End-User GUI Customization

Christof Lutteroth
Department of
Computer Science
The University of Auckland
38 Princes Street
Auckland 1020, NZ
lutteroth@cs.auckland.ac.nz

Gerald Weber
Department of
Computer Science
The University of Auckland
38 Princes Street
Auckland 1020, NZ
g.weber@cs.auckland.ac.nz

ABSTRACT

Constraint-based description of GUI layout is a powerful technique, but having to define constraints manually is not user friendly. We propose a GUI editor for the Auckland Layout Model (ALM) that can handle constraint-based layout in a WYSIWIG manner, making it much easier to create or modify complex layouts. Furthermore, the GUI editor is built into the layout manager that is used during the runtime of a GUI application, making it accessible to the end-user. Users can switch from the operational mode of a GUI into the editing mode, and immediately adjust the GUI to their needs. GUI specifications can be managed in a platform-independent XML-based description language, leading to a document-oriented paradigm for GUIs. The implementation of GUIs currently changes from hard-coded GUIs to document-based approaches such as XAML and XUL. Sadly, this shift is currently performed as a mere reengineering of the development process and driving forces are chiefly productivity and maintainability. Our approach, in contrast, aims at enhancing user options and also platform-independence.

Keywords

Layout Manager, GUI, document orientation, end-user development, constraint programming, WYSIWYG.

Categories and Subject Descriptors

H5.2 [**Information interfaces and presentation (e.g. HCI): User Interfaces**]: GUI.

1. INTRODUCTION

XML-based GUI description languages have been around for some time and offer the chance to increase user interface robustness. Unfortunately, this transfer is not in itself adding anything to the functionality available to the user. Such GUI languages offer merely the same features as conventional GUIs, except for the prospect of the integration into web applications: web portals can use standard GUI features instead of only HTML and JavaScript. But also here, the features go not beyond what has been possible with Java applets before. The technological change is therefore rather motivated by new hopes for better software engineering of applications.

We present here intermediate results of a long-running research project into the Auckland Layout Model (ALM). ALM is a formally specified and platform-independent layout specification framework that is more expressive than common grid-based layout methods such as Java Gridbag layout [7]. Also it fosters generally a better software architecture, particularly in GUI layouts at the upper complexity limit [3]. In particular, this paper presents an editing approach for constraint-based GUI layout using ALM and an XML-based description language for GUI layout specifications, which offer true end-user advantages in an application-independent and platform-agnostic way. The end-user gains generic, multilevel customization and personalization options that are inherently safe and are always available, except if explicitly revoked by applications (something that is supposed to happen rarely). The end-user editing feature of ALM means that customization options such as adding and moving toolbars are available in every application without an effort on the side of the developer. Moreover the editing feature of ALM is even more generic and flexible than the handcrafted configuration options of applications programmed with conventional windows toolkits.

The paper is organized as follows. Section 2 introduces the basic concepts of ALM. Section 3 introduces the WYSIWYG GUI editor embedded into the ALM layout manager. Section 4 introduces an platform-independent XML format for ALM specifications. Section 5 discusses the full vision of document orientation and how this fits into a wider research approach. Section 6 relates our approach with important User Interface paradigms.

2. THE AUCKLAND LAYOUT MODEL

ALM offers several different levels of abstraction. On the lowest level, ALM is based on linear programming [11], i.e. on linear constraints and the minimization of a linear objective function. The problem of linear programming is formally well-defined, therefore it constitutes a kind of stable interface for the implementation of ALM: it can be implemented on any linear programming solver. On top of this very basic but complete interface, ALM offers higher-order constructs with features typically used for GUI layout: soft constraints, i.e. constraints that may be violated if necessary; abstractions for rectangular areas, which contain controls and offer parameters for preferred sizes, alignment and padding; abstractions for rows and columns, with functionality that enables easy reordering and elision. In the following we will summarize ALM's most important features; a more complete account can be found in [6].

2.1 Linear Constraints

In ALM, the controls of a GUI are aligned at virtual gridlines called *tabstops*, or tabs for short. Tabs are either vertical or horizontal, also known as x- and y-tabs. X-tabs are represented as variables holding an x-coordinate, and the latter ones as variables holding a y-coordinate.

If we consider a layout with x-tabstops $x_0, \dots, x_m$, $m \in \mathbb{N}$, and y-tabstops $y_0, \dots, y_n$, $n \in \mathbb{N}$, then a linear constraint for that layout can take the form

$$a_0 x_0 + \cdots + a_m x_m + b_0 y_0 + \cdots + b_n y_n \ OP \ c$$

with the coefficients $a_0, \dots, a_m, b_0, \dots, b_n$ and the right side c being real numbers, and the operand OP being one of $\{\leq, =, \geq\}$. A layout can contain an arbitrary number of such constraints; usually most of the coefficients in a constraint are zero. It is possible to use different units for different constraints. A value may be defined in pixels, which makes the actual size dependent on the graphics hardware, or in real-world units like cm, which always produces the same size. In the following sections we will examine different types of linear constraints, and describe how these can be useful for GUI layout. We will only consider equalities, but the concepts can be transferred to inequalities.

2.1.1 Absolute Constraints

We use absolute constraints in order to place x- or y-tabstops at particular x- or y-positions of the UI, respectively, or set the width or height between tabstops to a fixed value. If we want, for example, to set x-tabstop x_3 at position 50, we simply use the constraint

$$x_3 = 50.$$

In order to set the width of the area between x_1 and x_2 to 100, we would use the constraint

$$x_2 - x_1 = 100.$$

Such constraints are a very straightforward way to define the absolute properties of a UI, i.e. the properties that do not change when, e.g., resizing the window the UI is displayed in. Note that absolute constraints may be impossible to satisfy under some circumstances. If, for example, the available display area is only 10cm wide, the width between two x-tabstops cannot exceed this value.

2.1.2 Relative Constraints

In contrast to absolute constraints, relative constraints describe the position of tabstops or proportion of areas relative to others. This is useful in order to adapt the layout to changing circumstances, like UI display size or resolution. The layout manager recalculates the layout when such a change occurs.

Relative constraints can be used in order to position tabstops at positions relative to other tabstops. One might, for example, want to position an x-tabstop x_2 exactly between two other x-tabstops x_1 and x_3. Let us assume that $x_1 \leq x_3$, then the constraint can be expressed as follows:

$$x_2 - x_1 = x_3 - x_2 \Leftrightarrow -x_1 + 2x_2 - x_3 = 0.$$

Similarly, we can center an area that is delimited by x-tabstops x_2 and x_3, $x_2 \leq x_3$, horizontally between two other x-tabstops x_1 and x_4, $x_1 \leq x_4$:

$$x_2 - x_1 = x_4 - x_3 \Leftrightarrow -x_1 + x_2 + x_3 - x_4 = 0.$$

We only need to make sure that the area we want to center does not exceed the boundaries of x_1 and x_4 by specifying that $x_1 \leq x_2$ or $x_3 \leq x_4$.

Another usage for relative constraints is the specification of an area's proportions relative to those of another one. If, for example, we want the width between x-tabstops x_1 and x_2 to be twice as much as the the width between x_3 and x_4, we would use the following constraint:

$$x_2 - x_1 = 2(x_4 - x_3) \Leftrightarrow -x_1 + x_2 + 2x_3 - 2x_4 = 0.$$

Since a constraint can contain x-tabstops as well as y-tabstops, it is also possible to specify the aspect ratio of an area. We could, for example, specify the aspect ratio for an area $(x_1, y_1, x_2, y_2, \mathit{moviepanel})$, which might contain a control for displaying a video. Because we do not want the video to be shown with an arbitrary, distorted aspect ratio, we could, for example, set the ratio of width and height of this area to 16:9. This would be achieved with the following constraint:

$$\frac{x_2 - x_1}{y_2 - y_1} = \frac{16}{9} \Leftrightarrow -x_1 + x_2 + \frac{16}{9} y_1 - \frac{16}{9} y_2 = 0.$$

The aforementioned constraints are hard, i.e. if they are contained in the specification of a layout, then this layout will satisfy them strictly. Sometimes, however, we want to specify constraints that may not be satisfied fully if circumstances do not permit so. Such constraints are called *soft constraints*; they are natural in applications for user interfaces and have been used in the past [1].

Soft constraints are important in order to prevent overconstrained specifications which would be infeasible otherwise. For example, a specification may become overconstrained when several sets of constraints are merged. Therefore, soft constraints are relevant when layouts are specified in a modular manner, i.e. composed using several partial specifications. ALM supports soft constraints as an abstraction layer on top of the linear programming solver. Soft constraints can be handled in exactly the same manner as hard constraints; they are implemented as a subclass of hard constraints. In addition, they can be prioritized using penalty parameters for positive and negative deviations from their exact solution. Several approaches for prioritizing constraints have been devised, such as constraint hierarchies [2] that make sure that important constraints are satisfied first.

On a certain level, it is advisable to use only soft constraints, so that a specification will always have a solution. The different levels of a constraint hierarchy can be related to access privileges: hard constraints should only be accessible to trusted developers, since they may render the specification infeasible. Similarly, soft constraints with high priority should be chosen carefully since they may detrimentally affect all constraints with lower priority.

2.2 Areas

The controls of a GUI are organized in rectangular *areas*, which are bound by a pair of x-tabstops and a pair of y-tabstops each. In general, an area a is defined as follows:

$$a =_{def} (x_1, y_1, x_2, y_2, content)$$

The x-tabstops x_1 and x_2 delimit the area on the x-axis, with x_1 being to the left or on the same position as x_2; the y-tabstops y_1 and y_2 delimit it on the y-axis, with y_1 being above or on the same position as y_2. *content* can be a control of the GUI, but can also be *empty*. Empty areas are useful, e.g., for defining margins or padding, or for extending a layout specification with ordinal information. Like this, a set of area definitions result in a partial order on the x- and on the y-tabstops.

One can think of an area as a group of adjacent cells in the table created by the tabstops that are merged into a rectangle to house one of the graphical elements of the UI. This approach covers the common colspan and rowspan features of other concepts for tabular layout: colspan means that adjacent table cells lying on a horizontal line can be merged into a single cell; rowspan means that adjacent table cells lying on a vertical line can be merged, i.e. the resulting cell spans over several rows or columns. ALM generalizes colspan and rowspan by permitting areas that extend between any two x-tabs and any two y-tabs. In addition to this, ALM permits overlapping areas – something which cannot be achieved by merging adjacent cells with colspan and rowspan. This is because tabs may run through areas, and any tab can be used to delimit an arbitrary number of areas.

2.2.1 Additional Area Parameters

ALM supports the specification of parameters for areas that allow the layout manager to find optimal dimensions for them. One of those properties is the preferred size. This value expresses the size that an area would need in order to fulfill its function in an optimal manner. Usually it is based on the size of the content that is displayed in a control. In many GUI toolkits, such values are made available by all controls, and are thus taken directly from them by a layout manager. That way the layout can be adapted immediately when preferred sizes change, e.g. when the content shown in a control changes. Like this, screen real-estate is not unnecessarily wasted, and areas can be used in a modular fashion: each area contains properties describing its demands, while the system as a whole finds a global solution that respects the desires of each area as much as possible. Analogously to soft constraints, the demands of areas can be prioritized, so that areas that are considered more important can be given preference over less important areas.

The following list describes some of the area parameters:

Preferred size s_{pref}**:** the preferred width w_{pref} and height h_{pref} of an area. These optional values can often be calculated by a GUI toolkit, and can therefore be directly taken from there.

Minimum size s_{min}**:** the minimum width w_{min} and height h_{min} of an area. Both values are optional.

Maximum size s_{max}**:** the maximum width w_{max} and height h_{max} of an area. Both values are optional.

Shrink penalty p_{shrink}**:** a coefficient for each dimension that indicates the reluctance on the part of the area to take on sizes that are smaller than the preferred size. This value is optional, and sensible default can be chosen dependent on the type of the control in the area. Controls that contain important information, such as buttons with text, should not be shrunk easily and should therefore have a higher shrink penalty.

Expand penalty p_{expand}**:** a coefficient for each dimension that indicates the reluctance on the part of the area to take on sizes that are larger than the preferred size. This value is optional, and again, a sensible default can be obtained by considering the control type. Controls such as textboxes where users enter data can benefit more from additional space than controls with fixed information content, such as buttons, and therefore the former should have a smaller expand penalty.

Having useful default values for the different parameters of the model makes specifications more compact and hides the complexity of unused model features. A GUI developer needs, for example, only consider penalty coefficients if they want to model something that differs significantly from the default behavior. Like this, developers can learn the features of the model step by step, and can use the model even with a limited understanding.

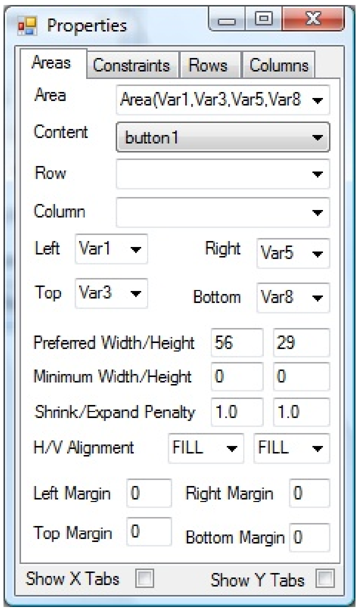
2.3 Rows and Columns

A useful pattern that enhances maintainability of table definitions is the *separate tabstop* pattern, which means that separate tabstops are used to delimit different logical objects, even if the tabstops end up on the same coordinates. ALM offers abstractions for columns and rows, which contain two x- or y-tabstops each. Instead of using tabstops directly, an area can be arranged by placing it into a particular row and column. The rows and columns offer methods that make it possible to adjust their size and order dynamically. For example, each row has an optional `Previous` and `Next` property referring to the row directly above and below. Rows can be removed from and inserted between existing rows. The order in which the columns appear in the table is encapsulated in a separate set of constraints, managed by ALM and independent of the definition of the columns themselves. Hence, the order of the columns can be changed very easily without interfering with the rest of the specification. In fact, all the parts of a specification, such as areas, rows and columns, also allow developers to add their own constraints which are then managed together with the respective part in a modular fashion.

3. END-USER CUSTOMIZATION

The main contribution of this work is a new customization option that is directly built into the layout manager, and that is therefore available at any time in every application using ALM. The customization function presented here is only the first step in a development programme that will ultimately support all features of document orientation as it is presented in Section 5.

How a GUI is switched from the operational mode into the editing mode can be defined by the developers of the GUI: the layout manager offers a method `Edit` that can be called anytime. After switching to editing mode, the GUI looks exactly the same, with the exception that certain parts of the GUI may be highlighted for easier editing, e.g. in the form of a selection rectangle around one of the areas. The


Properties

Areas | Constraints | Rows | Columns

Area: Area(Var1,Var3,Var5,Var8

Content: button1

Row:

Column:

Left: Var1 Right: Var5

Top: Var3 Bottom: Var8

Preferred Width/Height: 56 29

Minimum Width/Height: 0 0

Shrink/Expand Penalty: 1.0 1.0

H/V Alignment: FILL FILL

Left Margin: 0 Right Margin: 0

Top Margin: 0 Bottom Margin: 0

Show X Tabs ☐ Show Y Tabs ☐

Figure 1: The properties window in the Area mode.

behavior of the GUI in editing mode is completely different: the controls do not offer their normal possibilities of interaction; instead they behave similarly to purely graphical objects, as one would expect in an editor, as entities that can be selected, moved, inserted and deleted.

In editing mode, an additional window appears, which we call the properties window. The properties window allows users to switch between several modes of the editor, and inspect and change properties of the different entities that make up the specification of the GUI. When the properties window is closed, the GUI is switched back to the operational mode. There is an editing mode for areas, a mode for constraints, and modes for rows and columns. In the following we will look at the different modes.

3.1 Editing Areas

Figure 1 shows the properties window in the area editing mode. By clicking on the tabs at the top of the window, the editing mode can be changed. In this window, all the parameters of areas can be manipulated, with all changes becoming immediately visible in the GUI that is edited. For example, new tabs for an area's boundaries can be selected from drop-down lists.

The most important functions for areas are also available through direct manipulation [12] in the GUI: Fig. 2 shows how areas can be swapped by dragging and dropping one area onto another. Both the source area as well as the current target area are marked by a selection rectangle, and as soon as the mouse is released the areas marked by the rectangles are swapped. The button in the top-left corner will be in the center where the grayed out text box is, and vice versa.

Similarly, also the boundaries of an area can be changed through direct manipulation by selecting an area and dragging the sides or corners of the selection rectangle. As Fig. 3 illustrates, as soon as the selection rectangle is dragged, all suitable tabstops in the GUI become visible. A second rectangle that highlights possible new boundaries for the se-

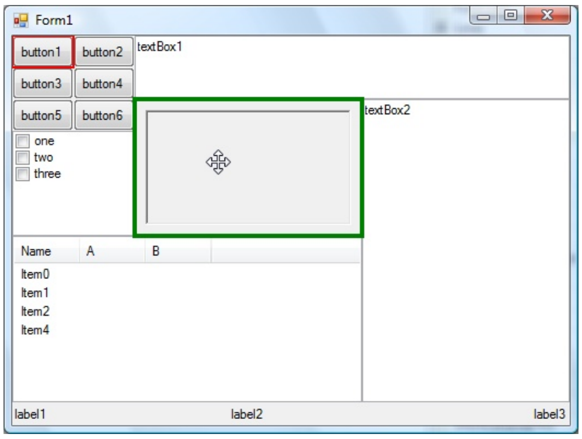


Figure 2: Dragging of areas for swap operation.

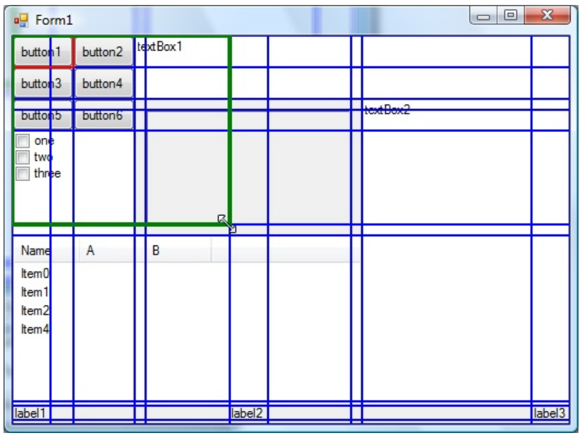


Figure 3: Dragging of area boundaries.

lected area appears, and it can be snapped to the suitable tabstops. If the mouse button is released, the boundaries of the area are changed to the new boundaries.

After a couple of customization steps the GUI could look as illustrated in Fig. 4. Apart from several swapped areas and changed boundaries, this layout has an additional area at the bottom left. This was created by splitting the area above it horizontally. Both the original and the new area are marked in the screenshot. Horizontal and vertical splitting and deletion of areas is supported through the context menu: the user can right-click on the area to be modified and select the appropriate operation from the menu.

3.2 Editing Constraints

In the constraints editing mode, the properties window shows only a selection list of all constraints, a textual representation of the selected constraints, and the penalty parameters in case the selected constraint is a soft constraint. The GUI window looks like the one in Fig. 5: the tabstops of the layout are visible, and the tabstops that occur as variables in the currently selected constraint are highlighted. Tabstops can be added to and removed from a constraint by right-clicking on them in the GUI can selecting appropriate operations from the context menu. In the screenshot, the selected constraint specifies the height of the second row of

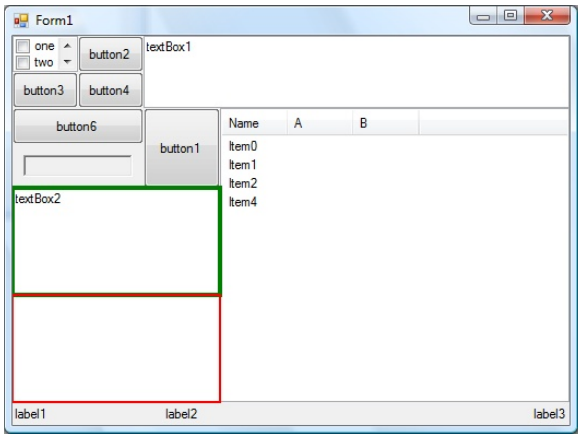


Figure 4: Customized layout.

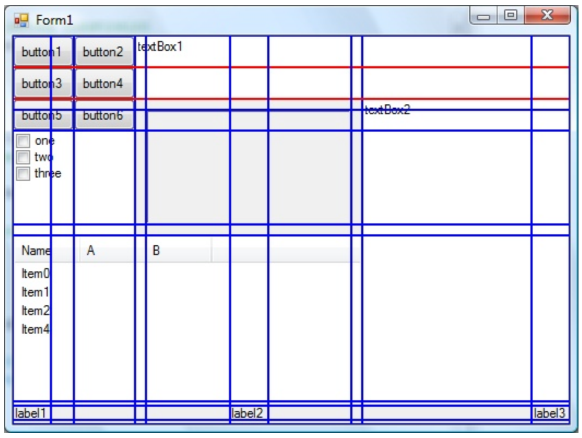


Figure 5: Editing constraints.

buttons at the top left.

Besides textual editing of constraints in the properties window, absolute constraints – which are the most common constraints – can be modified through direct manipulation. The editor recognizes if a constraint specifies an absolute position or distance, and lets the user drag the respective tabs in the GUI. If a tab is dropped, the selected constraint is adjusted appropriately.

3.3 Editing Rows and Columns

The editor also offers modes for editing the rows and columns in a GUI specification. In the row mode, all the rows are marked, as illustrated in Fig. 6. The selected row is highlighted. The order of rows in a GUI can be changed by dragging the selected row. In the screenshot, the second row from the top is dragged to a new location between the third and fourth row, which is indicated by the marker at this new location. As soon as the mouse button is released, the dragged row is removed from its original location, the first row is linked with the third row to close the gap, and the dragged row is inserted at its new position. By right-clicking on a row operations for deleting a row and splitting a row horizontally can be invoked from the context menu. The mode for editing columns works analogously to the mode for rows.

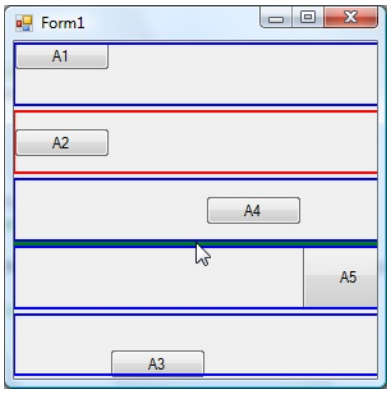


Figure 6: Editing constraints.

4. DOCUMENT-ORIENTED REPRESENTATION OF GUIS

ALM GUIs can be represented in a platform-independent way as XML documents using the XML ALM Object Notation (XALMON, pronounced “salmon”). This format differs from other XML-based user interface description languages such as XUL and XAML in that it provides a separation of content and layout, and that it is platform-independent. By contrast, XAML and XUL are tied to their respective GUI frameworks.

In a XALMON specification, the controls are only addressed by a symbolic name, and the format makes no assumptions about the GUI toolkit. This leads to a clean separation between the GUI content and the ALM layout, which makes it possible to ignore the differences between the controls of particular GUI toolkits when using XALMON. In all the supported GUI toolkits, i.e. Java Swing, .NET Forms and the Haiku Interface Kit, each GUI control can be given a textual name that can be accessed during runtime as a string. The controls in the GUI of an application are defined programmatically, i.e. in the source code of the application, but without having to worry about layout. The XALMON specification, which describes the layout, refers to controls by their textual name. In the example below, the control `button1` is such as symbolic name, referring to a button control.

XALMON files can be created by every user interface that uses ALM, and they can also be loaded by each such user interface. In the editing mode, functions for loading and saving a specification are available, and these functions can also be invoked from within an application through methods of the layout manager. For a fully document-oriented GUI specification solution [4], XALMON is still lacking some features such as a content description language and a decomposition mechanism. But it already fulfills the vision of document orientation with respect to single GUI layouts. The following listing shows a small XALMON example:

```
<almlayout>
  <xtab> x1 </xtab>
  <ytab> y1 </ytab>
  <row>
    <name> r1 </name>
    <next> r2 </next>
  </row>
  <row> <name> r2 </name> </row>
```

```
 9      <area>
10        <name> button1 </name>
11        <left> left </left>
12        <right> x1 </right>
13        <row> r1 </row>
14        <leftmargin> 10 </leftmargin>
15        <halignment> center </halignment>
16        <expandpenalty> 1 </expandpenalty>
17      </area>
18      <constraint>
19        <leftside>
20          <summand>
21            <coeff> -1 <coeff> <var> y1 </var>
22          </summand>
23          <summand>
24            <coeff> 1 <coeff> <var> r1.top </var>
25          </summand>
26        </leftside>
27        <op> = </op>
28        <rightside> 0 </rightside>
29        <penaltyneg> 1 </penaltyneg>
30        <penaltypos> 1 </penaltypos>
31      </constraint>
32    </almlayout>
```

The whole specification is enclosed in an `almlayout` tag. In lines 1-2 the tabstops `x1` and `y1` are defined. Lines 4-8 define two rows `r1` and `r2`, with `r1` being directly above `r2`. Lines 9-18 define an area containing the control with the symbolic name `button1`. Since each control can be contained in at most one area, this name can also be used as a symbolic name for the area. The area is contained in row `r1`, begins at the left side of the GUI, and is bounded on the right side by `x1`. In lines 14-16 various additional area parameters are specified. Lines 18-31 define the soft constraint $-\texttt{y1} + \texttt{r1.top} = 0$, with `r1.top` being the top y-tab of row `r1`.

5. THE DOCUMENT-ORIENTED APPROACH

The document oriented approach [4] uses the document metaphor to add natural semantics to user interfaces. User interfaces are managed as documents that can be edited directly by the user, with the editing functionality being part of the UI rendering infrastructure. As such, the work presented here is a natural milestone within the document-oriented programme, as it allows users to treat GUIs as documents with respect to presentation and editing. The full document-oriented approach extends the customization option to the GUI content and offers potentials that are summarized in the following.

5.1 Implementation Perspective

The implementation perspective of the document-oriented approach is shared with current document-based approaches such as Mozilla XUL and Microsoft XAML.

Separation of concerns All levels of GUI layout are automatically separated from implementation of the functionality.

Small footprint A document-oriented GUI is comparable to a HTML page. It can be rendered by a generic viewer and hence no installation is necessary.

New cross platform potentials The document-oriented description offers the potential to use one format for all platforms, thus ensuring compatibility and supporting other features such as unified look-and-feel. It is possible to use cross platform GUI libraries, or to give a cross-platform mapping between fundamental GUI concepts (such as buttons).

Non-universality Definitions of GUIs through code offer a Turing-universal language to specify the GUI. This makes GUI description theoretically unanalyzable and practically error prone. It is also often unneeded, hence document-orientation uses here non-universal languages that are easier to analyze.

5.2 Design Perspective

Unification of GUI editor and GUI framework GUI editor functionality is available at runtime and thus can be unified with functions such as resizable separators.

Decomposition mechanism For the document-oriented approach, reuse of documents is not necessarily restricted to a single application. For example, if a document represents a print-dialogue, the same dialogue can be reused across several applications and thus realizes settings that are shared between applications.

5.3 Analysis Perspective

The analysis perspective is the one most closely related to the user view. Our view of analysis is here that it is the model that the user and the developer agree on. Analysis covers the functional requirements of an application.

Definition A document-oriented GUI is a precise definition of what is the GUI and what is the rest of the program. With a code-based GUI definition, no real distinction can be made.

Simplified controls In a standard GUI, labels and text input fields are distinct. In our approach, they differ only in the rights the user has. While a label is read-only, and maybe has a slightly different visual style, the text input field can be edited.

Comprehensive customization In document-orientation it is easy to grant users sweeping customization powers.

Explicit semantics This is important for example for auxiliary dialogues. In [4] we showed that the scope of options even within the same application varies widely. Typical scopes are: session, document, application, office suite, user, platform. The scope is regularly not presented. In the document-oriented approach, the scope of changes to options becomes obvious.

5.4 Quality Perspective

Isolation The GUI can be separated from the other program, even put into a different process such as the browser, thus enhancing system stability.

Robustness The chances of incorrect GUIs are reduced. An example type of error are incompletely rendered

GUI components that miss vital options such as a cancel button. An example is shown in Fig. 7. Such errors are more likely in code-based GUIs due to the inherently higher complexity of their representation. In document-oriented GUIs such errors can not only be traced by the user, but also corrected.

Completeness Those GUI features that relate directly to the fundamental GUI model, such as customization and editability, can be offered in every application, thus ensuring the completeness of the options that the user has.

Conformity to Expectations Through cross platform capabilities, the look-and-feel can be unified. Look-and-feel works on the level of reflexes and is an important quality attribute.

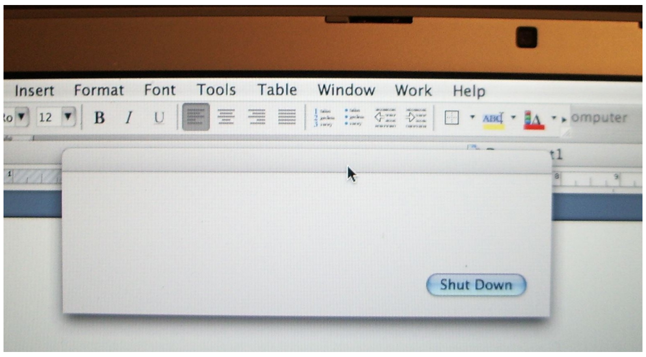


Figure 7: An incompletely rendered popup window without information and cancel button.

6. RELATED PARADIGMS

XML-based GUI-description languages have been around for some time, for instance in the form of Mozilla XUL. Their acceptance was limited, for unknown reasons. With a major platform now switching to this paradigm in the form of XAML, it is likely that XML-based GUI descriptions will become more widespread.

The document-oriented approach is able to increase our understanding of the connection between several important high-level models for user interfaces. All of these approaches can be understood as being based on a representation of the GUI as a document.

6.1 The Seeheim Model

The Seeheim Model [10] proposes a layered approach for user interfaces. The interface is separated into three different components that we want to refer to here as layers: the presentation layer, the dialogue control layer, and the application interface model layer. The original Seeheim model also includes a circumvention path for this layering, which has been, however, of lesser interest. The Seeheim model is on a very high abstraction level, but the fundamental idea of a multi-layer architecture remains timely. A document-oriented description is an excellent way to specify a layer, since the document format can ensure strict adherence to the layer abstraction. The ALM document format achieves a clear layering within the presentation component.

6.2 The Ousterhout Dichotomy

Ousterhout has claimed a trend to a clear split between scripting languages and system programming languages [9]. His position is first an observation of different languages, but then it motivates a separate use of these languages. Scripting languages are favored for high-level gluing purposes, and user interfaces are considered an instance of this. Tcl/Tk is an instance of such a user interface scripting language, JavaScript would be a different one. XALMON takes the role of a non-universal scripting language. Single XALMON specifications lead to continuous constraint evaluations that give rise to many machine instructions, which is a hallmark of a scripting language.

6.3 Smalltalk MVC and Model 2

Model-View-Controller is a classical approach to GUI design, and still valid in its various forms. It is quite different from the Model 2 for web applications, which is however also known as Web MVC [5]. In Model 2 architectures the view is naturally realized using documents.

6.4 Separation of Content and Presentation

The design guideline of separating content and presentation is an interesting notion here because it is not yet referring to a particular architecture. It is often understood to be an instance of the general guideline of separation of concerns. This view might be helpful, but it can also put the focus too narrowly on the software engineering issues. In practice, this guideline gives rise to the use of style sheets and is therefore somewhat different from the other concepts discussed here, and in fact orthogonal. The same principle can be found in HTML cascading style sheets [8], GUI skins, XML style sheets, and actually, document master templates in office suites. XALMON offers a precise separation of layout from other concerns.

7. CONCLUSION

The shift to XML has been hailed as a paradigm shift in computing, but XML-based user interface description languages have yet to prove that they offer advantages to the end-user. In this paper we present the end-user customization options of the ALM layout manager, which use a platform-independent XML format. The layout manager offers a GUI editor for every application. The customization options go beyond what is offered in other, grid-based layout managers. This contribution is part of a wider research into the concept of document orientation, with the goal of making application behavior more intuitive and allow the end-user to regain control of the user interface.

8. ACKNOWLEDGMENTS

We would like to thank Gyurme Dahdul for his initial work on the GUI editor. This work has been funded by AARN Innovation Ltd and the Foundation for Research, Science and Technology of New Zealand under grant number AARN0501.

9. REFERENCES

[1] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[2] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp Symb. Comput.*, 5(3):223–270, 1992.

[3] G. W. Christof Lutteroth. Modular specification of GUI layout using constraints. In *Proceedings of ASWEC 2008 – 19th Australian Conference on Software Engineering*. IEEE Press, 2008.

[4] D. Draheim, C. Lutteroth, and G. Weber. Graphical user interfaces as documents. In *Proceedings of CHINZ 2006 – 7th International Conference of the ACM's Special Interest Group on Computer-Human Interaction*. ACM Press, 2006.

[5] D. Draheim and G. Weber. Specification and Generation of Model 2 Web Interfaces. In *APCHI 2004 - 6th Asia-Pacific Conference on Computer-Human Interaction*, LNCS 3101. Springer, June 2004.

[6] C. Lutteroth, R. Strandh, and G. Weber. Optimal GUI layout as a problem of linear programming. Technical Report UoA-SE-2007-6, Software Engineering, The University of Auckland, August 2007.

[7] C. Lutteroth, R. Strandh, and G. Weber. Domain specific high-level constraints for user interface layout. *Constraints*, 13(3), 2008.

[8] J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Thousand Oaks, CA, USA, 1999.

[9] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[10] G. E. Pfaff, editor. *User Interface Management Systems*. Springer, Secaucus, NJ, USA, 1985.

[11] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[12] B. Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *IUI '97: Proceedings of the 2nd International Conference on Intelligent User Interfaces*, pages 33–39. ACM Press, 1997.