Improving End-User GUI Customization with Transclusion

Lung-Chen Lee Christof Lutteroth Gerald Weber

Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
Email: llee051@aucklanduni.ac.nz, lutteroth@cs.auckland.ac.nz, gerald@cs.auckland.ac.nz

Abstract

Usually the possibilities for end users to customize GUIs to their requirements are limited. We present a GUI specification and customization system, the Auckland Interface Model (AIM), that represents GUIs as documents that can be loaded, saved and changed by the end user during runtime. GUI layout and GUI content can be customized independently, and GUIs can be decomposed using transclusion. In this paper, we explain why transclusion is an important feature for GUI customization that does not only facilitate the maintenance of a GUI, but also supports its consistency and clarity. Transclusion makes it easier to reuse GUI specifications, and support different customization scopes. AIM was implemented on several platforms and evaluated using the cognitive dimensions framework.

Keywords: Transclusion, GUI customization, end-user development.

1 INTRODUCTION

Graphical user interfaces (GUIs) are the most common type of user interface in modern software applications. Developing a good GUI requires a significant amount of work because of usability considerations such as learnability and suitability for a particular task. It is usually hard to understand the exact requirements of users upfront. As a result, it benefits both developers and end-users if a GUI can be customized easily. Developers can use GUI customization as a rapid development tool, i.e. to react to user feedback quickly. End-users can use GUI customization to satisfy special requirements, such as simplified GUIs for novice users or larger widgets for users with visual impairments. However, most GUI applications offer only limited customization options. Making a GUI customizable usually involves significant extra work for the developers.

In this paper we describe how GUI customization can be improved by supporting transclusion of GUI specification parts. We describe a document-oriented GUI customization system (Draheim et al. 2006), the Auckland Interface Model (AIM) (Lutteroth & Weber 2008), which has been implemented on the Java, .NET, and Haiku platforms. Document orientation means that GUIs can be treated like documents, i.e. they can be modified, saved and loaded at runtime. Transclusion means that a document includes parts of another document by reference. In AIM, transclusion can be used to decompose GUI specifications, and to use and adapt specification parts in a different context. From a developer’s perspective, this facilitates reuse and maintenance. From an end-user perspective, this facilitates end-user development, consistency among GUIs, and scoping of application data.

This paper begins with a discussion of the benefits and the current limitations of GUI customization. The following section introduces the concepts and advantages of the document-oriented approach as well as its implementation in the Auckland Interface Model, followed by a discussion of how document decomposition with transclusion improves the maintenance and the clarity of GUIs. Finally, our approach is evaluated with the Cognitive Dimensions framework.

2 RELATED WORK

2.1 Findings on End-User Customization

Different users may have different requirements for the user interface of an application. Although a GUI designer may capture the common requirements for most users, it may satisfy users’ individual needs much better by allowing the users to customize the application. A study by Findlater et al. (Findlater & McGrenere 2004) showed that the users were able to customize the user interface to increase the efficiency. The majority of users also preferred a personalized interface. A later study (McGrenere et al. 2007) on user’s customization on a word processing software with two types of menu interfaces also shows the users do customize effectively when they are provided with an easy to use customization mechanism. The users prefer the adaptable menu system which they can control over the system-controlled adaptive menu system.

Page et al. (Page et al. 1996) presented a study on the how users customize a word processor. The study finds that 92% of the participants customize the software in some aspects. It indicates that users do customize software applications when they are presented with the options. The results show that the users tend to customize the application more if they use the software more. The work they surveyed suggested that the common obstacles to customizing software were the lack of time and the difficulty to customize the application. The study recommends a mindset of expecting the users to customize the software. The customization should be made easy for them to use. The users should be able to switch between normal task and the customization activities with little disruption.

In the discussion about the past, present and future of user interface software tools presented by Myers et al. (Myers et al. 2000), it was suggested that a “gentle slope system” that the user only needs to learn incrementally would attract more users to overcome
the initial threshold of customization. It also proposes that the capabilities should be supported by the underlying toolkit instead of by the individual applications. This is in line with the Auckland Interface Model, which provides the customization capabilities to all applications that use this system.

The studies above indicate that the users indeed customize the user interfaces of the applications and it helps their efficiency and satisfaction of using the software application. However, at present, many applications provide the end-user customization. The customization is usually limited and may not persist between sessions or it may have troubles in a multi-user environment (Kim & Lutteroth 2009). Providing graphical user interface customization functionalities in the application requires the developers to spend a significant amount of effort.

The traditional approach of developing the GUI of a software application is to include the GUI as code in the program, which is mixed with the code for other functional parts of the program. There are some limitations with this approach. It is more difficult to develop and maintain the program logic and the user interface independently. It may cause difficulties for the programmer and the user interface designer to work on their parts separately. It also makes it hard to customize a user interface. As the user interface is represented as code, the customization requires changing the code and recompiling the program. Furthermore, because the user interface is described in a general purpose programming language, it is possible that the user interface is generated by a complicated section of code. It is difficult for the developers to maintain and modify as it is hard to statically analyze the GUIs by only reading the code without executing the program.

2.2 End-User Customization Technology

Systems with various aims have been developed to enhance the editing of the GUI and they have varying effects on GUI customization for the end-users. Existing approaches for GUI customization work either in the windowing system, or rely on special widget toolkits.

Implementing customization features into the windowing system has the advantage that all applications can be customized. However, customization is very generic and does not take into account the exact structure of a GUI. For example, User Interface Façades (Stuezlzing et al. 2006) is a system that allows the user to copy and paste screen regions to duplicate the GUI controls and arrange them for their convenient use.

Using special widget toolkits drastically restricts the number of applications that can be customized, but customization features can be closely integrated with the structure and logic of a GUI. For example, EUPHORIA (McCartney et al. 1995) is a user interface management system that allows the end-user to construct the user interface of distributed multimedia applications with custom widgets.

2.3 GUI Description Languages

The main benefit of using separate languages to specify GUIs is the separation of concerns (Draheim et al. 2006). Separating the specification of the GUI from the program logic makes the structure of an application clearer, and allows developers to edit the GUI without navigating the program logic. As such, GUI description languages are a software engineering technique more than an interaction principle.

Many GUI description languages have been designed to specify GUIs declaratively. Two better known examples are Mozilla XUL (Cheng 1999) and Microsoft XAML (Bishop 2006). There are other user interface languages such as UIML (Abrams et al. 1999) that have been developed but never gained broad acceptance. All these approaches use documents — typically XML — to specify a GUI, therefore we call them document-based.

In principle a document-based GUI is amenable for end-user development, if end-user development is understood as development through a highly skilled power user. It is possible for power end-users to edit the documents of GUI descriptions to change the GUI. In the case of the code-based GUI, the customization may involve changing the code and recompilation, which even a power end-user typically cannot do. The focus of the current GUI specification technologies is on improving the GUI construction and maintenance for UI designers and programmers.

2.4 Transclusion

The general concept of transclusion is to include one document into another document by reference. In this discussion, the document that includes another document is referred to as the transducing document or the transcluding page. The document that is included into another document is referred to as the transcluded document or the transcluded page.

The term “transclusion” was coined by Ted Nelson. He proposed a comprehensive system of viewing and sharing digital documents, known as Project Xanadu (Nelson 1981, 2007). Transclusion specifies how documents are stored, addressed and edited to construct different versions. It emphasizes that the connection between the transducing document and the transcluded document must be maintained.

Ted Nelson’s idea is that of a “dociverse” that enhances the viewing, editing and sharing of digital documents more than merely imitating paper (Simpson et al. 1996, Nelson et al. 2007). It provides a visualization of documents that indicates the connection between the quoted content and the original context, and allows readers to view the original context. Content is reused by referencing the original source without explicit copying of the content (Nelson 1995). A document is made of a sequence of global reference pointers that point to segments of content (Nelson n.d.). Changing a document is regarded as creating a new version of the document and is done by editing the list of reference pointers to point to a new sequence of segments. The characters at the existing content addresses never change.

Later approaches by other research groups adopt the main feature of maintaining the connection between a transducing and a transcluded document, but support only some of the features of Nelson’s original transclusion idea. For example, Di Iorio and Lumley investigated adding transclusion to XML (Di Iorio & Lumley 2009). It attempts to add a transclusion mechanism to XML without building an entirely new system like Project Xanadu. Another example is Krottmaier proposed a system to perform transclusion for LaTeX documents to enable the inclusion of content by reference (Krottmaier 2002). The transducing document embeds a macro, and a server processes the macro to compose the document with the transcluded content before sending it to a client.

3 DOCUMENT-ORIENTED GUIS

The document-oriented approach views a GUI as a structured document analogous to forms that can be created in office applications (Draheim et al. 2006).
Many capabilities of the GUI can be explained in terms of this metaphor. For example, the ability of the user to interact with many typical controls such as text areas and check boxes is analogous to filling out a predefined form.

Document-orientation means that the UI rendering infrastructure incorporates editing functionality, and therefore GUIs can be edited directly by end-users. The GUI framework behaves like a WYSIWYG office application. Since customization will typically be viewed as a separate activity from actually using the GUI, the GUI framework distinguishes an edit mode, where the GUI can be directly edited, and an operational mode. The developers can give editing access rights to end-users, and hence control the level of customization that is allowed.

In AIM, our implementation of a document-oriented system, the customization function is integrated into a layout manager. As a result, customization is available in every GUI that uses the layout manager, without additional development effort. After switching to the edit mode, parts of the GUI can be visually rearranged, removed or replaced. This is supported by direct manipulation, e.g. users can drag and drop parts of the GUI. More information about the edit mode of AIM can be found in (Lutteroth & Weber 2008).

AIM supports a particularly strong separation of concerns in GUI applications. Whereas other technologies support only a separation of user interface and application logic, AIM further separates parts of the GUI: it is possible to separate the GUI layout from the GUI content and the GUI data. GUI content refers to the widgets in a GUI, and GUI data to the data values that are represented in the widgets. The different concerns of GUI specification in AIM are described in the following.

3.1 GUI Layout

The Auckland Layout Model (ALM) is a mathematical model that allows precise, platform independent specification of GUI layouts, i.e. the visual arrangement of widgets in a GUI, and their dynamic behavior, i.e. how they change under resizing. The layout of a GUI is described by specifying horizontal and vertical tabstops – grid lines that are used to align widgets – and linear constraints. Each widget area is bound by a pair of x-tabstops and a pair of y-tabstops. ALM uses linear programming to compute the layout of a GUI.

GUI layouts can be loaded and saved in a platform-independent specification language. This language refers to the widgets in a layout by symbolic names. This means that the layout can be specified independently from the widgets and their content.

3.2 GUI Content

The GUI content of AIM is handled by the Auckland Value Model (AVM). AVM provides a description language to specify the GUI content, i.e. all the widgets in a GUI and their properties. AVM is platform independent, i.e. it defines abstract widget types. When an AVM specification is loaded, platform-specific adapters are used to instantiate and configure widgets so that they match the specification. AVM supports loading as well as saving.

3.3 GUI Data

AIM also enables the data that are presented in the widgets of a GUI to be customized. For example, such data could be strings in textboxes, selected items in lists, check boxes or radio button groups, or numerical values in sliders. The data in such widgets are stored as widget properties in an AVM description. By loading and saving AVM specifications, users can manage, reuse and share the data represented in a GUI.

4 DECOMPOSITION OF GUIs

Document-orientation supports the decomposition of GUIs, similar to the way it is done for other types of documents. GUI specifications in AIM are composed of parts such as layout areas and constraints in ALM layout specifications, and widgets with properties and data values in AVM content and data specifications. In order to reuse GUI specification parts, transclusion is employed.

When a change is made to a part, the change is reflected in all documents using the part. This is similar to the way a change in a style sheet affects all the documents that use the style sheet. It enables centralized maintenance, while still enabling special customizations in individual user documents. If transclusion is not used and the same part is duplicated in multiple documents, changes would have to be made to all the individual documents, which would frequently lead to inconsistencies. The following sections discuss advantages and usage scenarios of GUI decomposition using transclusion.

4.1 Sharing and Reuse

AIM documents can easily be copied and shared. The capability to share and reuse a customized GUI is a useful feature for end-users as well as for developers. For example, if one person in an office customizes the GUI appearance and the program settings of an application to be convenient for the tasks performed in that person’s team, it would be great for other team members to have the same adjustments to their GUI. Without document orientation, the person would have to repeat the customization steps on each of the team member’s computers. Naturally, it is much easier if the customized GUI can directly be shared with the other team members. Team members benefit from the customization, and retain the flexibility to perform individual adjustments. For developers the concept of reuse is very common, and in fact an essential principle of software engineering. Hence, having a reuse mechanism for GUI parts makes GUI development easier.

4.2 Consistency

A main application of GUI transclusion is to ensure consistency in GUIs. This may be consistency in the layout, the widget appearances, or the GUI data.

4.2.1 Consistent GUI Appearance

When GUI designers develop a GUI, they usually want it to look coherent overall. They are likely to design similar widgets so that they have a common style. For example, buttons should typically have the same background color, and textboxes should use the same font. This is similar to the use of style sheets as commonly used in word processing documents and web user interfaces.

In AIM, designers may create template widgets for a GUI, e.g. a template button with a particular color and font. If the buttons on the GUI all transclude the template button, they have the same appearance as default because the buttons reuse the properties of the template button. When the UI designers or the users
change the template button, all buttons change to the new appearance. As the designers frequently edit the user interface to try out new styles, it is easy to make a global change. It is less likely that individual widgets are forgotten and end up having inconsistent appearances.

A similar approach can be taken with GUI layouts. Sometimes features of a layout are reused. For example, many layouts have toolbars at the top and a status bar at the bottom. By reusing layout parts, consistency can be achieved more easily. For example, a GUI designer may decide to locate the toolbars at the side instead of the top. With the proper use of transclusion, such a change can automatically adjust all the GUI windows of an application suite accordingly.

### 4.2.2 Consistent Program Settings

Transclusion of GUI data is useful for options and preferences dialogue windows, where frequently different sets of settings need to be managed. For example, a user may want to use the same print dialogue settings in a web browser, a word processor and a PDF document viewer. Using AIM, different applications can use the same data for the settings in a dialogue.

The settings are stored as properties of widgets in an AVM description. The user can set an application to use the same AVM description for a part of the GUI as another application. Consequently, different applications can present the same print settings to the user.

AIM can also be used to manage different sets of data that are used in different contexts. For example, if the user needs different settings for printing photos than for printing presentation slides, the user may load different AVM descriptions of the print dialogue with different values of the GUI widgets.

Managing program settings in the GUI is also useful for system administrators. If the users' GUIS transclude the settings from the system wide setting, all users would have the same settings by default. When a system administrator changes a particular system-wide setting, all users get the new default setting. This avoids the possible inconsistencies when settings are changed individually for each user. An administrator can also use this mechanism to manage different settings for different groups of users.

### 4.3 Overriding

Transclusion allows different users to reuse the same GUI parts. In AIM, users can also customize the transcluded parts without breaking the relationship between the transcluder and the transcludee. Individual properties of a transcluded part can be overridden in the transcluding document. For example, a user may transclude all the system-wide print settings, but override the default paper size. The user may later discard their individual program settings to use the system-wide settings again. To do this, the user only needs to remove the personal settings that override the system-wide settings, i.e. the transclusion from the system-wide settings will still work.

In comparison, Cascading Style Sheets (CSS), which is used for web user interfaces but not for standard applications, also offers overriding of UI appearance to a limited extent. The difference is CSS does not allow as fine-grained customization as AIM does. CSS style sheets can only import other style sheets as a whole; it is not possible to import only parts of another style sheet, whereas AIM specifications can transclude individual widgets selectively.

### 4.4 Scoping

Document-orientation makes it possible to customize GUIs in different scopes, such as between sessions or across different applications. For example, an application may hold the only copy of a particular print dialogue window specification. The same dialog window is displayed to all users because the same GUI specification is used application-wide. GUI declaration units can also be specified in user-specific documents, thus allowing different users to have different dialog windows. Separate copies of GUI specifications for the same dialog window enable individual users to have customized user interfaces for the same application, while some parts of a GUI can stay the same for all users. Typical scopes would be user, organization, application, platform, session, and document.

The decomposition mechanism allows different scopes of customization to suit different roles. For example, the system administrator is able to change some settings on an application, and the change takes effect for all users when they use the application. Individual users may customize their own UI to suit their individual needs. The decomposed GUI declaration units are stored at appropriate locations to enable the different roles to access them.

For example, an IT administrator may supply default print settings to all users. As AIM supports different scopes of customization, when the IT administrator wants to change the default values for all users, she can edit the values contained in the AVM description of the print dialogue. The change affects all users. If an individual user wants settings different from the default settings, the user can edit the AVM description that is used only by the user.

Not all document concepts support fully recursive transclusion. HTML notoriously did not do so from the start. In contrast, in our implementation of the document-oriented approach, we support full transclusion, as well as standard document locations. This allows the user or application designer to store information per application, per user or per file. The concept of the scope of a GUI customization is thus mapped to the concept of a file location, a good example, how the document metaphor is mapping the semantics of the customization to familiar concepts.

### 4.5 Scope Visualization

This approach also motivates a refinement of transclusion: the location of the data is not hidden from the user, as it would happen in most other approaches, but actually displayed to the user; moreover it can be changed by the user. Since this location information is chiefly logical and non-obfuscated (i.e. it says “scope: user bert” or “scope: application myEditPad” instead of a URL or an equally opaque UNIX file path) it enhances documentation and transparency of the user interface. This is in stark contrast to many auxiliary documents in standard office applications, where the scope of a setting is very unpredictable, sometimes being the file, sometimes the office suite, sometimes the session, and this scope is not made explicit to the user; let alone being configurable. Finally, the fact that we see GUIS as documents editable by a WYSIWYG editor gives the user full control over GUIS.

In our design, using transclusion enables the system to provide the information about the scope of setting the value of a component on the interface. An example is for changing the settings in the print dialogue of various software applications. Some applications like the Microsoft Office applications use the same print dialogue for all open instances of the same application. If the users change the option to turn
of the controls. Extra visual elements are introduced in the GUI to convey this information.

One possible way to indicate the scopes of the different GUI controls is to decorate controls individually. Another way is to use visual containers to separate the controls in different groups. A visual container encloses the UI components that are concerned with settings that affect only this particular file, and another container encloses controls that belong to another scope. In the example shown in Figure 1, a group box is used to enclose the components. The caption of the container indicates the scope of the settings that are inside this container. In this example, the Printer setting is for this particular file, MyDoc1.odt. If the user edits this setting, it does not affect the same print dialog setting used in other files.

The example has another container that contains the settings for the particular user’s files that are opened by this application. The settings ‘Paper size’ and ‘Orientation’ are inside this container. If the user James changes the paper size from A4 to A3, the other files James opens using the same application Open Text will also have the paper size as A3 when he opens the print dialogue. The extra visual elements clarify which settings have which scopes of effect. The users are better informed when they use this GUI and they are more confident that the changes they make will not cause unintentional changes.

The user interface also allows the user to change the scope of GUI controls when the GUI is in the editing mode. There are several proposed ways of changing the scopes and they will be investigated in our research. As AIM allows the users to move the GUI controls when it is in the editing mode, one possible way to change the scope of a GUI control is to simply move it to inside of the container of the desired scope. Another possibility is to drag and drop the edge of the group container to include new widgets. For example, Figure 2 shows that the scope container for the file has been expanded to include the paper size setting as a setting for only this file.

5 IMPLEMENTATION

The GUI content of AIM is handled by the Auckland Value Model (AVM) component. AVM provides a content description language to specify the GUI widgets and GUI values, i.e. all the widgets in a GUI and the values they hold.

An engine called the AVM Engine processes the content description language. It facilitates instantiating widgets from the content specification, and saving widgets to the content description. A current implementation of the AVM Engine is written in the Java programming language. It converts widgets between the AVM description and the Java Swing GUI components. It also handles the transclusion of the widgets.

5.1 AVM Specification

AVM uses PDStore as its data access layer and it defines the model of the AVM specification. PDStore is a structured database system that uses GUIDs (globally unique identifiers) for all data instances. An association between instances also has a GUID, which is the role ID that is used to find the associated instances of an instance. An example in the AVM specification is that a button has a role ‘background’. We may use the ID of the ‘background’ role to access the instance in PDStore that represents the background value of the button. PDStore allows the AVM Engine to access the AVM specification stored in the database in a more object-oriented manner.

An AVM specification is defined as having a number of widgets. A widget has a value, which represents the value held by a widget such as the text of a text box. It also has a transclusion associated with it to specify which widget it transcludes. A widget is associated with an instance of a particular type of widget, e.g. a text box or a button. This concrete...
type of widget has a number of property values. This is summarized with the following diagrams. Figure 3 is a trimmed down model of the AVM specification showing two types of widgets each with two properties. Figure 4 is a sample of two instances of AVM specifications.

**Figure 3:** PD model of AVM specifications.

**Figure 4:** Two AVM specifications. Widget2 transcludes Widget3.

### 5.2 Transcluding Widget Values

A widget may hold its own value, in which case this value is used. If the value of a widget is null, the widget uses the value of the widget it transcludes. If the transcluded widget does not hold a value, either, it in turn finds its transcluded widget and uses the value of that widget. The search ends when it finds a value.

The following shows an example of the text box that holds the value of the paper size in a print dialogue. The text box in the user’s AVM specification transcludes the paper size text box in the department-wide AVM specification of the print dialogue. It in turn transcludes the size text box in the faculty-wide AVM specification. When the AVM Engine loads the paper size text box, it finds the its value is null. So it looks for the transcluded department-wide widget, which contains the value A4. in the end, the AVM Engine uses A4 in the paper size text box for the user’s print dialogue.

A case could be the department finds that printing on the larger A3 paper is the norm for this department, and decides to change the default paper size for all users in the department. It then replaces its the null value of the paper size widget to A3. The AVM Engine looks for the value in the transcluded widget, it finds A3 and uses it.

**Figure 5:** The widget PaperSize in UserSpec transcludes the value ‘A4’ from PaperSize in FacultySpec.

The following pseudo-code describes loading of transcluded widget values:

1. while value = null and current widget ≠ null do
2. Read the value of the current widget
3. Find the transcluded widget and make it the current widget for subsequent processing
4. end while

### 5.3 Transcluding Widget Properties

The widget properties, which determine the appearance of the widget, may also use the property values of the transcluded widget. For example, a GUI designer makes a button of light green background and dark blue text and wants it to be the default style of the buttons of the GUI. The designer edits all the buttons to transclude this button and leaves all property values as null. Figure 4 is an example. When the AVM Engine loads a button, it looks for the property values. It records all properties that have values. There is none in this case. It then finds the transcluded widget. It checks the widget is of the same type, i.e. a button. The AVM Engine then read property values for properties that do not have values yet. In this case, it records all the property values of the template button. As it has collected non-null values for all properties, the search for transcluded property values is complete. The AVM Engine uses the collected property values to set the appearance of the instantiated widget. For this example, the button would have the same appearance as the template button.

The following pseudo-code describes loading of transcluded widget properties:

1. Get the set of widget property role IDs of this type of widget from the widget adapter
2. while set of property role IDs is not empty and the current widget is not null do
3. Read the value of the current widget
4. Find the transcluded widget and make it the current widget for subsequent processing
5. end while
5.4 Changing Widget Values

When the user uses a new value for a widget that currently uses a transcluded value, the widget replaces the null value with the new value. The widget uses its own value and properties if it holds, while using transcluded values for other properties. Effectively, the widgets' own values override the transcluded values. If the user later decides to change back to using the transcluded value, it is done by changing the value to null to specify the value is be found in the transcluded widget.

5.5 Widget Adapters

The AVM Engine uses a set of widget adapter classes to process the various types of widgets. A widget adapter provides a standard API that allows AVM Engine to access the properties of a particular type of widgets. The API wraps the actual platform specific calls that access the widget properties. Therefore, the AVM Engine is not concerned with platform specific details. New widget adapters can also be easily added to the AVM Engine to provide the processing capability for a new type of widgets.

A widget adapter provides a standard API that allows the AVM Engine to access the properties of a particular type of widgets. The API of the widget adapter is as the following:

```java
public interface WidgetAdapter {
    JComponent New();
    GUID getTypeId();
    GUID getWidgetRoleId();
    Object getValue(JComponent widget);
    void setValue(JComponent widget, Object value);
    GUID[] getAllPropertyRoleIds();
    String getPropertyName(GUID propertyGUID);
    Object getProperty(JComponent widget, GUID propertyGUID);
    void setProperty(JComponent widget, GUID propertyGUID, Object value);
    Object getPropertyDefaultValue(GUID propertyGUID, Object value);
    GUID getPropertyDefaultValue();
    boolean hasPropertyDefaultValue(JComponent widget, GUID propertyGUID);
    void resetPropertyToDefaultValue(JComponent widget, GUID propertyGUID);
}
```

The method `New` is used for instantiating an actual widget in the program. The type of widget adapter determines the type of widget that is created. The `getTypeId` and `getWidgetRoleId` methods are used to tell the AVM Engine the type ID and role ID of this type of widget in the PD model. It enables the AVM Engine to retrieve the corresponding instance of this type of widget in the AVM specification.

`getValue` and `setValue` provide access to the property that is considered as the value of a particular type of widget. For example, the text inside the text box is considered as the value of a text box. The `getPropertyRoleIds` tells the AVM Engine all the properties in a type of widget. `getPropertyType` indicates the value type of a property, so the AVM Engine knows how to parse the property value. The property value is accessed by using `getProperty` and `setProperty`.

---

Figure 6: Widget1 uses the background color from YellowButtonWidget and uses the foreground color from ButtonWidget.
For a property, `getPropertyDefaultValue`, `hasPropertyDefaultValue`, and `resetPropertyToDefaultValue` allow the AVM Engine to retrieve the default value, check whether a widget holds the default value, and set it to the default value for the widget respectively.

The API encapsulates the platform specific calls that access the widget properties. Internally, the widget adapter translates the standard call from the AVM Engine to an actual method call on a GUI component instance in the underlying GUI platform. For example, the text of a text box is considered the value of the text box widget. When setting the text of a text box, the AVM Engine calls: `adapter.setValue(nameTextBox, “Robert”);` The `TextBoxAdapter` then translates it into: `name-TextBox.setText(“Robert”);`

New widget adapters can be developed by implementing the API. The new widget adapters can be added to the AVM Engine to enable it to process more types of widgets.

### 6 Evaluation

We used the Cognitive Dimensions framework (Green & Petre 1996) to evaluate the use of the transclusion mechanism within the document-oriented paradigm. This framework allows us to qualitatively evaluate our approach against a set of standard dimensions, and to find issues with the approach. This framework has been used successfully in the past (Reid & Plimmer 2008). We used the print-dialogue and the page settings dialogue in text processing applications, and compared their implementation in standard text processing applications with a mock-up implementation based on AIM. Dimensions that are not yet addressed are: progressive evaluation, viscosity, visibility and role-expressiveness.

#### 6.0.1 Abstraction Gradient

Abstraction is a central feature of the transclusion mechanism. Through transclusion, sets of parameter settings can be reused, for example as default settings. Through this reuse, the individual places where this reuse happens can be oblivious to the details of those parameter settings.

#### 6.0.2 Closeness of Mapping

The transclusion mechanism is a direct and immediately understandable representation of practices that can be found in every workplace even beyond the IT world. The idea that a certain assumption is by default valid for a large social context, and local deviating rules are explicitly stated, is a widespread notion. A typical example would be speed rules in traffic. There are rules by default, and these rules can be overridden by local changes.

#### 6.0.3 Consistency

The central consistency question is the scope of changes to parameters. In conventional GUIs, as outlined by the examples, the scope of changes is often not explained and moreover counterintuitive. In our approach, once a number of individual scopes have been encountered, such as “settings for department”, “settings for file myfile.txt”, the user will understand easily notions such as “settings for user Pat”.

#### 6.0.4 Diffuseness / terseness

The transclusion concept presented here needs very little symbols, as is evident from the examples. The only symbol necessary is a group box, and the syntax of the context, consisting of the colon separating the context type and the context instance (user: Pat) and the comma separating different context specifications.

The standard context types which will appear in most cases, such as user, organization, and perhaps file, have a middle position between being elements of the notation itself, or being just applications of the notation, albeit in widespread use.

#### 6.0.5 Error-proneness

The transclusion approach cannot prevent all errors, but it is not specifically error prone. The main errors that can be ascribed specifically to the transclusion approach are errors in the chosen context. A context too wide might create unwanted effects. This is usually prevented by access rights, i.e. the settings for the organization context can be used, but not changed by all users.

#### 6.0.6 Hard mental operations

The hard mental operations are not on the notational level. On the semantic level, it sometimes needs some thought where from to transclude a setting, but it is our impression that this is a direct consequence of the desired outcome. The lack of flexibility of current systems however, where one simply has no choice, seems hardly to be an advantage.

#### 6.0.7 Hidden dependencies

Through the context path, the transclusion approach clarifies dependencies that are normally hidden in GUIs.

#### 6.0.8 Juxtaposability

An element of juxtaposability is planned for later versions. It should then be possible to see for one value that is overridden both, the current overriding value as well as the overridden value.

#### 6.0.9 Premature commitment

In the normal editing process, if a transclusion path is changed and a value is overridden with an overriding value, the overridden value cannot be changed at this position. Another timing issue that is in connection with error-proneness is related to the scope of change. We consider the print dialogue and assume the paper size has currently the transclusion context “file”. Now we want to do two changes: change the paper size and change the transclusion context to “user”. Here a timing issue can arise. If one first changes the paper size and then changes the context, then this is not equivalent to first changing the context and then changing the paper size. In the first case we change a setting in a context and then immediately leave that context, thus the changed setting is not used. Intuitively this is likely an error. One of the features that is planned but not yet implemented is a “did you mean” warning in this case. If the user actually meant to do it the other way round and do the change in the new context, the system offers the possibility to bring things in order.
6.0.10 Secondary notation and escape from formalism

Certain mechanisms of the transclusion approach cannot be replaced by informal notions, since they are immediately evaluated in the running system. The transclusion path itself has to refer to an existing context or a new context has to be created. Secondary notation can take several forms in GUIs, and some of them, such as help systems, are beyond this presentation. The notation can in principle be enriched, for example with icons for contexts, however this requires changes to the framework.

A further secondary notation that is of interest here and that is readily available in the framework is contextual comment by the individual user, directly in the user interface. This is certainly not possible in conventional GUIs. In AIM this is right away possible. Every AIM GUI comes with an inbuilt editor, which allows end users for example to add labels. The user can annotate for example the paper size input field with comments on important usages. These comments again are stored in transcluded contexts and this determines the visibility of the informal comment. Therefore AIM acts additionally as a tool support for the dissemination of secondary notation, without losing informality.

Informal content is currently boxed, i.e. the comment takes the form of labels (the borders are however invisible) that are placed at certain points. This hints at further work, where more general annotations with a transparent pane would be thinkable.

6.1 Summary of Evaluation

The evaluation through the cognitive dimensions framework allows us to attribute features of the transclusion approach to standard dimensions. With this method we have found issues where further work is warranted, namely in the dimensions of juxtaposability, secondary notation and premature commitment. While the first two dimensions have hinted us at interesting nice-to-haves for further work, the last dimension, premature commitment showed us a more important issue that should be addressed with more urgency.

7 CONCLUSIONS

Current GUI customization technologies are limited and do not take advantage of decomposition mechanisms such as transclusion. The Auckland Interface Model (AIM) is an open-source, cross-platform technology that uses the document orientation to provide easier and more comprehensive customization for end-users. AIM separates the concerns of GUI layout, GUI content and GUI data, and enables end-users to treat GUIs as documents. This makes it possible to customize GUIs and decompose them using transclusion. The use of transclusion has advantages for GUI developers and end users: it facilitates reuse, consistency, introduces different scopes of customization, and clarifies to the end user where values in a GUI are actually stored. AIM was evaluated using the cognitive dimensions framework, and we could identify the dimension premature commitment, for instance, as a potential issue that should be addressed in further work.

References


