Reflection as a Principle for Better Usability

Christof Lutteroth, Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
{lutteroth, g.weber}@cs.auckland.ac.nz

Abstract

This paper explores the principle of reflection, which is well-known from the world of programming languages, and its relation to HCI. We define reflection in a wider sense that can be applied to the world of user interfaces, and argue that the new, generalized notion of reflection can benefit the usability of a system significantly. The paper discusses concrete approaches for the design of reflective user interfaces, and shows that the reflection principle is in fact already used in many existing applications.

1 Introduction

In the context of computer science, reflection is known as a principle from the domain of programming languages. In its most common form, it enables a program to get information about itself and its runtime environment. With this information a program can, for example, adapt itself to new data structures or changes in the runtime environment. The principle of reflection, however, is not just a programming concept. Although it has been discussed systematically in the programming domain, this is just one of its applications. In this paper we show that reflection is applicable to and relevant for the domain of user interface design and implementation as well.

Reflection in user interfaces is in fact ubiquitous. It is intrinsically related to important functions of a program, like functionality for help or features for program customization, and therefore affects its usability. The reflection principle can be used as a lens to gain a better understanding of user interfaces, and inspire new ways for improving them.

The notion of reflection in user interfaces is described in Sect. 2. In this section we discuss concepts and terminology that was elaborated in the context of programming languages, and use them in order to create a taxonomy for features found in user interfaces. To the best of our knowledge such an analogy has not been described before. In Sect. 3 we describe different approaches that can be used in order to create reflective user interfaces. Many of them are well-known but have never been discussed from this point of view. Section 4 describes some examples of reflective features in the user interfaces of relatively common application, and shows that – although this has never been discussed explicitly – there are in fact many reflective features in existing applications. It also delineates a software modeling application prototype which we used in order to examine the potential of reflective user interfaces. Section 5 discusses related research. The paper concludes with Sect. 6.

2 Reflection and HCI

As mentioned, reflection is common in many programming languages and can provide a high degree of flexibility during the runtime of a program. For an overview of reflection in different programming paradigms, see [5]. Reflection in a programming language refers to the capabilities of a program to read and modify information about itself or its runtime environment. This information is called *metadata*. On a HCI level, we refer to reflection as the ability of a user interface to represent and support the modification of itself and its application. The metadata is the information about the user interface or the application that is represented or modified.

Commonly two different kinds of reflection are distinguished: *structural* and *behavioral* reflection. Furthermore, each kind of reflection can be split up into two operations: *introspection* and *intercession*. In the following sections we want to describe these different aspects, which are summarized in the table of Fig. 1, and what they mean on the level of HCI.

2.1 Structural Introspection

In the context of programming languages, structural introspection means that a program can read information about its own structure, i.e. about its data structures and

	Introspection	Intercession
Structural	Representation of data structures & implementation	Modification of data structures & implementation
Behavioral	Representation of information about the UI	Modification of the UI

Figure 1. The different aspects of reflection in a user interface.

its program code. For a user interface it usually means that it presents not only application data but also information about the structure that this data has or can have, i.e. metadata. This provides insight to a user as to what data can be processed by an application and how the different elements of that data relate to each other. A data management application, for example, could help users find the information they need by revealing the data schema of its database. Although less common, structural introspection can also mean that information about a program's implementation is revealed in the user interface. For example, a system might reveal to the user that its functionality depends on some other program, which has to be installed first. This can help a user understand technical problems in case the application is not working correctly.

2.2 Structural Intercession

Structural intercession in a programming language means that types and/or program code can be modified by a program. For a user interface this usually means that it supports modification of structural information about the application's data; i.e. metadata about data structures can be changed. This enables users to adapt the structure of the application data to their needs. For example, templates of any kind fall into this category: they describe common aspects of a group of data instances and thus determine their structure. If we can use templates in a text processing application, it is much easier to make sure that documents are consistently structured or that a particular visual design is preserved. Structural intercession can also relate to the implementation of an application. This means that the user interface supports to some degree modification or configuration of the internal functionality. For example, a user interface might offer functionality for extending the system with plug-ins, so that the user can extend or reduce the functionality offered by the application.

2.3 Behavioral Introspection

Behavioral introspection in programming languages means that it is possible to read information about the behavior of the runtime environment. E.g. it is possible for a program to look into the code of the interpreter, i.e. the abstract machine, that executes it. We want to use this term if a user interface offer users the possibility to acquire information about how the system behaves towards the user. In other words, it means that the user interface reveals information about itself. For example, it might show what happens when a particular button is pressed, or how a particular setting of a control affects the system behavior. A system with behavioral introspection has a user interface that is in a way self-explanatory because it grants the user a look under the hood of the system into the underlying mechanisms. Its semantics are made perceivable for the end-user.

2.4 Behavioral Intercession

In programming languages, behavioral intercession means that you can change the behavior of the runtime environment, i.e. the way the runtime environment behaves towards a program. We want to use this term if a user interface makes it possible to change the way a program behaves towards the user. In other words, the user interface of a program can be configured. This is very important, for example, for professional users who need to tailor the user interface to their professional environment in order to maximize productivity, or for better accessibility of an application. An application might allow users to change the controls available on the user interface, or use alternative input and output devices.

3 Approaches for Reflection in User Interfaces

In this section we want to discuss some approaches for creating user interfaces with reflection capabilities. Some approaches are well-known and some are new. None of the approaches have been discussed in the context of reflection before.

3.1 Generic User Interfaces

Many applications are able to process different types of data. Most popular text processing applications, for example, allow the user to edit documents of different formats, such as ASCII text, ODF or HTML. This is possible because these applications use an underlying data model that is flexible enough to deal with the particularities of the different data types. Whereas the user interface generally stays the same, such an application informs the user about the characteristics of a data type in some way or other, thus providing some degree of structural introspection. A text processing application, for example, will not provide the same functionality for ASCII text documents as for HTML. Some applications provide more structural reflection by allowing the user to import and use data of a previously unknown data type. Some spreadsheet applications, for example, are able to import data from structured text files with different formatting or delimiting characters. An even higher degree of structural reflection is provided when an application allows the user to define and use completely new data types. This is the case, for example, in some desktop databases. The key element in all these applications is that they provide a generic user interface for several, potentially very different, data types. Internally this requires a flexible data model, and sometimes also involves structural reflection of data types by the program itself.

Another common way of implementing generic user interfaces is to apply the “don’t ask what kind” principle. This means that code is able to process data of different types by making use of dynamic binding. In object-oriented programming languages this is done with method polymorphism, whereas in other languages function hooks are used. It is commonly used in component technologies like Microsoft’s Object Linking and Embedding (OLE), which allow data of one application to be embedded and edited in others without them knowing about each other. E.g. it is possible to embed a spreadsheet into a text document and change the spreadsheet from within a text processor. This does not only allow applications to provide a higher degree of structural introspection into different data types, but also enables a higher degree of behavioral intercession by providing new means for the extension and configuration of the user interface.

3.2 Metadata Integration

In some cases the reflective parts of a user interface, i.e. those that deal with metadata, can be integrated with those that support data operations. While the data of an application serves as its productive input and output, its metadata may describe what data types exist and how they are structured. Often the metadata describes commonalities of data instances, which can be important for preserving the integrity of a system, or just useful for information reuse. For a text processor, for example, such metadata would typically include the page size setting, the default font and page formatting. The user interface could allow a user to set metadata globally, i.e. for all data instances, or with more sophisticated mechanisms. Many applications support a notion of templates, which can be used to set metadata for groups of data instances. In many applications the part of the user interface responsible for data is separate from the part allowing metadata access, as pointed out in Fig. 2.

In the world of programming languages metadata and data are usually treated in the same way. Metadata is just special kind of data, represented with the same data model and accessed with the same operations. Sometimes an analogous approach is possible in a user interface: data and metadata can be represented in the same user interface and possibly even modified with the same functions. We want to call this metadata integration because the user interface for metadata is integrated with the user interface for data, which usually means that the internal representations of data and metadata are integrated as well. This is depicted in Fig. 3.

A common problem with reflection in programming languages is known as *meta-confusion*, which means that the different data and metadata levels that might exist are easily

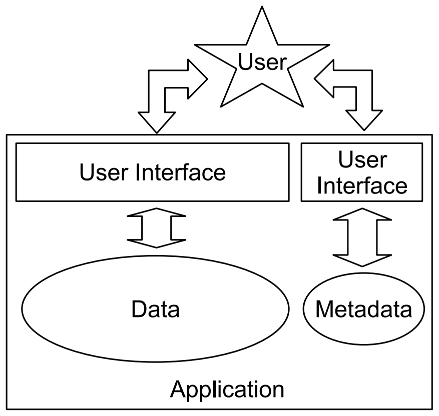


Figure 2. System with a non-reflective user interface.

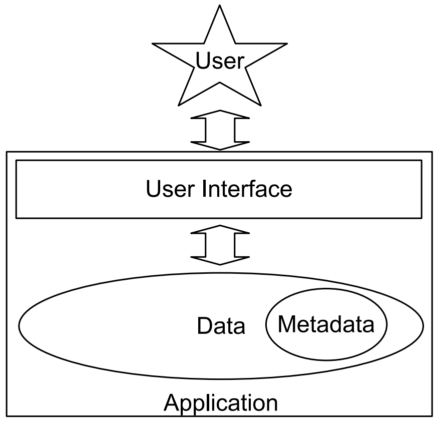


Figure 3. System with a reflective user interface.

confused. We have to be careful to avoid the same problem for metadata integration in the user interface: if it is not immediately clear if a user is modifying data or metadata, i.e. a simple data element or a metadata element that might change the way the application behaves, this will lead to mistakes. Metadata integration may enhance accessibility and result in a coherent internal design, but metadata and data should be distinguished clearly in order to prevent a loss of clarity.

3.3 Plug-in Architectures

A certain degree of reflection can be achieved by using an architecture that can be extended by plug-ins. Insertion and removal of plug-ins is often supported in the user interface so that it can be done by the end-user. Sometimes plug-ins just add internal functionality to a system, which results in structural intercession, but sometimes they also extend or modify the user interface, resulting in behavioral intercession. The ability of a program to let the user see what plug-ins are installed, and possibly how they are configured, enables a degree of introspection.

3.4 Direct Data Access

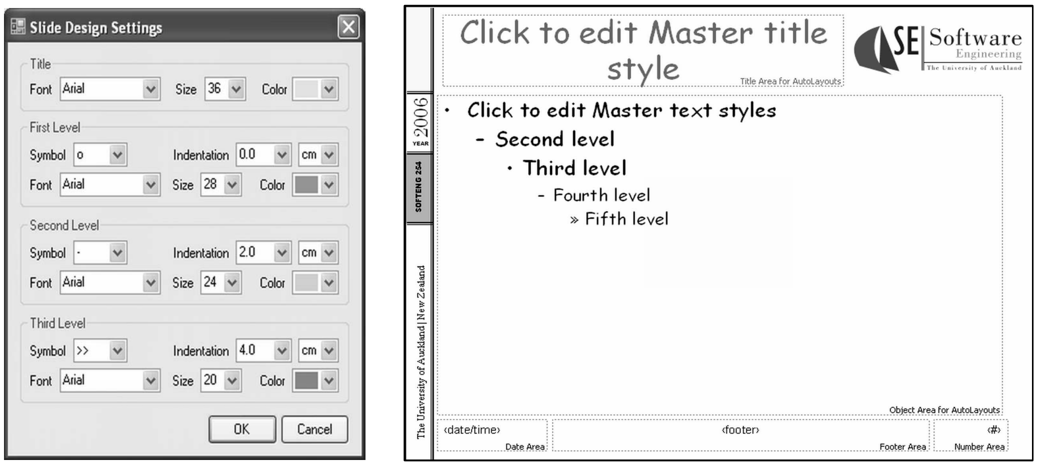
A different approach for reflection is possible if the way data is handled within an application comes very close to the way end-users should be able to handle the data. Instead of creating new layer of functionality for the end-user, we can provide a front-end for the existing data structures and operations. This can produce very elegant, flexible and minimalistic designs and expressive user interfaces. For example, such an application can leverage the reflection capabilities of its runtime environment, thus reusing a great deal of functionality. A data management application that should be able to let the user use new, formerly unknown data types, written in a programming language that supports reflection, could use existing features for data introspection, dynamic loading, and introspective access. Reflective capabilities of the user interface could thus be directly delegated to reflection in the implementation. A direct data access architecture is, for example, the naked objects approach [21].

3.5 Document-Oriented User Interfaces

A natural choice for systems like Fig. 3 would be to use the concept of document orientation [7]. This concept has much in common with the concept of direct data access, specifically with regard to metadata. In conventional applications, metadata is often presented in auxiliary dialogues like the print dialogue. Such auxiliary dialogues have the disadvantage that they present data that may have very different lifecycles, or scopes; some data live only for the time the window is open, other might be permanent immediately if they are changed in the dialogue, others again are only permanent after clicking ok. In the latter case there are even more subtle differentiations between data valid only for the document, and data valid for the user profile. The document-oriented approach tries to provide the user with an intuitive understanding of these lifecycles by using the document metaphor instead of auxiliary dialogues. The metainformation is part of a document, typically a certain part of the whole document. The presentation of the metainformation as part of the document is actually no big change from using an auxiliary dialogue, but it adds the desired clarification of the semantics. In principle, such document editors follow to some extent the principle of direct data access. The presentation as a part of the document, as it is proposed here, gives the user an immediate idea of the scope and lifecycle of the change: the change is valid for the document, and it becomes permanent in exactly the same fashion that other changes would become permanent. This intuitive understanding is a direct consequence of the fact that this metainformation has become integrated with the ordinary data; we can infer its lifecycle behavior from its property of being just data. For the software architecture this means that there is no need for the software engineer to create a separate operational process like opening an auxiliary dialogue. The different scopes of the metadata can be expressed by using a decomposition mechanism for documents. Such decomposition mechanisms are anyway important for many real-life applications of document editors. Documents of the size of a book should rather be decomposed into convenient blocks of chapter size. In document orientation, this same reuse mechanism can be used; for example the profile specific metainformation is stored as a single file in the profile of the user, and by including always the same file the desired scope of this information can be achieved. This approach is very favorable from a software engineering point of view since it offers a substantial reuse, in this case of the decomposition mechanism, and it offers a large degree of initial flexibility, which in turn can be either exposed to the user, for example by allowing to delete the reference to the profile-specific data, or it can be encapsulated and released only in fine doses (for example by making it impossible to drop the inclusion of the profile-wide metainformation).

3.6 Templates and Master Instances

One choice for reflective user interfaces that follows the document-oriented approach and also supports the intuitive use of reflection is the usage of master instances like master slides in slideshow editors. Master instances are different from style sheets in that a master instance resembles an ordinary part of a document, for example, in the slideshow editor it looks much like another slide and is therefore an


Slide Design Settings

Title
Font Arial Size 36 Color

First Level
Symbol o Indentation 0.0 cm
Font Arial Size 28 Color

Second Level
Symbol - Indentation 2.0 cm
Font Arial Size 24 Color

Third Level
Symbol >> Indentation 4.0 cm
Font Arial Size 20 Color

OK Cancel

Click to edit Master title style

- Click to edit Master text styles
  - Second level
    - Third level
      - Fourth level
        - » Fifth level

<date/time> <footer> <#>

Title Area for AutoLayouts
SE Software Engineering The University of Auckland
2006 YEAR
SOFTENG 254
The University of Auckland | New Zealand
Object Area for AutoLayouts
Date Area
Footer Area
Number Area

Figure 4. Auxiliary dialogue and master slide in a slideshow editor.

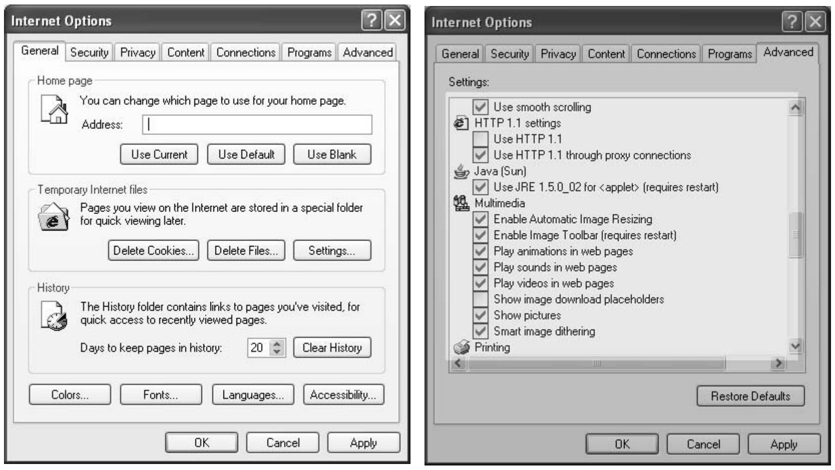
instance. The master instance can be formatted like an ordinary part of the document, but any formatting operation on this master instance is automatically applied to all parts of documents that are based on this master instance. This approach to present metainformation has now become standard in slideshow editors. These editors offer a master slide view as shown in Figure 4. This master slide has many properties of ordinary slides. However, formatting this master slide has immediate effects on all slides that are derived from this master slide. Contrast this with the auxiliary dialog window shown on the left hand side of Figure 4. This behaviour of the master slide is not simply the concept of having data and metainformation in the same presentation format. In this case the additional degree of unification is that the metainformation, for example the standard setting of the font size, is shown in the master slide not as a numerical value, but as the font size of a prototypical text. In this way the master font size can be manipulated with the same tools as the ad-hoc font size. Hence the change of the metainformation is exactly the same operation as the change of the information in the individual instances. The master instance concept can be implemented to different degrees. in the case of slide editors, the master instance is often hidden in special menu entries. The degree of integration could go even further, if the master slide is stored as an actual slide, perhaps in a separate repository, but in clear analogy to ordinary slides. There is one particular drawback of current slide editors, namely that the slide master is always file-specific, and a consistent change of the master slide across even a directory of talks is precluded. This problem would be solved naturally, but reusing the same master instance by the decomposition means that were mentioned in Section 3.5. From a software engineering viewpoint, the master instance concept can deliver several advantages to the software architecture: ideally, the manipulation of the master slide does not require additional functionality, but is done by the same code that does the instance manipulation. Slight differences in functionality, for example a restriction in the ability to delete special objects like the title, can be implemented with dynamic linking. The master instance concept can be used in many cases, as will be explained in section 4.3. A closely related concept are server pages, such as Java Server Pages (JSP) and Active Server Pages (ASP). For example, consider a JSP for presenting a table. If this is done the most natural way, then the JSP contains indeed a proper table, but with only one row [6]. Such templates, although they have actually program-like properties, can then be edited just like instances, for example the table grid lines can be formatted, and these edits have the intended effect on all the pages subsequently generated.

4 Examples

In this section we want to discuss what role reflection plays in existing technologies. We will see that reflection is used in several systems, although the underlying principle has never been discussed explicitly. We will also discuss cases where reflection is absent but could be applied to improve the usability of a system.

4.1 Unix-Style Operating Systems

Unix-style operating systems are known for a motto which says that “everything is a file”. This is true for system entities like console connections, connected hardware devices, some configuration settings and, of course, all ordinary data. The structure of the system can be introspected and certain settings can be changed, thus this approach is an example of structural introspection and intercession. While most files contain ordinary data, others contain information



You can change which page to use for your home page.

Address:

Pages you view on the Internet are stored in a special folder for quick viewing later.

The History folder contains links to pages you've visited, for quick access to recently viewed pages.

Days to keep pages in history: 20

- ✓ Use smooth scrolling
- HTTP 1.1 settings
  - ☐ Use HTTP 1.1
  - ✓ Use HTTP 1.1 through proxy connections
- Java (Sun)
  - ✓ Use JRE 1.5.0_02 for <applet> (requires restart)
- Multimedia
  - ✓ Enable Automatic Image Resizing
  - ✓ Enable Image Toolbar (requires restart)
  - ✓ Play animations in web pages
  - ✓ Play sounds in web pages
  - ✓ Play videos in web pages
  - ☐ Show image download placeholders
  - ✓ Show pictures
  - ✓ Smart image dithering
- Printing

Figure 5. Two different ways to present advanced settings.

about internal system entities, i.e. metadata. The metadata can be accessed and processed in the same manner as ordinary data files, e.g. using the same set of command line tools. With the file system being the user interface, data and metadata are handled similarly and thus a degree of metadata integration is achieved.

The idea of the file system as the operating system's main user interface is taken a step further in the Unix successor, the Plan 9 operating system from Bell labs [19]: even more system entities are represented as files, including windows, processes, and almost anything else available in the operating system. This is an example of how a homogeneous, integrated user interface for data and metadata can yield a very slim and elegant design that facilitates the system's usage. However, like in programming languages, the principle of reflection is still used on a rather technical level. The file system approach provides by itself rather an interface for developers than for end-users. But a consistent interface on a lower level of a system is much more likely to propagate into a clean interface on a higher level: a direct data access user interface layer on top of the file system could use the same reflective metaphor and thus provide the same advantages.

4.2 MS Windows

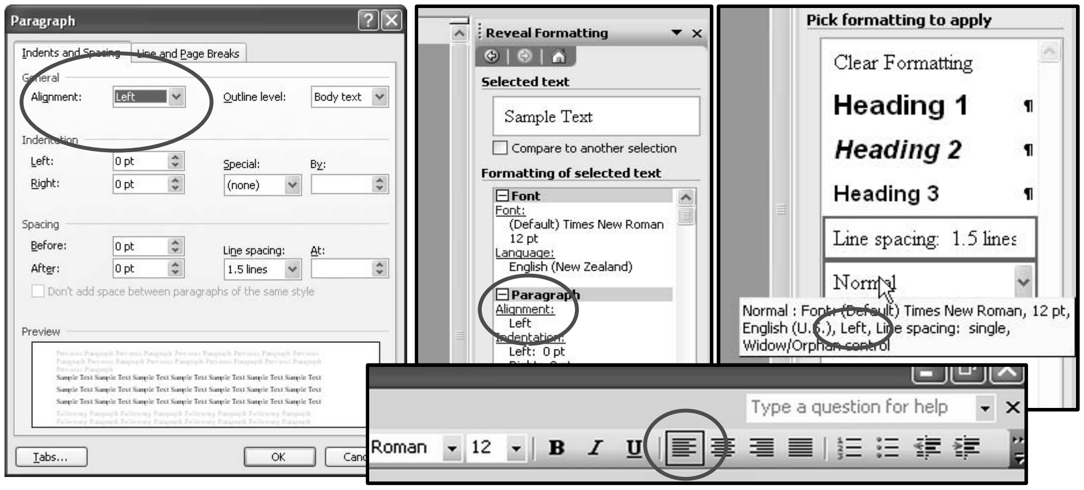
In the following we want to examine the user interface for some typical system configuration tasks in the Windows operating system.

A practical example for achieving behavioral introspection is a function for help. This gives users the possibility to look up parts of the user interface that they do not understand, and hence gain knowledge about system behavior. In order to enhance usability a help function can be context-sensitive. And it can offer even very small pieces of information, like the tool tip labels that appear when the mouse button is hovered over certain parts of some UIs.

There are also other ways of representing the underlying mechanisms and concepts of a system, which are less canonical and may require some creativity and innovation. In graphical desktop environments like MacOS and MS Windows, for example, windows can be minimized into icons in a task bar, and this is visualized by an animation that shows the window shrinking into its icon on the screen. While this may be superfluous eye-candy for computer savvy users, it suggests to unexperienced users that the minimized window is not gone and where it can be found. A similar effect is applied for window maximization.

The use of a registry based on a directory service is an example of structural reflection; the metainformation of many programs is presented in a single interface. This instance of a reflective user interface is a replacement for setting environment variables in a traditional batch file executed on startup. The reflective interface has the advantage of simple semantics: The environment variables for a certain application can be looked up in one particular place. In a batch file, perhaps even with conditional execution, it can soon be intransparent where the actual settings are made.

A closely related example are auxiliary dialogues for advanced settings, for example printer settings. In principle they follow the same logical structure, except that they are often found within applications as opposed to a central repository. Again, thinking in terms of a reflective interface makes it obvious that the more uniform presentation is the tree view that presents the set of settings rather like a document. The tabbed panel approach has a stronger emphasis on presentation, and this can lead to strange imbalances, if for example the number of settings is very different from rubric to rubric, leading to some empty and some very large tabs (Figure 5).


Paragraph
Indents and Spacing | Line and Page Breaks
General
Alignment: Left
Outline level: Body text
Indentation
Left: 0 pt
Right: 0 pt
Special: (none)
By:
Spacing
Before: 0 pt
After: 0 pt
Line spacing: 1.5 lines
At:
Don't add space between paragraphs of the same style
Preview

Reveal Formatting
Selected text
Sample Text
Compare to another selection
Formatting of selected text
Font
Font:
(Default) Times New Roman
12 pt
Language:
English (New Zealand)
Paragraph
Alignment:
Left
Indentation:
Left: 0 pt

Pick formatting to apply
Clear Formatting
Heading 1
Heading 2
Heading 3
Line spacing: 1.5 lines
Normal
Normal : Font: (Default) Times New Roman, 12 pt, English (U.S.), Left, Line spacing: single, Widow/Orphan control

Type a question for help
Roman 12

Figure 6. The same metainformation is presented in several different ways

4.3 Office Applications

In many applications, there are several different ways how the same metainformation is presented to the user. In a current version of the popular text processing system MS Word, font size can be set ad-hoc or for a paragraph type. For the ad-hoc setting of font size, we can identify at least two places where it can be set, both differing in presentation slightly. If the font size of a paragraph type can be seen and changed with the application, then this is a case of reflection in our terminology. For the font size of paragraphs, there are at least three different presentations used (Figure 6). This means that this example of metainformation is presented in three different ways. A user that is supposed to work with these different presentations in a consistent way must have an abstract model of the concept of font size. By virtue of this model he is able to recognize the same metainformation in the different presentations. A better alternative would be to use only one presentation, and the reflective approach would foster such a parsimonious solution.

The concept of a master object has become well known through slide editors, but this concept can be used far beyond. This starts to happen in some other graphical tools, but has not yet happened in the most popular tools. In Word processing tools for example, there is no concept of a master, although it would well be conceivable. Take again the example dialogues from Figure 6. Although they all are different presentations, they all are interfaces different from the document. Obviously there is a correspondence between the leftmost dialogues in Figure 4 and Figure 6. In the example word processor there is no correspondence to the master slide approach, although this is possible. A master paragraph could be a single paragraph that is formatted like an ordinary paragraph, but changes to its format are immediately visible in the changed documents.

Many desktop applications allow a user to configure the tool bars, i.e. the panels with buttons for common functions, by adding/removing buttons and moving a tool bar to different locations. Another example is the possibility to define shortcut key combinations for functions that a user wants to be accessible from the keyboard. Other examples for behavioral intercession include the setting of big fonts or buttons for better accessibility on the screen, changing the order, size and location of different panels, and configuration of the dialogues that a system offers.

4.4 Meta-CASE Tools

Meta-CASE tools, e.g. the one described in [23], support configuration and generation of specialized visual editors for 2D graph-like diagram types. Because they usually aim at being as generic as possible, i.e. being able to generate many different such editing tools, they have to explicitly deal with metadata that describes the structure of a diagram, its appearance, and the behavior of the editor. A user has to define a model for a diagram type, and then specify how each of the model elements are visualized. Providing both information about models and functionality for modifying it means that the user interface of a meta-CASE tool naturally supports structural reflection. Being able to observe and change the editing functionality that is provided for a generated diagram editor means that behavioral reflection is supported as well.

To illustrate the idea of meta-CASE tools, Fig. 7 shows a simple model for state machine diagrams and its relation to a visual representation. The model specifies types for states, transitions and labels on transitions, which are shown as filled circles. Associations between the types are shown as connections between the circles: each transition refers to one source and one target state, and has one label. The graphical representation of a state is a circle, and that of a transition an arrow between the circles of its source and target states. The label of a transition is placed on its corresponding arrow. In the same manner many different diagram types can be defined. Tools that can be generated with such specifications are, for example, editors for data models, process diagrams and electronic circuits.

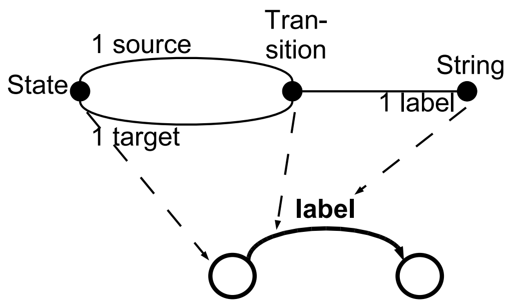


Figure 7. Model (top) and graphical representation (bottom) of a state machine diagram.

4.5 The AP1 System

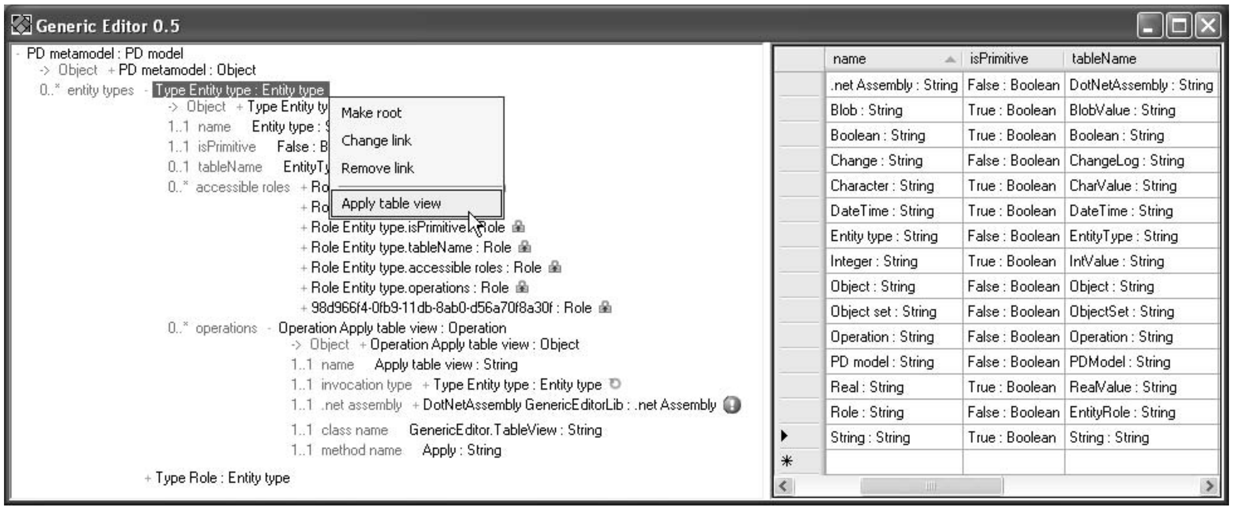
AP1 is a research prototype we are using to examine old and new concepts for reflection in user interfaces. It is a data management and manipulation application for software models, e.g. models for functional specifications, user interfaces and source code. But it is very generic so that it could be used for other data as well. AP1 consists of a data repository, which stores all data, and an extensible generic editor, which can represent data using different views and manipulate them using different operations. The window of the generic editor can be split up into panels, and each panel can contain a different view. New views and operations can be added to the system during runtime.

Figure 8 shows a screenshot of the generic editor. The left panel uses a generic tree view, which allows a user to view and edit any data in the repository. The black parts in the tree view are data instances and the gray parts are roles which can be used to connect instances to other instances. An instance is given as an identifier, a colon, and the type of the instance. Instances to the right of a role are connected to the parent instance above them by that role. The gray parts also contain information about multiplicities, i.e. how many instances can be connected by this role minimally and maximally, or information about inheritance relations between instances. Thus, information about the structure of data can always be made visible.

Data and metadata are stored in a single internal data model. All metadata is handled like data, i.e. represented in the same data model and with the same user interface. Most operations can be used on data as well as on metadata, e.g. editing operations or functionality for search, and both data and metadata use the same mechanisms for version control. The data shown in the left panel is the metamodel of the data model used for the repository. We can see two metatypes that are connected to the "PD metamodel" instance: "Type Entity type" and "Type Role". The right panel shows a table view, which can be used to view and edit the instances of a particular type. The instances shown here are all instances of "Type Entity type", so the table lists all the different types that are currently present in the repository. Metadata types can be extended just like data, and modified to a degree that ensures system integrity. It is also possible to link any data with any metadata. This can be used, for example, in order to attach documentation to a data type. AP1 supports subtyping as a non-destructive form of specialization: if users need a special form of a particular type, then they can create a subtype of it that reuses the structure that has already been defined, extends it, and can be used in place of the original type. The original structure remains undisturbed.

Unlike most common applications, the generic editor integrates the parts of the user interface for invoking functionality with the ones for viewing and entering data. There are no menu or tool bars that consume valuable screen real estate, but just the aforementioned views. Functionality is available in the form of operations, which are basically methods that can be invoked on the instances of a particular type. Operations can modify the repository as well as the user interface of the generic editor. The usual way to invoke an operation is to open the context menu of an instance that is represented in a view. The context menu contains all operations that are applicable to the type of the particular instance, and thus no irrelevant operations are shown. In the screenshot we see the context menu of instance "Type Entity type" of type "Entity type", which contains two operations for data manipulation, "Change link" and "Remove link", and two operations for user interface configuration, "Make root" and "Apply table view". Operation "Make root" changes the tree view so that the instance on which it is invoked is displayed at the root, and operation "Apply table view" changes the view of the panel to a table of all instances of type "Entity type", like the one shown in the right panel.

AP1 stores all configuration information in the repository so that the generic editor can be used to edit it. For example, all operations available to the user are instances in the repository, and we can add, delete and modify operations by changing corresponding instances and links between them. In the screenshot we see how operation "Apply table view" is associated with type "Entity type": instance "Operation Apply table view" is connected to instance "Type Entity type" by role "operations". Program



```
- PD metamodel : PD model
    -> Object  + PD metamodel : Object
    0..*  entity types  - Type Entity type : Entity type
                            -> Object  + Type Entity ty
                            1..1  name      Entity type : S
                            1..1  isPrimitive    False : B
                            0..1  tableName    EntityTy
                            0..*  accessible roles  + Ro
                                                    + Ro
                                                    + Role Entity type.isPrimitive : Role
                                                    + Role Entity type.tableName : Role
                                                    + Role Entity type.accessible roles : Role
                                                    + Role Entity type.operations : Role
                                                    + 98d966f4-0fb9-11db-8ab0-d56a70f8a30f : Role
                            0..*  operations  - Operation Apply table view : Operation
                                                -> Object  + Operation Apply table view : Object
                                                1..1  name     Apply table view : String
                                                1..1  invocation type   + Type Entity type : Entity type
                                                1..1  .net assembly   + DotNetAssembly GenericEditorLib : .net Assembly
                                                1..1  class name     GenericEditor.TableView : String
                                                1..1  method name     Apply : String
                      + Type Role : Entity type
```

	name	isPrimitive	tableName
	.net Assembly : String	False : Boolean	DotNetAssembly : String
	Blob : String	True : Boolean	BlobValue : String
	Boolean : String	True : Boolean	Boolean : String
	Change : String	False : Boolean	ChangeLog : String
	Character : String	True : Boolean	CharValue : String
	DateTime : String	True : Boolean	DateTime : String
	Entity type : String	False : Boolean	EntityType : String
	Integer : String	True : Boolean	IntValue : String
	Object : String	False : Boolean	Object : String
	Object set : String	False : Boolean	ObjectSet : String
	Operation : String	False : Boolean	Operation : String
	PD model : String	False : Boolean	PDModel : String
	Real : String	True : Boolean	RealValue : String
	Role : String	False : Boolean	EntityRole : String
▶	String : String	True : Boolean	String : String
*			

Figure 8. The generic editor of the AP1 prototype.

code is part of the repository as well, i.e. the program code of the operations, the views, and the repository itself. Also the user interface is stored in the repository and can be modified with generic views, and we are currently working on a specialized view that allows direct manipulation of GUI elements in a document-oriented fashion [7].

5 Related Work

Reflection, in one way or other, has also been described in other work as an important user interface concept, although not as explicitly as we do in this paper. For example, ISO 9241-10 [12] names self-descriptiveness, suitability for learning and suitability for individualization as general principles for achieving the ergonomic requirements of user interfaces. Self-descriptiveness is, in fact, a direct result of behavioral introspection, and related to ease of use for the casual user and the difficulty of learning. About the suitability of applications for learning is said that "rules and underlying concepts which are useful for learning should be made available to the user" so that "the user is able to obtain information on the model on which the application is based". This kind of transparency in a system is achieved by structural introspection. Suitability for individualization is directly related to structural and behavioral intercession.

One of the criteria for performance-centered user interfaces [15] is to "provide performers easy access to and control of desktop support components, interface presentation, and functions". Access to interface presentation and functions refers to behavioral reflection, while desktop support components include different kinds of help system. Help functionality in a program, e.g. as described in [11, 14], is a typical example of behavioral introspection, since its purpose is to explain to the user how to interact with the system in order to accomplish certain tasks. Different approaches for help generation have been described that can provide behavioral introspection in the user interface by performing behavioral introspection on an internal model of the system [22, 4, 18, 16]. [8] indicates that help systems that explain the functions of a program result in a better learning performance for users than help systems that merely provide a list of actions to perform. The former approach, i.e. offering insight about the system to the user, is the very heart of introspection.

Adaptability and extensibility are important for user interfaces as well. Approaches for adaptability have been discussed for web-based systems, e.g. [9], and graphical user interfaces, e.g. [7], and are examples of behavioral intercession. [10] describes an approach for adaptability and extensibility of user interfaces that is based on metadata about UI components and related to reflection in programming languages. Those problems have also been addressed in toolkits like the ones described in [3, 13] and user interface management systems, e.g. [20].

The instrumental interaction model [2, 1] reflects concepts and functions of an application as instruments, which are first-class objects in the user interface. It relies on design principles that are inspired by programming languages, with one of them being reification, i.e. the representation of metainformation as ordinary data. Consequently, this methodology targets the creation of reflective user interfaces. One of three desired properties of user interfaces that are mentioned is reinterpretability, meaning that "users can change input/output devices, add or remove interaction techniques, even program their own functions", which is pursued by making interactions first-class objects of an application. This does, in fact, describe the principle of behavioral intercession.

Some of the usability heuristics described in [17] are related to reflection as well: "visibility of system status" can be achieved by introspection. "Recognition rather than recall" is supported when metainformation is visible in the user interface, and this information can be helpful in order to "help users recognize, diagnose, and recover from errors". "Flexibility and efficiency of use" is supported with behavioral intercession, and "help and documentation" is provided through behavioral introspection.

6 Conclusion

In this paper we described how the principle of reflection can benefit the domain of user interfaces. The different aspects of reflection form a taxonomy for many application features that strongly affect usability. We have discussed several approaches to reflection in user interfaces and have examined user interfaces for reflective features. Examples show that these concepts can be found in and are relevant for real applications. Our own research prototype examines new ideas for reflective user interfaces and serves as a vehicle for future work in this area. To the best of our knowledge the the reflection principle has never been systematically applied to the domain of user interfaces before.

References

[1] M. Beaudouin-Lafon. Interactions as First-Class Objects. In *Proceedings of the ACM CHI 2005 Workshop on the Future of User Interface Design Tools*. ACM Press, 2005.

[2] M. Beaudouin-Lafon and W. Mackay. Reification, polymorphism and reuse: three principles for designing visual interfaces. *Proceedings of the working conference on Advanced visual interfaces*, pages 102–109, 2000.

[3] B. B. Bederson, J. Meyer, and L. Good. Jazz: an extensible zoomable user interface graphics toolkit in java. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 171–180, New York, NY, USA, 2000. ACM Press.

[4] D. E. Caldwell and M. White. Cogenthelp: a tool for authoring dynamically generated help for java guis. In *SIGDOC '97: Proceedings of the 15th annual international conference on Computer documentation*, pages 17–22, New York, NY, USA, 1997. ACM Press.

[5] F. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Aplications in AI*, 1995.

[6] D. Draheim, E. Fehr, and G. Weber. JSPick - a server pages design recovery. In *7th European Conference on Software Maintenance and Reengineering*, LNCS. IEEE Press, March 2003.

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

[8] S. Dutke and T. Reimer. Evaluation of two types of online help for application software. *Journal of Computer Assisted Learning*, 16(4):307–315, 2000.

[9] J. Fink, A. Kobsa, and A. Nill. User-oriented Adaptivity and Adaptability in the AVANTI Project. In *Proceedings of the Conference 'Designing for the Web: Empirical Studies'*, 1996.

[10] J. Grundy and J. Hosking. Developing adaptable user interfaces for component-based systems. *Interacting with Computers*, 14(3):175–194, 2002.

[11] K. Halsted and J. Roberts. Eclipse help system: an open source user assistance offering. *Proceedings of the 20th annual international conference on Computer documentation*, pages 49–59, 2002.

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

[13] E. Lecolinet. A molecular architecture for creating advanced GUIs. *Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 135–144, 2003.

[14] J. Masthoff and A. Gupta. Design and evaluation of just-in-time help in a multi-modal user interface. *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 204–205, 2002.

[15] K. L. McGraw. Defining and designing the performance-centered interface: moving beyond the user-centered interface. *interactions*, 4(2):19–26, 1997.

[16] B. Myers, D. Weitzman, A. Ko, and D. Chau. Answering why and why not questions in user interfaces. *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 397–406, 2006.

[17] J. Nielsen. Ten Usability Heuristics, 2004.

[18] S. Pangoli and F. Paterno. Automatic generation of task-oriented help. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 181–187, New York, NY, USA, 1995. ACM Press.

[19] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[20] J. H. Pittman and C. J. Kitrick. Vuims: a visual user interface management system. In *UIST '90: Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 36–46, New York, NY, USA, 1990. ACM Press.

[21] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley, 2002.

[22] P. N. Sukaviriya, J. Muthukumarasamy, A. Spaans, and H. J. J. de Graaff. Automatic generation of textual, audio, and animated help in uide: the user interface design. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 44–52, New York, NY, USA, 1994. ACM Press.

[23] N. Zhu, J. Grundy, and J. Hosking. Pounamu: a metatool for multi-view visual language environment construction. In *Proceedings of VL/HCC'04 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Press, 2004.