Prerequisites for a Personalizable User Interface

David R. Karger
MIT CSAIL
200 Technology Sq., Cambridge, MA 02139 USA
karger@theory.lcs.mit.edu

Dennis Quan
IBM T. J. Watson Research Center
1 Rogers Street, Cambridge, MA 02142 USA
dennisq@us.ibm.com

ABSTRACT

Interfaces that support customization and can adapt to individuals' specific use patterns may be more effective than ones designed to be "one size fits all". However, to test whether such interfaces benefit users in actual everyday applications, a lot of underlying application infrastructure must be reimplemented to accommodate customization and adaptation. In this paper we discuss Haystack, a platform for building information applications in which user interface concepts such as commands, views, and widgets and even application data itself is described as a semantic network, providing a flexible environment for prototyping notions of customization. Haystack's data model can host information that used to be managed by multiple applications, meaning that users can use Haystack to interact with their data under a single paradigm, and adaptability and customization capabilities can be provided across multiple domains at once.

Author Keywords

User interface customization, Semantic Web.

ACM Classification Keywords

H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

Personalization holds great potential to improve people's ability to interact with information: different people care about different information and prefer to examine it and manipulate it in different ways.

Unfortunately, today's application model throws up numerous obstacles to effective use of personalization. Features that users would like to customize are often hard-wired into monolithic assemblies of fixed, private data models, information displays, and operations. Similarly, the problem of allowing applications to study their users' behavior and to automatically adapt themselves is hindered by a lack of hooks into the user interfaces of applications.

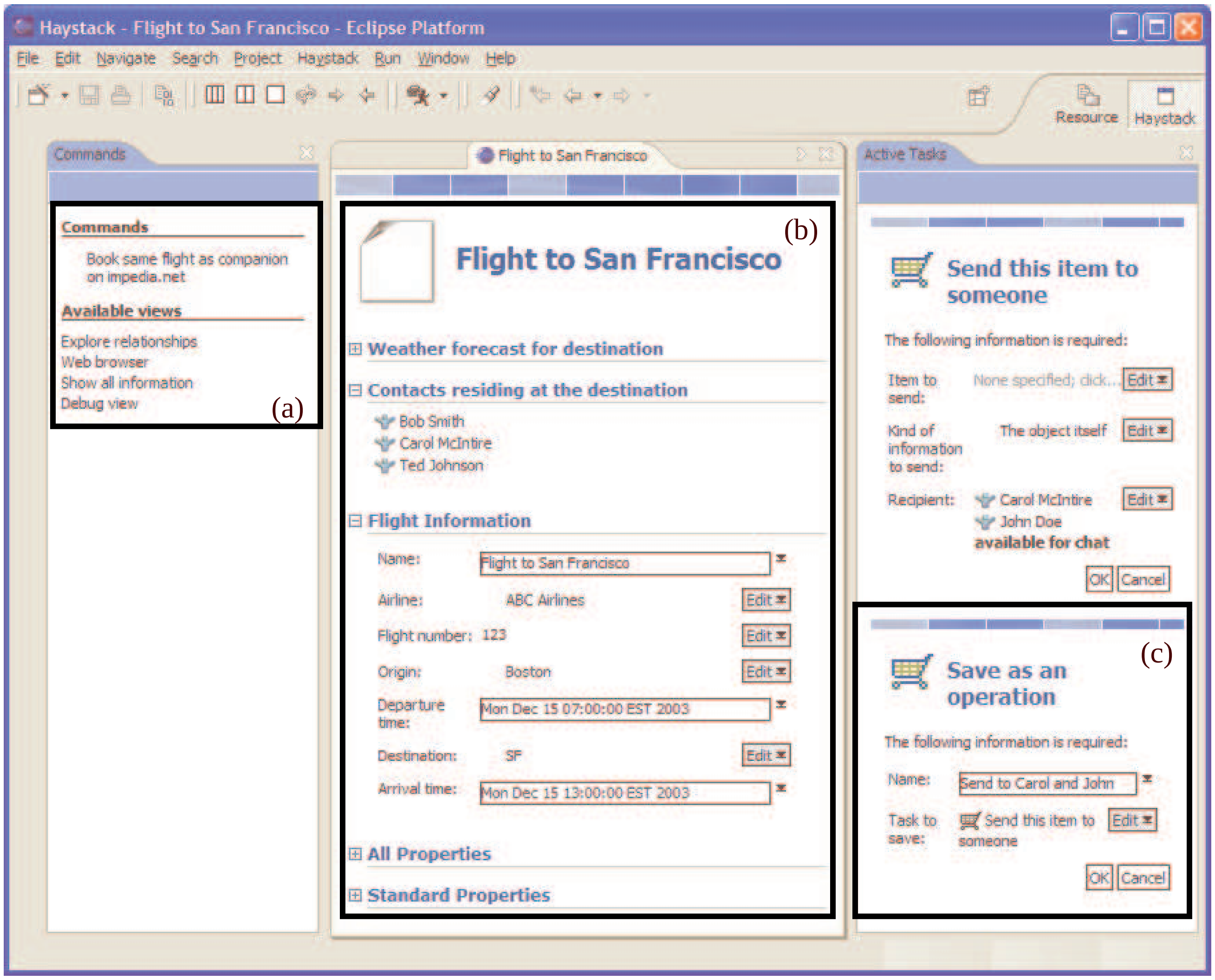
As a result, providing support for customization often requires applications to undergo a significant overhaul—a nontrivial burden for the researcher/developer and an impossibility for the end user. In addition, a single activity performed by a user may involve multiple applications, yet the problem of studying cross-application customization is even more daunting because multiple applications must be rewritten to support the same style of customization.

Enabling effective personalization—both automated and user-driven—requires a new approach to application development, in which:

- The data model (and not just the data) can be flexibly modified at runtime, for example, to reflect novel properties that might interest a particular user;
- The user interface must flex with the data model, displaying information that was not part of the model when the UI was first defined; and
- Views (ways of displaying information) and operations (ways of manipulating information), rather than being hard-wired in code, must be first class objects that can be examined and modified by end users or by agents acting on their behalf at runtime.

This reification of data model and interface information that has typically been implicitly buried in code is a necessary first step in the design of any richly-customizable system.

We have built Haystack, a system that effectively meets the requirements outlined above [1, 6], depicted in Figure 1. Haystack uses a semantic network model (RDF, a popular World Wide Web standard [3]) to uniformly represent and interconnect all of a user's information, views, and operations. Haystack's user interface kernel, Ozone, interprets the semantic network for instructions on how to show information and how to let the user manipulate it. This design allows end users as well as researchers to customize their environments by annotating the interface information in the semantic network. The data model also provides a natural repository of information about the user's actions, which we intend to use as input for machine learning agents that will automate some of the customization process for the user.


Haystack - Flight to San Francisco - Eclipse Platform
File Edit Navigate Search Project Haystack Run Window Help
Resource
Haystack
Commands
Flight to San Francisco
Active Tasks
(a)
(b)
(c)

Commands

Book same flight as companion on impedia.net

Available views

Explore relationships
Web browser
Show all information
Debug view

Flight to San Francisco

⊞ Weather forecast for destination

⊟ Contacts residing at the destination

Bob Smith
Carol McIntire
Ted Johnson

⊟ Flight Information

Name:	Flight to San Francisco	
Airline:	ABC Airlines	Edit
Flight number:	123	Edit
Origin:	Boston	Edit
Departure time:	Mon Dec 15 07:00:00 EST 2003	
Destination:	SF	Edit
Arrival time:	Mon Dec 15 13:00:00 EST 2003	

⊞ All Properties

⊞ Standard Properties

Send this item to someone

The following information is required:

Item to send:	None specified; click...	Edit
Kind of information to send:	The object itself	Edit
Recipient:	Carol McIntire John Doe **available for chat**	Edit

OK Cancel

Save as an operation

The following information is required:

Name:	Send to Carol and John	
Task to save:	Send this item to someone	Edit

OK Cancel

Figure 1: Screenshot of Haystack. (a) extensible list of operations and views; (b) view of flight with embedded views of component objects; (c) creating the curried “Send to Carol and John” operation from the “Send this item to someone” operation

A FLEXIBLE DATA MODEL

A customizable user interface is of limited value in the presence of a fixed data model. If certain information that a user cares about cannot be expressed in the data model, then no amount of user interface customization can help that user work with such information. Today’s applications exhibit two distinct kinds of fixity. Within a single application, it is often difficulty for a user to extend the data model to reflect, for example, additional attributes of the information that the developer did not think to include.

Other problems arise when multiple applications are involved. On the one hand, a user may discover a need to connect information that is “owned” by different applications. The incompatible formats and data models of the two applications make it difficult to record such connections, and neither application likely has a user interface designed to show or manipulate such connections. Conversely, there are certain data types, such as documents, people, or messages, that are managed by multiple applications using distinct models and formats. In such a setting, customizations made by a user of an interface within one application fail to propagate to other applications manipulating the same data.

The problems outlined above led us to design a common, unified data representation for all of the information a user works with. We chose a semantic network as a kind of “Turing universal” data representation. In a semantic network, arbitrary objects are connected by arbitrary binary relations (higher arity relations can be represented by creating intermediate nodes). In order to maximize Haystack’s compatibility with other systems, we have chosen to use RDF, a Semantic Web representation standard supported by the W3C [5]. Using RDF lets Haystack interact “natively” with sources of RDF such as RSS news feeds.

Like a typical application, Haystack comes with a “default” data model: for example, it models documents using the properties (such as author, title, and body) defined in the Dublin Core schema [4]. The default schemata cover many data types, such as e-mails, contacts, MP3 files, photographs, research papers, and text documents. Still, customizing the model is easy: any user can define additional document attributes (such as color, current reader, or quality) that are then treated by the system with the same attention as the default properties.[1] Haystack also attempts to maximize reuse: for example, e-mail messages are a subclass of documents, so they inherit all of the Dublin Core properties. This means that a user's customization of the document data model propagates to all kinds of documents, not just those handled by a specific application.

VIEWS

Most applications in use today render information from an internal data model to some sort of interface. Unfortunately, many applications only provide one or an extremely limited number of ways for viewing information and cannot be extended with new visualization styles, hampering customization. In Haystack, there may be multiple *views* present for any kind of object, and in fact views can be modified or created at runtime. In this sense, views are treated as first class objects in Haystack because a single presentation mechanism is not monolithically coupled to any portion of an application.

Our approach to designing views is driven indirectly by the need to cope with our flexible data model and directly by the desire to support customized information display. Much user interface real estate is allocated to the display of objects in their relation to each other. We might, for example, display an iconic tiling of the files in a directory, indicating a containment relationship. Alternatively, we might display a list of e-mail messages in rows, using distinct columns to present each message's (relationship to a) sender, subject, and date. A calendar view displays in each day a list of appointments, and an address book has a standard format for displaying an individual by listing properties such as name, address, phone number, and notes in some nicely formatted layout. The address itself may be a complex object with different sub-properties such as street, city, and country that need to be laid out.

This common phenomenon of using a hierarchical layout of information to show relationships between and internal to complex objects motivates an object-oriented approach to user interface presentation in which a view of some object *X* may be created by (i) letting *X* decide which of its properties and relationships to other objects need to be shown, (ii) recursively asking the objects required by *X* to create views of themselves, and (iii) allowing *X* to lay those views out in a way that indicates *X*'s relation to the viewed object. As a concrete example, when rendering a mail message we might consider it important to render the sender; we do so by asking the sender to produce a view of itself and then laying out that view of the sender somewhere in the view of the mail message. The sender object, in rendering itself, may recursively ask the sender's address to produce a rendering of itself that the sender view could incorporate.

[1] Haystack's schemata are themselves represented in the semantic network (using DAML+OIL from the DAML project) so modification of the schema is simply another instance of modifying data in the semantic network (and is supported by the same user interfaces as the rest of the system).

Our interface design makes customization much easier than in the traditional application-centric approach. The view production rules for a given object type are defined once and then reused whenever that type of object is shown; thus, a user can customize the view once and see the customization whenever its objects are displayed. At the same time, our delegation approach means that a user's customizations of one view do not impact any other view: views treat their contained sub-views opaquely, so changes to sub-views will be ignored by the containing view. (We do however provide ways for allowing views to cooperate and coordinate with each other.)

Furthermore, Haystack's data model is intentionally decoupled from the views that are responsible for presenting the data. Because views are decoupled from the underlying data model, information from multiple applications can be brought together in a single presentation. As a result, customizations made to a view remain in effect regardless of which application the data came from.

For a developer to construct views that are conducive to customization, Haystack provides a user interface toolkit that enables reusable user interface components, such as list views and buttons, to be packaged and assembled with a declarative, HTML-like language. To make customization even easier, we attempt to maximize the portion of the view generation that can be described declaratively, in data, as opposed to imperatively, in code. For example, many object views are created simply by laying out some decoration together with views of related objects, as a mail message view might lay out the sender, subject, date, and body of the message. Such views can be declared simply by listing the properties to be inspected and the scale and layout of their views in the containing view. These annotated lists can easily be stored in our semantic network and viewed using our standard interfaces; thus user interface customization becomes as simple as other data manipulation activities.

OPERATIONS

In addition to visual customization, command customization is another common means for users to personalize an application to better suit their work patterns. Such customization comes in many forms: macros, alteration of toolbars and menus, most recently used lists, etc. The prerequisite for this functionality is an abstraction that allows commands to be treated as first class objects; that is, one can create custom lists of commands, create composite commands (macros) that specify individual commands to be executed in a specific order, and so on. However, many application frameworks do not treat commands as first class objects, relegating them to code buried in event handlers.

In Haystack, commands are abstracted into *operations*, which are composed of two parts. The first part is the metadata that characterizes the types of parameters the operation accepts, the name of the operation, an icon, etc. This metadata is stored in Haystack's semantic network. The other part is an implementation written in a programming language such as Java.

Treating operations as first class makes supporting many forms of customization easy. Since a menu is simply a collection of commands, displayed in a particular view, a user can create menus for any desired application by applying the same tools they use to create other collections. Operations can be annotated by the user with useful reminders about usage, creating a customized help system.

Furthermore, operations can be wired together in a number of ways at runtime. In addition to a full-featured scripting language called Adenine that is useful for macros, Haystack provides the ability to partially evaluate ("curry") operations [2]. For example, the print command in many programs provides many options such as number of copies, double-sided printing, and portrait/landscape mode. However, for a user who wishes to print either single-sided with no staples or double-sided with staples, he or she must reconfigure the options in the dialog box each time. Instead of requiring a macro to be written, currying allows the user to select a set of print options and to save the state of the customized command as a new command, say "print double sided with staples".

LEARNING USERS' BEHAVIOR

Often users will not be willing or may not understand how to perform certain customizations, even if they may lead to large productivity gains. A natural approach is to apply machine learning techniques in order to make some of these customizations on behalf of the user. To make automated customization successful, several important elements are needed. First, learning algorithms work better when they are given more training data. Because Haystack enables information from multiple applications to be accessed from a single environment, a learning algorithm only needs to be hooked into this single environment in order to observe the user's behavior over a variety of different situations. Additionally, Haystack makes it easy to obtain training data since invocations of operations, browsing history, and other session data are recorded in the data model, providing a single abstraction to the feature space of interest.

Because Haystack provides a highly expressive data model, learning agents inherit a useful facility for recording observations. Many problems with agent-driven user interfaces arise as a result of insufficient information being provided to the user—i.e., a lack of transparency. For example, when an agent makes a nonsensical recommendation to users, it is useful to be able to introspect and/or modify the agent's observations to determine what led to the recommendation and fix the problem. Haystack's data model facilitates the recording of fine-grained observational details. Also, metadata originating from other sources such as the user's browsing history can be correlated with agents' own records *in situ*, providing for more detailed explanations.

A semantic network is a good way not only to describe observations of the user's behavior but also the conclusions learned. Besides custom tracking structures, conclusions can also take the form of a view or an operation. For example, an agent could be tasked with monitoring which operations are most frequently invoked and create a custom view that incorporates them. An agent might also create curried operations based on common usage patterns automatically. Additionally, when conclusions are stored in the data model, users benefit from being able to manipulate conclusions using the same tools for manipulating other information because of the uniformity of the semantic network representation. In essence, learning agents become data-to-data transforms working against the common semantic network.

SUMMARY

We propose that a number of the barriers to supporting personalization can be lowered by using an application framework built on the notion of pervasive customizability. In particular, Haystack uses its expressive semantic network data model not only to incorporate various forms of information but also to model the user interface. The key point is that targets for customization need to be treated as mutable data, not immutable code. Information is presented to users by an extensible family of views, which are customizable and whose specifications are themselves described in our data model. Similarly, user interface commands are specified in the data model and implemented through a common operation abstraction, enabling them to be composed at runtime. In addition to facilitating customization at runtime, the notion of "UI specifications as data" allows agents to learn about users' preferences and behavior by monitoring the data model and to potentially offer suggestions in turn back into the data model.

ACKNOWLEDGMENTS

This work was supported by the MIT-NTT collaboration and the MIT Oxygen project.

REFERENCES

1. Quan, D., Huynh, D., and Karger, D. Haystack: A Platform for Authoring End User Semantic Web Applications. *Proceedings of Int'l Semantic Web Conf. 2003*.
2. Quan, D., Huynh, D., Karger, D., and Miller, R. User Interface Continuations. *Proceedings of UIST 2003*.
3. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
4. Dublin Core Metadata Initiative. http://dublincore.org/.
5. Berners-Lee, T., Hendler, J., and Lassila, O. "The Semantic Web" in *Scientific American*, May 2001.
6. Haystack project home page. http://haystack.lcs.mit.edu/.