Chapter 1

VISTO: A MORE DECLARATIVE GUI FRAMEWORK

Kris Aerts and Karel De Vlamincx – (*Kris.Aerts@kuleuven.ac.be*)
Katholieke Universiteit Leuven, Flanders, Belgium

Abstract By inverting the widget - call back relationship and by providing selectors, a suitable abstraction for concrete widget choices, Visto provides a more declarative approach to defining user interfaces, building on the declarativeness of the lazy functional language Haskell.

1. CONTEXT

The functional programming research community has always paid a special interest in declarativity. Although there exists no water proof definition of what a declarative system is, a system can be called more declarative than another if it focuses more on *what* should be done instead of on *how* it should be done.

The Haskell [4] *quicksort* definition below can therefore be considered far more declarative than the equivalent Pascal or Java procedure, because the latter extensively spells out *how* the ordered sequence is to be calculated (through various variable manipulations), whereas the Haskell program simply says *what* to do: append the ordered sequence of smaller elements before the ordered sequence of larger elements.

$$\begin{aligned} \text{quicksort } [] &= [] \\ \text{quicksort } (x : xs) &= \text{quicksort } [y \mid y \leftarrow xs, y < x] \\ &\quad ++ (x : (\text{quicksort } [y \mid y \leftarrow xs, y > x])) \end{aligned}$$

Typical GUI programming is *not* declarative. It focuses on the *how* in 2 ways:

- first one chooses a particular kind of widget (a button, a menu or a key short cut, ...) instead of providing suitable abstractions,
- thereafter one adds a call back function to the widget to provide the desired functionality. In this sequence one first chooses *how* something is to be invoked (through which widget) and afterwards adds *what* is to be done (which call back function) instead of vice versa.

Visto is a prototype result of research in the functional community focused on finding a more declarative user interface definition. To this purpose we provide abstractions over widgets and insist that the GUI ('widget') is added to the functionality ('call back').

Because Visto finds its roots in textual programming, no graphical design tools have been implemented yet, although some are planned for future stages.

2. VISTO'S MENTAL MODEL

Most GUI systems consider the user interface as a collection of widgets totally independent from the application. Their programming language provides widget combinators or they implement a tool for drawing and positioning the widgets. Adding functionality is typically accomplished by attaching call back functions to the widgets. Changing the user interface then becomes a difficult task, requiring not only replacing the widgets, but also defining new call back functions, as their type depends on the widget.

Instead of adding behaviour and functionality to a user interface, Visto insists that a user interface is added to the defined functionality.

The traditional technique is also potentially dangerous: the drawn user interface may look good, but if it doesn't offer all the desired functionality, it's no good user interface. Only if we clearly identify the desired functionality of the application and know which functions should be present in the user interface, we can constitute a good interface. Visto does precisely that.

The objects in Visto and their methods describe the functionality of the application. This is the start basis of any Visto project. In a second phase the programmer identifies which methods are to be called from within the user interface. To that purpose he associates GUI invokers, such as buttons, key presses and menu items with methods. A selected method can be associated with any number of invokers to allow alternatives in the interface and to please both expert and non expert users. This also allows for easy modification: just change, add or remove some invokers and recompile!

Another requirement of any user interface is consistent information display. Visto offers support for that too. If objects wish to display information, they have to define a draw method. If they do so, Visto takes care that this draw method is applied every time the object receives a message. The development of the Visto objects is therefore automatically centred around the visualisation of information.

3. STAGES OF DEVELOPMENT

In this section we present the suggested itinerary for developing a proper Visto application. After all, Visto is more a design methodology than an advanced tool.

3.1 FUNCTIONAL APPLICATION CODE

Visto is designed for the implementation of a graphical user interface on top of (existing) Haskell programs. Its design philosophy therefore inherits many functional features, such as the requirement that every Visto method returns a return value and the use of a single state variable per object. These matters are no key issue for this conference and will be abstracted away in this paper.

We have chosen a functional source language because of its distinct declarative nature. This offers many advantages such as provable correctness, strong typing and concise, readable and maintainable code [2]. Haskell, the standard lazy functional language, was the natural choice.

Although the GUI of the finished application will be event-driven, the Haskell code of the application layer doesn't have to take this into account. It can be safely developed using traditional software methodology techniques.

3.2 VISTO OBJECT DEFINITION

Visto – for VISual State Transforming Objects – is a prototype based object oriented system [5] with special features for deriving objects from objects and for the model-view relationship. The visual aspect of Visto is important as we expect the programmer to define objects that correspond to visual *user* entities. In a word processor the *visual* model uses texts, whereas the application layer may prefer to use paragraphs, undo buffers and other logical entities hidden for the user.

We do not expect the user to work with these logical entities, but rather with *visual* objects. The Visto programmer will define a set of objects, and, for each of these objects, a number of methods for actions on the objects. These methods constitute the complete set of actions the user interface can use. Screen refreshes are automatically performed using the **draw** method.

3.3 CHOOSING INVOKERS

Objects typically communicate through messages, thereby invoking each others' methods. In Visto methods can be invoked both by objects sending messages and by the GUI itself.

The methods defined in the previous stage constitute the complete set of actions that the GUI may invoke. The next stage is choosing which actions will effectively be present. This is done by associating invokers with methods. Note that we have first considered which actions on the different objects may be needed and only afterwards decided which ones will be present in the GUI.

Because we start from the methods, it is easier to ensure that the application provides all the needed functionality and not just a nice looking interface.

The programmer can associate a single method with *any* number of invokers. This allows different alternatives to be addressed in a simple way, e.g. a menu item and a button for the first time user or a key short cut for the expert user.

Changing the user interface then becomes quite simple, like replacing a button invoker with a menu invoker. This is quite different from the traditional approach where the type and parameter count of the call back function depends on the widget used. So changing a widget may incorporate quite some work.

Following the example of J. Johnson [3] we provide an abstraction for the actual widgets, called selectors [1]. In this technique we focus on the declarative aspect of the widgets: selecting the appropriate data. Examples of selectors are:

- Selecting a single value from a range or from a discrete set,
- Selecting a specified number of values,
- Selecting between n_1 and n_2 values,
- Special interest selectors such as currency, time, ...

This stage offers two steps with an enhanced declarativity: focusing on the functionality over the widgets by adding invokers to methods and not vice versa, and abstracting away from the actual widgets by through selectors.

3.4 CREATING THE FINAL GUI: LAYING OUT AND INSTANTIATING INVOKERS

This final stage will be performed in a (not yet implemented) tool for directly drawing the user interface using direct manipulation. The tool will first analyse the Visto program and collect all the used invokers. These invokers will then constitute the precise set of things the programmer has to put in the user interface. It can be compared with a puzzle where the puzzler has all the pieces in front of him and assembles them together until all pieces fit, while being assured of the completeness of the result. The same applies to Visto. Once all the selectors have been instantiated and inserted in the interface design, the programmer is assured that the interface provides all the desired functionality. This is different from traditional tools where the programmer has little or no guidance on which widgets to use. Such an interface may then look good, but miss some essential functionality.

When placing the selectors on screen, the programmer chooses between various visual instances, such as for example an entry, a slider bar, button rows and columns, key short cuts, ... A time can be selected by combining such range selectors for hour, minutes and seconds or by manipulating the hands of a clock, ...

A different user interface with identical functionality can then be created by instantiating the selectors differently or by drawing other combinations. Changing the functionality of the user interface is accomplished by adding, removing or changing some invokers in the previous stage.

This way the semantically different changes in the user interface (changing outlook or functionality) have to take place in different stages of development, which is always a desired feature of a methodology: keeping separate decisions separate.

4. A SHORT EXAMPLE

Tradition dictates the counter example. Although this example combines many features in one object, such as state and behaviour, visual representation and invoker, which in real life should be better separated, it can still be used to clarify the framework.

```
interface Counter
  up :: Int → Int
```

```
object PlusOne Counter Int 0 {
  draw = Label
    text (show state),
  up = λ x → let s' = state + x in (s', s'),
  GUIup = Button "Increase with" << - DiscreteRange 1 10;
    Key " + " << - 1}
```

The **interface** simply states which methods an object has to provide in order to be a *Counter*: *up* which takes an *Int* with which to increase the counter.

PlusOne is a *Counter* with a state of type *Int* with initial value 0. This object supplies a **draw** method, so the state of the object is shown on a label.

GUIup gives two invokers for the *up* method: after clicking the button “Increase with” the counter is increased with the value selected by the *DiscreteRange* selector (selecting a number between 1 and 10¹), or by pressing the key ‘+’ in which case the counter is increased by 1.

These invokers have been defined completely separate from the methods. To change the user interface we don’t have to mess with call back functions, but only have to redefine the **GUIup** body.

If we consider the object *PlusOne* as the model (with already a view present in its own definition – to make the first example self-contained –), we can define another view using the **expand** derivation.

¹In the screen shot we can see that a requester is displayed because the input 11 was not in the proper range.

```

expandInterface CounterView from Counter
  -up
expand PlusOneView CounterView String "Dutch" from PlusOne {
  draw = Label
    let counter = superState
    text (translate state counter) }

```

As a view should not change its model, we remove the *up* method. This can be done in Visto by preceding with an underscore the method name in the derived interface. The view object *PlusOneView* only defines a **draw** method, which simply displays the number in textual form. We use the keyword **superState** to gain access to the model's state. Note that the view object can have a state on its own. Although we haven't defined a method to change that state here, *PlusOneView* contains the language in which to translate the number. The (self defined) Haskell function *translate* takes a language and a number and returns the value in textual form.

5. CONCLUSIONS

The Visto methodology of concentrating on the visual objects, defining methods for manipulating that (displayed) information and supplying GUI-invokers for the methods opens a brand new perspective on user interface design. Especially the move from call back functions to GUI-invokers is an important shift.

Although Visto has not reached an entirely mature phase, this fresh declarative approach deserves further research.

References

- [1] K. Aerts, K. De Vlaminck: *A GUI on top of a functional language (abstract)* Proceedings of ACM SIGPLAN International Conference on Functional Programming, 1997, p.308
- [2] P. Hudak, M.P. Jones: *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity* Available from <http://www.haskell.org/papers/NSWC/jfp.ps>, 1994, 16 pages
- [3] J. Johnson: *Going Beyond User-Interface Widgets*. CHI'92 Proceedings, 1992 , 273-279.
- [4] S. Peyton Jones, J. Hughes (editors): *Haskell 98: A Non-strict, Purely Functional Language* Report on the Programming Language Haskell 98, Available from <http://www.haskell.org/>.
- [5] W.R. Smith: *Using a Prototype-based Language for User Interface: The Newton Project's Experience* In ACM Sigplan Notices, OOPSLA, (1995), Vol. 30, no. 10, 61-72