Multi-Platform Document-Oriented GUIs

James Kim, Christof Lutteroth

Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
Email: jkim202@aucklanduni.ac.nz, lutteroth@cs.auckland.ac.nz

Abstract

In recent years, increasing complexity of graphical user interfaces (GUIs) of applications has led to problems in GUI management, since there is no single layout to fulfill every user's needs. GUI editors have been developed to enhance end-user options but they commonly fail to preserve personalized GUIs. This paper presents an extension to the GUI editor built into the Auckland Layout Model (ALM) that can permanently store user-defined GUI layouts and reproduce them on different platforms. A novel technique called the document-oriented approach has been exploited to empower end-user customization, which allows GUI layouts to be dynamically edited, saved using a standardized XML-based GUI description language, and loaded in a platform-independent manner.

Keywords: GUI customization, platform-independent GUIs, ALM

1 Introduction

At present, most computer applications provide a graphical user interface (GUI), which consists of widgets (or controls), e.g. buttons and text areas. Many applications typically organize controls in the GUI using a layout manager. Layout managers primarily enable GUIs to adapt to changing environments, for example, automatically adjusting sizes of controls on resizing the application window. A GUI layout can significantly influence usability of an application. However, current layout managers do not solve the problem of increasing GUI complexity. Recent applications overload the user interface with too many controls disregarding suitability for users, which often complicates the process of achieving a simple task, especially for novice users. It is evident that there can be no single GUI layout that fulfills every user's requirements. The easiest solution for this is to allow end-users to edit GUIs, so that GUIs can be adapted to individual needs. In fact, many applications offer end-user customizability such as resizing side panels and hiding certain items on toolbars. However, this is somewhat limited. Moreover, personalized GUIs are usually not preserved. While the layout is maintained between sessions in some applications, layout customization falls apart in a multi-user environment.

This paper presents an extension to the GUI editor built into the Auckland Layout Model (ALM). ALM (Lutteroth et al. 2008) is a platform-independent constraint-based layout specification framework. A novel technique called the document-oriented approach (Draheim et al. 2006) has been exploited to empower dynamic end-user customization of GUI layouts. In particular, the editor has been developed to enable GUI layouts to be customized at run-time, stored permanently in document form, and loaded on multiple platforms. Since the editor is embedded in the ALM layout manager, it is readily accessible in any application using such a layout manager.

The paper is organized as follows. Section 2 examines related work. Section 3 introduces the basic concepts of the ALM layout manager. Section 4 presents the key editing features of the editor. Section 5 explains the utilization of the document-oriented approach. Section 6 explains support for multiple platforms. Section 7 discusses some of the challenges encountered when implementing the document-oriented approach for different platforms. Section 8 presents evaluation results. Section 9 discusses current limitations and future work.

2 Related Work

Contributions of this project relate to two main areas: end-user customization of GUIs and GUI description languages. In this section, some of the related work in each of these areas will be highlighted.

A lot of work has already been carried out in the area of end-user GUI customization. The Self-4.0 user interface (Smith et al. 1995) presents a flexible method for editing GUI layouts. It primarily aims to remove the distinction between a running application and an application being edited, which is commonly known as a WYSIWYG (What-You-See-Is-What-You-Get) editor. Self allows users to create or remove controls in a dynamic environment. Moreover, it provides scripting support, which allows users to specify functions for controls.

A user interface management system, EUPHORIA (McCartney et al. 1995), handles end-user construction of GUIs without any need for programming. It supports run-time construction, allowing GUIs to be created and modified interactively. EUPHORIA uses a graph-based constraint algorithm to solve a set of relationships between GUI components.

Similarly, Haystack (Karger & Quan 2004) allows personalization of user interfaces at run-time. Based on the model-view-controller architecture, the GUI layout and various operations can be created and manipulated flexibly. Moreover, it possesses limited capability to learn about user preferences and behavior.

ALM provides similar benefits by providing a WYSIWYG GUI editor, which can be promptly accessed in every application at run-time. Current GUI editors commonly fail to provide support for preserving personalized GUIs. On the contrary, ALM

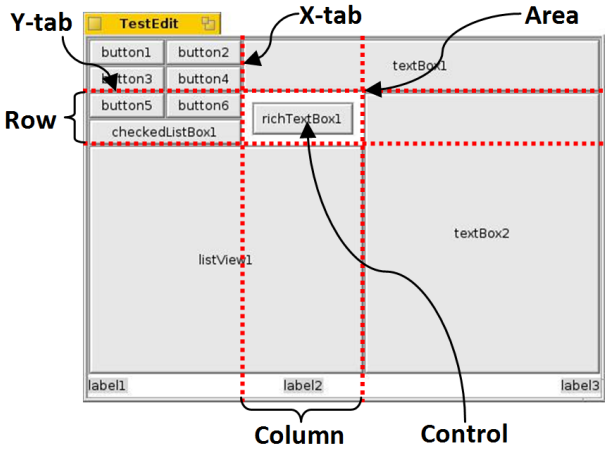

Figure 1: A GUI using the ALM layout manager.

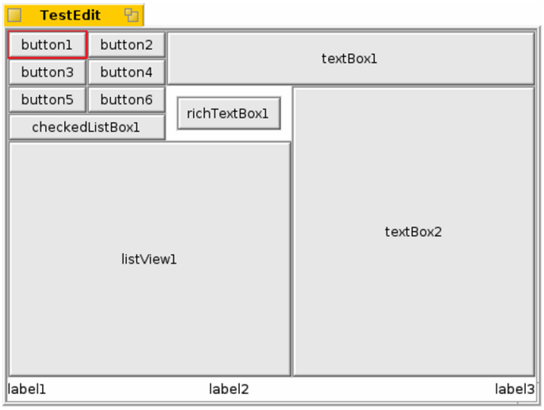

Figure 2: The GUI in editing mode.

uniquely provides saving and loading features.

XML-based GUI description languages are becoming more widespread for document representation of GUIs. UIML (User Interface Markup Language) (Abrams et al. 1999) primarily eliminates differences of interfaces in Internet appliances such as desktops, cell phones, and PDAs. Also, UIML eliminates the need for hand-coding user interfaces for different types of appliances by offering an abstract declarative language, while preserving the expressiveness of GUI toolkits such as Java AWT or Swing. UsiXML (USer Interface eXtensible Markup Language) (Limbourg et al. 2004) employs multiple abstraction levels. User interfaces can be specified at one or more levels of abstractions, and mappings between the different levels can be maintained.

XUL (XML User Interface Language) (Cheng 1999) is developed for the Mozilla Firefox web browser. It aims to integrate powerful GUIs into the web browser. Similarly, XAML (Extensible Application Markup Language) is developed for the Microsoft Windows platform (Draheim et al. 2006). However, both XUL and XAML are fixed to specific GUI frameworks. In contrast, ALM's GUI description language makes no assumption about the GUI toolkit by making a clean separation between the layout and the content.

3 The Auckland Layout Manager

ALM is a novel technique for specifying 2D layouts in a GUI. A layout specification is formed using a set of constraints based on linear algebra. An optimal layout is calculated using linear programming (Dantzig 1963). Linear programming is a rather complex concept, therefore the ALM layout manager provides higher order abstractions for easier specification of layouts. On top of the formally well-defined linear programming, higher order abstractions include soft constraints, areas, and rows and columns.

Figure 1 shows a GUI created using the ALM layout manager. Controls in the GUI are organized using rectangular *areas*, which are bound by virtual grid lines called *tabstops*, or *tabs* for short. Specifically, an area is enclosed inside a pair of vertical tabs called *x-tabs* and a pair of horizontal tabs called *y-tabs*. X-tabs and y-tabs hold x-coordinates and y-coordinates, respectively, within the GUI coordinate system.

Coordinates of tabs, and hence the positions and sizes of areas, are mainly determined by linear constraints. There are two types of constraints: absolute constraints and relative constraints. Absolute constraints are used to place tabs at fixed positions or to set the width or the height between tabs to a particular value. Therefore, these tabs are not affected when resizing the GUI window. By contrast, relative constraints are used to place tabs at positions relative to other tabs. In this case, coordinates of tabs are constantly adjusted as the GUI window is resized in order to continue adhering to the constraints. In addition, soft constraints can be specified, which are basically flexible constraints that can be violated by setting positive and negative deviations from the exact solution. Soft constraints primarily solve the problem of over-constrained specifications, hence a feasible layout can always be provided.

The dimensions of areas depend on constraints and additional area parameters, including preferred size, minimum size and maximum size. Each area may contain a control. By default, a control fills the entire area. However, a control can be freely placed within its area as desired. Some of the parameters for the area content, i.e. the control, include left margin, top margin, right margin, bottom margin, horizontal alignment, and vertical alignment. In Figure 1, the control richTextBox1's left, top, right, and bottom margins are set to 10. Also, the horizontal alignment of label1, label2, and label3 have been set to left, center, and right, respectively.

Abstractions for rows and columns simplify reordering and elision. As illustrated in Figure 1, a row is represented by a pair of y-tabs. Likewise, a column is represented by a pair of x-tabs. By employing the table metaphor, rows and columns can be utilized to define, resize, and rearrange areas easily without a direct reference to tabs. Note that ALM can not only be applied to GUIs but also to other types of user interfaces such as those found in submit-response style systems (Draheim & Weber 2004).

4 End-User Customization

The layout specification of a GUI is initially specified in the source code of an application. ALM provides a WYSIWYG GUI editor (Lutteroth & Weber 2008), which enables end-user customization of the layout specification. Specifically, end-users are presented with a functionality that allows them to manipulate GUI layouts directly in a dynamic environment, i.e. at application run-time. Moreover, the editor is immediately available on request in every application using ALM, since it is incorporated in the ALM layout manager. This is also advantageous for application developers because it eliminates the need to implement such a customization feature explicitly.

A GUI is started in the *operational mode*, i.e. the mode in which it can be used normally. Anytime during run-time, it can be switched to the *editing mode*

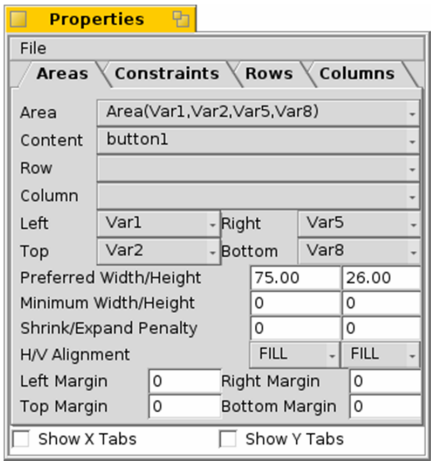

Figure 3: The properties window.

where it can be customized. Internally, this switch is performed by calling a method Edit. How the transition between operational mode and editing mode can be performed by the end-user is entirely up to the application developer. Typically, the developer defines a shortcut key. Figure 2 shows a GUI in editing mode. Changes in visual appearance from operational mode to editing mode are minor, and consist mostly of the addition of colored rectangles, which are markers for editing operations. It is important that controls in editing mode are no longer functional, while their visual appearances are maintained. This is achieved by converting controls into bitmap images during mode transition. The WYSIWYG-style of the GUI editor makes the end-user customization operations more intuitive.

During editing, a *properties window* is shown that makes the properties of editable GUI elements readily accessible. The properties window itself has been built using the ALM layout manager for coherence. The ability to use the ALM layout manager within itself illustrates the emphasis on reuse in ALM's architecture. The properties window displays specification details of the GUI layout depending on the active *editing mode*. There are editing modes for modifying areas, constraints, rows, and columns. However, this paper will only cover the area editing mode. Figure 3 shows the properties window in the area editing mode, which displays useful information about the selected area. Both windows in the editor are always synchronized. Editing the GUI directly updates the properties window. Similarly, when new values are entered into the drop-down lists and text fields in the properties window, changes are immediately reflected on the GUI window. The GUI returns to operational mode simply by closing the properties window. Therefore, switching between editing mode and operational mode is very simple. In the following, the most essential features of the editor in area editing mode will be illustrated.

Currently, the editor allows areas to be customized in various ways. Firstly, editing operations on the GUI window will be introduced. In editing mode, a red rectangle border around an area indicates the selected area, i.e. button1 in Figure 2. One of the most frequently used operations is swapping positions of two areas. For example, in an image editing application, one may want to position the most used controls closer to the editing region. Figure 4 demonstrates the swapping operation, which is simply performed using the drag-and-drop action. In the first phase, button1 is selected with a mouse and dragged above textBox2. Notice that the target area is highlighted

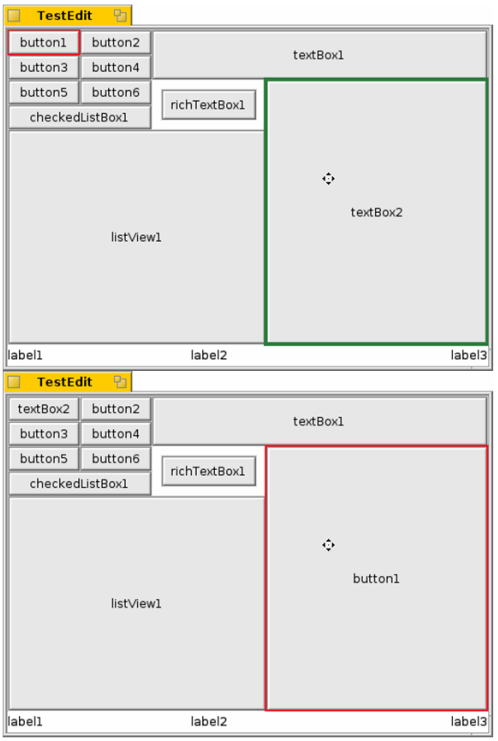

Figure 4: An area being dragged (top) and the GUI after swapping areas (bottom).

with a green rectangle border, i.e. textBox2. Also, the cross mouse cursor is another indication that an area is currently being dragged. When the mouse button is released, positions of the areas involved are swapped. Drag-and-drop is a commonly used action in computer applications, and therefore an intuitive procedure.

Another common editing operation is resizing an area. In the same image editing application example, the user might want to expand the size of the editing region for processing a large image. Figure 5 illustrates the resizing operation. Firstly, when a border of an area, i.e. button1, is clicked, all available tabs become visible in blue. Note that the area does not need to be selected first. The border of areas can be recognized by the mouse cursor, which transforms into an appropriate bi-directional arrow when it approaches an area boundary. Again, use of the drag-and-drop action is promoted. When the mouse cursor approaches a tab while being dragged, it snaps to the tab, providing a preview of the new area using a green rectangle. Note that its original size is still displayed in red. Finally, releasing the mouse button completes the resizing operation. If the mouse button is released without a new tab, i.e. without a green rectangle, then the operation is simply canceled and the GUI remains unchanged.

Furthermore, an application may contain controls that are never used by a certain user. In that case, unused controls can be removed from the GUI. Figure 6 shows the removal operation. Initially, a pop-up menu is opened by clicking the right mouse button on an area that contains the control to be removed. Subsequently, the control in the selected area, i.e. richTextBox1, is removed by selecting the menu item "Remove Area Content". However, instead of deleting the removed control completely, the control is stored in the ALM layout manager and can be reused later if required. For instance, the removed control can be used to fill an empty area or to replace a control in another area. This is done by selecting the target area and choosing the control in the Content drop-down

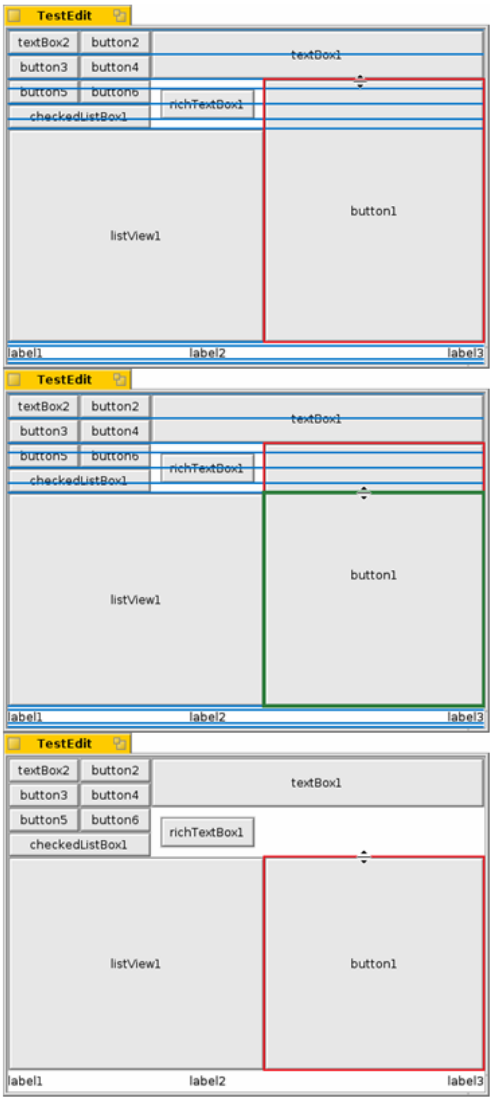

Figure 5: Border of an area selected (top), border of the area being dragged (middle) and the GUI after resizing the area (bottom).

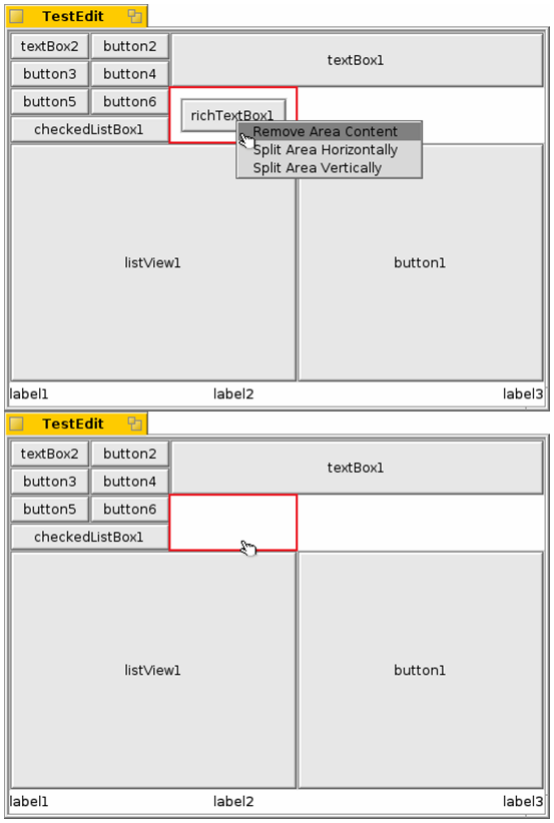

Figure 6: Right-click menu (top) and the GUI after removing the content in the selected area (bottom).

list, as seen in Figure 3.

As mentioned earlier, details of the selected area are displayed in the properties window. In addition to the operations supported in the GUI window, further editing operations can be performed using the properties window. Two of these operations are illustrated in Figure 7. Firstly, the horizontal alignment of the selected area, i.e. listView1, is changed from FILL to CENTER. Similarly, the top margin of the same area is set to 100. Notice that the control inside the area is affected by such operations, not the area itself. These operations are intended to arrange controls within the area without affecting the rest of the GUI layout.

Figure 8 shows the resulting GUI in operational mode after a series of editing operations. During the transition from editing mode to operational mode, controls retrieve their normal functionality.

5 The Document-Oriented Approach

On top of support for end-user customization, the main focus of this project was to improve the editor by exploiting the document-oriented approach. Document-orientation solves the problem of current GUI editors failing to preserve customized GUIs. In Figure 9, features of the editor after utilizing the document-oriented approach have been illustrated. Firstly, by utilizing the document metaphor, GUIs can essentially be edited, saved permanently and loaded just like a document in any word processor. Moreover, the same tool is used for editing and displaying the GUI as illustrated earlier, i.e. a WYSIWYG GUI editor.

As well as possessing the fundamental properties of documents, document-oriented GUIs are represented as documents. Basically, specifications of the GUI layout are stored in a document format. Such a format should contain enough information to allow the GUI to be constructed from standalone documents. ALM uses XML ALM dOcument Notation (XALMON) (Lutteroth & Weber 2008), which is a standardized XML-based GUI description language. While the editor offers end-user customization at runtime, static customization is provided by XALMON specifications, which can easily be edited using any text editor. Below is a part of an example XALMON specification, created by saving a GUI using the editor.

```
1   <almlayout>
2      <area>
3         <name> button1 </name>
4         <left> Var1 </left>
5         <top> Var2 </top>
6         <right> Var5 </right>
7         <bottom> Var8 </bottom>
8      </area>
9      <area>
10        <name> button2 </name>
11        <left> Var5 </left>
12        <top> Var2 </top>
13        <right> Var6 </right>
14        <bottom> Var8 </bottom>
15        <topmargin> 10 </topmargin>
16        <bottommargin> 10 </bottommargin>
17        <halignment> center </halignment>
18     </area>
19     <constraint>
20        <leftside>
```

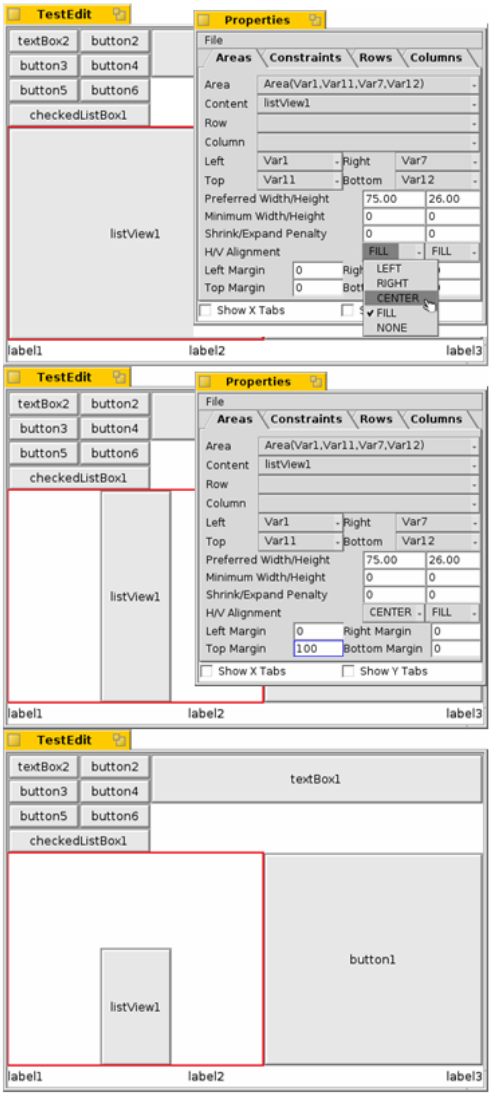

Figure 7: Horizontal alignment of the selected area being changed (top), the GUI after setting the horizontal alignment (middle) and the GUI after setting the top margin (bottom).

```
21            <summand>
22               <coeff> 2 <coeff>
23               <var> Var1 </var>
24            </summand>
25            <summand>
26               <coeff> -1 <coeff>
27               <var> Var5 </var>
28            </summand>
29         </leftside>
30         <op> EQ </op>
31         <rightside> 0 </rightside>
32      </constraint>
33   </almlayout>
```

The `almlayout` tag encloses the whole layout specification, including areas and constraints. It also stores rows and columns, if such abstractions have been used. As a standard, each area tag, e.g. lines 2-8, consists of a symbolic name for the control, and the left, top, right and bottom tabs the area is bound to. Lines 15-17 show optional parameters such as margins and alignments. Lines 19-32 implies a linear constraint $2Var1 - Var5 = 0$.

XALMON essentially separates the layout and the content. Basically, contents are controls, which are defined in the application code. A clean separation is achieved by referring to controls only by their symbolic names, e.g. button1, which is accessible at application run-time. That is, the differences between controls of various GUI toolkits, e.g. .NET Forms, Java Swing, and the Haiku Interface Kit, are ignored completely. Thus, platform independence is fully supported.

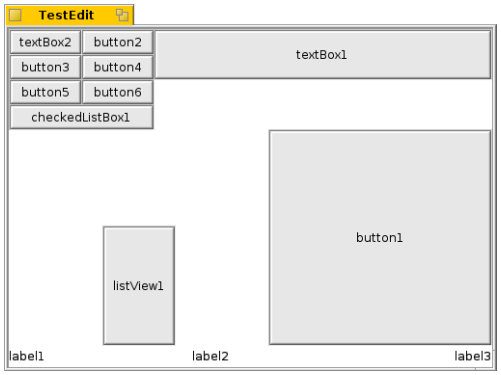

Figure 8: The customized GUI in operational mode.

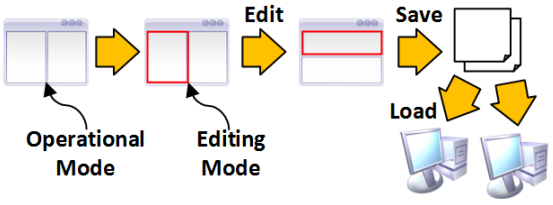

Figure 9: The editor utilizing the document-oriented approach.

XALMON only holds enough information necessary to reconstruct the GUI layout. That is, when a layout is saved, no redundant information is stored. As a rule, XALMON does not save parameters with default values. For example, area alignments and margins will not be saved if they have not been modified from their default values, which are FILL and 0, respectively. Also, ALM-generated constructs are not stored in XALMON. For instance, when a user creates an area, some linear constraints are implicitly generated by the ALM layout manager because higher order constructs use constructs in the lower levels. Duplicates produce needlessly lengthy documents and it may also cause conflicts, destroying the GUI layout altogether. Therefore, only the user-defined constructs are stored.

On loading a layout from the editor, the existing layout specifications are initially removed from the ALM layout manager. Then, the GUI is reconstructed entirely using the XALMON specification. The reconstruction process is performed in the order they appear in the XALMON document. That is, in the above example, areas are added to the new layout first, and then the constraint is added. During this procedure, tabs are added on demand, i.e. if they do not exist in the specification already. Section 7.2 explains the loading process in more detail.

6 Multi-Platform Support

In the context of computing, a platform refers to a level of abstraction for a runtime environment such as an operating system or a virtual machine. ALM is supported on multiple platforms. At the moment, it is available on the Windows and the Haiku (Haiku Inc. 2008) operating systems, where it is written in C# and C++, respectively. There is also a Java version, which in principle should run on all Java-supported platforms. Also, due to ALM's stable yet

simple architecture, ALM can potentially be ported to any other platforms, if necessary. Initially, the editor was only available for the Windows version, hence one component of the project involved porting the editor from C# to C++ and Java.

Porting is the process of translating software from one programming language and/or platform to another. The amount of time required to port a piece of software depends on many factors. Obviously, the most influential factor is the quantity of code, followed by knowledge and experience of the person doing the porting work. It is essential to understand the differences, both superficial and conceptual, between programming languages. Superficial differences mainly include syntactical differences, naming conventions, and coding conventions. These distinctions are usually adapted relatively quickly. At the conceptual level, differences include the framework architecture, data structures, and memory management. In contrast to superficial differences, conceptual differences make porting non-trivial, since much effort and time is required to understand the underlying concepts. It often requires a great amount of research, reading, and seeking for solutions. If the differences between the source and the target programming language are fairly small, e.g. C# and Java, it is possible to just copy, paste and modify large portions of code, thus saving a considerable amount of time.

A majority of software utilizes external libraries for code reuse, which is a recommended practice. However, it is very important to consider compatibility issues when multiple programming languages are involved. Using different libraries for each programming language will not only complicate porting, but it will also inevitably cause maintenance difficulties. Therefore, it is worthwhile to invest a reasonable amount of time looking for a library that is suitable for all languages, if possible. Platform-independent libraries commonly provide wrappers for target platforms. For example, ALM utilizes an open source library for solving linear programming problems, which can be used from all of the programming languages involved, i.e. C#, C++, and Java.

The impact that an integrated development environment (IDE) has on porting cannot be ignored either. Depending on the IDE, helpful features may be provided, which will reduce effort and time required to port a piece of code. For example, Eclipse for Java has a powerful auto-completion function and a dynamic error checker. In particular, the amount of code that needs to be manually typed and the time spent to debug are directed affected by the IDE.

Moreover, modularity of software can largely contribute in reducing the amount of time required to understand the code, if done correctly. ALM allows layouts to be specified in a modular fashion by providing various abstractions. In addition, abstractions are organized in separate classes. Well broken down software is easy to manage, that is, easy to modify and maintain consistently on multiple platforms.

7 Challenges

Our work was done in two main phases. The first phase involved porting the C# editor to Haiku and Java. The second phase involved integrating the document-oriented approach, specifically implementing save and load functions for the editor. In the following, major challenges encountered during these stages will be discussed, which can be regarded as a guideline for future development.

7.1 Porting to Haiku

Anyone porting software to an unfamiliar platform will certainly encounter various problems throughout, and overcoming these challenges will be a learning experience. This section will outline the key conceptual differences between the Windows (C#) and the Haiku (C++) implementation, which caused difficulties while porting the editor. Specifically, the Haiku version required extensive use of the Haiku API, which includes data structures, file system access functions, and a GUI toolkit, among other things.

Firstly, memory leaks in low level programming languages like C++ are one of the most frequent problems. Memory leaks are caused by applications holding unused memory, diminish system performance, and in the worst case may cause the application or the system to stop running. Memory usage has to be explicitly managed in C++, unlike C# where unused memory is automatically freed through garbage collection. Memory has to be allocated before it can be used, and it has to be explicitly deallocated in order to free the memory that is no longer needed. However, if the memory is freed too early, this may lead to memory access violations. Hence, it is important to correctly determine when exactly a chunk of memory is not needed anymore.

Events, e.g. mouse clicks or keyboard presses, are handled quite differently between programming languages. For example, events in C# are managed by using a delegate, a type that references a method, to assign a callback method. In contrast, events are handled through message passing in Haiku. For instance, when a mouse button is pressed, a message is passed to a system-wide application server. Information about the event, such as mouse button and mouse location, can be obtained from the application server to implement handlers for specific event messages.

Another major difference is converting controls, e.g. buttons, into pictures. In C#, it is relatively straightforward, since a method for direct conversion from a control to a bitmap is already provided. It can be used promptly without knowing any of its internals. However, drawing bitmaps using the Haiku API is not as simple. Bitmaps of controls need to be drawn using double buffering, which in the context of computer graphics refers to a technique used to remove artifacts such as flickering. Basically, when a bitmap is drawn, the drawing process is hidden by using a separate buffer. Then, the bitmap is added to the GUI window after it has been fully drawn.

7.2 Loading Layout Specifications

Saving specifications of a GUI layout is relatively straightforward. Relevant data from the specification are collected recursively and written to an empty document by following a specific format, i.e. XALMON. On the contrary, loading a layout is a challenging task, which requires reading data from an XML-based document, also known as XML parsing, and reconstructing the layout specification. Main difficulties related to loading a GUI layout are outlined as follows.

Firstly, the lack of a universal XML parser led to different libraries being used for each version of ALM. Initially, there were some concerns as to whether it would be maintainable for future updates. However, once an XML parser is programmed in each version, it is unlikely to be modified unless major changes occur in ALM's underlying architecture or in XALMON. Moreover, the structure of each parser will remain unchanged, since changes will only need to be made in the common handlers, e.g. code for adding areas.

The process of reconstructing a GUI using data obtained from an XML-based document is quite complex. As mentioned earlier, the existing layout specification is initially removed from the ALM layout manager. The removal process is a recursive one where abstractions are deleted in the order from highest to lowest, since higher level constructs utilize constructs in the lower levels. That is, rows and columns are deleted first followed by the deletion of areas, constraints, and tabs, respectively. Subsequently, new abstractions are added to the specification in the order they appear, dynamically as the document is being parsed. For an abstraction to be added, a set of certain arguments is required. For example, the method for adding an area must have a set of bounding tabs as arguments. Its parameters such as the content name, tabs, alignments, and margins are first saved in temporary data structures. Only after its end tag, i.e. `</area>`, the area is added to the specification followed by setting additional parameters. Also, it is important that all abstractions share a common hash table to temporarily store tabs to avoid adding redundant tabs to the specification. A key-value pair in the hash table holds the name of a tab, e.g. Var5, and the actual object. When any tab-related tag is parsed, e.g. var, left, top, right, or bottom, the name of the tab is checked against keys in the hash table. If there is a match, the corresponding value is used. Otherwise, the tab is added to the specification and the hash table. The hash table is created at the start of XML parsing and lives until the new layout specification is completely produced.

8 Evaluation

The editor was improved to empower end-user customization of GUI layouts by providing saving and loading features. However, we need evidence that such functions truly add value. Thus, it was advisable to assess the usefulness of the editor by conducting an empirical evaluation. Through the evaluation, both the usability and functional aspects of the editor were measured. In the following, the method of the evaluation will be explained, and then the results will be discussed.

8.1 Method

A total of 10 participants were chosen randomly from the computing labs at the University of Auckland. 8 of those participants were studying Software Engineering, and 2 participants had no programming experience. The background information and the purpose of the editor were explained clearly to the participants. Then, participants were asked to conduct a set of tasks with the editor. The tasks differed slightly among participants. An example set of tasks is as follows:

1. Check controls are working
2. Switch to editing mode
3. Swap button1 with textbox1
4. Resize textbox2
5. Center align textbox2
6. Change bottom margin of button2 to 10
7. Save layout
8. Load default layout (default.xml)
9. Load saved layout (your_name.xml)

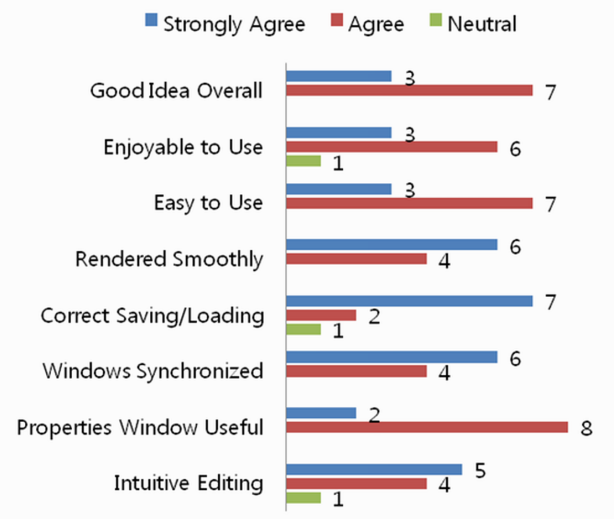

Figure 10: A graph of 10 participants' opinions about the editor.

10. Switch to operational mode

After carrying out these operations, participants were given an opportunity to explore the saved document, i.e. the XALMON specification. Then, they were asked to complete a questionnaire using the Likert scale, i.e. using the options strongly agree, agree, neutral, disagree, and strongly disagree for each item. The questions on the questionnaire are shown below.

1. Using the edit methods, i.e. using mouse and keyboard, was intuitive
2. The properties window provided useful information
3. The GUI window and the properties window were synchronized correctly
4. The layout was loaded as expected, i.e. the layout before saving equals the layout after loading
5. The GUI was rendered smoothly throughout edit, save, and load
6. Overall, the editor was easy to use
7. Overall, the editor was enjoyable to use
8. Overall, the editor is a good idea

At the end of the questionnaire, participants were asked to provide additional feedback about the editor.

8.2 Results

In general, the editor scored well. Figure 10 shows participants' opinions about the editor. Results indicate no disagreements in all questions, which may have been caused by acquiescence bias where participants have the tendency to agree with all questions. Nevertheless, the evaluation provided useful feedback about the editor to date.

In particular, functional features of the editor have been extremely satisfactory. That is, 70% of the participants strongly agreed that the saved layout was loaded correctly. Also, 60% of the participants strongly agreed that the GUI rendered smoothly throughout the editing, saving and loading process, and that information on the properties window synchronized with the GUI window.

Although there were slight differences in tasks carried out by each participant, all participants performed at least one swapping operation. It was observed that 8 participants performed the swap operation through the normal drag-and-drop action on the GUI window, which they thought was quite intuitive. However, one participant used the properties window to swap the controls, which involved many subtasks. Another participant started navigating through the properties window and later realized the quicker approach.

Most participants were able to quickly adapt to the GUI window, but many had difficulties with configuring properties of an area such as setting alignments and margins. While all participants agreed that the properties window provided useful information, many have complained that it was rather difficult to understand. In particular, it was mentioned that abbreviated labels such as H/V Alignment were confusing, and names of tabs such as Var5 were not meaningful. Similar comments were made about the XALMON specification.

Overall, a majority agreed that the editor was easy to use (30% strongly agreed, 70% agreed) and enjoyable to use (30% strongly agreed, 60% agreed). Also, the use of editing methods was found to be intuitive. However, a number of participants made a mistake during the editing process and had difficulties recovering from it. They pointed out that undo and redo functions would be beneficial. Otherwise, all participants liked the idea of the editor overall.

9 Future Work

Currently, the editor has some known limitations. Firstly, it would be advisable to provide undo and redo functions, which will enable recovery from mistakes, and therefore increase flexibility of end-user customization. It is also essential to provide full support for constraints, rows, and columns in the editor.

At the moment, only part of the full document-oriented approach has been applied to the editor. It still lacks some features of document orientation, including a decomposition mechanism and a content description language (Lutteroth & Weber 2008). For example, the editor could further extend its scope by allowing addition of new controls through the GUI window, rather than having to define all controls in the source code of an application. In order to provide such a function, it is extremely important to consider cross-platform and access control issues.

The editor currently lacks a document selection algorithm. Since the save and load facility makes it usable in a multi-user environment, it would be helpful to produce an algorithm that automatically determines when to load the default layout and when to load a user-specific layout. For increased usability, it is also important to provide a more user-friendly properties window, perhaps through the use of more meaningful identifiers for abstractions and labels, and the use of tooltips.

10 Conclusions

The ALM editor enables dynamic end-user customization of GUI layouts in an application-independent and platform-independent manner. It is built into the ALM layout manager, and is thus automatically available for all applications that define their layout using ALM. ALM and the editor are available for Windows (C#), the Haiku platform (C++) and all Java-supported platforms. The ALM editor implements the document-oriented approach to give end-users more control over the GUI. It allows personalized GUI layouts to be saved and loaded similar to documents, a novel feature that can solve the problem of increasing GUI complexity. Some limitations were identified, but the approach was evidenced to be valid.

References

Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S. & Shuster, J. (1999), 'UIML: An appliance-independent XML user interface language', *Computer Networks* **31**(11), 1695–1708.

Cheng, T. (1999), XUL – creating localizable XML GUIs, *in* 'Proceeding of the 15th International Unicode Conference'.

Dantzig, G. B. (1963), *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ.

Draheim, D., Lutteroth, C. & Weber, G. (2006), 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.

Draheim, D. & Weber, G. (2004), *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*, Springer.

Haiku Inc. (2008), 'The Haiku Operating System'. http://www.haiku-os.org/.

Karger, D. R. & Quan, D. (2004), Prerequisites for a personalizable user interface, *in* 'Proceedings of Intelligent User Interface 2004 Workshop on Behavior-based User Interface Customization'.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M. & Trevisan, D. (2004), USIXML: A User Interface Description Language for Context-Sensitive User Interfaces, *in* 'AVI'04: Proceedings of the ACM Workshop on Developing User Interfaces with XML', ACM Press, pp. 55–62.

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

Lutteroth, C. & Weber, G. (2008), End-user GUI customization, *in* 'Proceedings of CHINZ 2008 - 9th International Conference of the ACM's Special Interest Group on Computer-Human Interaction', ACM Press.

McCartney, T., Goldman, K. & Saff, D. (1995), 'EUPHORIA: End-user construction of direct manipulation user interfaces for distributed applications', *Software - Concepts and Tools* **16**(4), 147–159.

Smith, R., Maloney, J. & Ungar, D. (1995), 'The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility', *OOPSLA '95: Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* pp. 47–60.