The Semantic User Interface Paradigm for Presenting Semi-structured Information

David F. Huynh DFHUYNH@AI.MIT.EDU
Dennis Quan DQUAN@MIT.EDU
Vineet Sinha VINEET@AI.MIT.EDU
David Karger KARGER@THEORY.LCS.MIT.EDU
MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA 02139 USA

1. Introduction

The Haystack project (Huynh *et al.*, 2002; Haystack, 2002) seeks to store and manage semi-structured data in the user's personal information corpus. There arises a need to present and allow the user to interact with such semi-structured data. Unfortunately, this semi-structured data is often modeled in several schemas, some of which are created or, if already existing, extended by the user. The Haystack user interface cannot be hardwired to handle only fixed system schemas and cannot use compiled user interface resource templates to present the data. The user interface needs to be flexibly constructed at runtime based on the semantics of the semi-structured data to be displayed. This paper describes the Semantic User Interface paradigm that we have created for dynamically constructing the user interface of Haystack.

2. Problems with Existing UI Technologies

Most existing user interface technologies rely heavily on the use of resource templates for UI constructs such as dialog boxes and menus. The static nature of these resource templates is not suitable for displaying semi-structured data whose constitution is not known *a priori* or is changed and extended frequently. UI designers must explicitly make their UI dynamically adaptable to the changing data. Furthermore, UI work is rarely reusable. Each UI designer must add his or her own code to present data that is already displayed in proper form somewhere else by someone else.

There do exist some mechanisms for embedding UI components within one another (*e.g.*, Object Linking & Embedding (OLE) framework, Model-View-Controller (MVC) paradigm), but these mechanisms are lacking. In the MVC paradigm, a UI designer wishing to reuse an existing view must explicitly specify that view's implementation in his or her UI construction code. Should the view's implementation be replaced, the UI designer's code becomes outdated. The OLE framework resolves this problem by providing a dynamic binding scheme that looks up and then embeds view implementations dynamically at runtime. However, for each piece of data whose view is desired, the OLE framework can only instantiate one view—often the content view. The UI designer cannot specify the type of view to embed.

Because existing UI technologies are not powerful enough to support flexible reuse of UI components, each UI designer is left to improvise his or her own user interface design. He or she writes code to display almost each and every type of data that his or her application deals with. Even in the same application, different features written by different UI designers contain different code fragments to display the same type of data. The different code fragments provide different UI capabilities to their corresponding UI elements. In many cases, the piece of data that the user wants to interact with is readily displayed, but its UI representation is no more than dead pixels on the screen, affording no means for interaction, so that the user is forced to take a different UI route to manipulate it. This is a subtle but prevalent and severe inconsistency in today's user interfaces.

3. Constructing UI Dynamically

In order to build a consistent user interface, the Semantic User Interface paradigm facilitates and encourages extensive reuse of UI components. The paradigm specifies a dynamic binding scheme for embedding views much like that used in the OLE paradigm. However, while OLE and MVC rarely use more than one level of embedding, the SUI paradigm encourages arbitrarily deep nesting of views. Furthermore, SUI views need not be rectangular child windows. They can be inline segments of text that flow through several lines. This flexibility makes views versatile and easy to embed anywhere. In fact, the SUI paradigm strongly advocates that each UI designer specializes in handling only the types of data that he or she knows best and embeds views made by other UI designers for other types of data.

The mapping from a piece of information, or a type of information, to the views capable of rendering it is stored entirely as metadata in the RDF store of Haystack (Huynh *et al.*, 2002). The metadata describes the types and formats of data that each view is capable of presenting as well as the contexts in which each view is appropriate. Note that each piece of information can have more than one view: an audio file can be summarized in one line of text based on its play time, title, etc., or it can be viewed in an audio player that takes up a whole window. The former view is appropriate where a short description is desired, and the latter should be used when the user focuses solely on the audio file.

The Haystack user interface infrastructure provides a component called the *view selector* that performs this mapping automatically. While designing a particular view, a UI designer can insert view selectors to embed inner views within this view. At runtime, the view selectors look up and instantiate appropriate inner views. In essence, the view selectors act as the level of indirection in the dynamic binding scheme of the SUI paradigm. They compose the UI dynamically as a hierarchy of nested views.

4. Keeping UI Dynamic

In order to make a view a faithful representation of the corresponding piece of information, the SUI paradigm specifies that the view should register for notifications from the RDF store upon any change to the information that it displays. For instance, “stacker” views have been designed to present collections dynamically. (A collection is a mathematical set of objects.) Given a collection, a stacker view constructs a view selector for each element in that collection and stacks the view selectors in some specified sorting order. The stacker view also registers for notifications upon any change to that collection. If a new element is added to the collection, the stacker view constructs a new view selector for it. If an existing element is removed, the stacker view removes the corresponding view selector.

A UI designer who makes use of a stacker view needs only specify the sorting order for the elements and the specifications for the dynamically constructed view selectors so that appropriate views of elements are produced. The stacker views have effectively raised the level of abstraction for rendering collections. In the future, grouping and other high level presentation logics will be supported.

5. Supporting UI Features Uniformly

Because each piece of data is presented by a view, for any pixel on the screen, there is a systematic way to detect which of the currently displayed views enclose that pixel and to trace back to the corresponding pieces of data being presented by those views. That is, for every rendered pixel there is a connection back to the data that pixel represents.

Based on these live connections between elements on the screen and data they represent, we can easily and systematically provide features such as context menus and drag and drop consistently throughout the Haystack user interface. For instance, when the user right-clicks, we can trace back through all of the pieces of data that the clicked pixel represents and construct a context menu listing all actions applicable to those pieces of data. The result of supporting these features consistently is a user interface in which UI elements presenting the same piece of data afford the same set of actions corresponding to that data.

6. Proposed Benefits

We propose that the SUI paradigm has several benefits. First, its scheme for constructing the UI by dynamically nesting views raises the level of abstraction for UI designers. The designer of the outer view needs not know the details of how the inner view is constructed but can simply delegate the task of constructing the proper inner view to the view selector. In addition, certain views such as the stacker views allow the designer to specify UI construction specifications such as sorting orders in a higher level of semantics.

Furthermore, the stacker views, with their ability to update the UI dynamically, effectively decouple the information processing tasks from the UI presentation tasks. That is, one can concentrate on managing the elements inside a collection without concerning oneself with how that collection is being displayed. Since the UI gives a faithful representation of the data, the view of that collection always reflects the contents of the collection.

Finally, the result of systematic UI construction is a uniform user interface in which features such as context menus and drag and drop can be provided pervasively throughout the whole application, making the application behave consistently and predictably to the user.

References

Haystack (2002). http://haystack.lcs.mit.edu/.

Huynh, D., Karger, D., and Quan, D. (2002). Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. *Semantic Web Workshop, The Eleventh World Wide Web Conference 2002 (WWW2002)*. Honolulu, HI. http://haystack.lcs.mit.edu/papers/sww02.pdf.

Acknowledgements

This work was supported by the MIT-NTT collaboration, the MIT Oxygen project, a Packard Foundation fellowship, and IBM.