Semantic-Web-Backed GUI Applications

Axel Rauschmayer, Axel.Rauschmayer@ifi.lmu.de

Institut für Informatik, LMU München

Position Paper

1 Abstract and Introduction

End User Semantic Web Interaction is a field that is concerned with optimizing user interfaces for semantic web applications. In this position paper, we present a different angle to this problem: instead of looking at the Semantic Web from a user interface perspective, we would like to look at user interfaces from a Semantic Web perspective. Specifically, we are interested in graphical user interfaces (GUIs) as used by applications in mainstream operating systems such as Windows. We argue that the ease with which data can be integrated, queried and manipulated in Semantic Web applications is something that can be transferred to GUI applications. To that end, we outline where current GUIs fall short and how ideas from the Semantic Web can help. Note: Even though exceptions sometimes apply, our assertions are deliberately general in order to paint a concise picture.

2 Current Problems

In this section, we enumerate several flaws of GUIs that can be remedied by Semantic Web ideas.

Information Overload. GUIs suffer from information overload. When building vocabularies for manipulating data, applications rely on lists of text (menus) and/or icon bars which take too long to mentally digest.

Static Information Layout. When displaying a document, GUIs also display supporting widgets with editing information, meta-data etc. This information is either arranged in a fixed way or has to be customized by hand, one item at a time.

One-Dimensional Categorization. In order to categorize information (including program functionality), current GUIs use hierarchies. An example is preference management being implemented as a tree of nested dialogs. But hierarchies are one-dimensional and hinder more direct, associative, access, because they force a fixed way of categorization on the user.

Separate Meta-Level. Making meta-information available at the object level is called *reification*. In user interfaces, a lack of reification manifests itself in the following ways:

- When a user interface element does not behave as wanted, it is often difficult to find the preference setting that lead to the behavior. For example: “How do I turn off the red underlines for incorrectly spelled words?” or “Why is that menu entry grayed out?”.
- If an entity is displayed for a certain purpose, other information *not* related to that purpose cannot be directly accessed. For example: If there is a dialog for picking an application to open a document, one cannot otherwise examine the applications that are listed.
- When an error message is displayed, there is no way of copying the message text.
- Documentation and program are disconnected. So there is no systematic way to browse and discover features. Sometimes there *are* links between online help and program, but they are too few and the help content is structured one-dimensionally (see above).

Note that just filling up context menus with shortcuts to meta-information is *not* a solution, because one still faces information overload problems and because the meta-model does not become clear.

Lack of Integration. On one hand, application data such as contacts and bookmarks reside in separate data islands. Thus, properly combining and linking that data is not possible. On the other hand, application functionality also suffers from one-dimensional categorization: It is not clear where to put cross-cutting functionality. For example, whatever application displays a set of images should also provide operations for editing them and displaying a slide show. But doing so should not overburden already cramped user interfaces.

Little Context-Awareness. Current approaches to context-awareness are limited precisely because it is a problem where multi-dimensional categorization applies. Every application has its way of grouping its data under some name (email accounts, web browser users, network configurations etc.), but there are many cross-cutting effects here: For example, depending on where I currently am, I might need different bookmarks; or the phone number I have to dial for a contact changes. Note that contexts are not only spatial, but could also be related to time, circumstance (work or leisure), mood, health etc.

Limited Task-Orientation. The popularity of *software wizards* shows that a task-oriented approach to solving a problem is often more in line with human thinking. Usually, users do not go to the computer and think “I want to use Word”, they think: “I have to write a letter”. A further hint at the validity of this observation is that once someone has a favorite program, he uses it for almost everything: there are many people who write letters in Excel or text documents in PowerPoint. Still, wizards have limitations. They are by definition cross-cutting, but never give clues—visual or otherwise—as to what they are cutting across (see the paragraph on the "separate meta-level"). They are also constricted to single applications and rarely very flexible: Why can't I tell my film editing software to transcode three movies to a different format and then to switch off the computer?

3 Sketch of a Solution

Both the architecture and the user interface of an application are stored in an RDF graph. The interface consists of two parts: the *RDF browser* that includes a query widget and the *result list* that displays the results of browsing and querying. The latter part is the actual user interface, whereas the former part is more of a meta-component.

Objects and Operations. An application is not a large monolithic piece of software any more, but is decomposed into *user interface objects* (UIOs). UIOs are a hybrid of conceptual information and their graphical representation: the conceptual information is stored in RDF and contains a rich mix of data, documentation, keywords etc., the graphical representation is a comparatively small GUI widget. Whenever a UIO is visible in the user interface (the result list) then the browser allows one to access and further explore the packaged information. Furthermore, UIOs are annotated with what *operations* can be performed on them. Operations can also be considered objects and are stored and visualized in the same manner as UIOs.

Linking and Integration. Using the linking and integration abilities of RDF, one can continue the process that has already started with UIOs: to organize program functionality and to link it with various kinds of external information. The following things could be added as RDF-based annotations:

- An explicit model of the application architecture. Compare this to modern application scripting frameworks such as AppleScript that also expose the data model of an application.
- Context information.
- A simple task-based wizard can be viewed as a dossier on the subject of the task that links UIOs, operations and explanatory text.

Browsing and Querying. Filtering and querying will be constantly performed while using an application.

- Browsing is used for discovering functionality and for associative access to related information, some of which is obvious (such as documentation), some of which is less obvious: related operations can be derived by observing whether they apply to similar objects; object-operation relations are bidirectional, so one can also find out what objects an operation applies to; etc. Even more information can be accessed by following links into the architectural information of the application.

– Queries are used to filter the available amount of information, thus reducing information overload. For example, one could only show an operation if its name contains a given text or if it can be applied to the currently displayed UIOs. Both kinds of filtering lead to quicker operation lookup that via nested menus. More advanced queries can take multi-dimensional categorizations and context information into consideration and lead to on-the-fly creation of customized user interfaces that are context-aware and task-specific.

Other Advantages. As more architectural information on an application is accessible to users, they can refer to that information when reporting a bug which improves bug report quality.

4 Related Work

Spotlight and Dashboard. Both technologies are integrated[1] into Mac OS X: SPOTLIGHT is an indexing engine for querying both available objects (files) and operations (programs). DASHBOARD displays information as a collection of small, separate widgets.

Eclipse. The Eclipse IDE has an enhanced preferences interface where one can both filter the available information and click on hyperlinks to related preferences.

Emacs. On the Emacs home page it is called the "extensible, customizable, self-documenting real-time display editor" which already points to the incredible amount of innovative user interface ideas that come packaged with it: self-description, user interface by query, tab completion for application commands etc. Nevertheless, it is geared towards the more technical users, which sometimes leads to configurability prevailing over usability.

5 Further Challenges

Here are a few more examples of capabilities that GUIs need:

– Program history: In GUIs, there is no history in what has been done (for example: "What files have I recently deleted?") and many operations such as moving a window cannot be undone. Command line interfaces fare better here.
– User interface continuations: in a web browser, one has the ability to put aside any user interface state in a separate window. Traditional GUIs cannot do that.

[1] Note that it is quite probable that many of the user interface ideas in these technologies were inspired by third party applications for Mac OS X. We are not citing these applications simply because more people tend to be familiar with Apple's products.

– Inconsistent keyboard shortcuts: As soon as one uses Eclipse, Emacs and Word, one knows how much of an annoyance incompatible command keys can be. Standardization can go a long way, but maybe one has to completely rethink the idea of keyboard shortcuts, as they have mnemonic limitations: there are only so many combinations of an alphanumeric key plus modifier keys that one can easily remember.
– Notification management: There is an increasing amount of programs that operate without human supervision. Examples are MP3-encoded music playing in the background, server applications, web servers or any kind of longer task where one leaves the computer unattended. Somehow status information from these applications has to be managed: Should the status information be displayed on an external micro-display of my media-center display? Should an email be sent if something goes wrong or an SMS? Solving this problem shares many traits with imbuing applications with advanced task-orientation: Complex behavior has to be specified in an easy manner.

6 Conclusion

We have outlined how the current user interface metaphor reaches its limits, especially when it comes to managing its own information (which is at the meta level; we are ignoring the object level). In order to remedy these problems, we have outlined where partial solution are already appearing today and how these could be seamlessly integrated and improved on by using technologies and ideas from the semantic web.