Graphical User Interfaces as Documents

Dirk Draheim
Department of
Computer Science
Universität Mannheim
A 5, 6
68161 Mannheim, Germany
draheim@acm.org

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
The representation of GUIs as documents is a technological trend that has been present for some years, but is only now about to significantly change the way in which most user interfaces are developed. This paper examines this change, explains the reasons behind it and the concepts involved. It compares the old fashioned way of programming user interfaces as code units with the document-based paradigm, explaining why the latter is preferable. Furthermore, it discusses how the document-based paradigm can be extended to a very comprehensive and well defined customization approach for GUIs, the document-oriented approach, which supports the paradigms of end-user development and robust content.

Author Keywords
GUI, document orientation, end-user development, WYSIWYG.

ACM Classification Keywords
H5.2 Information interfaces and presentation (e.g., HCI): User Interfaces [GUI].

INTRODUCTION
Within the last eight years, different markup languages for the description of graphical user interfaces have emerged [2, 11, 13]. These languages are usually based on the XML format and offer the ability to describe WIMP-style graphical user interfaces as we know them from common desktop operating systems. GUI definitions in these languages are textual XML documents, which are interpreted and visualized by a GUI rendering system and linked to a program logic. This approach is different to the traditional approach, which represents GUIs in the program code of an application. It is more similar to the approach used for the user interface of web applications, but web applications usually do not offer the richness and interaction of stand-alone GUI applications. With the proliferation of document-based user interfaces, however, this is likely to change. They have the potential to bridge the gap between the flexible web applications and their stand-alone counterparts.

Despite the fact that these markup languages, like XUL for instance, have been around now for several years, the document-based GUI approach is scarcely used. There exist several technological implementations, but until now they have generally failed to attract the attention of software developers. But this is going to change. For good reasons many companies and organizations have already committed themselves to document-based GUIs in some way or other, and the remaining question is which of the many GUI markup languages will first gain general acceptance.

In order to understand the importance of that change, it is necessary to look at the different user interface technologies. However, the objective of this paper is not so much to discuss the technological side but to shed light on the concepts that those technologies implement. These concepts are of academic interest as they have a considerable impact on usability, robustness of UIs and the feasibility of end-user development.

In addition to providing an analysis of existing technologies and concepts, this paper presents a conceptual contribution which we call the document-oriented GUI paradigm. We suggest a document approach with access control in which the same WYSIWIG system is used for editing and rendering of GUIs in a framework. The ramifications of this approach include not only a simplified technological design, but also advanced customizability and guarantees about the robustness of a GUI.

In the following sections we will therefore present and compare four paradigms: code-based GUIs, GUI-oriented documents, document-based GUIs and finally document-oriented GUIs. In the end we sum up our conclusions.

CODE-BASED DESCRIPTION OF GUIS
Traditionally, GUIs are represented in code units not different in principle from the rest of an application's executable code. The GUI is described in the form of program statements that create GUI controls, set their properties, link them together etc. which makes it a requirement for people deal-

CHINZ 2006, July 6-7, 2006, Christchurch, New Zealand.

ing with this representation to have programming skills. It can be even worse: the code for the GUI can be arbitrarily intermingled with the other program code, thus making both much harder to understand, change and maintain.

Program code is generally Turing-complete, which means that it is possible to describe GUIs in an arbitrarily sophisticated manner. It is possible to describe a GUI by means of a complicated algorithm, e.g. for optimization purposes, even though clarity will suffer. In general, it is impossible to analyze code-based descriptions of GUIs statically, e.g. just by looking at the program code.

To mitigate the necessity of programming in the process of GUI creation, there exist many visual tools for editing GUIs. These tools usually let a user compose a GUI in a more or less WYSIWIG-style and then generate program code for that GUI which can be integrated into the application being developed. In Fig. 1, for example, we see the Visual Studio IDE showing a GUI design on the left and the corresponding generated code on the right side.

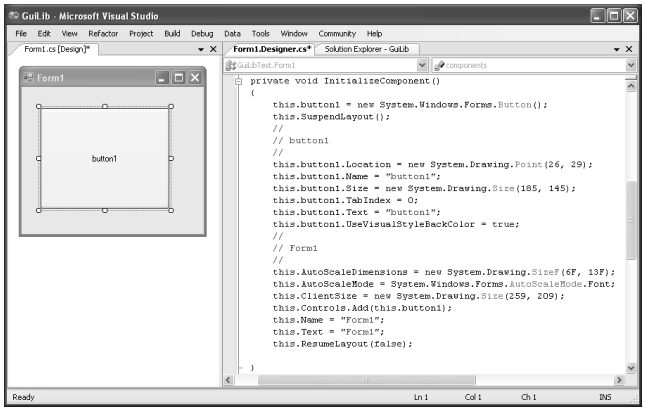

Figure 1. Visual GUI design and corresponding generated C# program code in the Visual Studio IDE.

GUI-ORIENTED DOCUMENTS

Some technologies deal with the relationship between documents and GUIs from a different direction: instead of trying to improve traditional code-based GUIs, they start with the traditional notion of documents. This notion understands documents as compositions of static elements which represent information, but do not allow any input from the user. The strategy of such technologies is to enrich documents with GUI elements, thus giving them basic capabilities for user interaction. However, such technologies usually do not reach the richness and level of interaction of real GUIs, as the concept is based on and restricted by the notion of a static document. Documents are merely used as makeshift GUIs and cannot satisfy the needs of professional GUI design as, for example, outlined in [3]. In the following sections we will examine typical technologies of this kind.

HTML

HTML started out as a document format; its original use was the publication of a phone directory on the network of the CERN research laboratory. It is in its structure very much oriented at static textual documents. HTML already supported the concept of navigation through hyperlinks, but was lacking other types of controls. Instead, HTML supported since long the representation of different static content, like images or mathematical formulas. Only as the WWW became more and more popular, was HTML extended by basic GUI-like controls like buttons and forms. This makes HTML a typical example of a language for documents in which GUI elements can be embedded, although they do not really integrate in a natural manner.

A good example illustrating the GUI-oriented characteristics of HTML are Wikis [1]. A Wiki is essentially a storage for static documents, but its functionality also supports management, retrieval, creation and modification of documents. This functionality makes use of the GUI-like elements that were introduced into HTML, but fails to achieve the degree of interaction and graphics-orientation of a real GUI. The editor for documents in a Wiki is usually text-oriented and the graphical appearance of a document can only be modified by using a specialized document language. This is a sharp contrast to the intuitiveness and clarity of WYSIWIG editors, as they can be realized with real GUIs, and hampers end-user development [12].

One of our priorities in this paper is the unification between editor and viewer. The Wikis address this issue rather on the access rights level, by advocating the right to change to be granted to everybody with the right to read. In most implementations, however, the edit view has to be explicitly entered, and is totally different in presentation as well as in logical structure: the edit view does not contain the actual screen objects that are changed, leading to robustness issues [4]. In contrast, we will propose an approach that not only supports different access rights, but makes them the only difference between an editor and a viewer: the viewer is identical to the editor started in read-only mode. There is a connection to Web 2.0 approaches that try to overcome the read/edit dichotomy in Wikis. Such Wiki-like collaborative work approaches [7] provide solutions for global editability of documents with access control, but do not address the shortcomings that such documents have regarding their capability to describe full-blown GUIs.

The limitations of HTML as the main language of the web lead in turn to limitations of today's web-technology based clients. With the new alternatives of real GUI languages coming up, they are merely embarking on a lucky chance in today's technology landscape. The novelty of the WWW, with its ability to display graphics and navigate through hyperlinks, has long worn out, and description languages for real GUIs could very well lead to new alternatives. Having a GUI instead of merely a static document is not a loss because traditional documents are, in electronic form, displayed in GUIs. Consequently, the concept of a GUI encompasses that of a traditional document.

Office Applications

The primary domain of office applications like MS Word or OpenOffice Writer is the creation of printable documents.

Hence, such applications are usually based on a traditional static document model that is not entirely suitable for describing GUIs. Such a model is usually page based, positioning of elements is oriented at the text flow, and it is required to set fixed dimensions for the document elements. Nevertheless, in particular the widespread use of paper forms has inspired the addition of GUI elements for data input that can be used within the documents. These features have led to a new use case for text processors and their documents. In the new use case, one person has the role of developing a form and then sends it to persons who have the role of filling out only the created form fields, but electronically. Then they send it to the person whose role is to process the form. This approach has very important differences to traditional GUIs.

- This technology can be used in organizations as a way for end-users to produce a substitute for enterprise applications. Enterprise applications can offer this as an alternative pathway for data input: even if the enterprise application has an online form, it also offers the possibility to check out the form as an office document and fill it out offline.
- Unlike in HTML, creation of new forms is naturally done in a WYSIWIG fashion, which makes end-user development much easier and more widespread.
- Unlike with systems that use web forms, the users can create drafts and store them, and keep copies of their submitted forms. This is a notorious drawback of web browsers; they do not save filled out forms correctly.
- The limited capabilities of these applications are not suitable for the development of applications based on the observer pattern, since they do not support push-behavior.

If we look at most of the current advanced document formats, text documents as well as spreadsheets, and the accompanying tools, we see that many of them support browser-like active behaviour. This shows that, in principle, user interfaces can be built with them. But this is more like a quick solution as it may compromise the look and feel of the resulting applications.

THE DOCUMENT-BASED APPROACH

As we have mentioned, the concept of a GUI is more powerful than that of a traditional document. The traditional code-based approach for representing GUIs has, however, certain drawbacks. In this section we want to discuss the idea of using documents in languages that are tailored to the domain of GUIs, and the existing technologies based on this idea. The motivation is to combine the expressiveness of GUIs with the advantages that are implied in the use of documents. The result is an equivalence between GUIs and documents.

Talking about GUIs implies that there must also be program logic eventually. A GUI without program logic does not serve any real purpose. Consequently, documents that describe GUIs are only part of a system that uses such technology. This is illustrated in Fig. 2. The GUI on the left side is described by a document, the program logic on the right side is given as program code. Controls of the GUI are connected to the program logic by events. Events are usually actions performed by a user on the GUI, e.g. clicking a button. The GUI document does usually not contain the program logic itself, but just information about which part of the program logic should be invoked for each event. The parts of the program logic that are invoked by the GUI are called event handlers. The transition from code-based to document-based GUIs has also been described, for example, in [2].

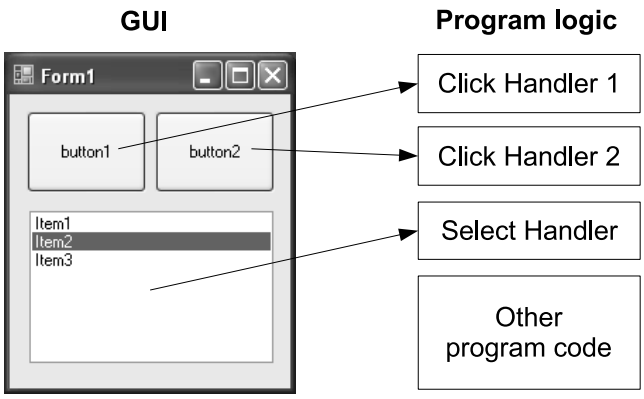

Figure 2. Structure of an application with a document-based GUI.

In the following sections we will look at two of the most promising document technologies for the description of GUIs. We will discuss their potential to change the way in which software is used and developed, and eventually explain the advantages offered by document-based GUIs in general.

Mozilla XUL

XUL stands for XML User Interface Language and was developed by the Mozilla Foundation, which also developed the Firefox browser and other web-related desktop applications. XUL is primarily used for the GUIs of the Mozilla applications, but since these applications consequently include a rendering system for XUL GUIs, they are also suitable for rendering other GUIs. This is most significant for the Firefox browser, as a browser's main job is to provide a user interface to the resources of the web. As we have discussed, the documents of the web are mainly GUI-oriented, but not fully GUI-capable, and the ability to render XUL documents with the same ease as HTML documents closes this gap. As a result, Mozilla offers a browser-centric approach to GUIs that allows for full-blown GUI applications over the web.

The most prominent example of a web-enabled XUL GUI is the Mozilla Amazon Browser. In a suitable Browser, this application can be started by simply opening a URL and offers a GUI to the Amazon online shopping system. The usability and look and feel is exactly that of a stand-alone GUI, while access is analogous to opening a HTML page. Figure 3 shows a screenshot of the GUI on the left side, and a screenshot of the normal web UI on the right. Both UIs run in different tabs of the same browser window.

While it is impressive to see the Mozilla browser switch with ease between these two UI paradigms, there are unfortunately comparatively few applications on the web that use XUL. Although Mozilla's browser holds a good market share, most users use the MS Internet Explorer which does not support XUL, of course. So it is not really astonishing that there seems to be hardly any other real-world example of a XUL application like the Mozilla Amazon browser. One has to say that XUL, despite its nearly eight years of existence, has failed to gain significant popularity as yet.

Microsoft XAML

XAML stands for Extensible Application Markup Language and is, like XUL, an XML-based language for the description of full-blown GUIs that was designed by Microsoft for the new Vista version of its Windows operating system [14]. In contrast to XUL, which is mainly used in the context of the Mozilla browser, XAML will be processed by a part of the operating system, the Windows Presentation Foundation (WPF). Therefore this can be called an operating-system-centered UI approach.

XAML has a good chance to become the first widespread document-based approach to GUIs. And what this could mean with regard to other technologies, in particular those of the Internet, can be just what we already see from the few examples like the Mozilla Amazon Browser in Fig. 3. XAML plus the safe execution environment of the .NET platform is likely to produce more and more web applications with real GUIs. Right now, web developers need to use a mix of several technologies in order to produce web sites that mimic the look and feel of real GUIs. Usually this involves a lot of technical details and programming skills. A widespread document-based GUI technology like XAML with the possibility to connect program logic in an easy and secure manner could alleviate these requirements and give end-user development of GUI web applications a boost. As a consequence, the superior possibilities of real GUIs can leverage better usability. Potentially, the distinction between web sites and GUI applications will blur, and there will just be GUI applications which can be loaded from the net and run either online or offline.

Advantages of the Document-based Approach

The following sections discuss the advantages of document-based GUIs over the traditional code-based ones, which have already been described in a previous section. The paradigm of GUI-oriented documents, as outlined in the section before, bears some similarity with the document-based one and consequently shares some of its advantages. But, as already mentioned, this approach does not support the creation of professional full-blown GUIs.

Separation of Concerns

Separation of concerns [9] is a very important principle of systems design. It means that solutions addressing different requirements in a system should be separate during development, thus keeping its structural clarity intact and facilitating development. One common instance of such a separation is the separation of user interface and program logic, as it is illustrated in Fig. 2. This kind of separation is very common, and also present in other UI modeling approaches like, for example, the form-oriented analysis approach [5]. Document-based GUIs encourage or even enforce such separation because the GUI is given in a document language and this language is tailored to the description of GUIs. If there is support for program logic in such a document language, then it is a marginal feature, and program code cannot be arbitrarily mingled with the GUI description but has to follow its structure. Furthermore, there is a clear notion for connecting a GUI to program logic, i.e. the interface between GUI and program logic is well-defined. All this helps to ensure that GUI designers and programmers can work on their respective parts of the system without interfering with each other.

Another aspect of separation of GUI and program logic is the possibility to easily have multiple different GUIs for the same application. This makes it possible to have different GUIs for different kinds of users, e.g. special GUIs for users with disabilities or GUIs in different languages. Consequently, this approach inherently offers solutions for accessibility and internationalization.

Compatibility

Code-based descriptions of GUIs depend much more on the technical specifics of their particular presentation environment than document-based ones. Different kinds of program code, i.e. in different programming languages, for different hardware, operating systems and other software components, use different execution mechanisms, data formats, linking methods and external code libraries. Code-based GUI descriptions are essentially program code, so an execution platform has to be compatible with the GUI description in all these aspects in order to be able to render the GUI. Cross-platform GUI libraries like GTK+ and abstract-machine-based language platforms like Java or .NET can mitigate this problem, but not completely eliminate it. This is because it is founded in the generally higher complexity of program code compared to a domain-specific document-based GUI description language. GUI documents have to be interpreted in some manner anyway, and the higher-level representation allows an interpreter to deal with the document in a more flexible and error tolerant manner. Also, many document-based GUI description languages are suitable for single authoring [10]. A code-based GUI, on the other side, is generally much more fragile and will not execute properly if even minor technical details do not fit.

Small Footprint and Isolation

With regard to system resources and footprint a document-based GUI is similar to an HTML document on the web. An application with a code-based GUI usually requires an installation on each machine the application is used on, and the installation often requires more access rights than a regular user has. Such an installation takes time and bears safety and security risks, as it may potentially render a system dysfunctional. In contrast to that, applications with document-based GUIs do mostly work without installation and are very easy to access, as the example of the Mozilla Amazon Browser

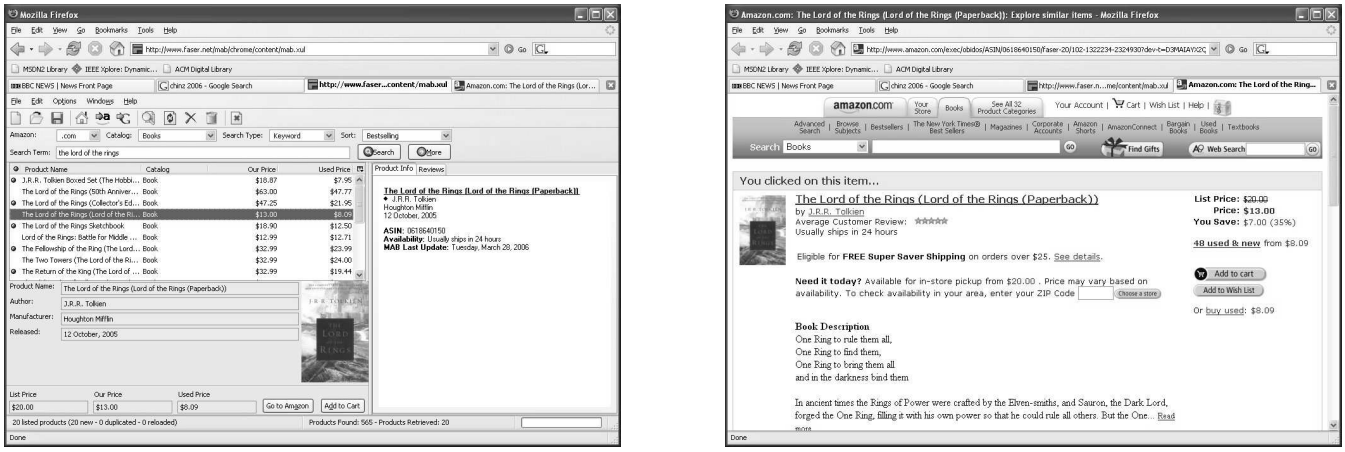

Figure 3. The Mozilla Amazon Browser GUI and the Amazon web UI.

shows. As documents, such GUIs are in general self-contained and portable. The GUI document serves as a central access point to the whole application and can also be accessed over the Internet. Program logic can be loaded on demand. All this facilitates the development of lightweight applications.

Because GUI documents do not run by themselves but have to be processed by a separate rendering system, this rendering system can be designed in a way that guarantees that multiple GUIs are isolated from each other. This is important to prevent a misbehaving GUI from interfering with another. The Mozilla Amazon Browser GUI, for example, is restricted to a single tab of the browser window.

Editability

Because they are documents, GUI documents are editable. This makes it possible for a user to change a GUI, which can usually be done with a simple text editor. This can be, as it was with HTML, a catalyst for end-user development. In contrast to this, code-based GUIs are hard-coded, and, once compiled, they cannot be changed without significant effort.

Non-Universality and Abstraction

GUI document languages are in a sense non-universal. Universality is not necessary for a language that is tailored to GUI, specifically if we recall the concept of separation of concerns. The GUI document language should only focus on the GUI description — not being able to do anything else in this language can avoid a lot of problems. Non-universality can be seen to be one of the most important benefits of GUI document languages. One might think at first that non-universality is a shortcoming. But one has to be aware that universality has its own disadvantages. First of all, universal programming languages allow arbitrary complexity, and shielding end-users from the complexity of program code facilitates end-user development. And there is an even stronger argument: while it is impossible in general to analyse the code of a universal language statically, reducing the capabilities of a language can make static analysis feasible and efficient. Such static analysis of a GUI, i.e. analysis before the GUI is actually executed, can detect safety and security flaws, so that faulty behaviour can be avoided beforehand. Moreover, a reduced complexity of the language makes it much easier for a rendering system to modify a GUI on the fly, e.g. for adapting it to particular layout or look-and-feel settings.

Note that the fact that most GUI document languages are textual is unrelated to the question of non-universality. Program source-code is usually textual, but universal. On the other hand, a non-universal GUI document could be stored as a set of serialized binary objects. As long as an editor for GUI documents is available, this does not affect the user.

The GUI document language is tailored to the domain of GUIs, so it offers a higher level of abstraction for that domain than a universal programming language, which has no such specialization. A GUI language can offer, for example, higher level constructs or shorthands for typical GUI constructions and offer a simplified interface to the developer. As a result, such abstractions facilitate end-user development as well.

THE DOCUMENT-ORIENTED APPROACH

We propose a new approach for GUIs which is compatible with the document-based one, but takes the idea of GUIs as document a step further. In this new approach GUIs are also documents. However, they are edited and displayed with the same tool. The difference between the GUI when it is created and the GUI when it is displayed for actual use lies in the access rights the user has. This approach has several implications and can be seen as a new design pattern for the development of GUIs. We call it the document-oriented approach.

There is a certain analogy to the GUI functionality of some office applications. As we have discussed in the section about GUI-oriented documents, a user can create documents with simple GUIs in such applications, e.g. for entering data into forms. Then, such a document enhanced by GUI elements can be sent to other users, who usually open the document and use the GUI in it with the office application it-

self. Like in the document-oriented approach, the same tool is used for creation and use, thereby ensuring that GUIs are WYSIWIG. But as we have said, the GUI functionality for documents in office applications is incomplete. The user will usually see the GUI in the window of the office application, embedded into the office application's GUI, and not by itself. It is also not so easy or even impossible to connect one's own program logic to the GUI. And without making the whole document read-only, there are few appropriate ways to control write access to the controls of the GUI.

Our paradigm shift is not so much concerned with the actual implementation of the GUI and the editing view. The aim is rather to establish a simpler model of the user interface framework. The document-oriented paradigm offers a number of advantages over traditional code-based GUIs and also the document-based ones. These advantages are described in the following sections.

Unification of GUI Development Tool and GUI Framework

For current GUIs, visual editors have become standard. They can be called WYSIWYG editors, as they present the emerging GUI very similar to the actual running GUI, as can be seen in the screenshot example of Fig. 4. Yet the GUI controls used in these editors to render the drafts might be different to the GUI framework that will be invoked at runtime. Every element of the runtime GUI is mirrored in an element of the editor that allows the editing and customization of this object – functions that are not supported by the controls as they are rendered in a running GUI. The editing process usually ends with a generation step that generates a representation of the edited GUI.

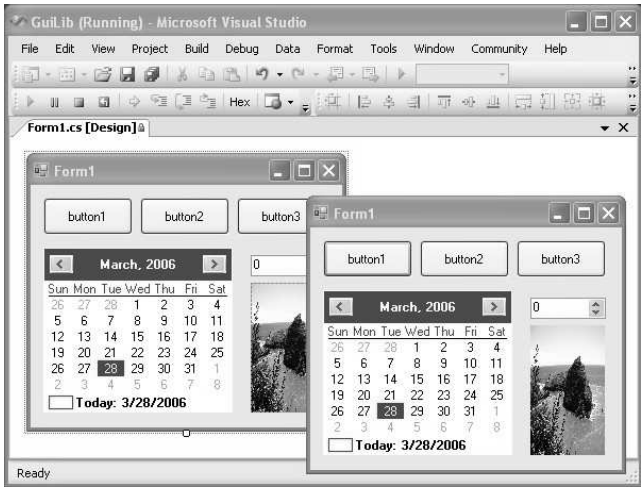

Figure 4. GUI in Visual Studio design tool and as running program.

In our approach we overcome this distinction: the running GUI is presented by the same framework that is used in editing. As a consequence, controls are in principle always editable like in a visual GUI editing tool. However, the GUI framework implements a mechanism for controlling access to the different properties of a GUI, which means that particular modifications can be forbidden depending on the context in which the GUI is shown. The GUI developer will usually have all the access rights and can make arbitrary modifications to the GUI, while the end-user would probably have limited editing rights. An end-user might be allowed to change the sizes or order of certain controls, but not the bindings between the controls and the program logic.

Robustness

The document-oriented model implicitly supports the concept of robust content creation [4] and facilitates configuration of GUIs. Because a document-oriented GUI is rendered with the same software that is used for its creation, the way it is shown during usage is identical to the way it is shown during creation. This means that the WYSIWIG property of the GUI editor can be guaranteed. Even if in editing mode the GUI will indicate the additional rights by additional controls, the GUI will generally look the same, no matter if seen by the developer or user. There cannot be a mismatch between the rendering functionality of the editor and the viewer that could lead to an unviewable GUI since the conceptual identity of viewer and editor makes it impossible to edit something that cannot be viewed, or view something that cannot be edited.

Simplified GUI Editor Design

The equivalence of the developer and user views has implications for the way we construct a WYSIWYG document editor as well. Usually, the GUIs as they appear when they are edited are different to the GUIs when they are used: in a WYSIWIG editor controls of the GUI usually have additional and modified functionality for changing a control's properties. E.g. a bounding box is shown that can be dragged in order to resize a control. It is, of course, a challenge for an editor to show the controls that are edited – possibly by actually using them – but augmenting and modifying their usual behaviour to suit the purpose of the editor. In the document-oriented paradigm we would implement such editing functionality for each control by default, i.e. every control comes with the functionality that is needed to resize or position it etc. This makes the internal design of the editor much easier. Putting functionality for editing into each control makes implementation of controls more difficult, but since they are heavily reused this is well worth the effort.

Simplified Controls

The relation of labels and text input fields illustrates most clearly the principle of using rights. 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. Analogously, a list box where textual entries can be inserted and deleted is essentially the same as a list box with static content, only that the former permits write access to its entries. Even the full-blown WYSIWYG editor itself can be used as a sort of input field in a program, allowing a user to input a whole document, which is possible due to the recursive character of the document paradigm. As a result, the editing capabilities of the controls generalize and simplify the way in which they can be used in programs, leading to a whole range of new possibilities.

Comprehensive Customization

Many of the discussed technologies, like XUL and XAML, define GUI parts like windows for auxiliary dialogues in single XML files. Since these technologies use a declarative approach, we want to call such units of code *GUI declaration units*.

Our approach offers a comprehensive customization approach. Currently, even in the new document-based implementations of user interfaces, the GUI declaration units describing the application interface are seen as a part of the application. In contrast, the document-oriented approach sees the GUI declaration units as part of the individual user documents. In the first approximation, each user document contains its own copy of the GUI declaration units for each auxiliary dialogue like a print dialogue. In current technologies for GUI declaration units like XUL and XAML, this works with by setting the prefill-values declared in these GUI declaration units.

Another example is a feature to resize different panels in a window. If the GUI developer wants to grant this customization option to the user, he then grants the appropriate editing right to the end-user. Vice versa, the resizing can be blocked by withdrawing this access right.

Decomposition Mechanism

If one wants to have more elaborate options to configure the auxiliary dialogues, then one can employ reuse of the GUI declaration units across different user documents. Note that the difference to the document-based approach is now, that the scope of reuse is not necessary restricted to the application. The document-oriented approach works well with decomposition mechanisms like style sheets. The aim of this decomposition is multiple reuse of parts of GUI declaration units with the aim of centralized maintenance: A single change in the style sheet changes the property under question in all the documents that use this style sheet. A conceptually simpler way than style sheets is however to use of an inclusion concept. We prefer to view such an inclusion as an instance of the transclusion principle [8]. Transclusion is the inclusion of a document into another document by reference. The inclusion takes place every time the user document is opened. This process is dynamic enough to enable centralized maintenance, but it is not (necessarily) supposed to deliver updates instantaneously to running applications. The recurring questions of how to deal with the conflicting goals of having a timely update of the changed information and on the other hand how to enable the user to work consistently on one document, uninterrupted by updates, is a different problem that rather belongs in the area of form-oriented interfaces [5].

The Scope of Changes Becomes Obvious

The transclusion approach offers the chance to increase transparency of the user interface. In current user interfaces, there is no way for the user to tell the scope for a certain option, as the following case study shows. 0 In the case of two classical auxiliary dialogues, the print dialogue and the page setup in word processors, the print dialogue is often in the scope of the current application invocation, valid for all open documents, but not persistent. The page setup dialogue on the other hand is in the scope of the current document and it is stored persistently in the document. But in two office suites (MS Office and Open Office for XP) for example, there is no way for the user to tell the difference. Quite contrary, in one office suite, both auxiliary dialogues are accessible under the file menu. A further difference between both office suites is the subdialogue of the print dialogue that allows the user to choose the number of pages per sheet. In the one office suite, clicking “ok” in the subdialogue is only temporary unless the actual printout is performed. In the other office suite, “ok” in the subdialogue does make this change stick for the application session.

With the transclusion principle of document-orientation we can achieve transparency for the user here without making actual changes in the look and feel. The following will hence show that document-orientation does not necessarily enforce a new dialogue structure; it only adds enlightening information. In the discussed case, the sensible introduction of document-orientation would be to associate a file location with each auxiliary dialogue, and to show this file location for example in the header of the auxiliary dialogue. For the page setup, the shown location would be the current file, while for the print dialogue the location would be stored in the user profile. This way the user would have the chance to identify the scope of his action. This proposal is not tested in usability studies, but we argue that this additional information could not be detrimental. We point out that this service of document-orientation is in accordance with the demand of ISO 9241-10 regarding the suitability of applications for learning: “The user is able to obtain information on the model on which the application is based” [6].

The transclusion approach, if fully employed, would of course allow more possibilities for reuse of customizations and settings. Take again a print dialogue as an example. If a user wants to have usually the same print settings for browser and text editor, for example a printer next to the user’s office, then he chooses the same document as print dialogue for both applications. However if the user needs often different print properties for the slide presenter, for example a colour printer, then the user can use a different print dialogue document for the slide editor.

CONCLUSION

We described the traditional code-based approach and the newer document-based approach for the description of GUIs and explained why the latter one will most likely change the way we deal with GUIs. We also presented the document-oriented approach, which goes beyond the document-based one. This novel approach offers a range of new advantages, like a comprehensive and exhaustive concept of GUI customization, robustness and new possibilities in the way controls can be used. The research opens the path for interesting further framework development as well as empirical studies.

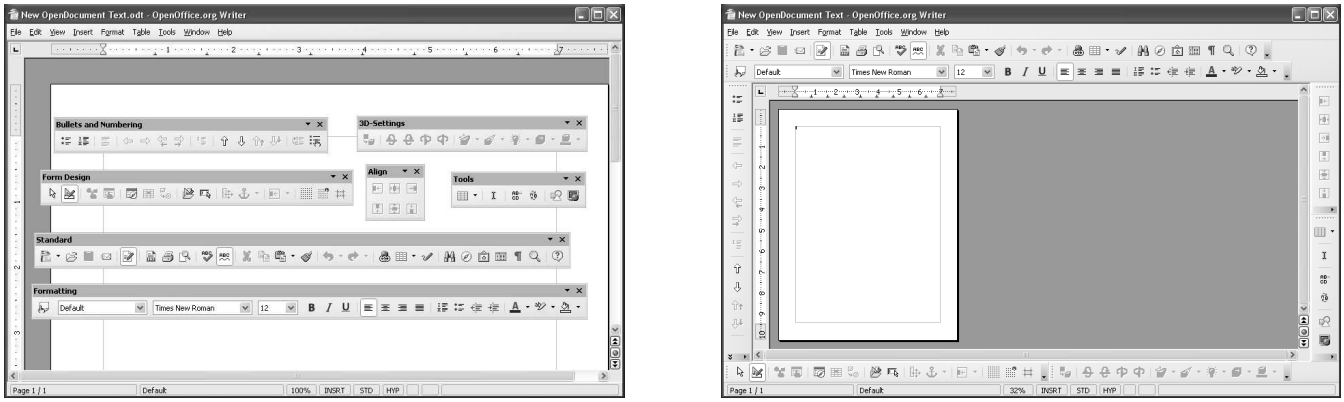

Figure 5. Traditional customization of GUIs.

REFERENCES

1. *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*. ACM Press, New York, NY, USA, 2005.

2. J. Bishop and N. Horspool. Developing principles of gui programming using views. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 373–377, New York, NY, USA, 2004. ACM Press.

3. S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating gui tools for designers and programmers. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 267–276, New York, NY, USA, 2004. ACM Press.

4. D. Draheim, C. Lutteroth, and G. Weber. Robust content creation with form-oriented user interfaces. In *CHINZ '05: Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction*, pages 45–52, New York, NY, USA, 2005. ACM Press.

5. D. Draheim and G. Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

6. International Organization for Standardization. *Ergonomic Requirements for Office Work with Visual Display Terminals (VDT) – Part 10: Dialogue Prinicples*. ISO 9241-10, 1996.

7. A. D. Iorio and F. Vitali. From the writable web to global editability. In *HYPERTEXT '05: Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 35–45, New York, NY, USA, 2005. ACM Press.

8. T. H. Nelson. The heart of connection: hypermedia unified by transclusion. *Commun. ACM*, 38(8):31–33, 1995.

9. D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

10. R. Simon, F. Wegscheider, and K. Tolar. Tool-supported single authoring for device independence and multimodality. In *MobileHCI '05: Proceedings of the 7th international conference on Human computer interaction with mobile devices & services*, pages 91–98, New York, NY, USA, 2005. ACM Press.

11. N. Souchon and J. Vanderdonckt. A review of xml-compliant user interface description languages. In *DSV-IS*, pages 377–391, 2003.

12. A. Sutcliffe and N. Mehandjiev. End-User Development. *Communications of the ACM*, 47(9):31–32, 2004.

13. S. Trewin, G. Zimmermann, and G. Vanderheiden. Abstract user interface representations: how well do they support universal access? *SIGCAPH Comput. Phys. Handicap.*, (73-74):77–84, 2002.

14. A. Wolfe. Toolkit: Longhorn ties platform apps to core operating system. *Queue*, 2(6):16–19, 2004.