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 icons 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, as one still faces information overload problems.

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.

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.

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

- By filtering, one reduces the information overload. Filtering operations means that they are more directly accessible, instead of time-consuming lookup in nested menus. Filtering by current context (what data is being displayed) leads to context-sensitive operation listings.
- 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; compare this to modern application scripting frameworks such as AppleScript.
- RDF queries allow one to dynamically rearrange the user interface. Now one can customize sets of items instead of single items. Moreover, RDF naturally supports multi-dimensional categorization. These queries can be said to create custom, task-specific applications.

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

Mac OS X has the Spotlight search technology for filtering both objects (files) and operations (programs) and Dashboard which displays information as a collection of small, separate widgets. The Eclipse IDE has an enhanced preferences interface where one can both filter the available information and click on hyperlinks to related preferences.

5 Further Ideas

There are many other user interface ideas that should be adapted to GUIs. Here are two examples:

- 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.