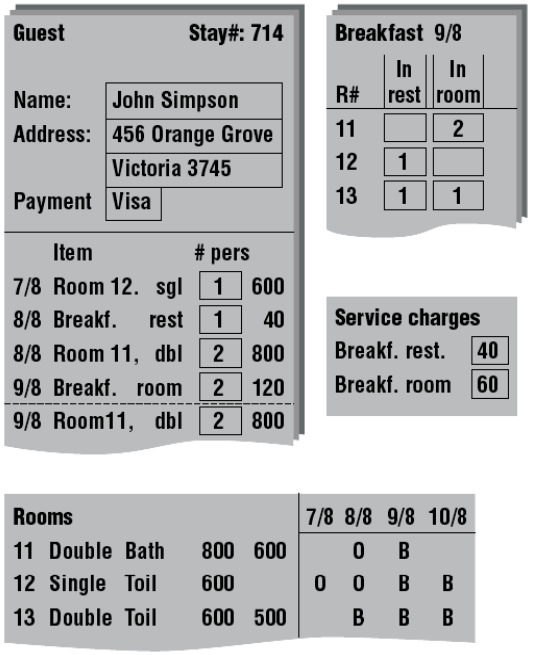

Figure 3. Virtual windows in final graphical form.

Check the design

In the final step the designer checks completeness (task/data coverage) and understandability, i.e. do users perceive the intended concepts satisfactorily? In case defects are found, the designer should revise the virtual windows.

Dialog design

In our line of work, dialog design usually involves function design (what actions should be available to the user in each window), navigation design (how does the user control the flow among windows), and user guidance design (help/messaging). Navigation and guidance issues depend heavily on the target system platform, which is why the integration of virtual windows, functions (including navigation), and guidance facilities should take place knowing the platform's characteristics.

When this final design step is taken the design is fine-grained enough that it is possible to do a usability test based on a prototype; the defects found here feed into the design cycle and inform the next iteration, if one is necessary.

VIRTUAL WINDOWS BENEFITS

Individually, the elements of the virtual windows design method are known and recognized by the HCI community [5]. It is firmly founded on task analysis, short-cycled iterative design, and usability evaluation. However, as its authors also point out, we find that the VW approach introduces three valuable variations of common elements in a design phase:

Early graphical design

Articulating pieces of related data and forming logical windows early on gives the designer a good feel of the design artifact, which in turn lets her or him make informed design compromises. Giving priority to a concrete manifestation of data (its graphical appearance) and letting this inform the delineation of the abstract outline is important.

Continuous checks with realistic data

This point is a consequence of the former; it is impossible to judge the graphical design if it does not show realistic data in the text fields, menus, or other kind of articulation chosen. Representative data must be part of the design environment to inform design decisions.

Late design of functions

Finding the right articulation of data (that the functions work on) should have first priority; function design follows data articulation, not vice versa. Postponing function design achieves this effect.

OUR MOTIVATION

The primary motivation for the work presented here is to facilitate realistic or close-to-realistic evaluation of user interface designs based on virtual windows. First of all to inform the design process (from initial sketches to understandability tests), secondly to produce prototypes that can be used for usability evaluation. Thirdly, to form a basis for transformations to final systems. The transformation part is not covered in this paper.

We set up a number of initial requirements for our tool design. They are listed here with a comment suggesting their most important rationale:

Acknowledge the virtual windows approach
to leverage the benefits of the design method

Facilitate prototyping of virtual window designs
to support usability evaluations

Allow short cycles of design-test-evaluate
to promote iterative development

Support abstraction (design-time and in domain)
to handle large and complex data sets

Declarative specification style
to avoid complexity of explicit flow-of-control

Two-way constraints link data and articulation
to provide transparent data/view synchronization

Implicit input event handling
to hide the complexity inherent in asynchronous events

Most of these requirements are motivated by existing research in design and implementation of interactive systems, e.g. [6], [7], [8], and have been addressed in a number of tools; however, none of them have acknowledged the virtual windows design method. On its own, this requirement spawns additional expectations:

- The VW semantics must be maintained during design iterations

- Specifications must be robust to frequent changes as a consequence of incremental refinement
- Support for advanced graphical displays. So-called 'black' boxes, i.e. screen regions that an application controls directly, must be avoided.

We do not detail how the individual requirements are met; in the following we prioritize a coherent description of the tool design.

LANGUAGE AS TOOL

Owing to its graphical nature, the Virtual Windows approach calls for graphical tool support. The most natural way of designing graphical appearance is 'by demonstration', so to speak. However, as shown earlier, VW design is not exclusively about appearance but about appearance in relation to a meaning given by some underlying data.

One may sketch appearance freely, and then infer a suitable data model as suggested in [9], or alternatively, adopt a data model first and design appearance accordingly. Design practice often ends up being a mix of both; domain conventions dictate certain models of domain data while other concerns (for security, data persistence, performance, etc.) and the designer's preferences motivate final data modeling.

However, establishing the relationship between data and appearance in the context of use is a necessary step towards system implementation even if either data or appearance may have precedence during the design. We use the term *articulation* to mean the use-dependent relation between data and appearance that is characteristic of an interactive system.

Since articulation depends on conditional checks of data values at runtime, and since we want to provide abstraction mechanisms to deal with complex domains, it follows that the kind of virtual window specification we seek is close enough to being programming that a suitable diagramming technique is non-trivial [10]. As a consequence, the Yuio language that we propose here is a textual language for user interface specification. Future work may integrate visual tools to specify inherently visual features of such specifications.

Tool for virtual prototyping

This section outlines the architecture of the pseudo platform we have built (in Java) to realize *virtual prototypes* based on Yuio specifications. It also acts as a straightforward user interface development environment (UIDE) as it allows the designer to edit specifications and see results immediately. We label it a 'pseudo platform' as it has limited capabilities and (with a few exceptions) does not integrate with the underlying real (Java) platform widget set.

We have not designed Yuio with only prototyping in mind; the intention is to provide a language for expressing user interfaces based on virtual windows per se. Yuio cannot and should not express complete systems either. The scope of a Yuio specification depends on the environment that runs it; this paper describes the particular environment we built to facilitate prototyping.

A virtual prototype is a stand-alone application with its main window titled *Workspace*. The Workspace is where the effects of Yuio programs show; at any point the designer can see and edit the running program in a separate *Editor* window. During usability evaluations the designer usually keeps the Editor window closed.

When a virtual prototype runs, it integrates a Yuio program, a typed data layer, and a set of articulation primitives. The latter mostly have names similar to widgets found in user interface toolkits, such as Form, List, Table, and so on, but as is explained later on, they are not simply widgets.

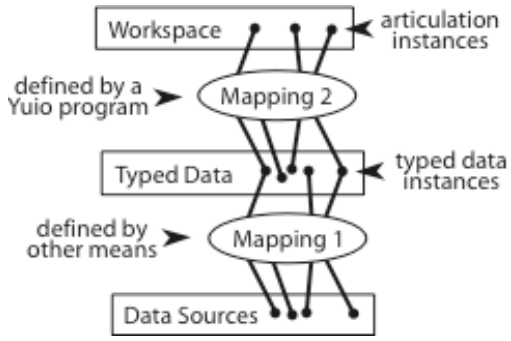

Figure 4. Yuio programs articulate typed data

In practice, data sources are data object graphs, application data structures, databases, or any other kind of structured data. Figure 4 shows the runtime architecture of a virtual prototype. *Mapping 1* takes place independently of the Yuio program by means dependent on the data sources; we use a *data interactor* abstraction to shape this mapping. For example, a data interactor takes data describing a guest (e.g. an object of class Guest, or a row in a Guest table) and forms an instance of type *Guest*. It also establishes instance *attributes*, e.g. the attributes of a Guest (Name, Passport, etc.). Instance attributes are themselves considered typed instances.

On the basis of such typed data instances a Yuio program defines *Mapping 2* which results in instances of articulation types. These show up in the workspace and provide an interactive environment for the user, i.e. they handle user input events (e.g. click, drag, and type) and manipulate the typed data layer. If the data interaction (Mapping 1) is bidirectional, then this manipulation may cause changes to the underlying data sources.

YUIO LANGUAGE BASICS

At runtime, a virtual prototype *continually* exchanges data with its data sources, and instantiates articulation types according to a given Yuio specification. Yuio specifications express the relationship between data and its articulation in the context of using an interactive system. The language builds on the notion of types and dynamic type inheritance to support specification of virtual windows and their behavior in use. The basic premise is that virtual windows are essentially type-specific articulations of data that support the user's understanding of that data in a particular context, e.g. while carrying out a task.

The typed data layer

A Yuio specification relies on a typed data layer, but it has no language facilities to establish neither the layer nor the mapping to it (*Mapping 1* in Figure 4), thus the runtime environment must provide and maintain the layer.

The logical organization of the data layer corresponds to the domain data model – and can be derived from it – but imposes a certain abstraction on the entities, attributes and relationships. Just as an E/R diagram abstracts away the details of how entity relationships manifest (uni-directional, bi-directional, or inferred), the Yuio typed data layer abstracts away the difference in manifestation of entity attributes and entity relationships. Attribute and relationship is unified in a single abstraction: *slot value*. In this view, a given type of entity has a number of slot values, each being either an attribute *or* a relationship to another entity.

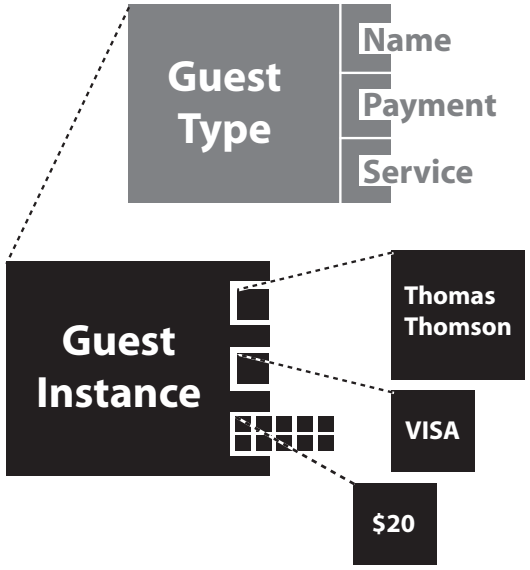

Figure 5. Yuio's data abstraction unifies attributes and relationships into slot values.

By comparing the illustration in Figure 5 to the data model in Figure 1, it can be seen that the attributes Name and Payment exist as slot values in the same manner as the relationship to a number of Service instances. Consequently, in a Yuio specification, the designer is able to refer to the slot values by name (Name, Payment, Service) without caring for the underlying manifestation, which is either an attribute or an entity relationship.

Note the formal distinction between *slot* and *slot value*. For example, an instance of Guest *always* has a Service slot, but not necessarily any Service slot *values*. Service slot values represent a guest's use of hotel services, so if the guest has just arrived, there won't be any.

In informal descriptions it is usually not necessary to distinguish between slot and slot values; in the following we use the term *attribute* informally to mean a slot with values.

Virtual Window as independent articulation

In our understanding a graphical virtual window defines an *independent* articulation for a single type of data. By independent is meant an articulation that abstracts away a concrete context for its data. For example, the *Rooms* window shows how to present instances of type Room collectively as a table. On its own it does not tell in context of which task(s) this presentation is useful.

Not every type of data needs an independent articulation; in the hotel system neither *Room state* nor *Service* has an independent articulation. Data from these entities has only *dependent* articulations, i.e. they are integral parts of other virtual windows.

In our perspective, the essence in the virtual windows approach is about (1) identifying which types need an independent articulation, and (2) finding proper dependent articulations of their associated types. The design of Yuio adopts this essential structure. Going from domain abstractions to user interface constructs, known in the user interface modeling community as *the mapping problem* [11], is with Yuio first and foremost a matter of declaring how types from the data layer map to *articulation types* (that represents user interface constructs) dependent on *context*. Secondly, it involves explicitly sharing state between instances of articulation types. In the following sections we will detail what constitutes context, and we describe the nature of the type mapping.

ARTICULATION TYPES

In the Yuio language named types represent the possible concrete user interface constructs. These types are *articulation types* whose semantics stem from the use environment (Element, List, Table, Form, Choice, etc.), whereas the *data types* discussed so far are abstractions from the domain (Guest, Room, Service, etc.). We use the hotel case to illustrate the language. A complete Yuio specification of the virtual windows in Figure 3 is in Appendix A.

Primarily, Yuio specifications express rules of dynamic type inheritance, more specifically of articulation type inheritance. The basic idiom is:

```
do <data type> as <articulation type>
```

which unconditionally binds a particular type of data to a particular type of articulation. 'Do' and 'as' are keywords of the language. As an example, consider the following statement: `do Guest as Form`

By this specification, whenever data of the type 'Guest' *comes into focus*, it inherits the abilities of the articulation type *Form*; logically, the Guest type dynamically acquires Form as its supertype. Since Form is a kind of articulation, the guest data may now appear visually. The actual effect is dependent on the environment; when a Yuio program runs in a virtual prototype, data of articulation types show up in the Workspace window. How data come into focus, and leaves again, is discussed later on.

The following less trivial example is a specification of the virtual window 'Guest' of Figure 3. Line numbers are added just to allow references in this text:

```
 1:   do CommonGuest as Form {
 2:      do 'First Name'* as Label {
 3:         do 'First Name' as Choice
 4:      }
 5:      do Address* as Label {
 6:         do Address as Choice
 7:      }
 8:      do Payment* as Label {
 9:         do Payment as Choice
10:      }
```

```
11:      do Service* as List {
12:         do Service as Row {
13:            do Date as Row.Column
14:            do 'Service Type' as {
15:               do Name as Row.Column
16:            }
17:            do Count as Choice
18:            do Price as Element
19:         }
20:      } with (.Title: "Charges")
21:   }
22:   do Guest as CommonGuest
```

Figure 6. Specializing a Form to make it a Guest form

For the sake of clarity we use the verbose syntactical form in the code examples. There are shorter versions available.

The effect on running this example is shown in Figure 7. It introduces several aspects of the language that we will seek to motivate in the following sections:

Linear composition
Statements are composed in a linear fashion to articulate the individual slots of an instance, e.g. lines 13-18 address the attributes of a service.

Specialization through hierarchical composition
Statements are composed hierarchically to specialize a particular articulation of a type, e.g. lines 2-20 comprise the statements (surrounded by curly brackets) that specialize the outermost statement in line 1.

Type slots are first-class citizens
Statements articulate individual types as well as their slots in a uniform way. As described earlier, a slot is a placeholder for a slot value. Syntactically, appending an asterisk to a type name denotes the slot for that type.

Articulation types may be interdependent
An articulation type may be dependent on another type to provide context for the articulation. One example is in lines 13 and 15 that use the *Row.Column* type. Here, line 12 provides the required *Row* type scope.

Before diving into the details with these aspects, we will introduce the conceptual framework we draw on to cope with the intrinsic complexity of user interface specification.

Figure 7. Articulation resulting from a guest instance

DEALING WITH COMPLEX DOMAINS

Yuio provides language mechanisms to break complex domains into conceptually simpler parts to ease specification. Most importantly, the designer can express single-inheritance relationships between articulation types. As an example continuing the hotel system design in Figure 6, consider that returning guests and VIP guests need special treatment.

```
23:   do ReturningGuest as CommonGuest {
24:      do 'Last Stay'* as Label {
25:         do 'Last Stay' as Element
26:      } with (.XPos:248, .YPos:0)
27:   }
28:   do VIPGuest as ReturningGuest {
29:      do 'Preferred Room'* as Label {
30:         do 'Preferred Room' as Choice
31:      } with (.XPos:210, .YPos:26)
32:   }
33:   do Guest where ('Last Stay' != Void)
34:      as ReturningGuest
35:   do Guest where (SpentLast12Months > 2000)
36:      as VIPGuest
```

Figure 8. Articulation types may form inheritance trees

The example in Figure 8 shows the use of the 'where' keyword to guard a statement by criteria. In this case, the guard causes special treatment of a Guest element based on the actual values of the attributes *Last Stay* and *SpentLast12Months* that exist at runtime. The effect of lines 33-36 is that Guest elements inherit a particular articulation type dependent on their attributes. A ReturningGuest is modeled as a special kind of CommonGuest, and VIPGuest as a special kind of ReturningGuest. Figure 9 shows the result of the specification when articulating a VIP guest.

Guards that depend on a single boolean attribute may be prefixed, e.g.:

```
do (isCheckedOut) Guest as SomeType
```

Note that the attributes *isCheckedOut, Last Stay, Preferred Room,* and *SpentLast12Months* used in the last two examples are not part of the initial data model in Figure 1, but exist in the virtual prototype we built to provide illustrations for this paper.

Figure 9. Result of specializing the guest form to fit VIP guests

Object-oriented features

Inheriting properties from a 'parent' abstraction, as is done in the example, is a classical feature of object orientation. We draw on this conceptual framework to promote clarity, reusability, and maintainability of Yuio specifications; however, Yuio is not an object-oriented language in the traditional sense. Throughout this paper we note the heritage of this tradition and point out relevant differences.

The basic inheritance mechanism in Yuio follows a common single-inheritance OO scheme. A number of articulation base types exist: Form, Label, Element, Choice, List, Table, Map, Button, Row, Column, etc. As usual, base types are a special. Yuio base types are special since a side effect of their instantiation is some sort of manifestation in the user interface. The details here follow later.

Linear composition

We think of a Yuio specification as a contract between the user interface designer and a technology provider. As such, a Yuio specification details *what* should take place in the user interface, not *how* it should take place. Thus, the sequence of statements does not imply flow of control, but (1) the sequence of evaluation and (2) the layout sequence of elements. Treatment of these two aspects differs in order to serve different purposes.

First of all, to allow incremental refinement it should be possible to specialize behavior at a certain point in a specification without changing existing statements. Secondly, the order in which elements appear in a user interface is important (a Name entry field usually appears before an Address entry field) and tightly bound to domain conventions. We achieve the desired effect by evaluating statements backwards, but maintaining layout sequence based on forward first-occurrence. This way specialization is a simple matter of extending a specification, and it is done without spoiling the ability to spot the layout sequence. Thus, in the example below (Figure 10), line 3 does not change the layout sequence.

```
1:  do Name* as SimpleInputField
2:  do Address* as SimpleInputField
3:  do Name* where (size of Name >100)
4:     as LongField
```

Figure 10. Specializing treatment of Name instances dependent on their length

The layout sequence remains (Name, Address) but whenever a given Name instance has a length larger then 100, the corresponding articulation type instances are (LongField, SimpleInputField) and not (SimpleInputField, SimpleInputField).

Specialization through hierarchical composition

In the trivial first example, *do Guest as Form*, it is obvious that further specialization is required. It is not sufficient to show a guest element as just any form; it has to be a guest form, i.e. a form that reveals the necessary attributes of a guest. As Figure 6 shows, this kind of specialization is done by placing a statement block (within curly brackets) immediately after the name of the articulation type, in casu *Form*. The context for the block specializing the Form is guest data, thus the statements in the block may refer directly to Guest attributes in order to articulate them individually, as it is done in the example.

Since Yuio supports inheritance among articulation types, as described earlier and shown in Figure 8, the type on which we base a specialization may itself be a specialized type. This is a common case in object-oriented modeling, and is supported in Yuio as well; the new type is logically a linear composition of its supertype and the specializing statement block.

Type slots are first-class citizens

When we studied the relationship between data and articulation in user interfaces of commercial information systems (most frequently built using conventional user interface toolkits [12]), we found a strikingly simple pattern: If type-dependent articulation takes place, it either wraps or juxtaposes any corresponding instance-dependent articulation. Type-dependent articulation is commonly a label, a wrapping of a list, or the name of a table column; instance-dependent articulation is then the data entry field next to a label, or the elements of a list or column.

This observation leads us to promote type slots to first-class citizens of the language and in this way be subjects to articulation just as the slot values are. Syntactically, an asterisk following a type name signifies a slot.

Articulation types may be interdependent

Often, an articulation type existentially depends on a certain context given by another type. As in Figure 6, a Row.Column is well defined only within the scope of a Row. If there is no Row, there can be no column in that row. Semantically, such dependent types are similar to the OO concept of nested classes (called inner classes in Java); in Yuio specifications nested types are useful to establish relationships across a hierarchically organized articulation. There are no restrictions on the number of nesting levels. Base types have a fixed set of nested types, but as with any type, sub-typing can manipulate this set.

The following is a simple example of a nested type. The articulation type *DoubleLineInput* is declared as a subtype of the base type Label with an extension that declares two nested types *FirstLine* and *SecondLine* each with the base type Choice as supertype.

```
 1:  do InputField as Label {
 2:     do FieldValue as Choice
 3:  }
 4:  do DoubleLineInput as Label {
 5:     do FirstLine as Choice
 6:     do SecondLine as Choice
 7:  }
 8:  do Guest as Form {
 9:     do Name* as InputField
10:        with (Name as FieldValue)
11:     do Address* as DoubleLineInput {
12:        do Address where (index = 0)
13:           as FirstLine
14:        do Address where (index = 1)
15:           as SecondLine
16:     }
17:  }
```

INTERACTIVITY

The foundation for interaction in a user interface based on a Yuio specification is the manifestation that is due to instantiation of the base types. As stated earlier, Yuio base types are special as their instantiations cause some kind of manifestation in the use environment.

When run in our virtual prototype environment, base type instantiations appear visually in the workspace window where the user can point, type, click, and drag. The workspace window provides a 2-dimensional space for the appearance in which each instance can occupy a region. In such a use environment, the workspace easily provides a mapping between the user's actions and the corresponding base type instances based on 2-d coordinates of the pointing device. Other mapping schemes are possible dependent on use environment characteristics.

User interface tools have traditionally dealt with dialog control in 3 ways: by state diagrams, based on a grammar, or as input events triggering event handlers. The event-based model is inherently asynchronous and often quite complex to configure, but it is also the most expressive.

Yuio's dialog control facilities are a mix of state diagrams and event handlers. At the language level a user's actions surface as attributes of base type instances, and are available to guard statements:

```
do Guest as Element
do (hovered) Guest as Element
   with (.Color: "Red")
do (clicked) Guest as SelectedGuest
```

With this specification Guest instances show up as plain elements, turn red when pointed at, and become a SelectedGuest instance when clicked. (The SelectedGuest type is not shown in the code.)

JOINING THE PIECES

In this last section, we will attempt to provide an overview of our work, and how the pieces fit together.

The design of Yuio adopts the notion of virtual windows, which are domain dependent graphical presentations of data organized to provide a proper framing of users' domain tasks. Programmatically, Yuio specifications articulate domain data to form virtual windows on the basis of a dynamic type mapping. The designer controls which mappings take place during interaction with the user by expressing rules that depend on domain data and on abstractions of the user's actions (e.g. clicks). Rule evaluation takes place at the discretion of the implementation.

Logically, rules specify when domain data dynamically acquires capabilities of articulation types, thus become articulation instances. Such instances have as side effect that they manifest in the user interface. There, they are subjected to user's actions (clicking, dragging, typing, and hovering), which may cause them to change state. Changing state may subsequently cause another type change.

Functions exist as type slots. Yuio lets the designer articulate slots as easily as slot values; a common articulation of a function is the *Button* and *Link* articulation types.

Getting into focus

As stated earlier, a Yuio specification details what happens when domain data comes into focus, but we haven't yet explained what this means. *Focus* is a concept that covers the domain data instances that have an independent articulation at a point in time during use. As an example, consider again the hotel system case. It is likely that the receptionist quite often will have more than one guest on screen at a time, i.e. have several instances of the guest window open at a time. Either because tasks are interleaved (a check-in *and* a booking), or perhaps when checking in a family.

In this case, some number of Guest instances constitutes Focus. Data instances move into Focus either at the request of the user, or by an external request (e.g. an application notification). Data instances move out of Focus either at the request of the user (e.g. hitting a close box), or because an instance disappears from the data layer.

RELATED WORK

Yuio shares the ambition to raise the level of abstraction in user interface development with the model-based tools. In comparison with these tools, Yuio specifications most easily compares with the *concrete user interface specification* [13]; however, in most MB-UIDEs the concrete specification is more or less a derivative of explicit abstract component models (among them usually a task model). MB-UIDEs promote multiple models so designers can address relevant design dimensions individually [14], but there are also concerns that the segregation and resulting complex model integration hinders efficient specification [15].

ITS [16] is a famous early model-based tool which employs *style rules* to control presentation and interaction style dependent on the runtime environment in terms of *dialog attributes* and the *data pool*. Yuio's rule evaluation achieves a comparable effect as an integral part of an object-oriented framework, whereas ITS' rules have a custom conceptual basis. Yuio's typed data layer features the *type slot* concept but is otherwise logically like the ITS data pool.

CONCLUDING REMARKS

This paper has described the constituents of the conceptual framework supporting the Yuio language: virtual window semantics, object-oriented type declaration, and dynamic type inheritance. Details of layout facilities, dialog control, and articulation base types have been omitted for brevity.

Even if Yuio as a language is still progressing, we believe its current position contributes to the research into efficient means for user interface development. Conceptually, Yuio strengthens the role of the concrete user interface specification. First by giving it clear semantics based on the notion of virtual windows, next by allowing descriptions in a human-readable declarative language, and finally by using proved object-oriented techniques to facilitate its specification.

Object orientation is characterized by the ability to use the same conceptual framework during analysis, design and implementation. This is useful as it allows development teams to re-use modeling and knowledge through these phases. In this perspective, Yuio represents a continuation of domain modeling, but at a higher though still concrete level.

REFERENCES

1. Myers, B.A., *User Interface Software Tools.* ACM Trans. Comput.-Hum. Interact., 1995. **2**(1): p. 64--103.
2. Markopoulos, P., et al. *Adept - a task based design environment*. in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*. 1992.
3. Lauesen, S. and M.B. Harning, *Virtual Windows: Linking User Tasks, Data Models, and Interface Design* IEEE Softw. , 2001 **18** (4): p. 67-75
4. da Silva, P.P. *User interface declarative models and development environments: A survey*. in *DSV-IS2000*. 2000. Limerick, Ireland: Springer-Verlag.
5. Maguire, M., *Methods to support human-centred design* Int. J. Hum.-Comput. Stud. , 2001 **55** (4): p. 587-634
6. Myers, B.A., et al., *Garnet: Comprehensive support for graphical, highly interactive user interfaces.* Computer, 1990. **23**(11): p. 71-85.
7. Singh, G., *Requirement for user interface programming languages*, in *Languages for developing user interfaces*, B.A. Myers, Editor. 1992. p. 115--123.
8. Zanden, B.V., *An active-value-spreadsheet model for interactive languages*, in *Languages for developing user interfaces*, B.A. Myers, Editor. 1992, Jones and Bartlett.
9. Landay, J.A. and B.A. Myers, *Interactive sketching for the early stages of user interface design*, in *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1995, ACM Press/Addison-Wesley Publishing Co.: Denver, Colorado, United States. p. 43-50.
10. Blackwell, A.F., et al., *Cognitive Factors in Programming with Diagrams.* Artif. Intell. Rev., 2001. **15**(1-2): p. 95-114.
11. Puerta, A. and J. Eisenstein, *Towards a general computational framework for model-based interface development systems* in *Proceedings of the 4th international conference on Intelligent user interfaces* 1999 ACM Press: Los Angeles, California, United States p. 171-178
12. Myers, B.A., S.E. Hudson, and R. Pausch, *Past, Present, and Future of User Interface Software Tools.* ACM Trans. Comput.-Hum. Interact., 2000(7): p. 3-28.
13. Szekely, P., *Retrospective and Challenges for Model-Based Interface Development*, in *Computer-Aided Design of User Interfaces*. 1996, Namur University Press.
14. Szekely, P.A., et al. *Declarative interface models for user interface construction tools: the MASTERMIND approach*. in *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*. 1996: Chapman \& Hall, Ltd.
15. da Silva, P.P., *Object modelling of interactive systems: The UMLi approach*, in *Department of Computer Science*. 2002, University Of Manchester.
16. Wiecha, C., et al., *ITS: a tool for rapidly developing interactive applications.* ACM Trans. Inf. Syst., 1990. **8**(3): p. 204-236.

APPENDIX A

```
do Guest as Form {
   do 'First Name'* as Label {
      do 'First Name' as Choice
   }
   do Address* as Label {
      do Address as Choice
   }
   do Payment* as Label {
      do Payment as Choice
   }
   do Service* as List {
      do Service as Row {
         do Date as Row.Column
         do 'Service Type' as {
            do Name as Row.Column
         }
         do Count as Choice
         do Price as Element
      }
   } with (.Title: "Charges")
}

>>  as Table {
   do Date as Table.Column
   do Room as Table.Row {
      do Room# as Element
      do Bath as Element
      do Bed# as Element
      do Price1 as Element
      do Price2 as Element
      do RoomState as {
         do Date as silent Table.Cell {
            do RoomState where(State="Occupied")
               as Element with (.Title:"O")
            do RoomState where(State="Booked")
               as Element with (.Title:"B")
         }
      }
   }
}

do ServiceFinder as Form {
   do 'Date Criteria'* as Label {
      do 'Date Criteria' as Choice
   }
   do Service* as List {
      do Service as silent Row {
         do Room as Element
         do BreakfastInRestaurant as Choice
         do BreakfastInRoom as Choice
      }
   }
} with ('Type Criteria'="Breakfast")
```