Vis/Vapourware Interface Synthesiser

Tuomo Valkonen
tuomov at iki.fi

2004-04-11

Abstract

We propose a thin framework for describing user interface semantics of programs. A user interface matching a user's preferences can then with the help of stylesheets be generated automatically based on this description while also ensuring system-wide consistency of user interfaces and taking the burden of designing user interfaces away from the application programmer.

Disclaimer This paper was first meant to be a short essay, started out of frustration of constantly explaining my views at a general level on how the user interface component of programs should be written. As the work progressed, it however grew to the current proportions. While the paper may at parts look like a scientific paper, it by no means strives to any "scientific standards" with respect to background research, for example.

1 Introduction

There is a common misconception in the world of software these days; that usability were somehow equivalent to a program having a particular type of user interface – we all know which – that we shall henceforth refer to as WIMP.[1] However, usability is a subjective matter, and especially power user and others who prefer the keyboard for one reason or another are these days usually left out of consideration when designing user interfaces. While there may be keyboard shortcuts to all or almost all features the program provides, these are often suboptimal due to the mouse-oriented layout and types of widgets and the chaotic nature of the desktop paradigm.

It can even be contested that WIMP was somehow especially friendly and intuitive to new users; see [War04]. Indeed, the user interfaces of complicated programs such as word processors are, in our experience, too cluttered to be friendly to a new user and, yet, the features are often not conveniently enough accessible to a power user who might, for example, prefer a command-based interface, but may be forced to use an inferior one due to other reasons. We can therefore claim that WIMP GUIs are a compromise between familiarity and features at the expense of usability.

Some work has been done on alternative window managers, by the author [Vala, Valb] and others [Ber, Gar04, Trs], to make the desktop more manageable. While

1. Windows, Icons, Menus and a Pointer

tiling as in, for example, Ion [Vala] seems to help keep programs with at most one top-level window per document well organised, some programs have a design that does not fit well into this model. An example are the multiple toolbox, palette and other auxiliary windows of the Gimp[2] that are needed to edit a document in yet another window. Also some programs have trouble with the tiled-and-tabbed model because they do not strictly follow the ICCCM [Ros] and expect a conventional window management model. It is our belief that if these problems were solved, the tiled approach would be very viable alternative to the currently dominant unorganised-pile-of-papers approach on modern high-resolution display hardware. To this end, programs' user interfaces would have to be written in a different manner.

Complex user interfaces also do not adapt well to small handheld devices and may need to be specifically ported to those.[3] Making programs available to visually impaired people poses a similar problem, where the 'display device' is not capable of everything that the device for which the user interface was originally designed for is.

To solve the problems described above, we do not propose to replace the modern WIMP GUI with something better. Infact, there is no such a thing; unlike some may think, *one type of user interface isn't good enough for everyone* or every device. What we propose is a semantic description plus stylesheet based approach to automatically generating user interfaces. This way the user is free to choose the kind of UI (s)he prefers instead of being forced to use what the designers of a piece of software thought of as a "good" compromise. An approach like ours should also ensure system-wide consistency of the user interface without software designers having to resort to style guides.

While some details and the (low, actually) level of abstraction of our approach may be new, the idea of automatically generating user interfaces based on a model or description is not. We compare the proposed approach to (a far from comprehensive list of) earlier work in Section 7.

But first, we begin in Section 2 with some unifying observations of programs' user interfaces. Then in Sections 3 and 4 we give an outline of our solution to the usability problem. In Section 5 we then give a draft of a minimalist application programming interface to the end of implementing our approach. Finally, in section 6 we give a few examples of how programs could be written using this API.

2 Observations

Most application programs can be grouped in the following classes by their user interfaces:

#1 One-shot: Either fully command-line based or a rather standard-looking dialog that "doesn't do anything" until 'Ok' is pressed. (Displaying a sub-dialog does not count as doing something.)

2. The GNU Image Manipulation Program; `http://www.gimp.org/`
3. Of course porting may also be required to reduce memory footprint etc., but we do not concern ourselves with such issues in this paper.

#2 Sequential: Different kinds of (simple) 'shells' and 'wizards'. These can be thought of as a sequence of executions of one-shot tools, even if it is actually the same program and same window all the time.

#3 Interactive: Bigger applications. For each 'document' or 'model' displayed by the application, there's a 'canvas' or 'view' where it draws the visible part of this document. When the program has multiple canvases, these may either be displayed as separate top-level windows when application is running under a windowing system, multiplexed with possible tabbing depending on the user interface style, tiled or float freely inside a main window (MDI). All kinds of standard UI style dependent 'controls' may be scattered around the canvases, either single or groups of.

We have put shells and wizards in the same class. While every wizard can be conveniently implemented as a shell with just a few possible actions to take at each step, many shells (/bin/*sh, interpreters for various languages) have such a vast amount of possible actions to take at each step that it is hard to imagine any other kind of an user interface besides other kinds of text input methods and alternative parametrisations that would usably serve the purpose. These kinds of shells are better realised as applications of class #3.

Programs of class #3 may have parts that behave like classes #1 or #2. The notion of 'document' above is to be understood in a very wide sense, and includes almost all kinds of data except the most trivial that a program would want to display and that isn't directly related to a single primitive 'control'. We classify documents in the following categories depending on their requirements and ability to use different display capabilities:

- Monospace text
- Data structures (tabular material, spreadsheets, directory listings, etc.)
- 2D vector graphics including formatted text
- 2D framebuffer graphics
- 3D graphics

In this paper, the notion of 'control' is understood to include not only widgets, but also key bindings, voice control and whatnot.

3 Solution outline

As already mentioned in the Introduction, we believe that application programmers should not have to be concerned with user interfaces. Instead they should only be asked to provide a description of the commands and data – henceforth called *'accessibles'* – provided by the program that a user might wish to access. Many of the most commonly used accessibles should have standardised names and semantics. For example, any program of class #3 that provides text editing functionality in a canvas should provide commands such as **edit.movement.bol** and **edit.delete.backward_char** for the 'go to beginning of this line of text' and 'backspace' actions, respectively.

A high-level toolkit – the Vis Interface Synthesiser, vapourware for the time being – could then based on this description, with the help of modules for each type of user interface, automatically generate a user interface matching the user's preferences.

While it should be possible to generate a reasonable interface from this description alone in simple cases, more complex programs may require some assistance, depending on the UI module in use. In the spirit of well-written HTML containing only tags with higher semantic significance, this assistance should be provided in terms of 'stylesheets' containing generic usage pattern hints and other extra information on the accessibles provided by the program. Each UI module could also allow for additional fine-tuning through an extra stylesheet specific to the UI style. Stylesheets are discussed in more detail in Section 4.

As a start, we can think of at least the following different user interface styles and modules for Vis:

ui-stream A linear text-based user interface. Couldn't support many canvas capabilities (see Section 3.3).

ui-tty Accessibles are mapped to bindings and minibuffers in the style of many text editors. Also modal binding-triggered menus and some kind of configuration editing system could be provided.

ui-ion Similar to ui-tty, but graphical in the style of Ion [Vala].

ui-qt A WIMP GUI using the Qt toolkit.

ui-gtk A WIMP GUI using the Gtk+ toolkit.

ui-html Why not? Couldn't support many canvas capabilities, of course (again, see Section 3.3).

The WIMP modules could have different variations or system-wide configurable parameters affecting aspects of control and multiple document placement, as there are many different conventions for those.

In the remainder of this section we shall explain what kind of facilities Vis should provide for each of the program classes mentioned in Section 2.

3.1 Applications of class #1

An application of this class should specify a hierarchy of accessibles related to its startup, and then ask Vis to 'realise' the program's 'main context'. If a command line UI is preferred, the library would then map the command line to this specification, while a WIMP module could generate a setup dialog (or dialogs) with controls to modify the settings and an 'Ok' button to proceed with the application.

After this 'realisation' phase is done, the program should be able to read its settings from the specification. If it needs to do output or requires user interaction after this, it would have to create canvas(es) with minimal capabilities, essentially behaving like applications of class #2.

Applications of other classes may also use the mechanism above for their startup parameters.

3.2 Applications of class #2

The idea here is to create a canvas with simple, if any, capabilities and keep changing its set of accessibles according to the phase[4] of operation. This phase-switching feature could also be used to implement different modes of operation of canvases for applications of class #3.

4. The word context is used in another meaning in Section 5.

3.3 Applications of class #3

An application of this class would register program-wide accessibles in a main context for the program. Additionally, when the program would want to display a document, it should request a canvas with the necessary capabilities for displaying the document, and then specify the accessibles available on this canvas. The toolkit and the user-chosen UI module would then generate the UI controls to do operations on this canvas.

In what follows we list some possible canvas capabilities:

Input:

input-chars Single character input events.
input-strings Whole lines of text as input.
input-rawkeys Raw key input events. Usage not recommended.
input-pointer Pointing device (for games, painting programs, text selection in editors).
input-stick Joystick-like devices.

Output:

output-cstream Equivalent of 'standard output'.
output-ccell Simple character cell display; for text editors and the like.
output-dstruct For viewing simple data structures.
output-opengl For drawing with OpenGL.
output-sdl For drawing with SDL.[5]
output-cairo For drawing with the Cairo[6] vector graphics library.
output-xlib Drawing support using Xlib. Usage not recommended, although this capability might be very useful during a 'transition period', if Vis ever was reality and accepted.

3.4 Remarks

The primary reason for having the **input-strings** capability in addition to **input-chars** is to integrate IRC/IM and whatnot programs' text input area with the general user interface look and feel and to provide **output-cstream** applications with an integrated command line. Note that **input-strings** might not necessarily be implemented as such a textbox or plain old command line prompt; it could in principle be even voice recognition.

Of the basic capabilities above, **ui-tty** can obviously only support **output-ccell**, **output-cstream**, **input-chars**, **input-strings** and possibly **input-pointer**. The **ui-cstream** module could only support **output-cstream** and **input-strings** while the rest of the UI modules mentioned above could in principle support any of the above-mentioned capabilities – but not necessarily simultaneously.

While APIs would have to be designed for some of the capabilities above and the implementations for different UI styles may differ a lot, it is not the point of Vis

5. Simple Directmedia Layer; `http://www.libsdl.org/`
6. `http://www.cairographics.org/`

to replace established libraries if one sufficiently well implements a capability independent of the UI, as can be seen from the set of graphical output capabilities above.

Indeed, Vis itself would not really care what all the capabilities are; in addition to the capabilities above, one can think of high-level output capabilities for all kinds of formats: **output-html** (any UI module), **output-ps** (UIs for graphical output devices), for example. It is debatable whether there's any point in such, though. Nevertheless, in principle the only restriction should be that unless the capability requires special hardware, a generic version of the capability should be implementable on top of one of the primitive capabilities above. Of course, different UI modules could provide specially optimised versions of the capability.

4 Tips to UI modules

Applications and users could provide two types of stylesheets to Vis: a generic one and ones specific to a single user interface module. The ones specific to UI modules could be used to control all the details of control and canvas placement and key mappings up to a full UI design. These are not discussed further in this document. Instead we concentrate on the generic stylesheets and the kind of information that should help any UI module make good guesses.

We remind the reader that a 'control' is to be understood broadly to include not only widgets and such, but also key bindings and anything else a user could use to alter the flow of the program. Also, the term 'accessible' refers to any command or data item provided by the program for the user to call, modify or view.

First of all, given the kind of API drafted in Section 5 the program provides through the API only hierarchical names for the accessibles it provides. Therefore,

- The stylesheet needs to provide natural-language names for accessibles in case they need to be visualised to the user.[7]

The hierarchy inherent in the accessible names along with type information and UI design conventions should provide the UI modules enough information to make reasonable groupings of controls and selections of control types. In case of a small program it is also possible to make all accessibles quickly available behind just a few controls. In case of larger programs with vast command and data sets, this is, however, impossible without making the program totally unusable. The UI generation code must therefore make decisions on what accessibles should be made quickly accessible and what are so unimportant that accessing them can be left to be done only through by, for example, entering the name of the accessible somewhere. To this end, the stylesheet should provide some a priori usage pattern information:

- How often is the accessible expected to be used (absolutely, relative to other accessibles)
- How important the accessible is? Some of the less-frequently used accessibles may be more important to have quickly accessible than the others.
- Is the accessible most likely used by users at what level? Newbies, average users, power users or all of them?
- Is it likely that the accessible will be used in sequence with another?

7. Icons and such belong in the UI-specific stylesheet(s).

By analysing the actual usage frequencies of accessibles, the UI could also adapt to the user and, for example, suggest keyboard mappings or teach the user by informing of faster ways to access an accessible – system-wide.

The stylesheet could also contain additional information such as documentation. Also, while additional stylesheets could be used for such, localisation should perhaps be done with the standard .po-file method.

5 Design document draft

5.1 Application programming interface

The Vis core API visible to the application programmer should be only a couple of handfuls of functions and classes to set up and maintain the description of the program. In the following we give such a minimal API, restricting ourselves to UI description. An actual API would, unfortunately, also be required to include a couple extra functions to wrap POSIX poll/select[8] and timers due to limitations and requirements of back-end UI libraries. Also some extra convenience functions might be needed, but in this paper we have chosen to keep the API down to a bare minimum.

The code below uses Lua [Ier03] syntax, but, there is nothing really Lua-specific about the API. Infact, given the powerful table mechanisms of Lua, an even simpler API could be devised.

While a library implementing the API has yet to be implemented, we shall switch to present tense for most of the remainder of this section to make the description seem more natural.

Vis module classes and functions

vis.Accessible and some actual data/accessible types inheriting this. The currently planned data types are the atomic types **vis.Bool**, **vis.Integer**, **vis.Double** and **vis.String**, the 'restricted atomic' types **vis.IntegerRange(a, b)**, **vis.DoubleRange(a, b)** and, with the **vi** strings, **vis.Enum(v1, v2, ...)**. Also we have the structure type **vis.Struct**, and the **vis.Set(·)** meta-type. Structures form the very basis of Vis accessible definitions.

vis.Context Something that an UI can be 'realised' for. Inherits **vis.Struct**.

vis.Canvas A canvas on which the program can draw data on and receive input using various capabilities. Inherits **vis.Context**.

vis.init(name, datadir, args) Initialise Vis. **Name** is the name of the program, **args** its command-line parameters, and **datadir** points to directory where Vis can find stylesheets and other specification files.

vis.create_canvas(name, capabilities, related) Create a new canvas with requested capabilities, if possible. The optional parameter **related** may point to an already-existing canvas to which this one is related to.

vis.maincontext() Return a 'main context' for the program. All program-wide available accessibles should go here.

8. Systems without the equivalent of POSIX poll/select are irrelevant.

vis.mainloop() Give control of execution flow to Vis or, more specifically, the chosen UI module.

Data type methods

vis.AnyAccessible:getv() Get value.

vis.AnyAccessible:setv(v) Set value, or callback in case of a command.

vis.Struct:get(name) Return a reference to an entry within the structure or substructures of it. If there are sets in the hierarchy (see Section 5.2), the return value will be a set (of sets) that is a cross-section of the sets in the hierarchy.

vis.Struct:def(name, type) Return a reference to and define a new accessible **name** in the structure or a possibly new substructure of it. **Type** indicates the type of the entity being defined. See Section 5.2 below for details.

vis.Struct:defconst(name, type) This convenience function additionally sets the **constant** attribute to true. See Section 5.2 for details.

Context and canvas methods

vis.Context:realise() Realise the context. After this **def** can no longer be used on structures included in the context before clearing the context. Vis may require user interaction while doing this to, for example, query the user for for startup parameters.

vis.Context:clear() Clear the context.

vis.Canvas:getcap(capability) Returns a handle to a capability that must have been requested for when the canvas was created. The type of the handle is specific to the capability.

5.2 The accessible hierarchy and identifiers

To define a accessible (command or data item), **vis.Struct:def** must be passed a valid name (henceforth called identifier) pertaining to the hierarchy of the items, and a data type. In some cases the identifier also reflects the data type. In this subsection we describe those relationships.

Identifiers consist of two or three parts: (1) a 'semantic space tag' ending in a slash ('/'), (2) a hierarchy of structure entries separated by periods ('.'), and (3) in case of attributes a colon (':') followed by attribute name.

The 'tag' is used to indicate which entity has specified the semantics of the accessible that the hierarchical part (2) refers to. For standard identifiers (Section 5.3) the tag is '**std**', while for others it should be the name of the program or library that uses or provides it. Note that the hierarchies for different tags live in the same namespace.

In the hierarchical part, periods are not only used to separate structure entries from the structure, but also command parameters from the command. There may be no periods after an atomic or restricted atomic data type, or a collection of these. Note that Vis automatically creates structures as it encounters periods not after a command in identifiers, so it should seldom be necessary to define a plain **vis.Struct** directly.

Entries corresponding to commands **vis.Command** must be indicated by adding parentheses ('()') as suffix to the entry name, while sets **vis.Set(·)** must be indicated with a square bracket suffix ('[]').

Finally, identifiers ending in a colon and a proper attribute name would be used to set attributes for previously defined data. The standard attributes could include

enabled Is the accessible enabled? (**vis.Bool**)
constant Is the data supposed to be modifiable by the user? (**vis.Bool**)
changed Function to be called when a data item is changed. (**vis.Command**)

As a simple example, we have the definition of the location and history of a simple browser. For more examples, see Section 6.

```
ctx:def('std/content.location', vis.String)
ctx:def('std/content.location:changed', vis.Command)
ctx:defconst('foobrowser/content.history[].url', vis.String)
ctx:defconst('foobrowser/content.history[].title', vis.String)
ctx:defconst('foobrowser/content.history[].when', vis.Double)
ctx:def('foobrowser/content.history[].goto()', vis.Command)
```

5.3 Standard identifiers

The full list of standardised identifiers and their semantics would be long and have to be constantly maintained. Thus such a full description is out of the scope of this paper. We will therefore only give here a rough outline of a part of a possible standard identifier hierarchy

Note that because the hierarchies for different 'tags' are in the same namespace, the semantics of an accessible with a different tag but falling within the standard hierarchy would be expected to have semantics closely related to the standard ones in that sub-tree. The '**std**' tag should *not* be used for new non-standard items within the standard hierarchy.

std/settings. Configurable program or canvas properties.
std/parameters. Startup parameters, such as command line options.
std/control. Program 'flow' control commands, such as exit, close, openfile.
std/seq. Phase sequencing. Primarily used in programs with UI of class #2.
std/info. Status information. Likely to be displayed e.g. in a statusbar.
std/view. Accessibles related to what part of a document is viewed within a canvas.
- **std/view.area.**
- **std/view.rotate.**
- **std/view.zoom.**
- **std/view.walk.**

std/content. Accessibles related to document content
- **std/content.search.** Search functionalities.
- **std/content.style.** Low-level font, colour and such control for consistent selectors.

std/edit. Text and other editing accessibles. These are separate from 'content' as some are used internally for non-documents.
- **std/edit.movement.**
- **std/edit.selection.**
- **std/edit.insert.**
- **std/edit.delete.**
- **std/edit.completion.**

std/edit.history.

std/subcontext. Groups of accessibles related to contexts[9] within canvases. Activity of a context is indicated with the **enabled** attribute of an entry here.

We shall once again add that this list is far from complete and only suggestive of what could be standardised.

6 Examples

This section attempts through actual examples in Lua [Ier03] to explain how Vis with the above API could be used to implement some simple programs. The accessible names are arbitrary and the standard names should be more carefully though-out if Vis was ever implemented.

6.1 Hello World

Our first example is, of course, a simple Hello World-style program that prompts for the user's name or gets it from the command line with an appropriate UI module. The program's command-line parameters are in the table **arg**.

```
require('vis')

-- A function to create a canvas for displaying a message
local function message(msg, finishfn)
    local mcv = vis.create_canvas('helloworld/message', {'output-cstream'})
    local f = mcv:getcap('output-cstream')
    f:write(msg)
    mcv:def('std/seq.next()', vis.Command):setv(finishfn)
    mcv:realise()
end

-- Initialise vis and our 'main context'.
vis.init('helloworld', '/usr/local/share/helloworld/vis/', arg)
local pgctx = vis.maincontext()
local user = pgctx:def('helloworld/parameters.user', vis.String)
                  :setv(os.getenv("USER"))

-- Realise main context. Actual user setting should be read.
pgctx:realise()

-- Create a canvas to display the 'Hello user!' message.
message('Hello ' .. user:getv() .. '!', function() os.exit() end)

-- Give control to Vis.
vis.mainloop()
```

The generic stylesheet could look as follows:

9. Not related to **vis.Context**.

```
info{
    identifier = 'helloworld/parameters.user',
    label = 'User name',
}
```

Finally, the stylesheets specific to UI-modules that support command-line parameters could include the following information.

```
info{
    identifier = 'helloworld/parameters.user',
    short_option = '-u',
}
```

The long '--user' option could be automatically deduced.

Note that there's no need to describe standard **std/*** accessibles in the stylesheets unless we want to change some aspect of them.

Of course, the above is relatively lot of work compared to a simple stdin/stdout based Hello World program, a like of which the simple **ui-cstream** module could implement for this example, or the kind of which could be implemented using the **output-cstream** and **input-strings** capabilities. A widget-based module could, however, display a dialog prompting for the user name and then another displaying the message. For that this seems very little work and the lessened amount of work should be even more pronounced in even slightly more complicated programs, as our next example demonstrates.

6.2 UI for a simple multiple image viewer

This is an example of a hypothetical multiple-image viewer using SDL and Vis. The SDL-specific code is omitted.

The main source file would be as follows:

```
require('vis')

-- Arrange for a new range of image to be displayed in cv.
local function set_viewarea(cv, view_area)
    -- [omitted]
end

-- Open a file in a new canvas
local function openfile(mainctx, std_control, param)
    -- Open the file and read data [mostly omitted]
    local fnam = param['filename']:getv()
    -- Create a canvas
    local cv = vis.create_canvas('viewer/image', {'output-sdl'})
    -- Do some SDL-specific setup [mostly omitted].
    local sdlsurface = cv:getcap('output-sdl')
    -- Define accessibles
    cv:def('std/control.close()', vis.Command):setv(vis.Canvas.destroy)
```

```
    cv:defconst('std/info.filename', vis.String):setv(fnam)
    cv:defconst('viewer/info.dimensions', vis.String):setv(image dimensions)
    cv:defconst('viewer/info.depth', vis.Integer):setv(image depth)
    local xrng = vis.IntegerRange(0, image width)
    cv:def('std/view.area.xmin', xrng):setv(0)
    cv:def('std/view.area.xmax', xrng):setv(canvas width)
    -- Repeat the above two definitions for y.
    cv:def('std/view.area:changed', vis.Command):setv(set_viewarea)
    -- Realise the canvas
    cv:realise()
end

-- Initialise
vis.init('viewer', '/usr/local/share/viewer/vis/', arg)
local pgctx = vis.maincontext()
pgctx:def('std/control.exit()', vis.Command)
      :setv(function() os.exit() end)
pgtcx:def('std/control.openfile()', vis.Command)
      :setv(openfile)
pgtcx:def('std/control.openfile().filename', vis.String)
pgctx:realise()

-- Give control to vis.
vis.mainloop()
```

We omit the stylesheets here as they should be very straightforward and contain descriptions of the non-standard **viewer/info.*** data. Then, a conventional WIMP module, for example, should be able to generate a UI with

- Either tabbing of multiple open files or separate window for each.
- For each window, a 'File' menu in a menubar, containing entries 'Open', 'Exit' and also 'Close' when there is an open file.
- Scrollbars – or a 'panner' as in, for example, the GV front-end to Ghostscript – to change view area,
- A statusbar displaying file size, image dimensions and bit-depth, and
- A few standard key bindings.

Note that the conventional semantically flawed 'File' menu of WIMP GUIs does not directly map to our hierarchy, and thus the module generating the above UI needs to use information on WIMP design conventions in creating the 'File' menu above.

The UI generated by the **ui-ion** could differ from the above at least by:

- No visible menubar, instead a main menu better matching our hierarchy is displayed at a corner of the canvas when a standard menu display key is pressed,
- The 'scrollbars' would only be used to indicate visible area and thus small, and
- Totally different kind of, and fuller, default bindings.

7 Related work

The idea of automatically generating user interfaces is, of course, not new at all. While our background research on the matter is *far from comprehensive*[10], it is our impression that most of the older research (80's, early 90's) on the matter is concentrated on the automatic and semi-automatic generation of WIMP GUIs based on a full model of the data processed by the program. In our approach we have chosen to concentrate on only generating UI for auxiliary data, which is *already given by the programmer in a form suitable for directly generating a reasonable UI*. The primary data processed by the program would in our approach be drawn by the program on the canvases. The different canvas capabilities may then optionally support modelling that data, as **output-dstruct** would at a very primitive level.

More recently, with the advent of widespread use of mobile devices and the consequential aspiration to run the same programs on platforms with varying display and input capabilities, research seems to have picked up on the notion of 'plastic user interfaces' or 'plasticity'; see [Thé99]. This notion is more related to our approach, and at least AUI of [Sch02] indeed has some similarities to what we have proposed in this paper. However, it only supports abstract drawing of primitive graphical objects on canvases, while our approach would support established graphics libraries through 'canvas capabilities'. The capabilities would also not be limited to graphics output.

Indeed, in general the existing frameworks seem to be geared towards relatively 'toy' user interfaces compared to ours. (At this point we want to remind the reader of the limited nature of our background research.) This is understandable considering the primary goal of running the same program on workstations as well as small mobile devices, and comparing it to our primary goal of allowing users to choose the type of user interface on a workstation computer. Our approach only abstracts 'C' and some of 'V' of model-view-controller while the mainstream in both earlier and recent research seems to strive towards abstracting both 'C' and 'V'.

8 Conclusion

We have presented a framework for writing programs independent of their user interfaces and generating the user interface automatically based on a semantic description of the program along with stylesheets. This kind of approach gives users the freedom to *easily* choose the kinds of user interfaces they prefer independent of the rest of the program (barring fascist no-configuration policies). This contrasts with the current situation where a user may be forced to use a sub-optimal user interface in an otherwise fine application because there are no alternatives or because policies dictate the use of a specific program. Our framework also enables relatively easy experimentation of new user interface concepts with existing programs. Additionally, the approach allows for user interface features such as learning and teaching without any extra application support.

Vis has not yet been implemented in practice, if ever will, as especially the WIMP modules would most likely be a tremendous amount of work, and the author has no desire to become knowledgeable enough in such toolkits to do that part of the

10. There are more interesting to do with free time than read *awful* HCI papers.

work. However, due to the tendency of users to resist change, for something like Vis to ever become an accepted standard for writing software, such modules would have to be implemented. Yet the biggest stumbling block of Vis may indeed be the resistance of developers to change, and to give users the control.

We can draw an analogy between the Web and user interfaces. HTML was originally a simple semantic language. As Web and the Internet became popular, 'Web developers' with interests other than usability wanted control over pixel-precise position of everything. As a result the language and the Web was ruined to a state comparable to modern user interfaces until HTML 4.0 was created along with Cascading Style Sheets to remedy the situation. CSS has still to see widespread usage, and one of the reasons for this is that not all browsers interpret it the same way and that is totally unacceptable to these Web developers.

References

[Ber] L. Bernhardsson: "The LarsWM window manager", Free software.
URL `http://www.fnurt.net/larswm/`

[Gar04] A. R. Garbe: "Improved GUI concepts for hackers", , Mar. 2004.
URL `http://wmi.berlios.de/docs/wmi-paper.pdf`

[Ier03] R. Ierusalimschy, L. H. de Figueiredo & W. Celes: *Lua 5.0 Reference Manual*, Tech. Rep. MCC-14/03, PUC-Rio, 2003.
URL `http://www.lua.org/manual/5.0/`

[Ros] D. Rosenthal: *Inter-Client Communications Conventions Manual*, X Consortium.
URL `http://tronche.com/gui/x/icccm/`

[Sch02] K. Schneider & J. Cordy: "Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems", in *Interactive Systems: Design, Specification and Verification*, no. 2220 in Lecture Notes in Computer Science, pp. 28–48, Springer Verlag, 2002.
URL `http://www.cs.usask.ca/faculty/kas/`

[Thé99] D. Thévenin & J. Coutaz: "Plasticity of User Interfaces: Framework and Research Agenda", in *7th IFIP Conference on Human-Computer Interaction – Interact'99*, pp. 110–117, Sep. 1999.
URL `http://research.nii.ac.jp/~thevenin/papers.en.html`

[Trs] "The TrsWM window manager", Free software.
URL `http://www.relex.ru/~yarick/trswm/`

[Vala] T. Valkonen: "The Ion window manager", Free software.
URL `http://iki.fi/tuomov/ion/`

[Valb] T. Valkonen: "The PWM window manager", Free software.
URL `http://iki.fi/tuomov/pwm/`

[War04] R. Wareham: "The Command Line – The Best Newbie Interface?", *OSNews.com*, Mar. 2004.
URL `http://osnews.com/story.php?news_id=6282`