# Tools for End-User Creation and Customization of Interfaces for Information Management Tasks

by

KARUN BAKSHI

B.S. Electrical Engineering
University of Maryland, College Park, 1996

B.S. Computer Science
University of Maryland, College Park, 1997

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2004

Signature of Author.........................................................................................................
Department of Electrical Engineering and Computer Science
March 15, 2006

Certified by ..........................................................................................................................
David R. Karger
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .........................................................................................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Students

**Tools for End-User Creation and Customization of Interfaces for Information Management Tasks**

by

KARUN BAKSHI

Submitted to the Department of Electrical Engineering and Computer Science
on March 15, 2006
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

## Abstract

Information based tasks rely on software applications that allow users to interact with information in some pre-defined manner deemed appropriate by the application developer or information/content provider. Whereas such an approach facilitates one way of working with the information, it does not (and cannot) take into account the unique needs of the user, e.g., the particular content of interest given the specific task being performed and expertise of the user, information visualization and interaction preferences of the user, etc. As a result, users must perform additional overhead information management activities in working with the software tools in order to accomplish their particular tasks.

In this thesis, we advocate breaking the "rigidity" of such applications by allowing users to create and customize their own task-oriented interfaces (information spaces) that aggregate and present task-specific information and tools on the same screen.

In developing a system that allows users to tailor an information space in a manner that suits their particular task and preferences, we recognize a set of desirable properties it must have, and the need for it to provide the user customization control over three primary aspects of information in their information space: content, presentation and manipulation. Haystack, a generalized information management system, encompasses many of the desirable properties at the system level and also provides many of the building blocks that are required to give users greater customization control. We thus approach our ultimate goal of enabling users to build and configure a personalized task-oriented interface by providing them with tools situated in Haystack that allow manipulating various primitives that control the three aspects of information spaces. A discussion of the design and implementation of each of the tools is provided.

The above solution allows users to develop information spaces that better match their unique conception of the task and eliminate much of the overhead resulting from "rigid" information management tools, resulting in productivity gains in recurring or long-running tasks.

Thesis Supervisor: David R. Karger
Title: Associate Professor of Electrical Engineering and Computer Science

# Acknowledgements

This goal has been a long time coming, and I have many people (and beings) to thank for their patience and support.

First and foremost, I would like to thank Professor Karger for giving me the freedom to choose a topic and his guidance in developing the idea.

Thank you God, for the opportunity to complete this goal, in this fashion. I never dreamed this would happen.

Thank you also to my parents and brother for their patience, support, encouragement and inspiration not just for this academic achievement, but in all my endeavors over the years. I cannot repay it.

Thank you also to all my teachers and friends over the years who invested time and patience in helping me get this far. In particular, thank you to Prof. Sidiropoulos who had faith in me and told me to "take it [GRE] again" so I could do better.

Thank you to all my friends in Haystack, who have helped me along in this program. Thanks Dave for always answering my questions, always fixing bugs promptly (with a smile, no less!) and ensuring the road was clear for me to make progress. I owe you. Thanks also to Dennis for answering many questions on Haystack's current design and implementation.

Thanks to the MIT CSAIL Oxygen Project and the Simile Project for their funding in conducting this research.

Finally, I am sure I've missed some people. So, thank you as well for helping me achieve this goal.

# Table of Contents

# List of Figures

## List of Tables

# Chapter 1 Introduction

Advances in processing, storage and networking have made enormous amounts of information accessible to people via the Internet (e.g., World Wide Web, e-mail, etc.). Also, users can now rapidly create new information using sophisticated software applications. Although a lot of information has become readily accessible and necessary for daily work, the current infrastructure for managing information is ill-suited for information-oriented activities.

The general information based task generally requires users to not only use diverse bits of information obtained from these various sources, but many times also requires them to use the same information in multiple tasks: the same information may need to be visualized and used in different ways depending on the task. The primary means that users have to manipulate the available information is software applications that generally have a well defined domain of application, and hence only deal with information deemed relevant to that particular domain. Inevitably, since there is no universal agreement on what constitutes a domain, many applications overlap in their functionality. Furthermore, the means of visualizing and interacting with information is also predetermined by the application. Whereas applications may be designed to support a number of well known tasks, they cannot be easily adapted to the user’s notion of a task which may be different than that anticipated by the application developer. Accordingly, since the information is not organized and presented in a manner that matches the task and user preferences, it becomes difficult to use and users must expend additional time and effort in collecting and extracting information relevant to the task at hand.

In this thesis, we outline a means for users to have greater control in defining how to interact with information relevant for their tasks by providing appropriate tools that allow them to create and customize information management interfaces.

The remainder of this chapter is devoted to understanding some examples of the above problems, and a closer look at what the underlying causes are. We then identify the problem we wish to solve, followed by a discussion of a high level approach to its solution.

## *1.1 Use Cases*

We illustrate the above observations on information usability further by considering two examples of information-oriented tasks. As demonstrated by these examples, a significant divide exists between the user's conception of information and task, and that of the supporting information management solution. It is not always possible for the user to impose her mental model onto the information in order to make it easy to assimilate and use. As a result, she is forced to manually or mentally bridge this gap.

### 1.1.1 Managing a Software Project

Consider a software project manager who needs to manage the tasks her team does, as well as the budget and schedule for the project. She might need various kinds of information for the various tasks involved in managing the project, e.g., contact information for team members, to-do list and assignment of action-items to individuals, e-mails corresponding to the project, outstanding bug reports, a budget spreadsheet and a schedule.

The current information management architecture supports the project manager's tasks through various applications modifying "their" data using proprietary formats. For example, in order to react to a customer e-mail about a software bug, she must switch from the e-mail client to the bug tracking software to enter a bug report. Then, she must switch back to the e-mail client, recall which software developer would be best suited to fixing the bug, look up his/her contact information, and send him/her a bug report number. The project manager may also need to meet with the developer, and hence negotiate a meeting time via e-mail, and update her personal calendar. After the meeting, the project manager will need to switch to the scheduling application to update the project schedule to reflect the time that will be consumed in fixing the bug. Another application switch may be needed to update the developer's outstanding tasks if the scheduling software does not support to-do lists for team members. Finally, she must switch to the spreadsheet software to update the budget on the project to take into account the resources consumed by the bug fix.

As can be seen, a significant amount of user effort is spent in finding information, mentally collating it and switching from application to application to perform various subtasks, not to mention possibly re-entering it in various applications, thereby duplicating data and potentially introducing a data integrity violation. In the ideal scenario, the user would not be subjected to context switches as he/she reviewed the project status or performed related tasks by having to go to different applications that managed different aspects of the project.

Thus, whereas the current information management infrastructure *facilitates* information oriented tasks (by allowing it to be accessed and manipulated), it leaves room for improvement in terms of *ease of use* from the user's task perspective. Whereas, a single application could be developed that included functionality for all project management tasks, it would still not solve the problem:

- What if the project manager also wanted to use the new application's functionality, such as calendar and e-mail, outside of the project management application, e.g., to see when her dentist's appointment is, or to receive mail from her spouse? Tracking such information in the project management application would not make sense, and tracking it separately would require duplication of stores as each application only maintains "its" data, thereby making it difficult to have a consolidated personal calendar view.
- What if the project manager's responsibilities changed, and he/she now had to also track news about a competitor's products? If this possibility had not been foreseen by the application developer and captured in the new application, the manager would have to return to "hopping" from the project management application to the news tracking application.
- What if the project manager did not use all the functionality in the new enhanced project management application? She would have to consciously and repetitively ignore parts of the functionality and buttons and widgets that expose it. Alternatively, she could be forced to use certain functionality unnecessarily: if the only way to access bug reports view was based on specifying a bug severity level, she may have to associate severities with bugs even though it was not required by the company's software process. Thus, there would be no way to customize her view of the information.

We can conclude that building another "super" application does not resolve the problem.

### 1.1.2 Catching Up On News

Consider another example of a user interested in the task of catching up on the news. Clearly, the notion of what does and does not constitute "news" is unique to the individual. One person (e.g., an Indian immigrant from New Delhi) may be interested in world headlines, business news, technology news, and New Delhi politics, whereas another (e.g., a baseball fan residing in Maryland who happens to be a stock analyst for the pharmaceutical industry) might be interested in Maryland state politics, baseball news, and business and technology news from the pharmaceutical industry. Whereas, the first person might be satisfied with CNN's website for the first 3 items, it is unlikely that he will get information on New Delhi politics from CNN. More than likely, he will need to visit an Indian or Delhi-based newspaper's website to get that kind of information. By extension, we can view the general activity of getting information as one corresponding to hunting and gathering; we must manually *seek out* what we need, since those providing it do not (and presently cannot) cater to individual needs.

Had the problem been restricted to news personalization, it would have been considered solved to some extent as evinced by the general purpose "information portals" such as

*Yahoo!* that aggregate multiple information sources and provide a table of contents and other options for managing and customizing the portal. In fact, in recognition of the varying information needs of people, Yahoo! has different web sites for different countries. Nevertheless, even for news, some problems remain:

- The user cannot aggregate information in case the content he is interested in is spread across the various *Yahoo!* sites.
- The information may not be organized in a way the individual user sees it (i.e., his world view), e.g., he may want to receive pharmaceutical industry news – both business and technology related, rather than business news and technology news, and having to sift it for information related to the pharmaceutical industry. The news provider may not cater to individual industries or organize information in such a manner as to easily allow the user to manually impose a different structure. As a result, the level of customizability that is available to the user is limited.
- People's information needs run deeper than conventional news. Even if CNN allowed the user the complete flexibility to specify the subset of information needed from the web site, clearly it cannot satisfy all of his needs, e.g., if he is interested in the IEEE signal processing electronic magazine to stay abreast of latest developments in the field as part of the "news" task, he still has to visit the IEEE website and log in separately to access the content.

## *1.2 Current Nature of Information Management Solutions*

We argue that the above problems in managing various aspects of information and tasks fundamentally stem from the current static nature of information and the supporting tools; In essence, information is "bottled up" by the application. Information producers have most of the control over how information is packaged, presented and can be manipulated. The information consumer whose productivity these decisions significantly affect has little say in these decisions. We briefly discuss some of these decisions and the impact they have.

### 1.2.1 Fixed Structure and Granularity of Information

The "pure" information that users require, devoid of its presentation, is perhaps the most critical ingredient of the task that the user is attempting to accomplish. Yet, users lack the ability to easily select *what* information to show and manipulate in a given context. Much of this work is mentally done by the user by selectively remembering, revisiting, focusing and ignoring information as appropriate. This method, however, does not scale well in the face of growing corpora of information. We investigate the various dimensions of this deficiency below.

#### 1.2.1.1 Relevant Subset Specification

Clearly, not everyone is interested in all available information. Furthermore, at any given time, a user is probably only interested in a further subset of items of general interest to him/her depending on the task at hand. However, applications currently make it difficult for a user to specify a subset of information of interest since the view of information has been predefined, and is always completely populated. It might contain more information than necessary, or the relevant information may be scattered across various views in the application. For example, in order to access information about all songs from a particular artist, a user may visit the web site of the recording label that produces the artist's albums. However, instead of providing a listing of artists and their songs, the recording label may provide information organized as albums it produced each year. For each album, it also lists the artist as well as the songs. In this case, the user cannot impose a structure on this information to aggregate and expose only the information of interest to her. Instead, she must manually go through the information to select relevant information. This process becomes problematic and inefficient when the corpus to be accessed is large and the result set may be relatively very small, or when the same query must be performed periodically. Part of the difficulty arises from the fact that in HTML-based web sites (and in many other cases), the presentation is tied to the content, and hence the semantics of the information are not exposed. In other cases, the information may simply not be semantically tagged to support queries that extract only relevant information.

Thus, the user cannot specify the subset of information of interest by specifying a condition it must satisfy in order to be relevant; he/she is forced to view the entire corpus even though the information set of current relevance is much smaller.

#### 1.2.1.2 Aggregation Specification

Information is currently fragmented because of representation (e.g., different file formats) or storage (e.g., different web sites), making it difficult to bridge gaps and place sets of information adjacent (physically or logically) to each other. Thus, users cannot arbitrarily aggregate or mix information from different sources, and co-locate it. The scenario outlined earlier with the software project manager hopping several applications to access information and operations is one example of this problem. Had the manager been able to specify the necessary information required from various sources, she would have been able to co-locate it to easily manage the project from a single UI. Furthermore, she could easily respond to the challenge of additional tasks, such as tracking competitors' products. Due to the fragmented nature, users are forced to collect information from one tool, and enter it in another. Alternatively, if the same information is maintained by multiple tools, the user must ensure that it is consistent across tools.

### 1.2.2 Fixed Presentation of Information

User control over information presentation is crucial, and in fact a significant portion of the functionality in many applications is devoted to allowing users to modify the presentation of the information in a manner consistent with how they want to manipulate it. Generally, however, the subset of functionality available in this respect is limited to that which is deemed useful by the application developer for the task at hand, leaving the user unable to make these decisions based on personal preferences. Users should have first class support (applicable to any information) for customization of information presentation. We consider the problem of information presentation as consisting of two parts: specification of high level layout and specification of the view of the aspect of interest.

#### 1.2.2.1 High Level Layout Specification

Related to the problem of lacking the ability to aggregate arbitrary content, is the lack of the ability to specify the layout of content. That this is a powerful and useful capability is demonstrated by the ability of many modern software applications to allow docking of content panes in different parts of the main application window. An example of such capability would be the Microsoft Developer Studio software development platform which allows docking of the compiler output pane, debug pane, source code browser, etc. However, this capability is not supported uniformly in applications. Another example of this capability resides in window managers that help users tile, cascade and otherwise organize their windows in the available real estate. Thus, layout capability is not only important in managing a set of related content within an application, but will also be important for *a priori*, unrelated content that the user has juxtaposed and related for his/her task.

#### 1.2.2.2 Aspect and Aspect View Specification

Developing the previous idea of subset selection further, not everyone is interested in all facets of the information being used. For example, in the case of contact information for friends, even though the user may have information on the job title and place of work of a friend, she may not be interested in seeing it listed in her address book; she may only want the phone number and e-mail address listed. (Such a capability is currently allowed in Microsoft Outlook.) Thus, the user may want to specify a particular *aspect* of the underlying information to work with [21]. This idea can be employed for example to keep information synchronized, where different parties view and/or modify different *aspects* of the information, e.g., the software project manager can update the skill set aspect of her team member, while the HR department may update the team member's employment status. Furthermore, shared information e.g., employee name may only be modifiable by one, both or neither parties. Interestingly, the notion of aspects in this sense is very similar to the notion of database views. However, aspects need not necessarily be a subset of properties of the underlying entity. They can be some computational closure on the entity, e.g., the age of the underlying entity based on its date of birth and the current date, or the size of the underlying collection of items [21].

Currently, little information manipulated by users is amenable to this type of control. We argue that the notion of aspects is sufficiently universal and should be available uniformly across data stores.

Subset and aspect specification together can determine the content of the information, but require a corresponding presentation specification to allow user interaction. Haystack currently has the notion of classes of views based on view size that can be applied to aspects as well [21]. However, as it stands currently in Haystack, all aspects are simple collections of properties that show the value of the underlying property.

We take the ideas presented by Quan, one step further by advocating the need for different styles of views for an aspect based on criteria other than size [21]. That is, whereas, Quan has identified one axis of variation of views for aspects, i.e., size, we suggest that the axes of type, and semantics of information are equally important, and other such axes may exist. For example, the items aspect of a list of coordinates may be viewed as coordinate pairs, or drawn as a curve. Furthermore, users should have the ability to select/create appropriate views for the aspect.

We posit that an extension of the document/view architecture used in software application development should be available as presentation customization capability to the user: any *aspect* can have multiple views as determined by the type and semantics of the aspect. Users can be given control over these preferences via tools that understand the semantics of the underlying aspect. Furthermore, for a given view, the user should have control over the rendering preferences based on the presentation primitives used for that view. For example, a view showing textual information should allow specifying font and color preferences, and order and/or layout of the information, whereas a view showing coordinate data rendered as curves should allow line style specifications such as color, thickness, style, etc. Another example would be one where the tool for designing the view provides particular widgets to interact with properties, e.g., a Boolean property can be controlled via a check box (if it can be changed), a slider widget may make sense for property having discrete values, or radio buttons may make sense if the property semantics require only one value (from a set of valid values) can be specified.

Since content customization is not available uniformly to the user, the idea of view specification of an aspect is missing to a large extent in current applications. Once again, this capability is present in some applications, but only partially. For example, Microsoft Outlook, in addition to allowing the user to select fields, allows the user to specify the order of the fields and font/color preferences for contacts [9]. We argue that users should have first class control over the presentation of an aspect, including its layout, rendering and other appropriate preferences.

### 1.2.3 Fixed Operations on Information

Similar to the case in presentation, many decisions about operations associated with the information are made by information producers: certain operations may only be available in certain views, and users must thus adapt how they perform a task based on the fixed views. Although there are exceptions to this (e.g., MS-Office), the user cannot add other operations available elsewhere to a view, nor remove unused operations to reduce clutter [8]. Furthermore, the application developer also generally decides which operations should have which means of access, e.g., menu, toolbar, shortcut, etc. Thus, the user who actually determines the task or the frequency of the operation invocation based on his/her task has little say in this important decision.

## *1.3 Problem Definition*

Based on the above discussion, we can conclude that information management tools that the user needs to accomplish his tasks are rarely organized in a manner that matches how the user thinks about the task and supporting information (either because of personal preferences, task needs, etc.): the information is scattered across applications and is presented as a one size fits all. Users are limited to how the information has been packaged and their experience of it from the author’s vantage, rather than being able to impose their own world view on it; they are passive recipients of information rather than active molders of it. As a result, users are forced to:

- manually collect information by opening various applications
- mentally select and associate items of interest, and reason with them, since they cannot be juxtaposed
- Reenter the data elsewhere, acting as the glue between applications because they do not understand each other's native information format.

Productivity is lost as users are forced to work with the tools and not able to configure the tools for the task at hand and their personal preferences.

Consequently, it makes sense for users to be able to aggregate the information needed for the task into a single task-oriented interface in order to minimize the overhead associated with preparing the information for the task. This capability becomes even more important for tasks that are long-lived or recurring such that the user must access the relevant pieces of information on multiple occasions.

What we have been arguing for is, in effect, task oriented interfaces. Current software does not always match the task or the user's current state-of-mind. Task-oriented user interfaces (as opposed to functionally oriented interfaces that group related operations and information irrespective of user tasks) recognize that information content, presentation and manipulation control should be relevant to the task at hand. As a result of recognizing this shift in UI design paradigms, researchers have attempted to study common tasks in order to reformulate user interfaces to better correspond to the user activities (some of which we discuss in the next chapter).

We take this idea one step further by advocating that a user should have the freedom to define the task, and the ability to configure a set of content, presentation and manipulation primitives required to create a corresponding task-oriented UI which maintains up-to-date content that a user can return to. As a result, the user requires the ability to perform such customization in a general manner, independent of the task, domain, etc., since the user's formulation of the task and supporting information and operations is unique, based upon his/her personal preferences, conception of information, skill level and other factors.

We define an *Information Space* as a user interface that serves as a single home base/console/workspace that co-locates information and operations that are relevant and related to each other somehow based on the user's view, and presents them in a way that is easy for the user to work with. Thus, the problems we discussed above occur because the user has no means of creating and working with an information space. Our goal then becomes to find a way to break the application abstraction barrier, so that information can flow freely into information spaces designed by users for their tasks.

### 1.3.1 Problem Scope

The problem we pose above can be solved by developing appropriate tools for the user that allow customizing which content is relevant to the task, how it should be presented, and which operations are applicable. We posit that in order to provide such computational support, the problems of fragmented information (due to storage location or representation) need to be resolved, so that information can be modeled and treated in the same way for use by the tools. Thus, it is critical that all information be rendered equal in terms of representation and semantic specification.

The set of technologies corresponding to XML, the Semantic Web and Web Services allow us to unify information in how it is represented and shared and render it amenable to user customization. XML as a semantic markup technology removes the burden of manually distilling a lot of information by rendering information machine processable [1]. Furthermore, XML marked up information is stored as text, removing the problem of proprietary formats. Thus, using XML, the problem of automation can be tackled, as the semantics of information are rendered machine understandable. Widespread adoption of web services, a means of exposing software functionality programmatically, should aid in alleviating the segmented nature of information, and make it easier to federate stores and aggregate information based on preferences [2]. Finally, the Semantic Web (metadata annotated web content) will exploit XML and web services in exposing information structure for machine processing, and allowing widespread programmatic access to it [3].

With these supporting technologies providing a common basis for all information, we can begin to tackle the higher level problem: the domain independent tools and framework required for user customization of context. The above mentioned technologies continue to support us in this endeavor. For example, the semantic markup using XML exposes information structure and can be used as labels to assist users in label-based queries that can be quickly resolved over large corpora, thereby helping he user to selectively work with information ("mold" the information).

Relying on the above technologies to solve the problem of information fragmentation, we limit the scope of our problem to developing an information space customization framework and supporting tools.

## *1.4 Thesis Outline*

The rest of the thesis is organized as follows:

- Chapter 2 discusses related work in order to understand different solutions to the problem we pose.
- Chapter 3 uses the knowledge gained from related work to come up with a set of requirements and desirable features and outlines an architecture that satisfies these needs. It also discusses the Haystack platform, how it helps us achieve our goal, and capabilities in it that are currently missing.
- Chapters 4, 5 and 6 discuss the tools and framework that constitute a solution to the above problem, and discuss the principles, concepts, design and implementation behind each of the tools.

- Chapter 7 concludes this thesis and discusses avenues for future research.

# Chapter 2 Related Work

In the previous chapter we identified a number of current weaknesses in the information management tools, and motivated the need to increase the level of usability of information from a user's perspective. To this end, we advocated the use of information spaces: task oriented interfaces that capture and appropriately present the information and tools a user requires for an information related activity. Furthermore, we supported the need for users to be able to define the task, and have appropriate control over customizing the corresponding information context.

In this chapter, we examine previous related work that collectively points to this conclusion and justifies the existence of the problem. Furthermore, we argue that in situations where the problem of context customization may not have been explicitly identified, the problems that have been uncovered are in fact symptomatic of this underlying problem and could be solved by solving it. In the process of reviewing prior art, we also inspect the solutions that have been employed in tackling it. As a result, we obtain insights into designing a solution that allows general creation and customization of interfaces for information-based tasks.

We organize the related work into three main sections that represent a logical progression of thought; a spectrum capturing the shift in control from developer-defined to shared, to user-defined (and developer exposed) information interfaces.

## *2.1 Task-based Information Spaces*

In this section, we discuss two examples of previous work that have clearly identified the need for a task-based interface that captures and co-locates appropriate resources when faced with an information-based task. The first example below is a research project, whereas the second one is a current commercial effort. Both examples are situated in e-mail (the canonical information management task), and interestingly tackle the same problem via different implementations, reinforcing our prior assertion that no perfect task interface exists, and hence the power to create and customize it should be in the hands of the ultimate user. Nevertheless, they both seek to exploit the power of a task based interface by providing a single, convenient interface that co-locates appropriate information and tools.

Both examples below rely on the feature of e-mail as a receptacle of various kinds of information, and provide information management functionality integrated into the mail client. Since e-mail offers a generic information sharing medium and related software is inevitably involved with some type of information management, we can safely consider e-mail management as a proxy for the general information-based task. Thus, by extension of the above argument, we assert that task-based interfaces are useful for information management in general.

### 2.1.1 Taskmaster

Bellotti et al. argue that e-mail users feel "overwhelmed and daunted by the time it takes to deal with all the work coming in through this medium." [4] As a result, they argue, the e-mail interface must be overhauled as it has been "co-opted … as a critical task management resource" and "e-mail tool features have remained relatively static in recent years, lagging behind users' evolving practices." They cite existing solutions to the problem as merely addressing some subset of the problem or amassing uncoordinated features. For example,

- Projects such as Re:Agent and MailCat only support with filing and organizational aspects of e-mail/project management.
- The Microsoft Outlook e-mail client smears the project context across the inbox, outbox and calendar which can all, only be viewed separately.

Hence, realizing that simply providing the ability to manage information does not imply that managing information is easy, they advocate re-designing the e-mail interface for project management by "embedding task-centric resources directly in the client." As such, they reach a conclusion similar to ours: unless the mismatch between users' changing needs and task conceptualization and that of the tool is alleviated, users' difficulties and feelings of overload will continue. Furthermore, the problem should be rectified by a task-based interface that makes relevant resources easy to access "at a glance, rather than scrolling around inspecting folders," i.e., the current UI must evolve to handle the evolving user tasks (from e-mail to project management).

The solution to the aforementioned problems, proposed by Bellotti, et al., is Taskmaster, a Visual Basic add-on for the Microsoft Outlook client designed based on a field study of the nature of task management activities in mail clients. Taskmaster primarily takes advantage of the heuristic that items in the same e-mail thread generally correspond to the same task, i.e. are the same *thrask*. Thus, (incoming, outgoing and draft) messages are grouped into project context thrasks based on such message data. However, users are allowed to adjust such automatic categorization by manual intervention. As a result, users see lists of thrasks, and can select a thrask to see a list of associated items, one of which can be previewed. Furthermore, realizing the first class status of other entities such as attachments and web page links in the task of project management, it allocates a (separate) thrask entry for each such items, thereby not only co-locating relevant content within the thrask collection, but also allowing more granular control over the project context's content. The attachments can be viewed in-place, without launching separate applications. Finally, Taskmaster allows the user to view summary information computed based on underlying data, e.g., nearest deadline for a thrask, contact information for people involved in related messages, etc.

The Taskmaster system was evaluated by users in another study and found to be useful on all three fronts. Certainly, the notion of creating a collection of related content is a powerful one that minimizes searching for relevant content by users, thereby increasing the usability of the information. Also, equality of content status bestows simplicity of interaction with the information, and selecting an aspect for it simplifies assimilation of information and reduces time spent on the task.

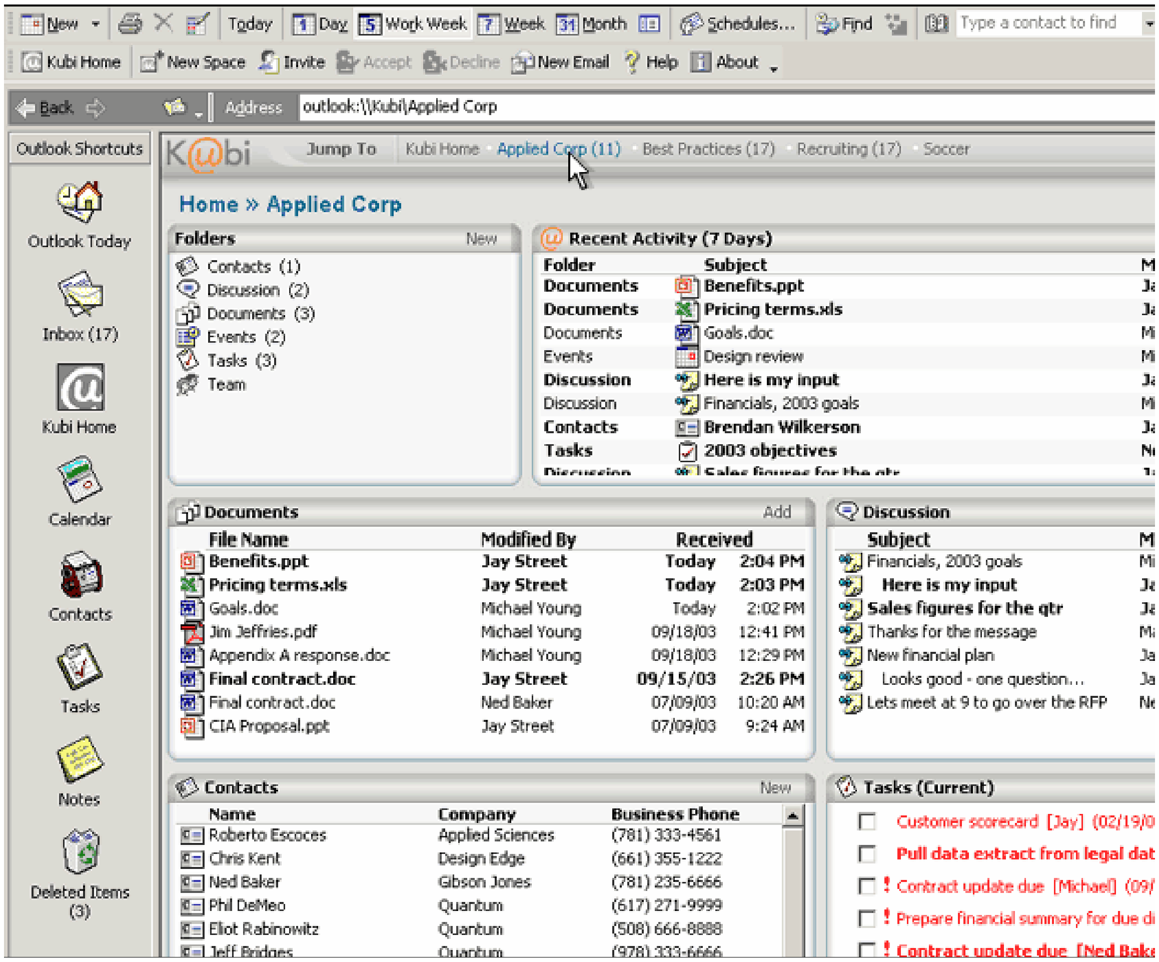
Although the ideas embodied by Taskmaster do increase usability of information from the perspective of content customization, users have little control over the layout or other presentational or manipulation capabilities. Even for content, the task is assumed to be a single project generally captured by a message thread or messages sharing a subject; an implicit assumption that that is the primary granularity of work that users will use. Thus, it relies on heuristics based on well known user behaviors for its advantage. As a result, it falls into the same kind of trap that the earlier e-mail interface faced: lack of flexibility. Although the proposed task interface solves the problem at hand (managing individual tasks), it would not work well if the user's primary collection of interest had different semantics, e.g., messages that were by anyone on the project team, who works for a different company. In such a case, the user would need to manually create and maintain the collection. Given one set of default semantics for collections, it is difficult to capture sets of items into a context that share a different relationship. Furthermore, multiple sets of related information cannot be related or viewed simultaneously. A similar argument can be made for the ability to specify the computation on the information. Users are limited in the types of content customization that are expressible.

### 2.1.2 Kubi Software

Collaboration software tends to exemplify task-based interfaces. The primary purpose of such software is to allow a single point of access to aggregated content and tools with respect to a common goal/project/task in order to minimize the context switch overhead experienced by users when gathering the relevant information. Such tools rely on stores that co-locate related information of pre-defined types, and provide some level of integration with the user by notifying him/her via e-mail of changes to the collaborative workspace/workflow information. A recent, more market driven, effort by Kubi Software takes this idea one step further by attempting to create "collaborative email"; collaborative project workspaces using the stores of popular e-mail clients (Outlook, Lotus Notes) that employ the extant e-mail messaging substrate and thereby further minimize the context switch overhead between the collaboration software and e-mail, for people who "live in email." Here, the fundamental motivation for the product is to co-locate the event notification facility of the collaborative software (e-mail) with the other capabilities and further simplifying the user's task.

Similar to Taskmaster, Kubi Software is pursuing a commercial effort to simplify project management in e-mail [5].  Even though Kubi's product, Kubi Client (see Figure 1), tackles the same problem, it treats the problem as one of collaborative project management in a business setting, as opposed to a single user's project. Thus, it hides and simplifies data replication. Nevertheless, from the perspective of any single e-mail user, the problem is still one of supporting efficient project management by making the relevant content and tools readily accessible.

Unlike Taskmaster, Kubi employs a different ontology for project information.  The primary abstraction is one of a project, which has associated contacts, discussion, documents, events and tasks.  This information is then co-located on the same canvas in little portal-style windows, in order to provide a task oriented console for the project. A screenshot of the Kubi user interface for MS-Outlook is provided below.



| Folder | Subject | M |
|---|---|---|
| Documents | Benefits.ppt | Ja |
| Documents | Pricing terms.xls | Ja |
| Documents | Goals.doc | Mi |
| Events | Design review | Mi |
| Discussion | Here is my input | Ja |
| Discussion | Financials, 2003 goals | Mi |
| Contacts | Brendan Wilkerson | Ja |
| Tasks | 2003 objectives | Ni |
| Discussion | Sales figures for the qtr | J |

| File Name | Modified By | Received | |
|---|---|---|---|
| Benefits.ppt | Jay Street | Today | 2:04 PM |
| Pricing terms.xls | Jay Street | Today | 2:03 PM |
| Goals.doc | Michael Young | Today | 2:02 PM |
| Jim Jeffries.pdf | Michael Young | 09/18/03 | 12:41 PM |
| Appendix A response.doc | Michael Young | 09/18/03 | 12:29 PM |
| Final contract.doc | Jay Street | 09/15/03 | 2:26 PM |
| Final contract.doc | Ned Baker | 07/09/03 | 10:20 AM |
| CIA Proposal.ppt | Jay Street | 07/09/03 | 9:24 AM |

| Subject | M |
|---|---|
| Financials, 2003 goals | Mi |
| Here is my input | Ja |
| Sales figures for the qtr | Ja |
| Thanks for the message | Mi |
| New financial plan | Ja |
| Looks good - one question... | Ja |
| Lets meet at 9 to go over the RFP | Ne |

| Name | Company | Business Phone |
|---|---|---|
| Roberto Escoces | Applied Sciences | (781) 333-4561 |
| Chris Kent | Design Edge | (661) 355-1222 |
| Ned Baker | Gibson Jones | (781) 235-6666 |
| Phil DeMeo | Quantum | (617) 271-9999 |
| Eliot Rabinowitz | Quantum | (508) 666-8888 |
| Jeff Bridges | Quantum | (978) 333-6666 |

- [ ] Customer scorecard [Jay] (02/19/0
- [ ] Pull data extract from legal dat
- [ ] ! Contract update due [Michael] (09/
- [ ] ! Prepare financial summary for due d
- [ ] ! Contract update due [Ned Bake

**Figure 1 Kubi Client User Interface**

Like Taskmaster, Kubi Software's solution also tends towards a task based interface that aggregates and displays relevant content and tools in a useful manner to the user. Although Kubi attempts to solve the same problem as Taskmaster using a similar task based approach that aggregates and presents relevant content to the user, its solution is significantly different; the data model and user interface share little in common. Interestingly, however, it suffers from the same types of defects that Taskmaster suffers from: the developer decides what is relevant and useful, i.e. developer imposed, single task ontology and model.

A Kubi Software enhanced e-mail client provides only pre-packaged types of information that can be used in fixed manners, thereby rendering the collaborative software usable in a single, or perhaps several closely related domains. For example, the Kubi Software solution is limited in the types of information that can be used in a project via folders: contacts, discussion (threaded e-mail), documents (attached files), events (calendar), task list (To-Do items), participants (contacts) and user-defined folders. Notably, all information corresponds to the inherent types *pre-defined* in the Personal Information Manager (PIM) like e-mail clients *by a different vendor*, albeit allowing dynamic selection of actual folder contents based on metadata that defines project context. The only way other information can be made available in the project context is if it is packaged in an appropriate file that the user explicitly specifies in a user-defined folder. Thus, Kubi Software's solution provides limited content customization in the collaborative task of team-based business projects; if a project required a blog or the real-time stock price of a company to be a critical knowledge input to the project, it could not be easily supported with the current project ontology since the underlying e-mail client does not support these notions of information. The software vendor(s) would have to make appropriate changes for this to be possible, and the user would have to wait for someone to "develop" software for his/her needs.

Furthermore, Kubi Software's presentation customizability is generally minimal: the UI views used to render this information reuse existing views (supplied by the e-mail client) in a fixed layout scheme. What if different people in the team want to have different views corresponding to their role in the team, e.g., marketing executive vs. engineering lead? Surely, these people would be more interested in certain aspects of the project's progress than in others.

## 2.2 User Configurable Information Spaces

It is not sufficient to simply present aggregated information and tools in a task-based interface with appropriate presentation to the user. Due to personal preferences, situational demands or varying task demands, users require the ability to further configure their information environment with simple interaction. Thus, a superior solution to the users' task needs would be to allow interfaces that can be configured. In this section, we discuss several examples of user configurability in information based tasks drawn from various domains. Although not all examples we consider below are comprehensive task-based interfaces that make all relevant resources for the activity readily available, they all attempt to provide some level of task support for users' information activities by providing appropriate customization and control over some aspect of information content, presentation or manipulation. (In fact, we argue that they solve problems that are symptomatic of the problem of context customization.) The examples are organized into two categories: Implicit Configuration and Explicit Configuration, and are discussed further below.

### 2.2.1 Implicit Configuration

Implicitly configurable information spaces adapt to the needs of the user without requiring manual specification by the user. Thus, they adapt based on user behavior to better support the task. We discuss a few of these adaptive user interfaces here, while realizing that a lot of work has been performed in this important area.

#### 2.2.1.1 Web Montage

Inspired based on the observation that world wide web access generally follows repetitive patterns, Web Montage is a system that was developed to aid the user in such "routine web browsing" (the task of regularly viewing particular web content, e.g. news, online documentation, comics) by making the most relevant information readily available during routine web excursions [6]. It accomplished this by automatically building a new personalized portal for the user based on his or her predicted context. Thus, if the user was predicted to be in the "developing software" context, online documentation was more likely to be available, as opposed to the "lunch" context, when news and comics might be available. The user context, in turn, was predicted based on a user model relying on various features extracted from web access logs including "time of day, the time since last access, and the recent pattern of topics."

Both the content and the layout of the portal were based on a prediction of the user's current context and the predicted value of previously accessed information or topics in that context. As a result, the user would not have to go seek out the information using fixed patterns of web access. Instead, Web Montage would aggregate and present the most relevant information in the most relevant form. For example, if the information was highly relevant, it would be coalesced into the montage directly. On the other hand, if it was less useful, or the overhead of navigation was acceptable, then only a link would be embedded. Of course, not all prior information was necessarily a candidate for inclusion, nor did it need to have been previously accessed (e.g., it could be the same topic as previously accessed content).

In some sense, by using a start page for the browser along with the Web Montage system adapting the page to suit the task it felt the user was currently involved in (within the task of browsing the web), the web browser appeared as an implicitly configurable information context. As a result, the user implicitly adapted the web browser to the task at hand: the user behavior was used to predict utility of content, as well as best presentation and layout of the content.

#### 2.2.1.2 Personal Information Geographies

For the task of researching and learning about a new topic over an extended period of time by issuing multiple search engine queries, Bauer has proposed the use of an "information map" to help not only aggregate relevant results from multiple, semantically similar queries on a single map canvas (highlighting the importance of organizing some knowledge with respect to other knowledge), but then also to allow its presentation to be personalized as the user's "sensemaking process" unfolds over time, by highlighting 'mountains' of information deemed relevant and diminishing areas of the map considered unimportant [7]. Hence, this approach attempts to *implicitly* support both content and presentation customization for the user's information space pertaining to the research task. Furthermore, Bauer points to the need for a repeatable "reference frame" that a user can return to over time when working on the same research task (an information space) and the fact that it is the user's use of information (i.e., task) that defines its reference frame. As a result, we realize the need for presentation that is both session (i.e., is updated based on previous manipulations or changes since previous interaction) as well as person specific (i.e., the same information is presented differently based on how a person interacts with it, e.g., different nodes in the information map are highlighted differently based on how the user's sensemaking process unfolded).

#### 2.2.1.3 Microsoft Office Suite

Office document authoring software applications have evolved over the past decade and a half to provide significant capabilities to their users. The additional functionality has resulted in ever growing menus and toolbars placing a significant cognitive burden on users to navigate and learn this complex maze of functionality. Depending on where they are in the learning curve, users may only use a subset of the available functionality, or prefer one way of accomplishing a goal even though multiple means of accomplishing it are available. As a result, much of the remaining interface is useless as far as the user is concerned, and just gets in the way. Realizing that this is an undesirable cognitive burden on the users, these applications have incorporated adaptive features such that only what is deemed relevant to the user is shown.

Microsoft Office is one such suite of software applications that present the user with tremendous document authoring power [8]. In order to make the user experience in accessing functionality simpler, it incorporates many "intelligent" features that support the user in his task of editing an office document, presentation, spreadsheet, etc. by only showing those operations that the user has used recently. For example, when a menu is opened, the recently used operations are shown aggregated at the top, with an affordance to view all items in the menu (see Figure 2). In this manner, users are not distracted by menu options they presently do not need or understand. Similarly, given limited screen real estate, toolbars available in the applications reconfigure themselves to ensure that the items the user has used in the past will be visible, i.e., rather than always hiding the rightmost items, they hide the ones the user has not used, in order to move items the user has used to the left, thereby increasing their chance of remaining visible. As a result, the applications "conform" to the user and the task at hand, aggregating relevant functionality to effectively transform an application supporting multiple tasks to become focused on supporting the user in his or her current task.

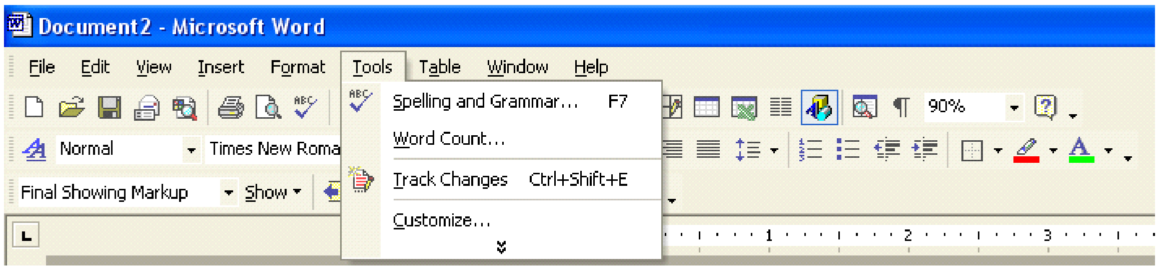


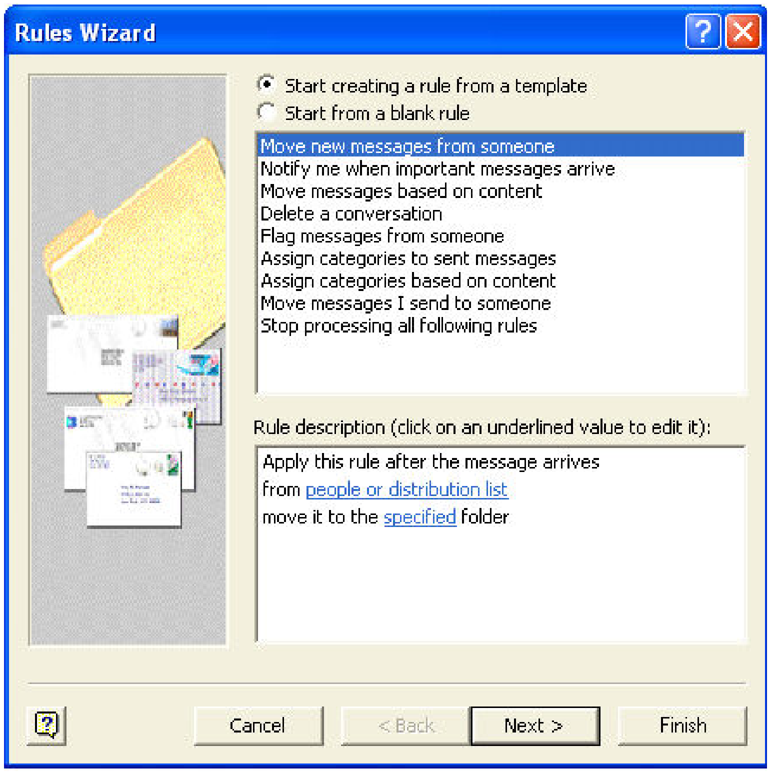
**Figure 2 Microsoft Word Application in the Microsoft Office Suite of applications customizes menus based on recent user actions**

### 2.2.2 Explicit Configuration

We define explicitly configurable information spaces as those that allow the user to explicitly specify his or her preference about some aspect of predefined or pre-implemented functionality in the information space, rather than deducing the preference based on user behavior. Explicit configuration is widely available in applications, and is indicative of the users' need to personalize their interface to provide resources that better support their task. We discuss work that can be characterized as allowing the user to customize the content, presentation or manipulation afforded by the interface. In the process, we understand how seemingly unrelated problems can be cast into the need for context customization as well as ideas and techniques to employ in such a solution.

#### 2.2.2.1 E-Mail Filtering

Most e-mail clients today feature some functionality to support automatic filing of messages and/or spam filtering. Microsoft Outlook is one such mail client that supports user-defined rules that trigger based on message arrival or message sending events [9]. A user may define a rule (see Figure 3) to check a set of (conjunctive) conditions based on message properties, and then perform one or more (conjunctive) actions based on whether the message satisfies the condition clause. Among the possible actions, the user can specify that the message be placed in a particular folder. As a result, the user is able to create a collection of messages that are related to each other by virtue of the fact that they satisfy a particular condition (the rule's condition clause). Thus, the user is able to customize the information content of a folder, and create a rudimentary context containing related information each time he/she switches to view the folder.


Rules Wizard

[✓ Start creating a rule from a template]
Start from a blank rule

Move new messages from someone
Notify me when important messages arrive
Move messages based on content
Delete a conversation
Flag messages from someone
Assign categories to sent messages
Assign categories based on content
Move messages I send to someone
Stop processing all following rules

Rule description (click on an underlined value to edit it):

Apply this rule after the message arrives
from people or distribution list
move it to the specified folder

**Figure 3 Microsoft Outlook Rules Wizard**

### 2.2.2.2 SHriMP

SHriMP (Simple Hierarchical Multi-Perspective) Views is a research effort directed at enhancing information visualization for large and hierarchical information spaces such as software design, knowledge management and flow diagrams [10]. It makes available tools to visualize information at various granularities within otherwise high resolution information spaces, e.g., it provides animated zooming and panning capabilities to "provide continuous orientation and contextual clues for the user" and to select information of interest. In addition to controlling content, SHriMP goes one step further by also allowing the user to determine which perspective he/she wants to use when viewing the detailed information. That is, it allows the user to customize the presentation of the information to the particular task at hand, e.g. viewing the source code for a class versus its documentation. Furthermore, the user can change between views as necessary with low context switch overhead.

### 2.2.2.3 Toolglass and Magic Lenses

A similar idea was embodied in the Toolglass and Magic Lenses See-Through Interface developed at Xerox PARC that allowed the user to interactively change his/her view of the information (space) by placing a see-through widget/lens on some underlying information entity to "reveal hidden information, to enhance data of interest, or to suppress distracting information" as needed for the low level manipulation task at hand [11]. Such an interface allowed the user to easily customize the presentation of the underlying information depending on the serendipitous change of operation he/she wanted to invoke or how he/she wanted to conceptualize the underlying information, resulting in quicker, easier and less error prone task completion.

#### 2.2.2.4 QuickSpace

Users evidently require control of not just additional relevant information, but also its presentation in an information space to make effective use of it to complete their tasks. Several research efforts have focused on providing users greater control along this axis of their information space. One example of this is the QuickSpace project at the Georgia Institute of Technology, which implements simple window management operations to allow users to quickly allocate greater space to their primary operating window while maintaining the overall layout of the desktop [12]. Interestingly, this solution is provided in response to the problem of not being able to "efficiently display all of the information available in …applications" on the desktop (all presumably opened to complete a task requiring information distributed across application boundaries); another instance of the need to aggregate relevant information. Furthermore, the solution attempts to do so while avoiding disturbing the user context as partially captured in the relative locations of the windows and their visible content, i.e., maintaining "information preservation." Future work recommended for the project includes other features that maximize the amount of space dedicated to relevant content for the task at hand, while minimizing irrelevant content; molding the physical space to conform to the relevant information.

#### 2.2.2.5 Web Browsers

In certain cases, where the tool used to view information is third party and unaware of the domain of the information, it is difficult for it to provide the right means of specifying content of interest, i.e., it cannot provide powerful/useful query primitives since it is not aware of the structure and semantics of the information (or the structure and semantics are not regular). In such cases, the best the tool can provide is presentation customizability to support individuality of tastes. Web browsers constitute one such example. Web browsers may be familiar with the structure of the information they present, but not the semantics thereof. In fact, HTML, the World Wide Web's lingua franca, was intended to capture semantics of the underlying documents. Nevertheless, the semantics it captured related to document structure and were intimately related to (and subsequently interpreted as) its presentation (e.g., tables, paragraphs, lists, etc.) not semantic structure. Thus, current web browsers are in the rare category of software that only understands presentation structure, not the semantic structure, of the information and hence primarily supports presentation customization for the task at hand, i.e., browsing information having unknown semantics. Examples of this type of software include Internet Explorer and Netscape Navigator. Both browsers allow the user to specify fonts and colors for text and hyperlinks, although only Netscape Navigator allows the user to override the web page author's presentation settings. Both browsers also allow the user to open a hyperlink in a new window and hence co-locate (likely semantically related) information. In addition, each browser supports additional presentation capabilities that are not necessarily shared, e.g., coloring visited hyperlinks, smooth scrolling, playing animations, etc.

The browsers' presentation capabilities are limited, apply uniformly and do so only at the highest granularity (MIME type or a webpage). For example, a color specification for hyperlinks applies to all hyperlinks on the web page, a specification to automatically playing a sound or video clip applies to all clips, regardless of where or how they are encountered and a user cannot specify that all names in the web page be shown in bold.

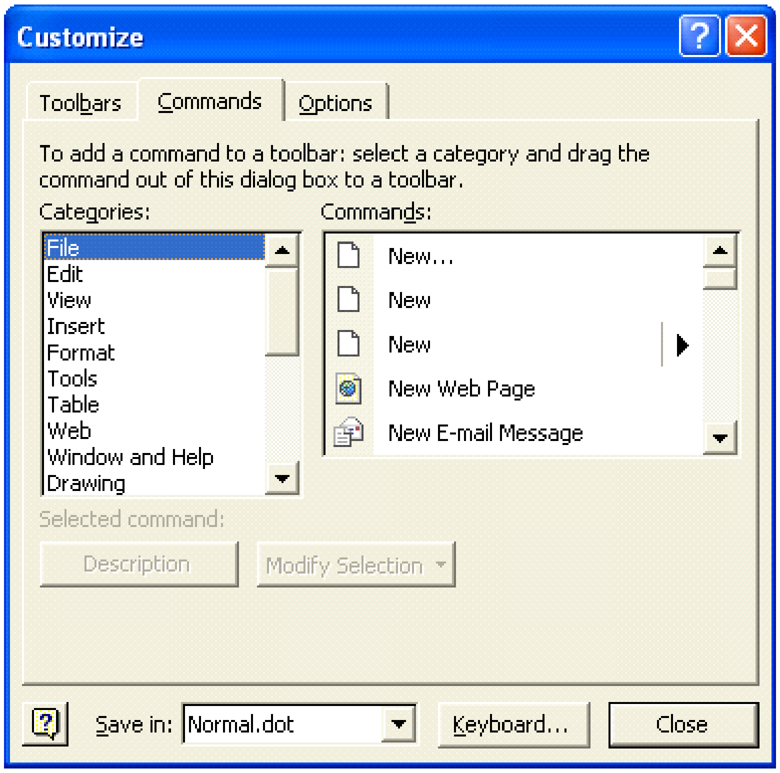
Presentation customizations that support individual viewing preferences and distinctions at a level of granularity lower than a web page currently are not widely supported because of lacking semantics (e.g., the browser does not know which pieces of text are names). Browsers could be made capable of supporting other presentation customizations, related to the semantics it does understand (e.g., paragraph, hyperlink coloring, etc.), but those are mostly structural semantics, not domain semantics. Most of these possibilities are not implemented, precisely because structural semantics do not necessarily (in fact, rarely) correspond to domain semantics and hence users gain little by being able to customize them. The majority of the presentation decisions in any web page are made by the author since he/she best understands its semantics. Thus, lacking semantics force end users to not only use and experience the information content as the author conceived it rather than as they do, but also its presentation.

It is interesting to note that the failure of users to shape their information space and experience due to lacking semantics is significant enough to spur major areas of research in capturing or assigning additional semantics to the content on the World Wide Web. Two efforts are notable in this respect: the Semantic Web and wrapper induction. The Semantic Web seeks to allow authors to annotate information with metadata such that it is rendered machine understandable in terms of domain semantics and not just structural semantics, thereby allowing the browsing tool to offer the user greater control over the content and presentation of the information (in addition to other benefits such as agent-based automation) [3,13]. In wrapper induction, software or the end user associates presentation and structure currently specified by the content provider on web pages as a proxy for semantic structure which is used to infer additional underlying domain semantics thereby enabling greater user control over both content and presentation. An example of wrapper induction would be attempting to infer from a webpage listing movies that all items in bold followed by a comma separated list correspond to a movie title in bold, followed by principal actors in the movie, thereby allowing the user to specify that only movie titles from the webpage are to be shown, and all other information is to be suppressed [14].

Thus, we realize that an understanding of semantics is crucial to allow exposing presentation customization functionality insofar as understanding them allows specifying what the presentation specification applies to. However, specifying what a presentation specification applies to is equivalent to specifying a subset of the content, and thus reduces to the problem of content customization – a means of specifying content of interest. Thus, information semantics are crucial to meeting our goal of allowing the user to create and customize an information space by controlling both the content and presentation of information.

### 2.2.2.6 Microsoft Office Suite

Similar to the idea of implicitly modifying menu items and toolbars, the Microsoft Office Suite of products also provides the user with the ability to explicitly configure various aspects of the applications [8]. For example, users can create/manage their own toolbars by specifying a set of most frequently used task-specific operations to aggregate that may not otherwise be implicitly co-located since they reside on different menus or toolbars (see Figure 4).



To add a command to a toolbar: select a category and drag the command out of this dialog box to a toolbar.

Save in: Normal.dot

**Figure 4 Microsoft Office interface allowing users to customize the operations available from toolbars**

Furthermore, the user can specify which toolbars to show at any given time. Finally, users can save these (and other) preferences in document templates. As a result, starting with a particular document template reconfigures the application to be task focused for that type of document. For example, the user can specify which toolbar should be available in a particular template by default, thereby placing the corresponding operations within easy reach upon application start up.

## *2.3 User Creatable Information Spaces*

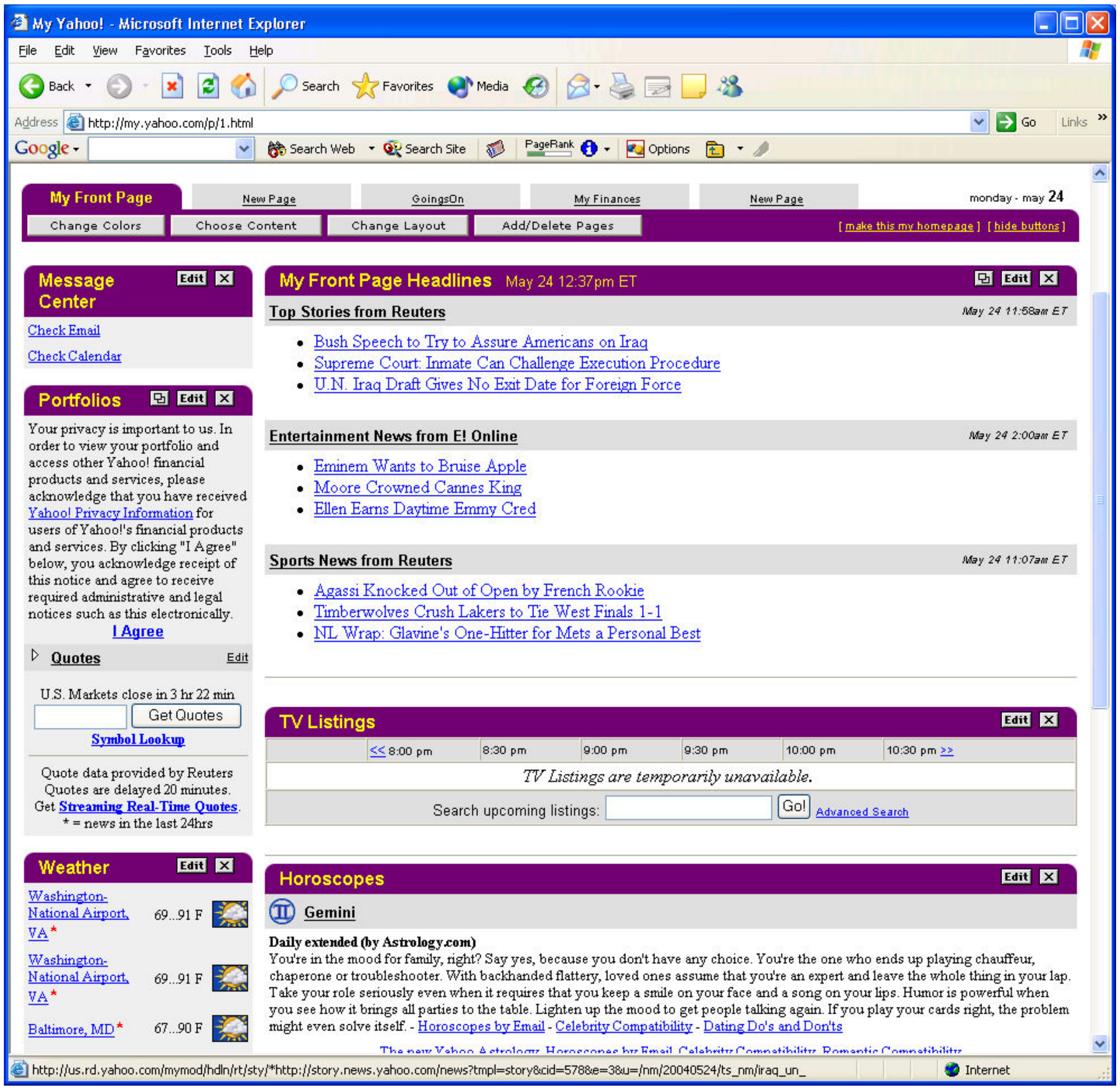
Having surveyed previous work that supported the notion of developer specified task centric interfaces as well as user configurable interfaces, we look at some information spaces that combine the two ideas where the developer exposes primitives that a user employs to create the context as he or she sees fit. Thus, the user is explicitly aware of a creation phase for the information context requiring relatively complex interaction, which is separate from a usage scenario. These examples acknowledge the need to give the user control of information context creation when he/she really is the best person to be creating it.

### 2.3.1 WWW Portals and News Sites

Content portals on the World Wide Web and news organization web sites nowadays provide a rich set of primitives that can be used to create personalized web pages that allow the user to filter and/or aggregate the content provided by the underlying organization(s). Many examples of such personalizable web pages are available, and we discuss one of them here: the *MyYahoo!* portal (see Figure 5) [15].

*Yahoo!*, a popular news, information, communication and e-commerce web portal, offers individuals the ability to configure the available information resources on the portal based on their personal needs (see figure below). Thus, the user has the ability to specify the content and presentation settings for *MyYahoo!*. *Yahoo!* offers modules of information that can be selected and grouped together to create contexts of related (as the user sees it) information or activities, e.g., a user defined personal information page consisting of mail, calendar, address book, and weather. The user may also select from pre-configured contexts roughly corresponding to newspaper sections that subscribe to certain modules of information, e.g., Finance, Technology, Entertainment, Travel, Sports. Furthermore, the user may specify parameters for the modules of information, e.g., which news sources to get information from, which companies to track, which zip code to get weather for, etc. Finally, each context can be configured to have a refresh rate, name, etc.

In addition to allowing user sophisticated control over information content and organization, *MyYahoo!* also allows the user significant latitude in its layout and presentation. For each context, the user may choose between a two (one narrow, one wide) or three column (two narrow, one wide) layout. The user may specify the order of the columns, the content of the columns, and the order of the contents of each column for a given layout scheme. Finally, the user has the ability to specify the background color/wallpaper, and the text font and color for the headings, sub-headings, text and hyperlinks, etc. for the sections. Oddly, the user can also "detach" certain information modules, so that another browser window showing just that module appears in order to allow the user to select a link on the new window and have it appear in the original window. However, this window management technique is not consistently available for all information modules.


My Yahoo! - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://my.yahoo.com/p/1.html

My Front Page | New Page | GoingsOn | My Finances | New Page — monday - may 24

Change Colors | Choose Content | Change Layout | Add/Delete Pages — [ make this my homepage ] [ hide buttons ]

**Message Center**

Check Email

Check Calendar

**Portfolios**

Your privacy is important to us. In order to view your portfolio and access other Yahoo! financial products and services, please acknowledge that you have received Yahoo! Privacy Information for users of Yahoo!'s financial products and services. By clicking "I Agree" below, you acknowledge receipt of this notice and agree to receive required administrative and legal notices such as this electronically.

I Agree

Quotes

U.S. Markets close in 3 hr 22 min

Get Quotes

Symbol Lookup

Quote data provided by Reuters
Quotes are delayed 20 minutes.
Get Streaming Real-Time Quotes.
* = news in the last 24hrs

**Weather**

| | | |
|---|---|---|
| Washington-National Airport, VA * | 69...91 F | |
| Washington-National Airport, VA * | 69...91 F | |
| Baltimore, MD * | 67...90 F | |

**My Front Page Headlines** May 24 12:37pm ET

Top Stories from Reuters — May 24 11:58am ET

- Bush Speech to Try to Assure Americans on Iraq
- Supreme Court: Inmate Can Challenge Execution Procedure
- U.N. Iraq Draft Gives No Exit Date for Foreign Force

Entertainment News from E! Online — May 24 2:00am ET

- Eminem Wants to Bruise Apple
- Moore Crowned Cannes King
- Ellen Earns Daytime Emmy Cred

Sports News from Reuters — May 24 11:07am ET

- Agassi Knocked Out of Open by French Rookie
- Timberwolves Crush Lakers to Tie West Finals 1-1
- NL Wrap: Glavine's One-Hitter for Mets a Personal Best

**TV Listings**

<< 8:00 pm | 8:30 pm | 9:00 pm | 9:30 pm | 10:00 pm | 10:30 pm >>

*TV Listings are temporarily unavailable.*

Search upcoming listings: Go! Advanced Search

**Horoscopes**

Gemini

**Daily extended (by Astrology.com)**
You're in the mood for family, right? Say yes, because you don't have any choice. You're the one who ends up playing chauffeur, chaperone or troubleshooter. With backhanded flattery, loved ones assume that you're an expert and leave the whole thing in your lap. Take your role seriously even when it requires that you keep a smile on your face and a song on your lips. Humor is powerful when you see how it brings all parties to the table. Lighten up the mood to get people talking again. If you play your cards right, the problem might even solve itself. - Horoscopes by Email - Celebrity Compatibility - Dating Do's and Don'ts

http://us.rd.yahoo.com/mymod/hdln/rt/sty/*http://story.news.yahoo.com/news?tmpl=story&cid=578&e=3&u=/nm/20040524/ts_nm/iraq_un_ — Internet

**Figure 5 A sample customized MyYahoo! news portal**

## 2.3.2 Virtual Desktops

Given the application driven nature of information management today, users generally have various applications open in order to access relevant content and presentation when accomplishing a task. It is not difficult to imagine computer desktops as capturing the notion of an information space that capture the content, tools and presentation aspects of the task. Not surprisingly, previous work has attempted to define the granularity of a user task at the desktop level, and shared a similar goal of allowing a user to be able to specify the required information content, presentation and tools separately for each task.

Card et al. propose such a virtual desktop workspace interface based on a Rooms abstraction [16]. Each room (or virtual desktop) corresponds to a separate user task workspace and specifies which tools are open, as well as the layout and presentation of their windows. Rooms may share windows, but the window presentation and/or location may be specific to the room. Furthermore, rooms that capture common tools and information may be included in multiple other rooms to serve as common "control panel[s]". The rooms also support a type of tool clipboard that allows users to switch between workspaces, while carrying their tools with them. Finally, just as tasks may have subtasks, or other relationships to other tasks, rooms can capture these relationships via the notion of Doors. A door allows a user to exit one workspace, in order to enter another (either forward, or backwards). In order to support easy orientation and navigation in such a connected environment, the system provides miniaturized renderings/previews of workspaces for easy identification, as well as a means of inspecting the graph connectivity of the rooms. Finally, rooms or subsets of rooms may be saved, and shared with others.

The Rooms system is interesting in several respects. Some aspects of the system are implicitly determined. For example, when a user moves or resizes windows in the course of the task, these settings are captured automatically, i.e., the user need not be aware that he/she is setting up the room. Nevertheless, other aspects such as included rooms or connecting doors must be explicitly specified. Also, the ability to link tasks is powerful as it allows easy access to related workspaces, thereby facilitating hierarchical task subdivisions (or other task conceptualizations) that make it simpler to accomplish and think about a complex task.

### 2.3.3 User-Defined Database Views

Database querying and view definition has been a long standing research area in Computer Science. Earlier databases, primarily relational in nature, required database administrators to be proficient in the database query language and appropriate schemas in order to retrieve relevant information or set up pre-defined views for other users. With the advent of the World Wide Web and other queryable multi-media repositories that require direct user access, being able to shape the result set of information via a good query/result interface has steadily gained more attention. Delaunay$^{MM}$ ($^{MM}$ stands for multi-media), a querying framework for distributed, heterogeneous multi-media data stores developed at the Worcester Polytechnic Institute, is one such attempt at allowing user configurability in result viewing [17]. Delaunay$^{MM}$ basically addresses the problem we pose, i.e. how can a user customize his/her information space. Hence, it answers both the questions of how to customize content (via query specification) and presentation (via layout/presentation specification).

However, in considering the ability of users to customize content with Delaunay$^{MM}$, it is important to keep in mind that it is primarily a database querying tool. Thus, defining the content of the information via a query is the task itself, not a step in achieving the task. Hence, it may be argued that it is not representative of the types of information spaces we wish to create, where specifying the content is one step in achieving the task, not the task itself. Furthermore, given the nature of querying, the information space that is generally created is transient (not returned to by the user), and hence probably not one a user would want to invest significant time in specifying the presentation of. Thus, the information spaces that generally result from its use are unlike the type of user-centered information spaces we have been describing that allow the user to amortize the set-up time and come back to become (re)situated in their context.

Nevertheless, Delaunay$^{MM}$ embodies some interesting ideas that we feel should be explored since they provide inspiration for some of our work. First, like Yahoo!, it acknowledges the need for an end-user to determine both the content and presentation of information. Also, whereas the act of querying is not new, allowing untrained end-users rather than database administrators to query complex relational data stores that use SQL (Simple Query Language) without *a priori* knowledge of the underlying schema or query language is indeed compelling, since it relates directly to our work of allowing users to specify content for their information spaces in a similar situation. Furthermore, since the queries can be saved and re-executed, and yield large result sets, the user may consider reviewing the results as a long term research task and hence we may consider the result set as a returnable context that he/she would spend time specifying the presentation of, thereby justifying its discussion as it relates to our work for specifying presentation of information by the end-user.

In order to query using Delaunay$^{MM}$, the user must first specify the data store. The system then dynamically queries the store for its underlying schema which is then presented to the user so he/she can specify the **select**, **from** and **where** components of the query. A similar abstraction is provided for querying the web via an object oriented data model and conversion of the query to WebSQL. The query is then translated to the appropriate SQL syntax and dispatched to the store. Although Delaunay$^{MM}$ does not hide the data model for the relational stores (i.e., the user must understand what **select**, **from** and **where** mean) to avoid introducing implementation complexity, it does avoid the intricacies of SQL syntax (especially different flavors thereof) and does not require detailed knowledge of the schema. Such an approach yields significant progress in simplifying the content customization for an end-user by creating a unified abstract data model for the disparate data sources and simplifying the query specification.

Delaunay$^{MM}$ also allows the user to specify the layout and format of the results of a query by specifying a "virtual document" for the result set, and corresponding style sheets for the pages of the document that bind to the query result. Users can customize the style sheets by dragging and dropping type specific widgets (e.g., image, text block, audio and video) that bind to particular components of particular queries, directed at a particular store. All such widgets inherit from an icon widget that specifies the data binding, physical location on the layout and the corresponding query. Users can then configure not just the layout of the results, but also the low level presentation specifications on the widgets, such as fonts for textual results. Finally, Delaunay$^{MM}$ supports the use of presentation templates as starting points for naïve users, while facilitating sophisticated rules based layout functionality for advanced users. Thus, Delaunay$^{MM}$ supports an intuitive and interactive UI for allowing the user to configure information presentation by specifying overall layout as well as low level rendering specifications of particular components (e.g., fonts, etc.).

#### 2.3.3.1 Web Content Collection

The advent and rapid adoption of the World Wide Web as the single largest, publicly accessible information store has led to an increasing realization that how users use information does not always match how information providers organize it (granularity of information) or envisage its usage context (domain of application). Also, there is the realization that information is widespread and incomplete at any single resource, and hence a sophisticated user will require the ability to collate information from various locations in pursuit of a single goal. Thus, many avenues of research have started to ask the question, "How do we allow the user to collect and organize relevant information (i.e., customize information content) for the task at hand while minimizing manual overhead in related information management activities that dilute the time spent on the task and result in unnecessary context switches?"

Hunter Gatherer is a recent system developed at the University of Toronto that targets this question [18]. In particular, it simplifies the ability to capture parts of a web page into a contextualized collection – a collection wherein all elements share something in common with each other, as determined by the collection's creator. Furthermore, it allows the user to "preview" the contents of the collection by having them rendered in a single webpage – a means to visually co-locate relevant information and in effect, allow the user to impose his/her world view on the information and reify it according to his/her needs. Some limited functionality in determining presentation in Hunter Gatherer resides in the ability to change the order of elements in the collection, and hence their order in the layout. Otherwise, the selected bits of information appear in their original rendering (including the embedded link behavior) on the preview page. The success of the tool (with field users insisting on using it after the 4 week field study ended) provides further validation to the realization of the need to allow users to gather information from disparate sources at an arbitrarily determined granularity to support a task not envisaged by the original information publisher; in other words, to create and mold their information space by customizing its content.

Hunter Gatherer contributes several key ideas as they relate to user information spaces. First, it acknowledges that users require multiple distributed/disjoint bits of information to accomplish a single task. Second, it realizes that information needs of users do not necessarily match the granularity of the supply; that is, often times, users need multiple bits of information and that they want to capture only a part of a web page where the relevant content resides (e.g., capturing different parts of an academic conference website such as deadlines, location, guidelines, etc., to create a conference paper submission workspace). Also, Hunter Gatherer allows provides a simple UI for accomplishing the aggregation task such that lay users (the World Wide Web's user base) can use it as well. Finally, Hunter Gatherer gives the user the ability to *create* a *new context* in which the information that he/she has collected is situated. Currently existing bookmarking functionality as a means of co-locating several related web pages in a folder only operates at web page granularity, leaving the user to re-scan the page each time it is referred to, in order to get the subset of content of interest. Alternatively, users can attempt to copy the information, but then must spend additional time and attention labeling it, and/or saving its URL for future reference to its original context.

## 2.4 Conclusion

Having reviewed prior work in the area of interfaces oriented towards users' tasks, we can conclude that such interfaces that aggregate and co-locate relevant information and tools are indeed useful. Collectively, the corpus of related work also argues that the user generally requires some level of control over his or her information space. However, we note that prior work tends to have one or more of the following problems:

1. The interface leverages domain specific insights and is generally inflexible when created by a developer, i.e., the user cannot configure it.
2. The interface does not give complete control to the user over his/her information context, i.e., only a few aspects of it are exposed to the user for configuration.
3. An interface that lets the user build his/her context generally provides primitives geared for a particular domain and hence is not *generally* applicable.

What is needed is a set of context creation primitives that are applicable to information, regardless of its domain such that users can rapidly create new contexts using information from multiple domains. Combining these insights with the understanding that information is increasingly amenable to machine processing in a general manner by virtue of metadata markup, we argue that it is possible to design and develop a general solution consisting of domain independent tools for creating and customizing all aspects of information contexts.

# Chapter 3 Architecture

The previous chapter discussed several examples of prior efforts seen as attempts at providing the user with an interface and content tailored (or tailorable) to her task. We also identified several weaknesses in prior approaches. In this chapter, we draw upon this survey to determine a set of desirable properties of a general solution to the problem of context customization and the resulting functional requirements. We then propose an architecture that fulfills these requirements, followed by a discussion of some of its strengths and weaknesses.

## *3.1 Desirable Attributes*

Based on the strengths and weaknesses of existing work, we propose that a general solution to the problem of context customization should have the following desirable properties:

1. **Domain Independent** – The user should be able to apply the solution to any information in any domain; the ability to customize a task context should not be limited based on the domain of the information, e.g., news. In other words, an appropriate data model should be used so that information of any type can be modeled.
2. **Domain Interoperable** – The user should be able to *aggregate* and *co-locate* information from various domains.
3. **User Editable** – A lay user should be able to create the task context by having control of appropriate domain independent primitives for specifying the content, tools and presentation of the context. The user should be able to interact with the context, independently of designing and creating it.
4. **User Maintainable** – The user should be able to "re-factor" an information context by changing any of its aspects as the task evolves, e.g. including or removing information, etc.
5. **Persistent Returnable Habitat** – A task-oriented information context should be like a habitat for the user, to which she returns to find all necessary resources to

handle the associated task. That is, the time the user invests in setting it up should be amortized over the number of times she returns to it; it need not be re-created/re-configured each time the task must be accomplished. Furthermore, the context should be persistent, i.e., up-to-date, showing the latest relevant information. As such, some level of automatic updating will be required.

6. **Low Overhead Task Switching** – We anticipate that the user will be involved in multiple recurring tasks, and hence will create multiple contexts. Given that users frequently switch between tasks for various reasons (a sub-task, interruption, etc.), it should be easy for the user to switch to a new, related or sub-task [16].
7. **Task Shareable/Synchronized Content** – The user should be able to share content across tasks, and it should remain synchronized (live) in all contexts, no matter which contexts the updates take place in.
8. **Extensible Framework** – Whereas we may not be able to address all aspects of user context customization, our initial solution should establish a framework to support adding other aspects of the solution as they are identified. Also, the solution should support domain specific extensions, on top of the domain independent tools initially available.
9. **Shareable Contexts** – The user should be able to share a description of a context that he/she has set up with another user, thereby resulting in further savings of effort.
10. **Semantic Web Interoperability** – As metadata annotated information is crucial to a successful solution to general information space customization and the Semantic Web is anticipated to become a large repository of such content, employing web services for information sharing, it should be easy to import information from the Semantic Web.

## *3.2 Architecture*

In this section, we propose a solution to the general problem of user information space customization. Our challenge as developers is to provide usefully packaged functionality that allows users to easily specify content, presentation and manipulation that can be combined to create simple yet powerful contexts.

The primary focus of our solution is on an explicit means of information space customization by the user. Although we have seen examples of implicit customization, we avoid this approach in an initial solution for simplicity. Consequently, we also avoid the problem of less expressive user customization that an implicit solution would entail; the user would not be able to easily convey her known preferences and would instead have to hope that the system learns them correctly and quickly.

Our architecture for an initial solution to the problem of information space customization relies on building a set of tools on top of an existing information management environment (Haystack) that provides many of the building blocks required for our solution, in addition to encompassing many of the above desirable properties at the system level. We first discuss what Haystack is, and what it provides in terms of building blocks and desirable properties. We then discuss the set of tools that we seek to build on top of it to create a framework that will aid the user in customizing various aspects of her information space.

### 3.2.1 Haystack

Haystack is a generic information management platform that provides a set of cooperating technologies and tools supporting end-user browsing and application development for the Semantic Web. It encompasses and makes available several key ideas and components that support creating, visualizing and manipulating Semantic Web content, which we describe further below [19].

1. **RDF Data Model** – At its core, Haystack employs a single data model consisting of a semantic network expressed using the Resource Description Framework (RDF), the standard for knowledge representation for the Semantic Web [3, 20]. A semantic network allows knowledge to be captured as a set of relationships between entities and is commonly represented as a graph with nodes (entities) and arcs (relationships).
2. **Adenine** – Haystack provides a domain-specific language (Adenine) for simplifying expression, manipulation and querying of RDF data. Furthermore, imperative Adenine code that manipulates the data can be compiled into declarative data using a target, portable, runtime ontology akin to Java bytecodes, thereby rendering a majority of the Haystack system declaratively specified. Adenine serves as the lingua franca of the system, enabling communication between (and implementation of) its various components via the generic blackboard-like RDF store.
3. **Haystack Services** – Haystack has a service manager that can host services that perform various tasks, including managing the RDF store, populating it, and performing other manipulations and analyses on the information, e.g., categorization, summarization, extraction, learning, recommendation. In essence, the service manager and the set of services it hosts comprise the component that delivers on the promise of automation on the Semantic Web.
4. **User Interface** – Haystack provides a user interface framework (Ozone) that provides interaction primitives and renders views of information entities. It consists of the Slide Ontology, an extensible, HTML-like, ontology for content layout and rendering that is used to create views of information entities. A view for an entity is automatically selected by Haystack based on the entity's type, and the context of use (e.g., available UI real estate).

   Furthermore, the framework supports context sensitive manipulation such as context menus and drag-and-drop operations that are sensitive to the type of the underlying information entity.

   Finally, imperative code in Adenine can be exposed to and invoked by the user via operations. Operations are parameterized Adenine methods that perform a (pre-specified) task for the user using the specified parameters. Relying on metadata annotations on the operations themselves, Haystack employs an automated technique (UI Continuations) for collecting the parameters required for the operations from users. Also, operations may be curried, i.e., the user may customize an operation by specifying values for certain parameters, but not all. The resulting curried operation can be saved, and used as a template for applying the operation in new contexts, that all share the same value for the saved parameter.

Given our goal of creation of custom information spaces, whose content and nature of use are determined by the user and unknown *a priori*, a general information environment that allows working with a variety of information equally well is an ideal first step in achieving that goal. Haystack constitutes such an information environment. It provides a number of benefits which we discuss below, and supplies facilities for building additional tools on a powerful substrate that the user can employ to specialize her information space. Together, these features allow us to meet many of our objectives for general information spaces outlined earlier and facilitate easily supporting others.

- **Semantic Network Data Model** – Due to its simple and generic data model, a semantic network (and hence RDF) allows capturing a broad range of knowledge from various domains simultaneously. Furthermore, although the Haystack data model supports schemas, it does not necessarily enforce them, yielding a semi-structured data model capable of easily accommodating and modeling exceptional situations. Finally, the relationships captured by the semantic network provide a rich set of metadata that the user can utilize to better specify information of interest. This property of Haystack allows us to meet desirable attributes 1, 2 and part of 3, as mentioned earlier.
- **Declarative System Specification** – All system components in Haystack are first class; that is all of them can be manipulated in the same manner since they are just data, captured in the RDF store. This allows a majority of the system (e.g., Adenine code, Ozone slide presentation, ontologies, operations, views, etc.) to be represented in the same fashion as data, and hence renders them amenable to similar user or programmatic manipulation. Furthermore, portions thereof can be easily shared between users or updated via data transfers. Thus, properties 3, 4 and 9 above can be supported.
- **RDF Store** – Using RDF as the standard for representing data allows Haystack to be compatible with the Semantic Web, thereby allowing information from the Semantic Web to be directly imported into its store, without the need for data translation. This property simplifies semantic web interoperability as mentioned in property 10 above.
- **Adenine** – Haystack supplies a built in RDF manipulation language that also provides query primitives. As a result, with appropriate UI support, the user can easily manage the specification of content, as properties 3 and 4 above require.
- **Agent Infrastructure** – Haystack services provide a critical extension point to facilitate automation of repetitive tasks. Also, they allow web services to be written to collect relevant information (possibly from the Semantic Web) for users. With this feature, Haystack makes it simpler to maintain content up to date, and thereby support desirable property 5 above.

- **Blackboard Architecture** – A blackboard style store architecture that allows various computations to communicate by using a single store allows content to remain synchronized across multiple accessing information spaces. As a result, it becomes simpler to meet the requirement of keeping the content across contexts synchronized and up to date as in 5 and 7 above.
- **Browsing UI Paradigm** – All information in Haystack is addressable, and can be visualized using various views, depending on the context in which it appears. Thus, Haystack's user interface paradigm is based on browsing to various entities (addresses), for which the UI infrastructure automatically selects a view to render. This allows meeting property 6 mentioned earlier.
- **Direct Manipulation** – Pervasive use of context menus and drag and drop within Haystack, provides a uniform user interface that increases usability across various information spaces.
- **View Architecture** – The notion of context-sensitive, per-entity views is a powerful one. As a result, Haystack provides the infrastructure for users to be able to specify how to view and/or interact with content in particular situations, thereby supporting properties 3, 4 above. Also, since views can be nested, they can be reused in defining other views – a capability, if appropriately exposed, would allow the user to control how to view a complex entity.
- **Operations** – Since operations can easily be curried (see section 5.2.1.2) and associated with various widgets, the user has significant control over configuring operations, as well as how they can be accessed (properties 3 and 4 above).

### 3.2.2 Context Customization Tools

As indicated by the description of the Haystack system and its benefits, many of the features we desire in user contexts are completely supported at the system level in a general information management environment like Haystack. Other features are present, but only available to the developer. Since our goal is to allow users to have some level of control over the specification of their information space, what is required is a means of exposing capabilities to users that allow them to "author" their own information contexts. Finally, the remaining features can be satisfied by developing the appropriate infrastructure. Thus, our primary task becomes to address property 8 (extensible framework).

Thus, we propose to provide the user with context customization control by providing the requisite infrastructure and a set of tools in Haystack that expose relevant abilities to the user. The set of tools we seek to provide should, in addition to providing the desirable features and capabilities outlined in this chapter, also address the flaws in existing information management solutions (as discussed in Chapter 1). The tools and infrastructure primarily fall into four categories:

- A set of tools that provide the user with a means of selecting information of interest, and the underlying infrastructure that keeps it current.
- A tool for visually aggregating and laying out selected information and operations using particular views in order to design the task interface (information space).
- A tool for creating custom views for information.

- Supporting tools required to work with the other tools.

### 3.2.3 Channel Manager

The Channel Manager is a tool that allows users to define channels of information; a channel of information is simply a set of information the user considers useful and comprises a collection of items related in some respect. The relationship between the items can be articulated by specifying in closed form (via a query or computation). Alternatively, the user may consider the items related in some manner that cannot be expressed or captured through a query, and thus can explicitly specify the collection by placing items in it. A channel of information is always maintained up-to-date by the system and can supply content to the various portions in various information spaces. This tool is described further in Chapter 4.

### 3.2.4 Information Space Designer

The Information Space Designer allows the user to specify space in a visual manner, by defining a high level layout of information. The user can use this tool to determine what is shown by specifying an information entity or channel (described below), as well as how to view such information. Furthermore, the user may specify which set of operations are to be made available in the context for particular entities. This tool allows the user to specify information space level behavior and/or presentation information, such as a title and description. Other context level customization should be made available to the user through this tool (e.g., such as the wiring of preview windows to selections of particular collections). The user may preview how the context will look and behave via the dynamic preview at design time. This tool relies on integrating other components specified by the user (e.g., channels and views) to yield the final user configured information context. This tool is described further in Chapter 5.

### 3.2.5 Information View Designer

The Information View Designer allows the user to create a view for an information entity. A view captures the layout and rendering preferences used to present a particular entity. It serves a lot of the same functions as the Information Space Designer, but at a lower granularity. The user designates the views designed here for use in the Information Space Designer (as well as recursively, as explained later).

A simple domain independent view designer is implemented to allow the user to inspect any property of any information object and to lay them out in two dimensions. Like the information space designer, it allows the user to preview how the view will look and behave. The Information View Designer is discussed further in Chapter 7.

### 3.2.6 Supporting Tools

The above tools require a set of supporting tools that supply additional functionality or information for the user. They are discussed further in the various chapters related to these main tools, as necessary.

# Chapter 4 Channel Management

The previous chapter outlined the basic architecture of our solution for user customizable information spaces. In this chapter, we discuss the Channel Manager tool for Haystack that addresses the issues surrounding one aspect thereof: content specification. We focus here on describing the content of interest, and combine this capability later (Chapter 5) with the notion of information space creation to understand how users can specify content of interest within a particular information space.

## 4.1 Techniques of Information Specification

Fundamentally, user specification of information of interest may be segmented based on the size of the information. The information of interest may either be a single entity, or a collection of items. This criterion is easy to understand; many times, users are able to specify particular information objects (singletons) of interest, e.g., the project manager's schedule. In other cases, users are more interested in "interesting" sets of information, e.g., all people working on a particular project. (Of course, in a general sense, one may conceptualize all information as consisting of collections, with singletons corresponding to collections having a single item. Nevertheless, little is gained by enforcing this ontological efficiency, and much is indeed lost in the additional complexity introduced by an abstract treatment of content in this manner; a singleton is a sufficiently common occurrence to warrant separation as a first class means of specifying content.)

In addition, information may also be segmented based on style of specification. This second criterion distinguishes information the user is interested in based on whether it is obtained by explicit specification (e.g., the entity corresponding to Karun Bakshi, David Huynh, etc.) or implicitly specified based on a description of the information of interest (e.g., people working on the Haystack project).

Single entities are often specified directly (as opposed to described, e.g., Karun's brother). (Indirect specification of content by describing it cannot necessarily guarantee a single item matching the description as in the case of Karun's brother. This is only possible if the schema enforces a unique value (e.g., John's Mother). Since Haystack supports a semistructured store that does not enforce schemas, in the general case, an indirect specification results in a collection.) On the other hand, collections of items may be specified explicitly by enumeration of the members, or implicitly via a description. It is important that these notions of information specification be supported by any environment seeking to allow the user greater control over his information space.

Haystack naturally supports the ability to explicitly specify items of interest (either single entities or collections) by identification. We briefly discuss the notion and implementation of identity/addressing in Haystack. In Haystack, all entities are referred to via the same naming convention: Uniform Resource Identifiers (URIs), the standard RDF naming mechanism which is natively available in Haystack. Thus, each addressable entity, whether it comprises a single item, a collection of items, or each of the members of a collection, has a unique URI. A URI is generally system generated, and meaningless to a user. A user is expected to associate a human-readable title or label for the entity. (Of course, a user is free to assign the same name to two nevertheless distinct entities.) Henceforth, when we discuss information objects or entities and the ability of users to refer to them, we assume the user specifies the entity via some means that allows the system to unambiguously infer the corresponding unique URI being referred to (e.g., using drag and drop). Thus, Haystack makes it simple to explicitly refer to items since every information object has a corresponding unique identifier.

In this chapter, we address user specification of collections of items since specifying a single item generally does not amount to more than identifying an information object explicitly (i.e., somehow identifying the exact URI of the object), and allows little additional user customization or automation.

## 4.2 Channels

Although Haystack allows users to explicitly construct collections by enumerating their members, what is missing is the capability for the user to describe the collection of information of interest (specifying information implicitly). Defining a means for users to implicitly specify collections would result in two distinct methods of specifying collections: either completely explicitly specified or completely implicitly specified. However, we realize that these alternatives are special cases of a single, more powerful abstraction: collections that can be cooperatively maintained by the user and the system. We coin the term *Channel* to refer to such an abstraction. With this abstraction, explicitly specified collections only require user input, and implicit collections are computed solely by the system. However, we retain the ability to let the user and the system to cooperatively build a collection: parts of the collection can be specified directly, and other parts can be computed.

We define a channel as consisting of a collection of information entities that satisfy certain properties specified by the user using a set of description primitives (as opposed to a collection whose members are explicitly specified by the user). A channel's contents are maintained automatically. The primitives used to define channels and the user interface used to configure those primitives is the primary topic of the rest of this chapter.

The notion of a persistent query of information is not entirely new. For example, RSS feeds or blogs can be considered persistent and fixed queries resulting in a collection of information related to a particular, predefined topic (controlled by the content provider) that is always up-to-date [22]. Unlike RSS and blogs however, with channels *users* can change/specify what constitutes the collection by changing the description primitives to select a subset of the available content, i.e., they need not rely on the "packaged" RSS feed or blog, but can rather choose a logical partition of the entire available content. E-mail rules (e.g., MS-Outlook) and information portals like *Yahoo!* constitute a better example of channels, where the user does have some level of control as to which set of information is desired [9, 15].Thus, users can impose their world view on the information corpus. However, these latter examples are situated in a particular domain (mail, news) and the capability for selecting appropriate content is lacking for arbitrary information. In contrast, channels in the context of a generalized information management tool (such as Haystack) that can represent and maintain information from myriad domains can allow users to aggregate otherwise scattered information into a single collection, thereby minimizing the overhead of "hunting and gathering" to collect it.

Because a channel allows naming and manipulating a collection whose membership may be unknown, it constitutes a flexible and abstract *unit* of content that can serve to supply underlying content in various contexts. It can also be used as a primitive for manipulation, e.g., using an algebra that combines information in useful ways. Channels provide several additional benefits:

- The modularity of channels lets users define information properties in a virtual manner ahead of time, without knowing what they will apply to. This is possible, because the nature of the channel's content is known *a priori* based on the channel's description. Thus, each individual item need not be annotated with certain properties; the fact that an item has certain properties that allows it to be a member of a channel, also allows it to (virtually) "inherit" the properties of the channel dynamically, e.g., security. For example, a channel can be designated as being secure and only accessible by certain users, without knowing the actual members of the channel. This decision can be made because the description for the channel indicates that it should consist of all items marked "Classified." If an item ceases to be marked "Classified," it ceases to belong to this channel and thus also ceases to have the corresponding security restrictions.
- Modularity also allows information channels to become reusable and redirectable. They can be used in different contexts and redirected to different portions of the UI (within, or outside an information space) where the corresponding subset of information is useful. Or, the information corresponding to a channel may be redirected to a different device altogether, e.g. if an employee becomes sick or seeks to work from home, the channels appropriate to the work project can be subscribed to from the home computer. In effect, channels allow us to irrigate our tasks with the necessary information.
- Channels as an abstraction are also useful in hiding the distributed and segmented nature of information by allowing aggregation of information from multiple

stores. For example, the notion of viewing e-mails related to a particular topic regardless of which e-mail account it may have arrived in is a powerful one.
- As an indicator of the current user task focus, channels allow an information management platform a simple but useful technique to perform gate-keeping actions by minimizing users' interruption with events or information unrelated to the current task. For example, a user working on a task requiring information from a set of channels need not be interrupted by newly arrived (or created) information that does not fall into any of the channels. When he/she switched to the context that requires the channels that those items did fall into, he/she will become aware of them.
- The persistent nature of channels ensures that they are always up-to-date, and thus eliminates the need to manually review a potentially large and dynamic corpus of information in order to appropriately organize it.

Channels are primarily a self-maintaining organizing mechanism for dynamic corpora of information; they allow the user to impose his/her world view on an otherwise raw and changing set of information by defining a set of persistent indices of relevant information that are then maintained up-to-date by the system. (An alternative view of channel definition is to view it as the act of defining a simple agent whose output is the collection of interest.) Thus, given a store being modified by the user, agents and incoming information, the user can create a stream of information that is important to him/her independent of the source or creation method of the information. For example, channels can be used to segment communication via e-mail, instant messaging and other content delivery web services into a social channel, work channel, bills channel, news channel or high-priority channel. As a result, the user can impose the semantics of "social" onto a collection of information, regardless of whether it consists of IM, e-mail, or both. Channels can also be used to maintain a list of to-do items which may include not just those the user has specified, but perhaps also those that a supervisor has assigned. Additionally, a channel may consist of a list of the people working on a project that is automatically updated as the underlying information changes. Finally, a channel can be the result of a combination of other channels: to-do items of all people working on a project.

## *4.3 Design*

In this section we discuss the design of the infrastructure that supports channels in Haystack. It consists of two main components: infrastructure that computes and maintains the channels, and a set of user interfaces that facilitates channels definition and viewing.

### 4.3.1 Channel Generation Infrastructure

The channel manager agent is responsible for keeping channels current. As a result, it periodically updates all channels based on their descriptions. In this process, the channel manager agent relies on the channel ontology (described below) and a simple, extensible channel definition language that allows composing computational primitives into descriptions of channels.

#### 4.3.1.1 Channel Ontology

The channel ontology describes the attributes a channel can have. A class named `channel:Channel` is declared as a type to be used to annotate channels. In addition to the title and description accompanying most entities in Haystack, channels have four main properties: `channel:active`, `channel:targetCollection`, `channel:updateSpecification` and `channel:setTransformInstance` (STI).

The `channel:active` attribute is a boolean valued property that allows toggling the current status of the channel. If it is set to off, the channel is not kept current. The `channel:targetCollection` attribute specifies the underlying collection that receives the items that match the channel description. The `channel:updateSpecification` property identifies an update specification object that in turn stipulates how and when the channel is to be updated, e.g. fixed times, periodic, event driven, etc. This property is currently not exposed to the user. Finally, the `channel:setTransformInstance` property points to an item of type `channel:SetTransformInstance` that represents a computation closure that specifies the description of the channel.

We digress here briefly to explain certain concepts in Haystack whose current implementation significantly affected the design and implementation of Channels related infrastructure. A computation closure captures the "instance of a method call". For example, a method that computes the maximum of two numbers, Max(n1, n2), can be called with various parameters. In order to capture a particular call instance having particular parameters, say 3 and 4.5, the computation closure would specify the method being called (Max) as well as the values for the parameters (n1 = 3, n2 = 4.5). In Haystack, methods that can be invoked by users (e.g., send e-mail) are called operations. In order for the user to invoke them, he/she may need to specify certain parameters such that the *operation closure* can be determined so that the operation can be invoked with the specified parameters. In order to collect parameters from the user to create a closure, Haystack uses the notion of a User Interface Continuation (UI Continuation). When a user attempts to invoke an operation, a UI continuation is displayed to prompt the user for the required arguments. Although, such a continuation may be specialized for the operation, Haystack can also display a default UI continuation based on what it knows about the nature of the parameters.

An object of type `channel:SetTransformInstance` represents a custom operation closure that was created for two main reasons. First, and most importantly, the current operation closure ontology requires closures to directly point to the values of the parameters using the argument name as the predicate. As a result, the current operation closure ontology does not allow annotating the arguments' values in a closure as to how they are to be interpreted – a capability we needed (as discussed with the topic of argument vectorization later).

Second, the parameter collection user interface for operation closures in Haystack, UI continuations, was too powerful in what it allowed users to accomplish, and thus was too lax in what it allowed/expected from users. For example, it allowed users to specify collections of values for a given argument, when the underlying operation's semantics only understand a single value for an argument, thereby confusing the user as he/she is unsure of exactly how many values are required, whether they need to be ordered, and how they will be used. In addition, although UI continuations for operation closures not only allow argument values to be created in place but also dragged and dropped, the affordance on the UI is unclear as it directed users to “click here to add” for resource arguments. Also, the user interface was not type safe, i.e. it allowed any resource to be specified for a **`daml:ObjectProperty`**, when in reality, a resource of a *particular* type was expected by the operation. Finally, the operation closure presented to the user was uninitialized, thereby yielding an interface that came up blank for all parameters, when it would make sense to have the values be some reasonable defaults. Due to these reasons, the UI available for operation closures was lacking from a usability perspective.

An object of type **`channel:SetTransformInstance`** has two properties **`channel:setTransform`** and **`channel:hasArguments`**. The first property specifies the computational primitive of type **`channel:SetTransform`** (a subclass of **`adenine:Method`** that only returns a set of RDF resources corresponding to information entities) that generates a collection of items representing the channel contents. The second property denotes a collection of named arguments to the primitive. (The term Set Transform is meant to imply that some set(s) of information is (are) manipulated to yield another set. Either the set(s) are explicitly specified as arguments, or the entire store(s) from which information is being extracted constitute(s) the set being transformed.)

The arguments for a Set Transform Instance (STI) consist of two types, **`channel:SetTransformActualResourceArgument`** and **`channel:SetTransformActualLiteralArgument`**.
Both argument types have four properties: **`channel:argumentURI`**, **`channel:argumentValue`**, **`channel:vectorizedArg`** and **`channel:vectorSource`**. The first property points to the named parameter for the set transform, whereas the second one captures the current value for the parameter to be used in the STI. The last two properties allow specifying whether the argument is a collection (vector) of values that is supplied by another object of type **`channel:SetTransformInstance`**. In retrospect, the distinction between the two types of arguments is not strictly necessary, but differentiating between resource and literal arguments at this high level made it simpler from an implementation perspective to create appropriate views, without a complicated query that had to determine the type of the argument (resource vs. literal) pointed to by **`channel:argumentURI`**. Also, in retrospect, the distinction between **`channel:argumentValue`** and **`channel:vectorSource`** could have been removed, and a single predicate could be used to capture the value of the argument, which may be used directly, or evaluated before use, based on the boolean **`channel:vectorizedArg`** property.

#### 4.3.1.2 Channel Definition Language

From the description above, it is clear that all channel descriptions are specified in terms of a single type of building block: a set transform. All primitives are known as Set Transforms since they are used to transform an initial corpus of information into something interesting, i.e. a channel. Our goal from the onset was to have a robust architecture that facilitated extension by only having one type of abstraction. A set transform is a simple computational abstraction that may take arguments, and always produces a set of items. We chose not to return a bag of items that allows duplicates as it would make the semantics of operations with returned items more complex than simple set operations understood by most people. Also, we decided not to return ordered lists, since that would also increase the user's burden, as he/she would need to specify a means of ordering the results for each set transform. An order for the items is also only generally useful in specifying the display of information, not in the specification of the information itself. Thus, in the interest of simplicity, we traded off the benefits of order, and duplicate elements. Nevertheless, these capabilities are meaningful, and may be added in the future.

The simple set transform abstraction makes it possible to extend the language of primitives and add to the user's toolkit for channel definition by annotating any imperative code that satisfies the definition, i.e., code that always returns a set of items. Also, allowing the results of set transforms to be re-directed as inputs to other set transforms using the `channel:vectorSource` property in vectorized arguments makes it possible to compose set transforms and increase the expressive power available to users.

Using the set transform abstraction, we defined a number of primitives that together constitute the channel definition language. In designing the set of primitives, our fundamental objective was to provide a few well chosen primitives that could form a basis set that allowed the user to richly express the description for a channel. We identified several categories of primitives that would achieve this objective. However, a set of transforms that do not interact together have minimal expressive power for the user and are of limited utility. Thus, several of the set transforms are provided specifically to allow composing set transforms and/or channels. The categories are as follows:

- **Set Operators** – Set transforms that compute the set intersection, union and difference of the collections resulting from the set transforms provided as arguments. These operators are equivalent to AND, OR and NOT (with an appropriately specified universal set to be used in set difference). These operators facilitate a simple algebra for results from any set transform (including other set operators) and are similarly closed over sets.
- **Query Primitives** – Set transforms that allow users to query RDF triples in order to extract either the subject, predicate or object, given a fixed value for one of the elements in the triple, and a condition on another element in the triple.
- **Identity/Null Primitives** – Set transforms that return an empty set (for creating default instances of channels or default arguments to set operators), or wrap an `hs:Collection` or channel's elements. These primitives allow reuse of existing channels or explicitly specified collections in creating other channels, i.e. channels may be piped into other channels to derive new channels.
- **Others** – Set transforms that perform arbitrary computations to return a set of items.

Using these primitives, a channel is no longer limited to just be described by a simple set transform (e.g., items of type person); rather, it can consist of a composition of several of them that can be combined in various ways. An interesting combination of set transforms is one that is a union of a computed channel and a fixed collection. The resulting channel is one that is cooperatively maintained by the user and the system, where the system allows the user to add items to the computed set. Similarly, a channel consisting of a difference of a computed channel and a user specified collection simulates allowing the user to veto certain results computed by the system. Finally, using both a union and difference, users can manually mold the information computed by the system. Thus, the set operators, along with the ability to wrap existing channels or collections and use them as arguments, allow us to create various flavors of channels.

Although a detailed listing of the available set transforms is provided in Appendix A, we delve into the set of query primitives here as they are all designed in a regular fashion, mimicking the triples nature of RDF.

#### *4.3.1.2.1 Query Primitives*

All query primitives available to the user wrap a simple Adenine (native Haystack language for RDF manipulation and querying) query consisting of three variables and having the form `?subject ?predicate ?object` [21]. The user is assisted in the query building process by a tool that presents the structure of the information in the store – the ontology browser (discussed later). The user can fix the value of one of the three variables, specify a condition on the value of another variable, and the query returns the values bound to the third variable for all matching triples. For example, if the subject is to be returned by the query, than either the predicate or object must be fixed by the user. Thus, a valid query would return a set of pairs. The specified condition is then applied to the appropriate member of each pair, and if it is satisfied, the other member is added to the result set. As a result, all query primitives require two arguments, the value for the fixed variable, and the closure for the condition test to apply to values bound to the conditioned variable.

Similar to the Set Transform Instance, which represents a computational closure of a particular set transform, a condition test closure specifies the computational closure for a user specified condition test. In fact, the closure has the same general structure as the set transform instance. A condition test closure is a class named, `channel:ConditionTestClosure` which has two properties, `channel:conditionTest`, and `channel:hasArguments`. The first property points to objects of type, `channel:ConditionTest` (a subclass of `adenine:Method`) which are adenine methods. The second argument, like set transform instance, points to a collection of arguments, of the same type that Set Transform Instances point to.

In the case of condition tests, we once again felt a need for a different type of closure whose semantics could not be fulfilled by pre-existing classes of closures. As already mentioned in the case of Set Transform Instances, we wanted to preserve the ability to specify not one value for an argument, but several values, possibly the result of a computation. For example, a future condition test might allow the user to test for equality with any of the members in a *computed* set of values (e.g., a channel), rather than a single explicitly specified value. This was not possible with the current operation closure class in Haystack, which only allows *explicitly* specifying a collection of values. Also, the existing closure class had additional UI drawbacks mentioned above, which could be mitigated to some extent by using the same types of underlying arguments (and views) as set transform instances. However, although the argument types to the condition test closure were the same as Set Transform Instances, we could not use the Set Transform Instance closure class, as the semantics of Condition Tests were such that only some, not all, of the arguments are collected from the user; the rest are supplied by the results of the base adenine query using the fixed variable. These are compared to the user specified values to compute a boolean return value indicating whether or not the condition has been satisfied. In fact, Condition Test Closures represent an interesting new type of closure, where the user and system cooperate to complete the task, and present an avenue for future research, e.g. who specifies which arguments, whether this is a static choice, or negotiable, etc. The set of condition tests that are currently implemented are listed in Appendix B.

The query primitives were deliberately designed to be simple to work with, and as a result, do not allow specifying complicated Adenine queries with multiple existentials (variables) that are used for "joins" that allow traversing the RDF graph. An example of such a query with multiple existentials would be **`?x :brother ?y :name "Barney"`**. Here, the return set of interest corresponds to **`?x`** and the user must specify multiple variables (**`?x`**, **`?y`**) and condition on those variables ("has brother whose name is Barney", "has name Barney"). Although powerful, such a capability was primarily not made available as it would probably be too complex for a user to work with (see below). Also, in the general case, all variables being used to perform the join would require conditions to be specified (assuming, the user does not want to always use an equality condition), yielding a more complex user interface and underlying infrastructure for evaluating these queries, for which we lacked implementation time.

Nevertheless, we avoided the concomitant loss of expressive power by incorporating the concept of vectorization. By vectorizing an argument, the user may specify not one value, or an explicit set of values, but a set transform instance that *computes* a set of values (which can also wrap an explicitly specified collection if needed). As a result, the user can specify that the values for the fixed argument of a query primitive are to be the results from another query primitive, thereby simulating the notion of a join. For example, in the above query, the user might create one query primitive **`FirstQueryPrimitive`** consisting of the query corresponding to **`?y :name "Barney"`**. The user can then specify that another query consists of **`?x :brother <Vectorized Argument – VectorSource: FirstQueryPrimitive>`**. The last parameter in the second query is specified as a vectorized argument, whose values are returned by **`FirstQueryPrimitive`**. The set corresponding to **`FirstQueryPrimitive`** is dynamically bound to the first query as would be done in the single expression above, and hence performs the join.

Finally, the resulting experience for the user should also be better as he/she incrementally specifies the join and thus can build the expression in parts in a modular fashion and test each part (using tools discussed in the UI section), without having to specify and debug one large expression at one time – a much more complex undertaking. In effect, by specifying sub-queries and piping their results into other queries one step at a time, the user binds their results to each existential in the larger expression being built.

Compared to existing query languages, e.g., SQL and Adenine, it may be argued that there exists no way for the user to specify a return set consisting of tuples. However, the channel abstraction was meant to provide a single set of items, not tuples of items. Even for a single set of items, one may ask why it is the case that users cannot select (project) the properties of interest, and thus receive tuples of properties of the items rather than the items themselves. Such an approach would return information without its context: for example, the tuple <age, height> is meaningless to the user, unless he or she knows whom it refers to. Thus, if the user wanted to later see a different set of properties, or see properties based on the type of the underlying item (e.g., if it is a dog, show the hair color also), he or she would not be able to since only one set of statically specified properties had been specified. In general, much flexibility is lost in making the decision on which properties are of interest at this stage. Finally, from our perspective, usability was important, and forcing users to specify tuples during the querying phase seemed unintuitive; users would more easily understand that after obtaining a set of entities matching a particular set of criteria, they are allowed to choose which properties to show for those entities, when designing/selecting a view for it.

The decision for our query model may seem arbitrary. Whereas it makes sense to only return values bound to a single variable since the semantics of Set Transforms require returning a set of items (not tuples), why can only one of the remaining two variables be conditioned? Fundamentally, this design decision was driven by the fact that the underlying database has impoverished support for conditions on queries. Specifically, the only condition that can be applied is equality. For example, `?x :age "21"`, returns all items whose age *exactly matches* `"21"`. Thus, for example, one cannot express a query that returns people whose age is greater than "21." Of course, this would not be a problem if there was an infrastructure in place for evaluating conditions on queries, and a mechanism for adding condition primitives to the database to enhance the expressive power of the query language. However, such support is currently lacking.

Thus, the conditions that set transforms support are evaluated *after* the query results are returned from the database. As a result, unconditioned queries must be dispatched to the database. If set transforms allowed conditions tests on both remaining variables (other than the return value), then a *completely unconditioned* query would have to be dispatched to the database, and the condition tests would need to be evaluated based on the returned results. However, a completely unconditioned query of the form `?x ?y ?z` would return the entire database. Such an operation would be very expensive not just in returning the data, but then also in applying the conditions to each of the triples. As a result, we force the user to specify a value for at least one of the variables in the query, so that the database can return a more tractable result set. Note however that there is no loss in expressive power, since the user can specify the two conditions using different queries, and then combine the results.

In the process however, we enhance the query power available to the user as compared to the Adenine expression evaluator alone. A set transform allows arbitrary conditions and the user can more accurately specify the information of interest. For example, the user can choose the condition in the first query primitive above to be that the name begins with “B” rather than it be *equal* to “Barney.”

#### 4.3.1.3 Channel Manager Agent

The channel manager agent behaves like an interpreter for the Channel Definition Language; it periodically evaluates the top level set transform instance for the channel, which may in turn invoke other transforms, and then places the results in the channel’s target collection (replacing the earlier “out-of-date” collection). As a result, changes to the store are not immediately reflected. In implementing the update method for channels, we decided to poll the store periodically. This was primarily done to avoid the performance degradation that would accompany an event based implementation; the store dispatches lots of event since it is being used as a blackboard for communication between all components in Haystack. In the future, perhaps we will allow the user to select for each channel, its update preferences (periodic vs. event based). Currently, the update specifications of channels are ignored (and thus not shown to the user).

We briefly discuss here the notion of vectorization. As was briefly discussed earlier, vectorizing an argument to a set transform allows specifying a set transform whose result supplies a collection of values (a vector) to the transform for that particular argument. Thus, the transform is invoked multiple times, once for each argument value. Since the user may arbitrarily vectorize any of the arguments, the channel manager must have well defined semantics for handling multiple vectorized arguments. One approach would be to assume that all vectors of arguments to a set transform are of the same size, and hence constitute parallel arrays from which tuples of argument values may be removed and used until the arrays have been consumed. This assumption however is not safe since the vectors of argument values may not all be the same size. A safer approach (that we have chosen to use) would be to perform a Cartesian product, and use the resulting tuples to invoke the set transform multiple times.

A second decision that the channel manager agent must make is what to do with the results of multiple invocations of the set transform with various arguments. Clearly, some set operation would make sense. Again, to minimize the complexity of defining channels, this decision is not exposed to the user, and the agent by default computes a union of all the result sets which we felt would generally be the semantics the user would want applied.

It may be argued that our earlier reasoning for not using existing operation closures no longer applies as we are now allowing multiple values for arguments as well. However, the current scenario is different from *explicitly* specifying a collection of values as operation closures currently allow since the items in the collection are the result of *computing* a set transform. Also, the user is explicitly requesting vectorization, and is aware of how various arguments will be combined. Accepting this argument, it may further be suggested that all arguments to set transforms be set transforms, which happen to wrap collections of single items if the corresponding arguments are to have single values. However, we reject this design alternative, as it adds a significant amount of overhead to the user for understanding this esoteric abstraction just for uniformity of implementation. The associated negative impact on usability and simplicity in the common case for the user may not be justifiable.

### 4.3.2 Channel Manager User Interface

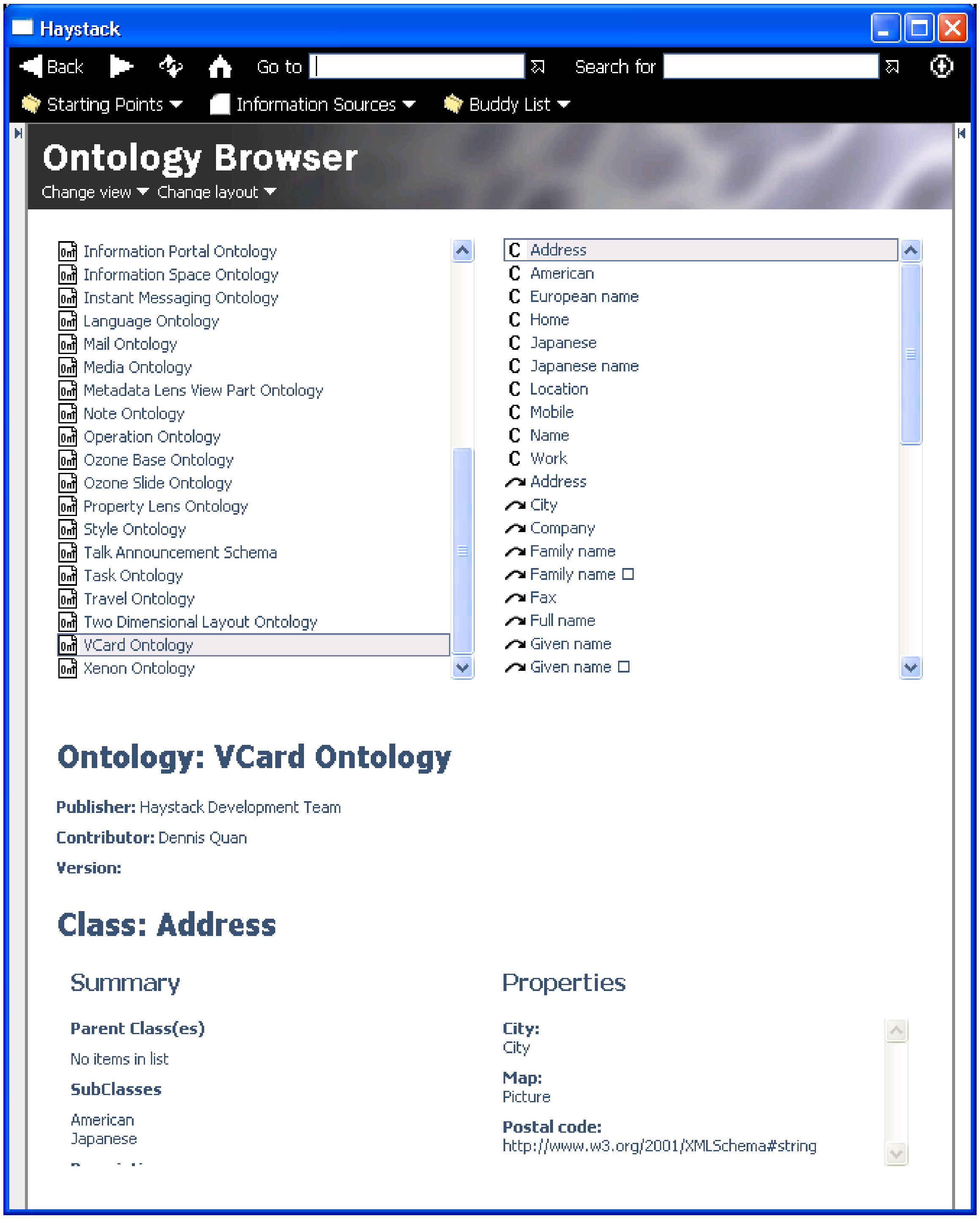
Four primary user interface components are provided for the user to interact with channels. These include: the ontology browser, the channel manager, the channel viewer, and the set transform instance viewer. Each is discussed further below.

#### 4.3.2.1 Ontology Browser

The ontology browser is a tool implemented in Haystack that allows users to explore the set of ontologies that are currently declared in Haystack. The ontology browser is meant to be used by users to understand the structure of information and learn more about the classes and properties in Haystack, e.g., which classes are available to model information, their associated properties, domain/range of properties, description, parent/subclasses of classes and properties, comments, etc. Figure 6 shows a screenshot of the ontology browser where the user is currently inspecting the Address class in the VCard Ontology.

The browser consists of three main panes: the ontology selection pane, the class/property selection pane and the preview pane. Selecting an ontology of interest from the list on the left populates the list in the right pane with corresponding classes and property defined by the ontology. The preview pane then displays the information about the selected ontology. Selecting a class or property from the list of ontology elements on the right then populates the remainder of preview pane, showing information about the selected item. Users can use drag items (classes or properties) from the Ontology Browser and drop them on the right pane. These can later be used to specify parameters for queries in the channel manager.

The ontology browser was designed to be an exploration tool that made it simple to browse information structure in Haystack. Thus, instead of having a single collection of all classes and properties, which would require a user to know the name of the class or property, it allows the user to select an ontology to quickly narrow down the domain of interest, and the corresponding items. Another important decision in this tool was whether to just show two different collection views, one for ontologies, and another for its elements, each having a separate preview pane. This alternative was rejected as it would have reduced the amount of horizontal space available to the ontology elements' preview pane, and thus required the user to scroll horizontally or vertically (if the information were oriented vertically). Thus, the decision to have two collection views would not have been space efficient. The current design allows the user to see everything at one glance.



Information Portal Ontology
Information Space Ontology
Instant Messaging Ontology
Language Ontology
Mail Ontology
Media Ontology
Metadata Lens View Part Ontology
Note Ontology
Operation Ontology
Ozone Base Ontology
Ozone Slide Ontology
Property Lens Ontology
Style Ontology
Talk Announcement Schema
Task Ontology
Travel Ontology
Two Dimensional Layout Ontology
VCard Ontology
Xenon Ontology

C Address
C American
C European name
C Home
C Japanese
C Japanese name
C Location
C Mobile
C Name
C Work
Address
City
Company
Family name
Family name □
Fax
Full name
Given name
Given name □

Ontology: VCard Ontology

**Publisher:** Haystack Development Team
**Contributor:** Dennis Quan
**Version:**

Class: Address

Summary

**Parent Class(es)**
No items in list
**SubClasses**
American
Japanese

Properties

**City:**
City
**Map:**
Picture
**Postal code:**
http://www.w3.org/2001/XMLSchema#string

**Figure 6 Screenshot of Ontology Browser**

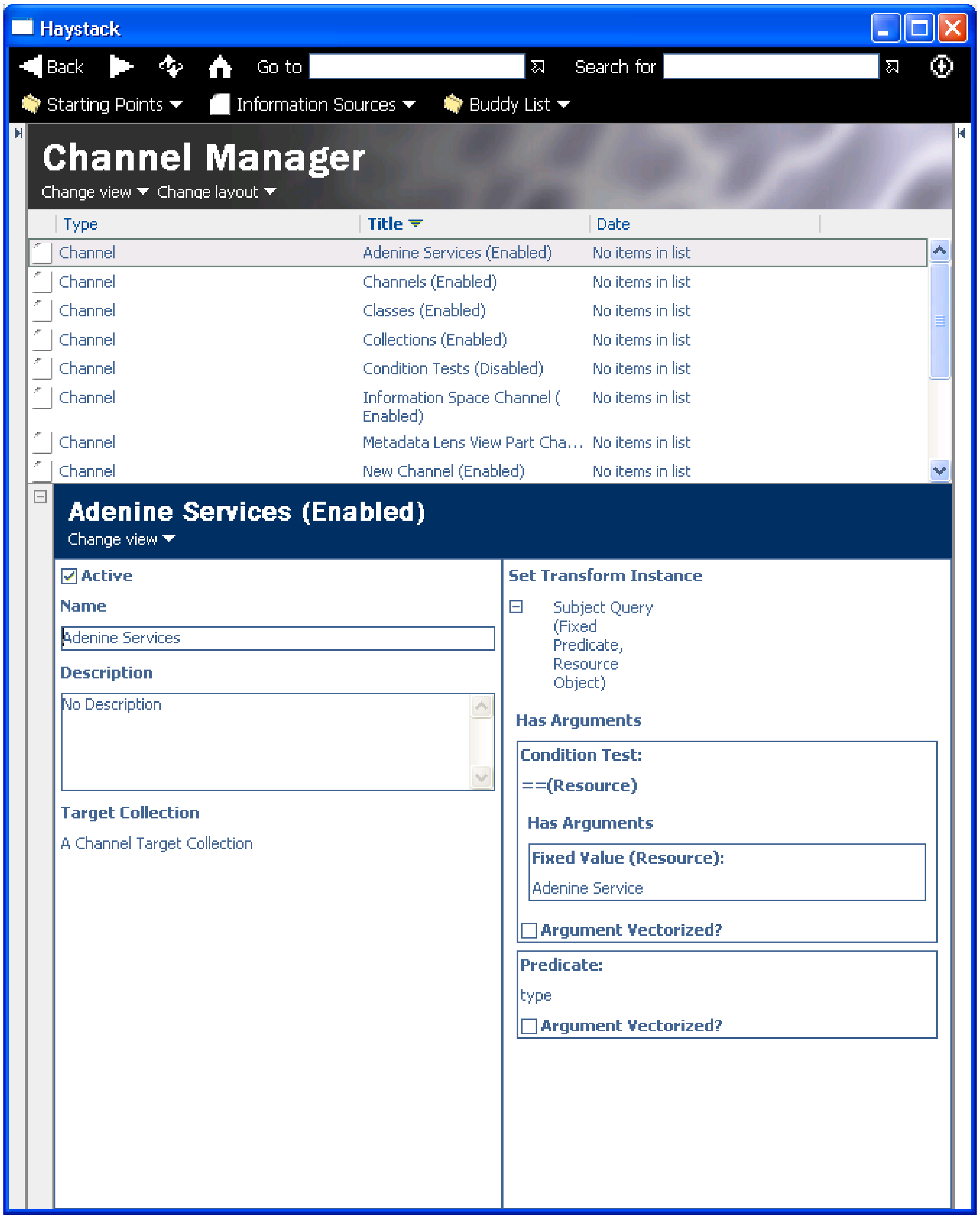
Perhaps the most important design decision in the implementation of the Ontology browser was whether the ontologies presented were statically declared or dynamically generated. Unlike the Lore project at Stanford which presents the user with a dynamic summary of the structure of the information in a semistructured database based on actual usage via DataGuides, we chose to implement a browser that presents only statically declared (appropriately annotated by developers) ontologies [23]. As a result, our browser does not capture undocumented use of classes and predicates. This was primarily done for performance reasons, in order to avoid the overhead of regenerating the DataGuide periodically. Unlike structured databases, schemas in semi-structured stores can easily exceed the size of the actual content stored with the schema. Nevertheless, since most ontologies *are* declared in Haystack, an ontology browser presenting a static structure of information in Haystack was a satisfactory solution.

#### 4.3.2.2 Channel Manager

The channel manager is the primary tool used to work with channels. It allows the user to create, copy, delete and toggle channels on and off. It also allows the user to edit the properties of any individual channel including the underlying set transform instance. Figure 7 illustrates, the Channel Manager.

The channel manager consists of a standard collection view, showing the set of available channels in the top portion, and a preview pane for the selected channel in the bottom portion. It supports pervasive use of context menus and drag and drop to make functionality available to users. Users may right click on the collection to add channels by selecting the “Add new item” operation, as is possible in the Haystack collection view. However, only channels can be added to the collection, as other types of items will be removed from the underlying collection (see the Channel Manager Implementation section for a full discussion). Also, the user may invoke the context menu on any channel in the collection, and gain access to common channel management operations that allow, copying, deleting, or toggling the underlying channel.

Any individual channel may be edited via the preview pane which is split into two main sub-panes. The channel’s set transform instance can be edited in the right pane, and all other channel properties are accessible via the left pane. Since STIs may be nested, the right pane provides a collapsible interface for specifying the arguments for the set transform instance. Like other operation closures, an expanded STI attempts to collect arguments from the user. Resource arguments can only be specified by drag and drop, whereas literal arguments may be entered via an edit box. Unlike current operation closures, all resource arguments specified for set transforms support type-safe drag and drop. That is, even though the closure user interface is general purpose and can be used for any set transform, it adapts to the underlying set transform to enforce type safety of the arguments. If the dragged item is not of the type expected by the set transform, the drag and drop operation does not succeed. If the “Argument Vectorized?” checkbox is checked, then the user must supply a set transform instance that computes the vector of values to be used for that particular argument.



# Channel Manager

| Type | Title | Date |
|---|---|---|
| Channel | Adenine Services (Enabled) | No items in list |
| Channel | Channels (Enabled) | No items in list |
| Channel | Classes (Enabled) | No items in list |
| Channel | Collections (Enabled) | No items in list |
| Channel | Condition Tests (Disabled) | No items in list |
| Channel | Information Space Channel ( Enabled) | No items in list |
| Channel | Metadata Lens View Part Cha… | No items in list |
| Channel | New Channel (Enabled) | No items in list |

**Adenine Services (Enabled)**

☑ **Active**

**Name**

Adenine Services

**Description**

No Description

**Target Collection**

A Channel Target Collection

**Set Transform Instance**

Subject Query (Fixed Predicate, Resource Object)

**Has Arguments**

**Condition Test:**

**==(Resource)**

**Has Arguments**

**Fixed Value (Resource):**

Adenine Service

☐ **Argument Vectorized?**

**Predicate:**

type

☐ **Argument Vectorized?**

**Figure 7 Screenshot of Channel Manager User Interface**

Of special note are the semantics for drag and drop for the two types of closures discussed earlier. In Haystack, an operation closure is created when the user clicks on an operation, and completes the resulting UI continuation that collects the requisite parameters. In the case of set transform instances and condition test closures, the existing means of creating a closure are not used (as previously discussed) and thus, some means must be specified to do so. In both cases, the default construction mechanism of channels and set transform instances automatically create *default* closures where needed, e.g., a new channel always has a null transform instance created. In order to specify a different transform, users can drag any set transform to the target closure, and underlying closure updates itself to become a default instance of that particular set transform. The same means of specifying condition tests is used, e.g., all query primitives have a default condition test closure, and the user may drag a new condition test to it. The system handles creating default instances of new closures as needed, and thus need only worry about configuring them.

Finally, as is visible from the figure above, condition tests, like set transform instances, also collect arguments from users. However, condition test closures do not seek values for all their arguments, as some are supplied by the system based on the results of queries (as discussed in the Query Primitives section). (Also, the ability to vectorize arguments for condition tests is currently not supported.)

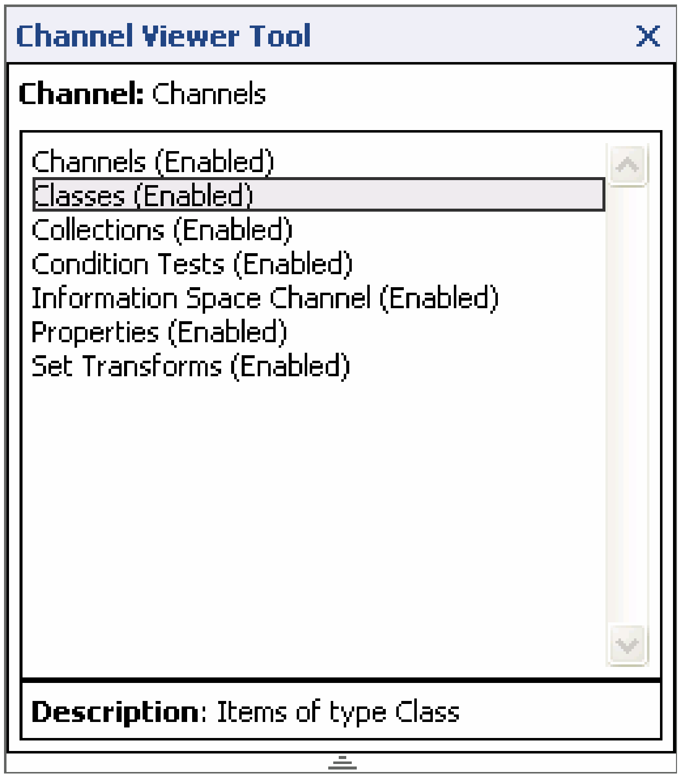
Several design decisions were made to enhance usability of the channel manager. First, default channels have been specified that produce collections of information useful in specifying set transform instances when the user defines his/her own channel, e.g., classes, properties, condition tests, etc. The user may use the channel viewer tools, to keep the information readily available when working with channel description. (The other default channels are defined for supporting the implementation of various managers, including the channel manager.) Also, all information objects used in defining the channel, from channels down to condition test closures, are always auto created and initialized with default values. Furthermore, the user interface, attempts to keep the information valid at all times. For example, a channel is always initialized with a valid null set transform instance that does not produce any results, a newly specified condition test such as the numeric <= is initialized with a default comparator value of 0.0 and replaces the previous condition test, etc. As a result, all information items are immediately valid and useful, and the user need only focus on the subset of information that is important. In the same spirit, the interface also provides a means for users to copy channels, in order to avoid creating and configuring them from scratch. As a result, users can leverage previous work, and only make incremental changes to define new channels. Unlike the current Haystack philosophy, destruction is made available as an operation to allow users to completely remove channel descriptions from the store (as opposed to deletion of an item from a collection). This is done in order to avoid channels that the user may have created that are of no use anymore, but that keep reappearing in the channel manager even after deletion since they still are of type Channel. Thus, the user is needlessly forced to contend with clutter in the channel manager. However, if a user wants to create a channel template that is of no use by itself, but is to be used for copying as a starting point for defining new channels, he/she can create a channel and disable it

The most important design decision made in the channel manager is that of the contents of the right sub-pane of the preview pane: the interface for editing set transform instances. Admittedly, an interface can quickly become complex if it attempts to collect parameters from a user, where the parameters themselves collect additional parameters. The current user interface design for editing set transforms consists of nested arguments. It allows collapsing some or all arguments thereby letting users focus on a particular portion of the channel description, e.g., users can hide the arguments for one set transform instance being used in a set union operation, so that more space can be allocated to the other argument. Whereas this approach is better than a user interface where both arguments to set union are always fully expanded, it is nevertheless lacking in usability. A better design than nested set transforms would be a graph editing interface, which intuitively highlights the ability of users to "pipe" channels into others. However, including such an interface was a complex undertaking requiring significant custom integration and possibly additional coding effort with existing graph editing tools to enforce the desired interconnection semantics and user interaction. As a result, this design alternative was not pursued in favor of a more functional design that could be completed quickly.

Another design decision related to the set transform instance specification was that of the query interface. Querying is anticipated to be used often in defining channels, and as a result, would benefit from a more intuitive interface than the current STI specification interface. Query interfaces have been well studied in the past, and many flavors are possible. These range from very simple command line interfaces that allow users to directly specify the query in the query language (e.g., SQL), to highly sophisticated and correspondingly more usable natural language systems (e.g., START, PRECISE) [24,25,26,]. A natural language interface, although powerful, would require significant effort to implement and deploy in Haystack. Somewhere between these extremes lie query-by-example (e.g., use of DataGuides for QBE in the Lore project, Filemaker Pro), wizards (e.g., MS-Access) and query designer (e.g., MS-Access) interfaces [27,28,29]. Our current interface tends towards the lower end of the spectrum just above a command line interface. Of these alternatives, query-by-example and wizards also seem sufficiently complex (and limited in applicability i.e., best suited to conjunctive queries) such as to merit a separate research project. However, a query designer interface to the set transforms based implementation of the query primitives, would benefit the channel manager's usability without significantly impacting our current research goals. Users would be able to intuitively specify a query rather than worry about selecting the right primitive. This capability will be added in the future.

#### 4.3.2.3 Channel Viewer

The Channel Viewer tool was developed to allow users to monitor the set of items in the channel when it is not being used in an information space, without detailed information on the items themselves in order to have a minimal UI footprint. Thus, for example, a person may want to monitor any urgent e-mails that come from home, while working on some task. The channel viewer tool always appears in the right pane of Haystack, and is shown in Figure 8. In this case, it is monitoring the channel that keeps track of the channels currently defined in the system.


Channel Viewer Tool

Channel: Channels

Channels (Enabled)
Classes (Enabled)
Collections (Enabled)
Condition Tests (Enabled)
Information Space Channel (Enabled)
Properties (Enabled)
Set Transforms (Enabled)

Description: Items of type Class

**Figure 8 Screenshot of Channel Viewer Tool**

The top of the tool specifies the channel being monitored, followed by a list of the members in the channels. The bottom of the tool shows a description of the selected item in the channel.

The user can simply drag any channel that needs to be monitored, and drop it on the channel viewer. The channel viewer then shows the members in the channel, and continues to keep them up-to-date. (Note, the channel is not copied.) The channel viewer behaves like any other applet element in the right pane, i.e., it can be collapsed, removed, etc.

We felt it important to develop the channel viewer for a number of reasons. When items are currently dragged to the right pane, they are always rendered using a view part of type `ozone:AppletViewPart`. However, in the case of channels, it is unclear whether the channel's properties are to be shown, or the items that have been computed to be its members? Although multiple view parts could be developed, one showing the channel's properties, and the other its contents, the problem would not be solved since Haystack *non-deterministically* selects from multiple view parts having the same view part class. Also, in the case where multiple instances of something are to be examined, one at a time, dragging each instance to the right pane overcrowds it since previous instances that are of no use still linger in the pane. For example, if the user wishes to examine the contents of Channel A, and then Channel B with no more use for Channel A, he would first need to manually remove Channel A, and then add Channel B. He cannot *replace* Channel A with Channel B. Finally, if the same item is added to the right pane multiple times, it is duplicated, and shown in the same view, again contributing to overcrowding, with little additional benefit.

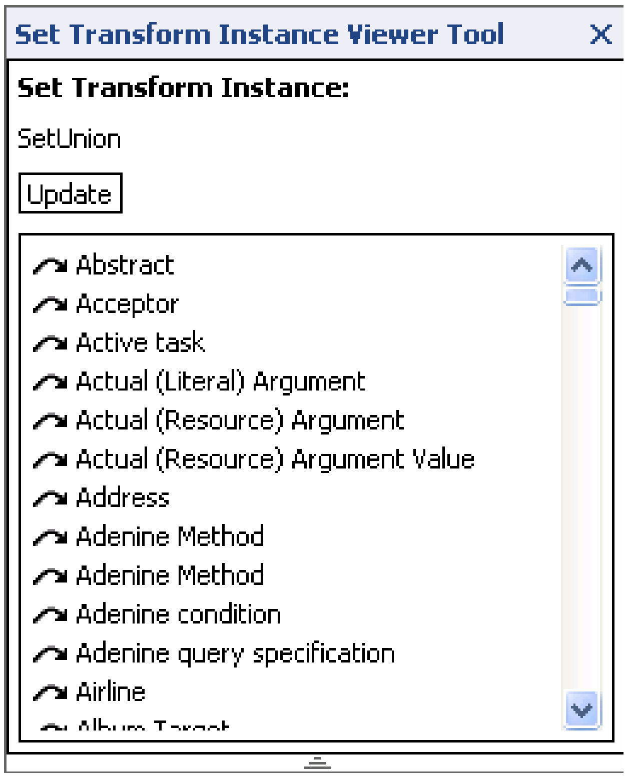
As a result, in a departure from current philosophy in Haystack, where items dragged to the right pane in Haystack are rendered in a particular view, we chose to introduce a different abstraction when developing the channel viewer: a tool. A tool has its own ontology and type, and when added to the right pane, is rendered in *its* applet view. A tool manages some content, and thus can control how to render the content. Another important principle is that the underlying content can be changed, e.g. by drag and drop, thereby removing the problem of overcrowding and lack of replacement semantics in the right pane. Finally, a tool allows having state associated with it or can host multiple pieces of information whose relationships determine the result that is produced and the content that is rendered. For example, a tool that lets users examine how two people are related would allow specifying any two people that can be dragged to the tool, and the tool would then show which set of mutual acquaintances connect them.

#### 4.3.2.4 Set Transform Instance Viewer

The set transform instance viewer is a prototyping tool that allows users to quickly see what results a set transform instance will produce, when they are working with the channel manager to create a new channel. This tool is especially useful when trying to build a query, as the user can check to make sure the different portions of the query are returning the results correctly. Figure 9 depicts the set transform instance viewer tool showing a union of properties of various types (**`daml:ObjectProperty`** and **`daml:DatatypeProperty)`**.

The tool lists the set transform being computed at the top. The update button is provided to allow the user to re-invoke the transform instance (after possibly having made some changes to the store) to ensure that the appropriate results are being produced. Finally, the bottom of the tool shows the set of items produced by invoking the transform instance.

As in the case of the channel viewer tool, the user can drag and drop a set transform instance on the tool, and the tool copies the set transform closure, rendering the copy in the tool independent of the one being used in the channel definition. (Although it would be useful to use the same STI as the one being used in the channel, so that the user need not drag and drop it each time, the STI is copied so that the user does not assume, that reinvoking the set transform will also update the channel. Also, keeping the tools independent allows channels to be deleted, without rendering the STI viewer tool inconsistent.)



SetUnion

Abstract
Acceptor
Active task
Actual (Literal) Argument
Actual (Resource) Argument
Actual (Resource) Argument Value
Address
Adenine Method
Adenine Method
Adenine condition
Adenine query specification
Airline

**Figure 9 Screenshot of Set Transform Instance Viewer Tool**

## *4.4 Implementation*

All code to support the above mentioned functionality was implemented in adenine, split across two adenine files: schemata/InformationChannel.ad contains code for the channels ontology, agent, and implementation of the set transforms and condition tests, and ui/InformationChannelManager.ad contains code for all the UI tools (except the Ontology Browser) discussed above. Below, we discuss the various components, and salient aspects of their implementation.

### 4.4.1 Channel Manager Agent Implementation

The Channel Manger agent is implemented as an Adenine Service that is periodically invoked by the Service Manager. Each time the agent runs, it queries for all channels, and processes each one, updating the ones that are currently enabled.

A channel is updated by reifying a `hs:Collection` (the native Haystack collection abstraction) and populating it. During each update cycle, the collection is only updated with the deltas to the existing set of members: addition of new members, and removal of members no longer satisfying the criteria (as opposed to removing all the old elements and adding all the new elements). This is done in order to minimize disruption to the user in case the channel is being watched by UI components.

Since channels can be used to supply content to multiple other channels, a channel may get updated multiple times: once for itself, and again when used in the context of computing another channel. In order to avoid computing a channel multiple times during the same agent update cycle, the agent employs a boolean flag to determine if a channel has been updated in the current agent invocation. The agent toggles its own flag each time it is invoked. As it processes each channel, if a channel has already been updated in a given invocation (i.e., the channel's update flag matches the agent's flag), it is not recomputed. (Each time a channel is processed by the agent, its flag is also toggled regardless of whether or not it is enabled. Thus, there is no chance that a channel may become enabled and not be updated on the very next update cycle.)

The Channel Manager agent can be viewed as an interpreter for the language consisting of set transforms. Each time it is invoked, it takes the set transform instance for the channel, and invokes it. As we already mentioned earlier, if the **`channel:vectorizedArg`** flag for the argument is set to true, the argument (a set transform instance specified by **`channel:vectorSource`**) is evaluated, i.e. the set transform instance is invoked. Otherwise, the argument is passed directly on to the set transform. We mentioned that vectorization of arguments is implemented by computing a Cartesian Product of the values resulting from invoking the set transform instance with the values of the other arguments. In order to minimize the memory overhead, this Cartesian Product is never pre-computed to create a table of argument value tuples in memory. Instead, each row of the table is computed by cycling through the indices of the various collections of values for the arguments. Thus, the agent trades off additional time for savings in memory.

The code for the channel manager agent is written in Adenine primarily because of speed and flexibility during the development process; it was easy to quickly prototype and evolve the framework for updating channels. However, having converged upon a better understanding of what the agent's tasks are, it seems a java implementation would be better and would gain from improvements in execution speed since the process of generating argument values for vectorized arguments is compute intensive. Also, an object oriented approach would lead to a more modular implementation, with minimal re-querying of agent metadata that could be stored in member variables.

### 4.4.2 Set Transforms and Condition Tests' Implementation

Set Transforms and Condition Tests are very query intensive, and thus implemented as Adenine methods; Adenine provides a compact means of expressing RDF queries and accessing the store, thereby easing development effort. Below, we describe the implementation of each class of set transforms.

The set transforms corresponding to set operations take two arguments, each argument being a set transform instance (closure). The closures are executed, and the results are then combined using the corresponding set operation, and returned.

Query set transforms consist of 3 sets of transforms, depending on whether the return values are in the subject, predicate or object position. As described earlier, all query set transforms take 2 arguments, one corresponding to the value to be used to fix one of the variables in the triple being queried, and the other corresponding to the condition closure that the values corresponding to the non-returned variable will be bound to, and tested against.

The dichotomy of resources and literals in RDF requires additional implementations of queries since conditions are type sensitive. Thus, once a set transform has obtained a set of pairs by querying with just the fixed variable, it must remove all those tuples that have values corresponding to the conditioned variable that do not match the condition test's argument type. For example, if the query is to return all subjects having predicate **`:foo`** and literals matching "bar", then all tuples (?x ?y) that result from the query **`?x :foo ?y`** must be processed to remove those where ?y corresponds to a resource. A similar case exists, if the fixed variable happens to be in the object position, rather than the predicate position. In this case, query transforms are duplicated because the UI needs to construct appropriate widgets based on the range of the parameters being collected. However, the underlying implementation of the query is reused for both the literal and resource version.

As a result of the inflation of primitives, 4 primitives each are needed for queries returning subjects and predicates. Two primitives are needed for queries returning objects (channels can only consist of resources, and thus only resources in the object position are returned).

The current set of conditions tests that have been implemented are binary relational conditions, e.g. equal, greater, etc. All conditions have one free variable that is supplied by the query that is performed; users supply the rest of the arguments. Condition tests are performed by iterating through the values in the 2-tuple and binding the appropriate member to the appropriate argument in the condition test closure. The condition test is then invoked, which then returns a Boolean value corresponding to whether or not the condition was satisfied.

The duplication problem resulting from the resource/literal dichotomy is repeated in the case of condition tests; some conditions such as equality are duplicated since appropriate UI elements need to be constructed to solicit appropriate values from the user. However, even when soliciting just literal arguments from users, additional types within literals (e.g., integer, string, and double) require further duplication of condition tests. A language that supports various types (e.g., object, double, integer, string, etc.) on top of the native RDF would be really useful here since these operator could then be overloaded at the language level, and all these implementations that inflate, can collapse to a single one.

### 4.4.3 Channel Manager Implementation

The UI development for the Channel Manager was also done in Adenine since it provides the Slide ontology for creating GUIs in a simple, declarative fashion, and also because it provides facilities for direct binding of UI elements to data, to create a live and responsive UI using data sources.

Using the View Architecture in Haystack, we have implemented two major views for Channels that allow us to create the channel manager. The first view (**`icm:ChannelInteractiveView`**) allows users to interact with the channel itself, to edit its properties. The second view, (**`icm:ChannelInteractiveContentView`**) allows users to work with the contents of the channel, i.e., the elements that have been computed to be channel members. This view uses a collection view to show the collection of items corresponding to the channel. Finally, one additional view part that overrides the default title view part has been implemented to allow showing a channel's enabled status.

Fundamentally, the channel manager is implemented by browsing to a channel ("Channel's Channel") that maintains the list of all channels, and using the view part class that implements the **`icm:ChannelInteractiveContentView`**. The other view of the channel is then used in the preview pane, to allow users to edit the channel that has been selected from the collection above.

We implemented several context menu and drag and drop operations that aggregate functionality which together allows the channel manager to behave as the central console for working with channels (hence the name). These operations are implemented as appropriately annotated Adenine methods.

The context menu operations that are available from a channel include the ability to toggle the channel's active status, copying the channel and deleting the channel from the store. The ability to create a channel is available by invoking the context menu on the collection of channels, using the standard Haystack mechanism to add items to a collection. Note however, since the collection underlying the channel manager is a channel itself, items that are not channels will be removed the next time the Channel's Channel is updated.

The view part that implements the **`icm:ChannelInteractiveView`** employs standard slide widgets for edit boxes and check boxes. It also embeds view containers for other resource properties, e.g., the target collection, set transforms instances and condition test closures. Custom views have been implemented for the latter two types of entities. In implementing these two views, additional views for their arguments are also implemented. Set transforms and condition leverage the view architecture in Haystack to select the correct argument view part: a resource argument embeds a view container showing the title of the resource, whereas a literal argument shows an edit box. A number of utility UI components were implemented to selectively show/hide portions of arguments (ui/utility.ad): if the argument is vectorized, the underlying **`channel:vectorSource`** is shown, otherwise the **`channel:argumentValue`** is shown.

Drag and drop operations support various capabilities: dragging entities as values for resource arguments, dragging a set transform or condition test (obtained from the corresponding channels – see below) onto a set transform instance or condition test closure to change the underlying target operation and re-initialize appropriate arguments. Finally, although a user can drag a collection to be specified as the new target collection for the channel, he/she should never have to do so (it is only provided for completeness).

As mentioned earlier, all items created in the course of user interaction, e.g., channels, set transform instances, and condition tests are initialized with properties set to default values. This is done by implementing appropriate constructors for each of the types of items, as well as other information objects they depend upon (e.g., arguments.). The Haystack infrastructure has been augmented by adding a method **`util:createDefaultEntity`** that creates an item of a particular type by calling a constructor for that type, annotated with the **`construct:defaultConstructor`** property set to true. Thus, we bypass the current Haystack creation mechanism that simply annotates a resource with a type, title, creator and date, without initializing it with appropriate property values. Resource properties of objects are constructed in a similar manner, and fall back on the Haystack default construction mechanism if a default constructor does not exist for items of the type expected by the property. Similarly, **`util:createDefaultLiteral`** is invoked to create default literal values of particular types, e.g., string, double, boolean, etc. Like the constructors, copy constructor and destructor methods have also been implemented to support the corresponding operations.

We have declared a number of channels that appear by default when Haystack is started consisting of entities of particular types. These channels can be used for constructing other channels and include collections, classes, properties, set transforms and condition tests. The hope is that users will create several instances of channel viewer tools in the right pane, and drag these channels there for ready access when constructing a new channel.

### 4.4.4 Supporting Tools' Implementation

The implementation of the tools supporting the Channel Manager (Channel Viewer Tool, Set Transform Instance Viewer Tool and Ontology Browser) is discussed in this section.

Both the viewer tools were implemented by creating a simple ontology for the tools, instantiating items of those types (using the construction mechanism discussed above) and adding them to the start pane collection (right hand pane). Since items in the right pane are displayed using the **`ozone:AppletViewPart`** view part class, appropriate view parts were implemented and annotated as being of type **`ozone:AppletViewPart`**.

The channel viewer tool has two properties, **`icm:channel`** (points to the channel being viewed) and **`icm:channelItemFocus`** (points to the currently selected item in the channel's collection). A drag and drop operation is implemented that replaces the underlying channel with a newly specified channel. The channel's target collection is displayed in the list view in the tool, followed by a description of the currently selected item in the list. Although the haystack collection views could be used to visualize the items in the channel, they would either provide too much power (e.g., a list of items with preview pane), or not enough (e.g., a list of items that cannot be selected or scrolled if the UI space available is limited). Thus, the simplest approach was to directly show the members of the collection. Defining another collection view did not make sense, as the semantics of the tool (just show a description of the selected item) would customize the implementation such that the new view for collections would probably not be used anywhere else.

The set transform instance viewer tool also has two properties, **`channel:setTransformInstance`** (points to the set transform instance whose results are being viewed), **`icm:setTransformInstanceResults`** (points to a **`hs:Collection`** containing the results of invoking the set transform instance). The collection pointed to by the second predicate is local to the tool instance. A drag and drop operation allows specifying a new set transform instance, whose description is copied, and pointed to by the **`channel:setTransformInstance`** predicate. Clicking on the "Update" button, invokes an adenine method that executes the set transform instance, and updates the collection pointed to by **`icm:setTransformInstanceResults`**. This collection is displayed in the list view on the tool. A haystack collection view was not reused for reasons similar to those above.

The Ontology Browser is implemented as a slide in Adenine (ui/OntologyBrowser.ad). It consists of two main portions, the top set of selector panes and the preview pane. The left selector pane displaying a list of ontologies has a datasource that tracks which element is currently selected. This list of classes and properties in the right selector pane is queried based on the ontology specified by this datasource. The preview pane is also split into two regions. The top region shows summary information about the currently selected ontology (using the datasource specified above), and the bottom pane shows information about the class or property selected in the right selector pane (also tracked by a datasource). Appropriate view parts were implemented to show summary information for ontologies, as well as detailed information on classes and properties.

## 4.5 Conclusion

In this chapter we have tackled the problem of weak (or lack of) control over content specification (regardless of domain). We have accomplished this by building a set of tools and adding to the Haystack infrastructure in a way that lets users express and manipulate information of interest. The notion of channels allows users to work with a unit of content in closed form, without listing the actual information entities. A set of primitives allow users great flexibility in building channels, and thus in controlling which subset of information entities in the store are of interest. Furthermore, an agent that updates channels, allows the content to always remain current. Finally, a set of UI tools allows users to build channels and view them. Together with the ability to identify single items of interest by explicitly specifying them, we now have a means for users to specify content in various different forms: single items vs. collections, and explicitly vs. implicitly.

# Chapter 5 Information Spaces

In the previous chapter, we discussed the notion of channels as a unit of content that the user can specify, and tools that support it in Haystack. Users can now specify information of interest as either single items or collection both explicitly and implicitly. Next, we require a means for users to be able to aggregate arbitrary content onto a single screen to define an *information space* – a console of related information and tools pertinent to a particular task. This chapter is devoted to understanding the infrastructure and tools added to Haystack to address this need.

## 5.1 Basic Capabilities

Before discussing the detailed design and implementation of information spaces in Haystack, we take a brief look at some of the basic capabilities such spaces should support:

- **Information Aggregation** – The information space should allow aggregation of various bits of information by allowing users to position them and allocate appropriate space in various parts of the information space.
- **Customize Information Space** – Users should be able to customize not just what is shown, and where it is shown, but also how it is shown, e.g. cosmetic attributes such as color, borders, etc., as well the views used for information entities. Also, users should be able to collect frequently used operations for tasks in information spaces.
- **Design and Usage Mode** – An information space should support two modes: a design mode that lets users customize the information space, and a usage mode that allows them to work with it. Furthermore, users should be able to easily switch modes, e.g., switching from usage mode to design mode to change some customization attribute, and then back to usage mode.
- **Collection of Information Spaces** – Users should be able to create and manage a collection of information spaces that correspond to their various tasks.

Furthermore, they should be able to copy and modify existing spaces to quickly create new ones.

- **Persistent Information Spaces** – Users should be able to return to information spaces and see updated information.

## *5.2 Design*

Support for information spaces in Haystack can be thought of as consisting of two primary components: information spaces and their views, and an information spaces manager. We discuss each in turn below.

### 5.2.1 Information Space

Any given information space in Haystack is supported by a set of interrelated objects, their ontologies and their associated views. We first briefly consider the underlying ontologies to see how they support the views. Then, we discuss the two views of information spaces (corresponding to design mode and usage mode) and their related design decisions.

#### 5.2.1.1 Underlying Ontologies

Three ontologies underlie information spaces: Information Space Ontology, Two Dimensional Layout Ontology and Information Portal Ontology.

The Information Space Ontology declares a class named `is:InformationSpace`, used to specify the class type of information spaces. In addition to the title and description associated with most entities in Haystack, the class has one other property: `is:informationSpaceLayout`, which points to an entity with type `tdl:TwoDimensionalLayout`, which captures the two dimensional layout of information portals in the space. An information space can have a collection of information portals laid out in some fashion, each one showing a particular entity or channel of information.

The `tdl:TwoDimensionalLayout` class is declared in the Two Dimensional Layout Ontology and recursively captures the layout of items in two dimensions, by splitting the space available to it among its children (also of type `tdl:TwoDimensionalLayout`). It has three main properties: `tdl:children`, `tdl:orientation`, `tdl:cellData`. The `tdl:children` property specifies a list of children of type `tdl:TwoDimensionalLayout`. The `tdl:orientation` property specifies whether the children are to be grouped into rows, or columns. A collection of nested layout entities yields a tree of items of type `tdl:TwoDimensionalLayout`. Only the leaf nodes of this tree are rendered as “cells” for the user. The remaining nodes are used to determine where to render the leaf nodes in the corresponding sub-tree. Finally, `tdl:cellData` specifies the content of the cells, i.e. the data that is to be laid out and managed by the leaf nodes. The cell data in the case of information spaces are information portals having type `ip:InformationPortal`. Thus, all non-leaf nodes have children, and a corresponding orientation. All leaf nodes have an associated entity as the cell data, and no children.

The two dimensional layout ontology was created to support a reusable, flexible layout scheme that would not force the user into two dimensional grid, with a fixed set of rows and columns. Thus, each node in the tree above can change the orientation for its children, yielding maximum flexibility in layout of cells.

An information portal has a title property, in addition to the following three properties: `ip:informationPortalContentSpecification`, `ip:informationPortalPresentationSpecification` and `ip:informationPortalManipulationSpecification`, pointing to entities of type `ip:InformationPortalContentSpecification`, `ip:InformationPortalPresentationSpecification` and `ip:InformationPortalManipulationSpecification` respectively .(URIs are case sensitive, and thus the same string in different cases may represent both the property name, as well as the class name). These classes capture the content, presentation and manipulation preferences of users. The class `ip:InformationPortalContentSpecification` has two properties (`ip:underlyingContent`, `ip:showUnderlyingChannelContent`), which capture the underlying content, and if relevant, whether or not the channel's contents are to be shown. The class `ip:InformationPortalPresentationSpecification` has four properties, `ip:showBorder`, `ip:showTitle`, `ip:channelItemsScrollable`, `ip:viewPartClassToUse`. The first two properties allow the user to select whether or not the portal is to be rendered with a border or title. The third property allows the user to specify whether the list of items of a channel (if shown) should be scrollable. The user may specify the view to use in displaying the underlying entity or all the entities in the channel's collection via the `ip:viewPartClassToUse` property. Finally, the `ip:InformationPortalManipulationSpecification` has a single property, `ip:availableOperations` that points to a list of operations the user would like to associate, and have readily available, with the information portal.

#### 5.2.1.2 Information Space Designer View

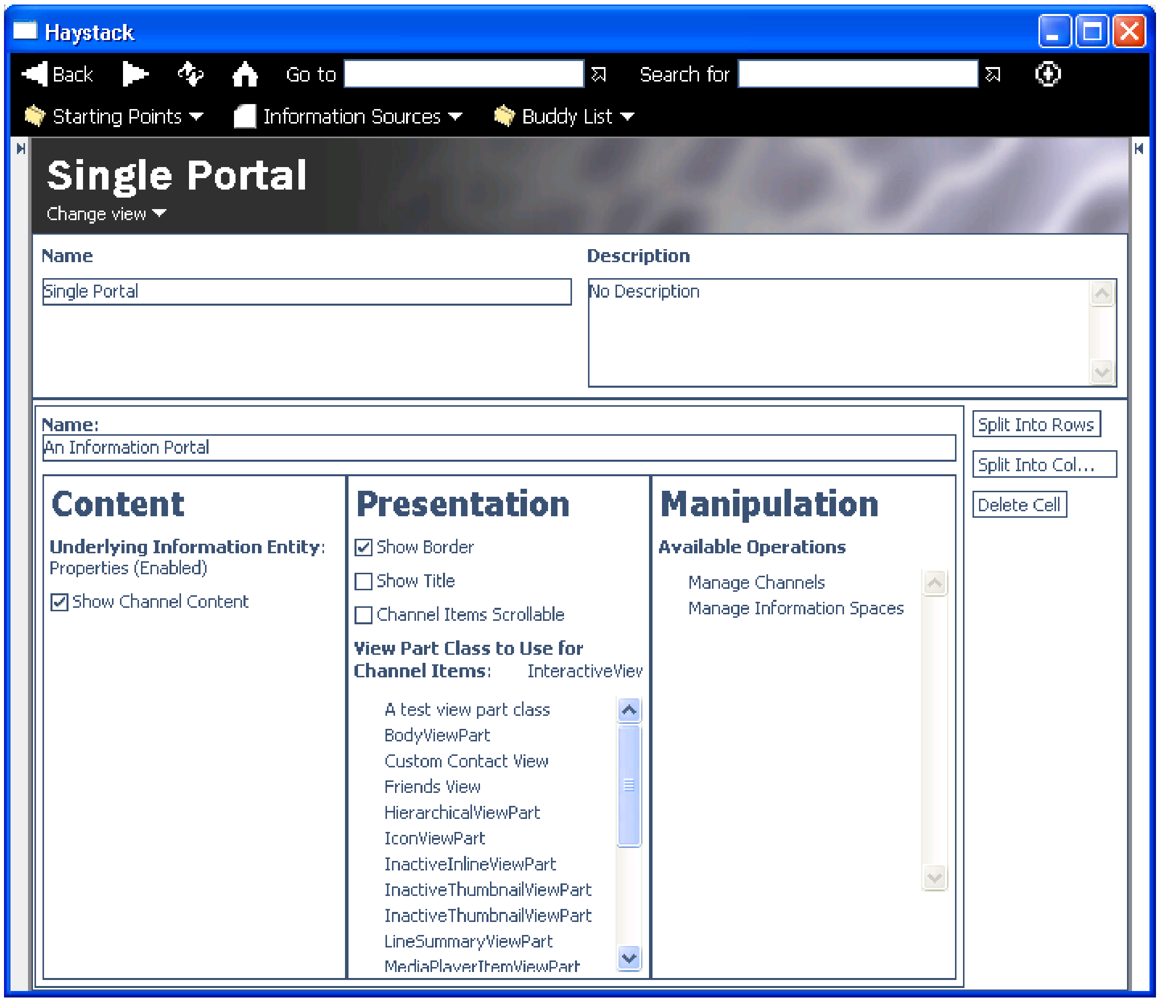
Information spaces can be designed by users via an interactive designer view (shown in Figure 10) that allows users to customize various aspects of the space. A new space is initialized with a single information portal.

The designer consists of two main sections: the top pane that allows editing the title, and description, and the bottom pane, that allows users to control what will be available when they are working with the information space.

The bottom pane consists of an information portal. An information portal allows users to view some information entity, or collection of entities specified using a channel. The left hand portion of the portal allows users to customize various properties of the portal. It allows users to enter the title for the information portal. Below the title, the user may customize the portal using three sub-panes corresponding to the content, presentation and manipulation preferences for the portal. The right hand portion of the portal allows users to further split the space occupied by the portal into additional portals using the various buttons.

The content portion of the portal pane allows users to specify which entity is to be shown in that portal (underlying entity) using a drag and drop operation. Since any item can be dragged to the content sub-pane, including channels, users must disambiguate whether the channel itself is to be viewed, or the members that comprise its contents. This is accomplished by checking the "Show Channel Content" check box appropriately, which only becomes available if the underlying entity is a Channel.

The Presentation sub-pane allows users to specify how the portal, as well as its contents, should look. Thus, using the appropriate check boxes, users may specify whether the portal is to show a border and title when the user is in usage mode.  The remaining presentation attributes correspond to the underlying entity for the portal, and are predicated on the type of the content being shown.


Haystack

Single Portal

Name: Single Portal

Description: No Description

Name: An Information Portal

**Content**

Underlying Information Entity: Properties (Enabled)

☑ Show Channel Content

**Presentation**

☑ Show Border

☐ Show Title

☐ Channel Items Scrollable

View Part Class to Use for Channel Items: InteractiveViev

A test view part class
BodyViewPart
Custom Contact View
Friends View
HierarchicalViewPart
IconViewPart
InactiveInlineViewPart
InactiveThumbnailViewPart
InactiveThumbnailViewPart
LineSummaryViewPart
MediaPlayerItemViewPart

**Manipulation**

Available Operations

Manage Channels
Manage Information Spaces

**Figure 10 Screenshot of Information Space Designer**

If it is the case that the members of a channel are being shown (i.e., the user has specified a channel as the underlying information entity, and checked the "Show Channel Contents"), then the "Channel Items Scrollable" check box becomes available and allows users to specify that they want a layout for the channel collection that has a scrollbar.

(Haystack also allows a layout showing a stacked set of items, which must be expanded to show additional items.) Users may specify which view to use for the items in the channel collection being displayed by the portal. This can be specified by dragging and dropping a view from the list of views that Haystack deems are applicable. Since the channel collection may be some (future unknown) heterogeneous mix of elements, the list of views only consists of views available for **`daml:Thing`**.

If the portal is not showing the elements of a channel, then the "Channel Items Scrollable" check box is not available. In this case, since the type of the underlying entity is known, the list of views from which the user may select, consists of a union of all views available for **`daml:Thing`**, as well as the views available for items having the same type as the underlying entity.

The last sub-pane available to the user for customizing the portal allows her the ability to specify a collection of operations she believes are useful to have handy when working with the information in the portal. In Haystack, all operations have URIs and are first class entities. That is, they are treated the same way as all other entities having a URI; they dragged into the collection of relevant operations, just as any item in Haystack.

The right hand portion of the portal pane allows users to click on buttons labeled "Split into Rows", "Split into Columns" and "Delete Cell" to appropriately segment the space into additional portals, or to remove portals. Any previous specifications for the portal (e.g., underlying entity, presentation preferences, etc.) are lost during this operation. The root portal (the single portal taking up the entire information space real estate) cannot be deleted.

Several salient features of the current design are noteworthy. We expect users will create information spaces containing several portals. One way users may reduce the resulting clutter in the interface is to use the Haystack navigate facility, to navigate to the underlying portal. All items that are navigated to are shown in the main Haystack pane. Thus, the portal gets allocated the maximum amount of space allowed by Haystack. A less drastic measure allows users to resize portals, or their sub-panes to focus on areas of interest. In the process, other portals or sub-panes reduce their size appropriately to cede space to the portion of the UI the user is interested in.

Since users are designing an information space, it would be useful for them to be able to see how the final space looks and behaves when being used. This can be accomplished by selecting the "Information Space Usage View" (see section 5.2.1.3) from the pull down menu labeled "Change view" located below the information space title.

Another important feature of the current information space design allows users to leverage a built-in capability in Haystack to specialize operations in order to customize the manipulation capabilities of an information portal. In general, operations that the user may invoke, may solicit him for values for arguments in order to carry it out (e.g., the "send e-mail" operation requires a recipient and a message at the very least). *Currying* is a technique in Haystack that allows users to specify some or all of the arguments, and save the resulting partially (or completely) specified operation closure as a new operation that, when invoked, will ask the user for any remaining argument values. Thus, a user may partially specify an operation that he feels will always require the same values for some of the arguments, and save it as a new operation. For example, the user may specify that the recipient for the "send e-mail" operation should always be "David Karger," and save this partial specification as a new "mail to David" operation. Thus, when the new operation is invoked, it will only ask the user for the body of the message, and always send the message to "David Karger." This operation, like all other operations, can then be associated with a particular information portal, for which the previously specified arguments make sense (e.g., in customizing a software bug tracking portal, an operation known as "notify the project manager" may be associated with the portal showing the bugs, which will always send a message to "David Karger"). Thus, the user may customize the manipulation capabilities of a portal by not just specifying which operations should be available in a particular portal, but also by pre-initializing some of its arguments with values corresponding to what makes sense in the context of the portal or information space being designed.

Several important design decisions were made in constructing the information space designer, and are discussed below.

Perhaps the most important design decision that affected the construction of information spaces by users was how to allow users to allocate space to portals. Clearly, several different methods of allocating space are possible. For example, automated layout of the information could be done using various heuristics as well as preferences. Such a mechanism could take into account not just the content to be shown, but also its view, and then algorithmically develop an optimal allocation, based on the size of the views for each of the information entities. However, such an approach would not be in the spirit of allowing users control over their information space. Furthermore, we argue, that "information visualization ergonomics", i.e., the set of decisions that a user makes for how to layout information based on frequency of use, order of use, relationship to each other, etc. would suffer. As a result, following in the footsteps of Delaunay$^{MM}$ and *Yahoo!*, we chose to allow users to explicitly specify the layout of information [15,17].

Even if information layout is explicitly specified by users, various possibilities exist for how users allocate the space. For example, users may associate information entities with various arbitrary places on the underlying canvas, much as a person pins something to a bulletin board. This approach has the benefit of allowing users to move the items' relative to each other, e.g., move one above the other, quite simply, without having to re-design the space. However, this alternative has some drawbacks: space may be wasted between items pinned to the board, and users must constantly resize or move elements to see their contents, and possibly resize/move others to make space. Also, a bulletin board metaphor makes more sense when a user needs to juxtapose information; when the relationship between information is more important than the information itself.

Given our understanding of the portal creation task from *Yahoo!*, and the ideas espoused by the QuickSpace project, it seemed that the goal of user controlled content layout was to allow users access to a canvas that is appropriately segmented and *completely* devoted to showing the content of interest, with the user being able to specify the location of items, and possibly their size directly or indirectly, e.g. by placing it in a narrow column in the *Yahoo!* portal as opposed to the wide column [12, 15]. Such an approach allowed users to allocate maximal space to the content of interest. Furthermore, it allowed the space to adapt to a local change: if a user increases the amount of space for one item, the space for other items *automatically* decreased without overlapping – an important feature desired by users, as demonstrated by QuickSpace.

We hypothesized that once users have worked on a task several times, they have a good understanding of what is needed, and what the optimal layout of information is, that maximizes their efficiency. Thus, changes to the layout of an information space should be rare, and hence the ability to move portals relative to each other will not be important in the long run. The two properties above should be more important. The other important consideration in our design was simplicity of creation of a new information space. As in various other parts of our implementation, we chose to make things work immediately, without requiring significant user configuration. In this case, users should quickly be able to segment the entire space, rather than be forced to position and size an information portal. Consequently, we designed our information spaces to capture the desirable features of *Yahoo!* and QuickSpace, both in the Design view as well as Usage view.

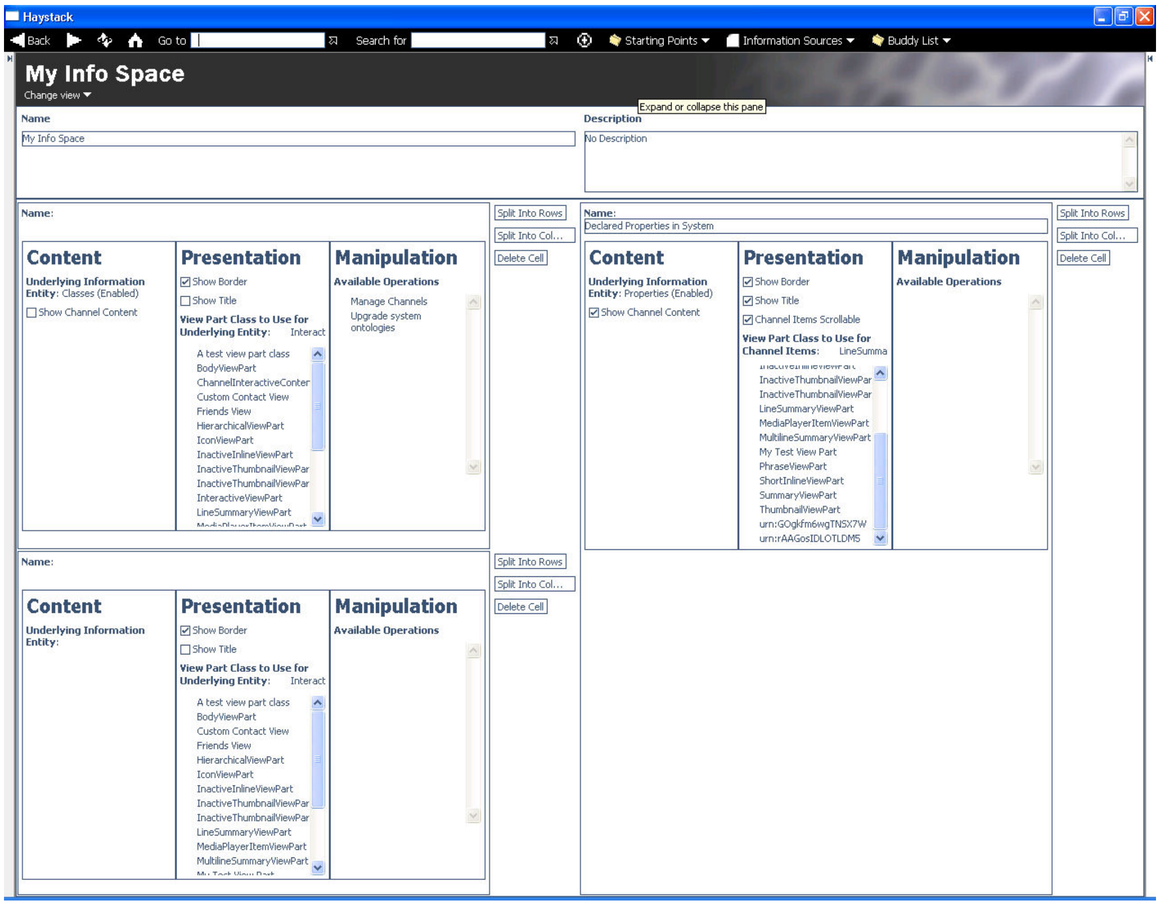
The decision to only allow the user to split cells into two rows or columns was another important design decision. Clearly, the underlying layout ontology could easily support more powerful layouts, based on an arbitrary number of children, rather than the two children semantics of the current splitting operations. However, we felt it was important to keep the operation of segmenting the usable space as simple and efficient ("one-click") as possible. Since portals could be resized to create the look of a non-binary tree driven layout, nothing was lost in terms of capability in segmenting the space.

Similar to other tools, another design decision was to only show the user those customization tools that are relevant based on his/her current settings. Thus, the UI is dynamic in that it shows and hides various UI components based on whether or not they make sense in the current context, e.g., the "Show Channel Content" check box is not visible if the underlying entity for the portal is not a channel.

One of the design decisions was made based partially on constraints due to current means of development of UIs in Haystack, and partially on future functionality we would like to support. The "Show Channel Content" check box is not strictly necessary. It is quite possible to specify that the elements of a channel are to be shown by specifying an appropriate view for the channel (e.g., **`icm:ChannelInteractiveContentView)`** that lists the elements of the channel. If a preview pane that accompanies this view is not desired, in favor of a simple list of items in the channel, then a new view for channels could be made available to the user. However, use of views for channels in this case creates two problems, and thus necessitates the check box. First, having a view for the channel, that showed the items in the channel, would deprive the user of the ability to specify how to show the *individual* items in the channel, since that would be statically specified by the developer of the view. Second, in the future, we would like to be able to support "wiring" of the user interface, i.e. the ability to allow parts of the user interface to control others, e.g. if one item in a collection is selected, a more detailed view of it should appear in a different portion of the information space. Such functionality would require the ability to expose for programmatic and user access, which item is currently selected in the collection. Due to the way the view architecture has been designed in Haystack, currently it is not possible to expose such properties of the view, i.e., a view for a channel showing its members would not be able to expose the currently selected item. Such functionality can only be exposed by directly using UI widgets to access the collection, without a level of indirection imposed by a view that maintains such a collection. Thus, if the check box is set to true, it allows the information space framework to know to use the UI widget for showing collections directly, so that it's "currently selected item" property can be made available in the future.

### 5.2.1.3 Information Space Usage View

The Information Space Usage View is another view of an information space, and is utilized  to let users actually work with the information in the information space, rather then specify their preferences. Figure 11 and Figure 12 depict both the design view and a usage view of an information space.



Name: My Info Space

Description: No Description

Name:

Content

Underlying Information Entity: Classes (Enabled)

☐ Show Channel Content

Presentation

☑ Show Border

☐ Show Title

View Part Class to Use for Underlying Entity: Interact

A test view part class
BodyViewPart
ChannelInteractiveConter
Custom Contact View
Friends View
HierarchicalViewPart
IconViewPart
InactiveInlineViewPart
InactiveThumbnailViewPar
InactiveThumbnailViewPar
InteractiveViewPart
LineSummaryViewPart

Manipulation

Available Operations

Manage Channels
Upgrade system ontologies

Name: Declared Properties in System

Content

Underlying Information Entity: Properties (Enabled)

☑ Show Channel Content

Presentation

☑ Show Border

☑ Show Title

☑ Channel Items Scrollable

View Part Class to Use for Channel Items: LineSumma

InactiveThumbnailViewPar
InactiveThumbnailViewPar
LineSummaryViewPart
MediaPlayerItemViewPart
MultilineSummaryViewPart
My Test View Part
PhraseViewPart
ShortInlineViewPart
SummaryViewPart
ThumbnailViewPart
urn:GOgkfm6wgTNSX7W
urn:rAAGosIDLOTLDM5

Manipulation

Available Operations

Name:

Content

Underlying Information Entity:

Presentation

☑ Show Border

☐ Show Title

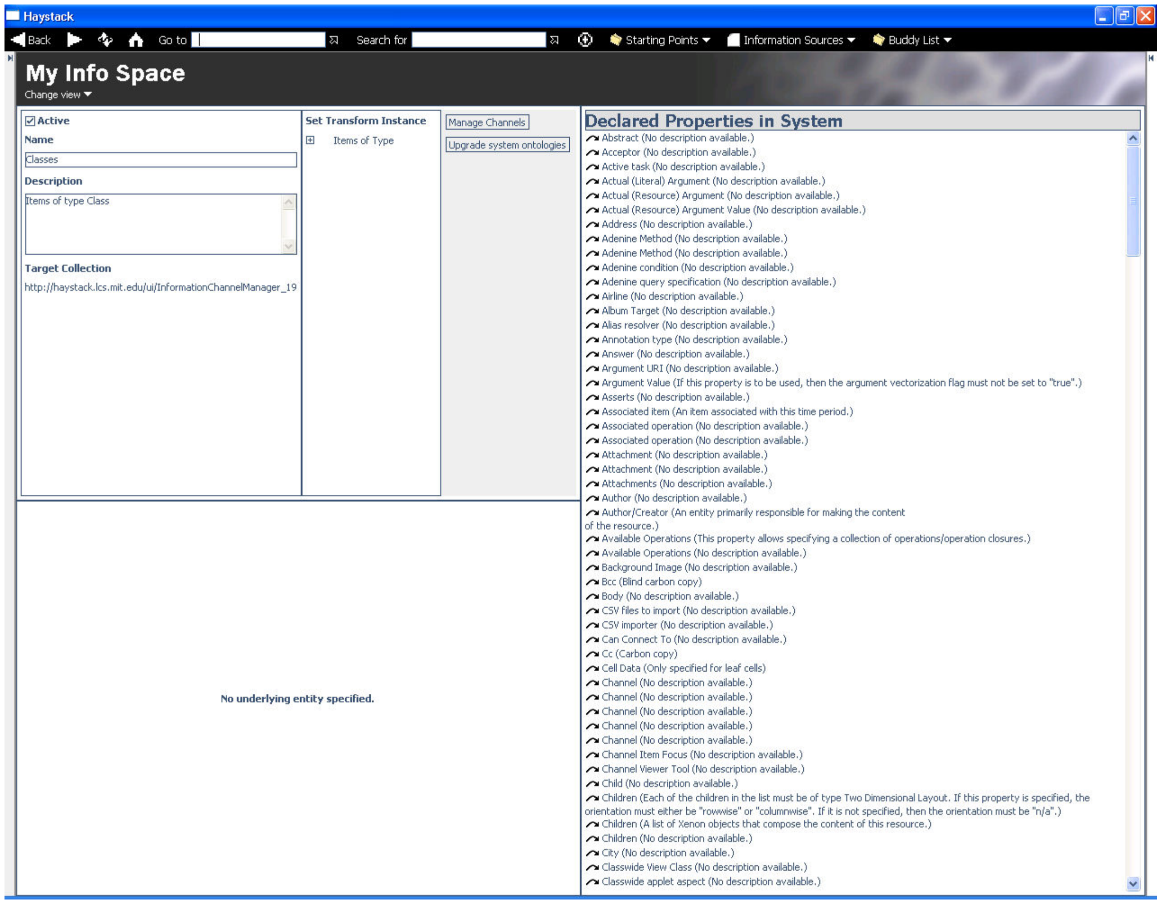
View Part Class to Use for Underlying Entity: Interact

A test view part class
BodyViewPart
Custom Contact View
Friends View
HierarchicalViewPart
IconViewPart
InactiveInlineViewPart
InactiveThumbnailViewPar
InactiveThumbnailViewPar
LineSummaryViewPart
MediaPlayerItemViewPart
MultilineSummaryViewPart

Manipulation

Available Operations

**Figure 11 Design view of the "My Info Space" Information Space**



My Info Space

☑ Active

Name

Classes

Description

Items of type Class

Target Collection

http://haystack.lcs.mit.edu/ui/InformationChannelManager_19

Set Transform Instance

⊞ Items of Type

No underlying entity specified.

Declared Properties in System

- Abstract (No description available.)
- Acceptor (No description available.)
- Active task (No description available.)
- Actual (Literal) Argument (No description available.)
- Actual (Resource) Argument (No description available.)
- Actual (Resource) Argument Value (No description available.)
- Address (No description available.)
- Adenine Method (No description available.)
- Adenine Method (No description available.)
- Adenine condition (No description available.)
- Adenine query specification (No description available.)
- Airline (No description available.)
- Album Target (No description available.)
- Alias resolver (No description available.)
- Annotation type (No description available.)
- Answer (No description available.)
- Argument URI (No description available.)
- Argument Value (If this property is to be used, then the argument vectorization flag must not be set to "true".)
- Asserts (No description available.)
- Associated item (An item associated with this time period.)
- Associated operation (No description available.)
- Associated operation (No description available.)
- Attachment (No description available.)
- Attachment (No description available.)
- Attachments (No description available.)
- Author (No description available.)
- Author/Creator (An entity primarily responsible for making the content of the resource.)
- Available Operations (This property allows specifying a collection of operations/operation closures.)
- Available Operations (No description available.)
- Background Image (No description available.)
- Bcc (Blind carbon copy)
- Body (No description available.)
- CSV files to import (No description available.)
- CSV importer (No description available.)
- Can Connect To (No description available.)
- Cc (Carbon copy)
- Cell Data (Only specified for leaf cells)
- Channel (No description available.)
- Channel (No description available.)
- Channel (No description available.)
- Channel (No description available.)
- Channel (No description available.)
- Channel Item Focus (No description available.)
- Channel Viewer Tool (No description available.)
- Child (No description available.)
- Children (Each of the children in the list must be of type Two Dimensional Layout. If this property is specified, the orientation must either be "rowwise" or "columnwise". If it is not specified, then the orientation must be "n/a".)
- Children (A list of Xenon objects that compose the content of this resource.)
- Children (No description available.)
- City (No description available.)
- Classwide View Class (No description available.)
- Classwide applet aspect (No description available.)

**Figure 12 Usage view of the "My Info Space" Information Space**

The title of the information space is displayed at the top. As can be seen, the layout of information in the usage view mimics that of the design view. Each portal is rendered according to the specifications in the design view, with respect to showing borders, titles, views, operations, etc. (If no underlying entity is specified, a default pane is shown that informs the user of missing information.) Each portal consists of three main parts: the title bar, the content pane, and the operations pane. The title bar shows the title of the portal as specified by the user, if it is to be displayed. The space below the title bar is split into a left and right pane. The left pane shows the contents of the portal according to the view that was specified in the design mode. The right pane shows the operations the user chose to make accessible.

As in the design mode, users can resize the various portals as needed. The entire user interface of Haystack is constructed by nesting views of entities, and the information presented in the information portals is no exception. Hence, in any information portal, users can manipulate information as they would in any other part of Haystack; users have access to context menus, and drag and drop capabilities, in addition to the affordances provided by various UI widgets, e.g. text fields, check boxes, etc. Furthermore, users can invoke the operations that they chose to make available in certain portals, by clicking on the appropriate buttons.

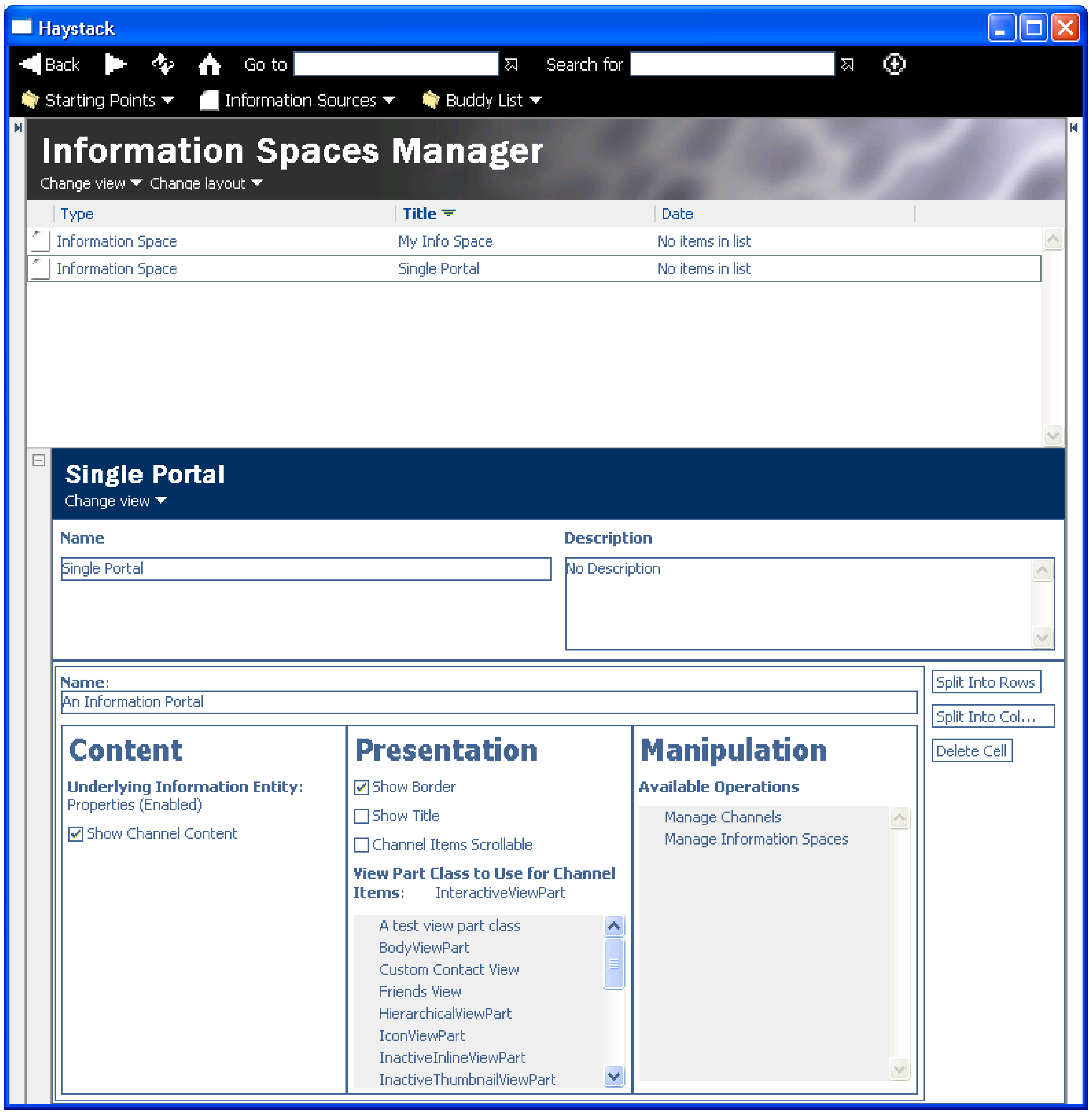
One particular design decision made in the development of the usage view for information spaces and portals was the use of buttons in the right pane of a portal, to show the operations the user had selected to be available in that context. It could be argued that, in keeping with the Haystack philosophy of context sensitive operations, these operations should have been added to the context menu of the items to which they apply, perhaps in exclusion to any other operations.

We argue that such an approach would be detrimental to the user. Operations on context menus are two or more clicks away. The user must first invoke the context menu, select the right context (since there may be multiple underlying entities) and then navigate one or more sub-menus before being able to access operations. This is fundamentally antithetical to the reason why users want to co-locate relevant operations in the first place: convenience of access. We argue that they should be "projected" onto the user by being made visible in a prominent manner, in much the same way that MS-Word "projects" frequently used operations via toolbars, or by hiding other operations from menus.

Another design decision that was made in developing the information space usage view was to arbitrarily select some styles for the information portal itself, e.g., a border width of 1, a title bar at the top having a fixed color and font style, etc. One reason for this was due to the fact, that Haystack currently implements graphics styles as a bag of properties, inherited by the children of a view, as opposed to properties of objects such as text, lines, etc. in the object-oriented sense. In fact, Haystack has no notion of objects of type text or line that have properties of color, font or thickness. Thus, the graphics styles cannot easily be made uniformly available without significant custom coding to collect these properties, and use them appropriately for rendering. Also, were they to be made available for portals, they would constitute a special case since the parents and grandparents of portals would also need to support this capability for uniformity, as would all views in Haystack. In order to support the functionality above, the UI rendering engine would need to support a notion of interoperating objects with particular views, so that users could specify the various properties of the objects. Alternatively, the user interface engine would need to support a rendering pipeline that takes an object and renders it with user-specified optional "decorations", e.g., borders, titles, etc. Neither possibility is currently supported by the Haystack UI rendering engine.

### 5.2.2 Information Spaces Manager

The Information Spaces Manager (shown below) is designed in the same manner as the Channel Manager, and supports similar functionality for working with all information spaces from a single user interface.



| Type | Title | Date |
|---|---|---|
| Information Space | My Info Space | No items in list |
| Information Space | Single Portal | No items in list |

Name: Single Portal
Description: No Description

Name: An Information Portal

Underlying Information Entity: Properties (Enabled)
☑ Show Channel Content

☑ Show Border
☐ Show Title
☐ Channel Items Scrollable
View Part Class to Use for Channel Items: InteractiveViewPart

A test view part class
BodyViewPart
Custom Contact View
Friends View
HierarchicalViewPart
IconViewPart
InactiveInlineViewPart
InactiveThumbnailViewPart

Available Operations
Manage Channels
Manage Information Spaces

**Figure 13 Screenshot of Information Spaces Manager**

An Information Spaces Channel is defined, and is viewed using the icm:ChannelInteractiveContentView view for channels, which simply wraps a collection view that is showing the current collection that underlies the channel. Thus, a collection of information spaces are available, and can be previewed and interacted with using either the design or usage view. Similar to the Channel Manager, various operations are made available for information spaces via context menus. These operations include the ability to add information spaces using the “Add new item” operation from the collection context. Also, any given information space can be copied or deleted by right clicking on it, and selecting the appropriate operation from the context menu.

## *5.3 Implementation*

In this section, we discuss the implementation of both the design and usage information space views. We forego discussing the Information Spaces Manager implementation as it is the same as the Channel Manager implementation. The relevant files consist of three pairs of files corresponding to the three ontologies, and their views. These files are, schemata/InformationSpace.ad, ui/InformationSpace.ad, schemata/TwoDimensionalLayout.ad, ui/TwoDimensionalLayoutPart.ad, schemata/InformationPortal.ad, ui/InformationPortal.ad.

Both the design and usage views for information spaces are implemented in a similar fashion. In both views, the portals in the space are laid out in two dimensions, by embedding a view (TwoDimensionalLayoutView) for the tdl:TwoDimensionalLayout entity captured by the is:informationSpaceLayout predicate. Similar to information spaces, information portals also implement two views, one for design mode, and one for usage mode. The appropriate view to use for portals when laying them out is specified by the information space by registering a context property for use by the TwoDimensionalLayoutView.

The TwoDimensionalLayoutView serves as the layout engine and deserves further elucidation. Given an entity of type tdl:TwoDimensionalLayout, it checks to see if the entity has any children. If it has children, the entity is rendered by laying out its children either in rows or columns, as specified by the orientation. If it does not have any children (i.e., is a leaf node), the view to be used to render the cell is determined by the context property that was registered by the information space. This view can either be a TwoDimensionalLayoutCellEditorView or TwoDimensionalLayoutCellUsageView.

The former view consists of two portions: a left hand pane showing the cell data with an editor view, and a right hand pane that allows users to edit the layout tree, by providing buttons that allow further splitting or deleting the cell. The cell data is embedded in a view container using a fixed view type that allows editing preferences for the information to be laid out in two dimensions, i.e., it requires the cell data to implement an editor/designer view. (In the discussion of the information space designer earlier, we stated that each portal consisted of the title, and three sub-panes, *as well as* the buttons for splitting and deleting the portal for simplicity of description; in reality, the buttons are provided by the view for the underlying leaf node of the layout tree, which further embeds the information portal in the left hand pane.)

The TwoDimensionalLayoutCellUsageView simply embeds the cell data in a view container using a view that simply shows the information, as it is to be presented, i.e., it requires the cell data to implement a usage view.

Since both a designer and usage view are available for them, information portals can be rendered appropriately by the two dimensional layout manager, as dictated by whether the information space hosting the layout manager is being shown in design or usage view.

We briefly discuss the implementation of the design view for information portals (since the usage view is driven by the view the user chooses for the underlying entity). Simply, the design view for an information portal, embeds three views corresponding to the content, presentation and manipulation specifications (see ontologies) associated with the portal. In fact, the ontological separation of information portal properties into these three categories was done not just to better organize the interface, but also because drag and drop operations in Haystack are context sensitive. Since any item can be specified as the underlying entity for an information portal, drag and drop directly onto an information portal entity would have prevented use of drag and drop for other user operations, such as specifying the desired view, or operations of interest. Thus, the use of three sets of properties avoided this problem since they could be declared as separate drop targets, rather than the single parent information portal. The content and presentation sub-panes employ some reusable utility code for showing and hiding user interface components based on a condition specified by a data source.

## 5.4 Conclusion

In this chapter, we have addressed the problem of giving users the ability to build a single task based interface. As a result, we have provided tools in Haystack that users can employ to segment a canvas into multiple portals that aggregate relevant information (single entities or channels) for the task. Furthermore, users can also specify how to view the information in the task interface by reusing views in their own contexts. Finally, users may specify or customize operations that are relevant to the task such that they are available in a convenient manner. Thus, users attain the ability to *create* a task interface simply by aggregating the relevant resources in a personally useful manner.

# Chapter 6 User Creatable Views

In the previous chapter, we presented methods for users to control the content, the views to display the content, as well as how the content was laid out. Even with this significant control over the presentation of information, a critical component was missing: users were required to re-use the views *already created by developers*. In customizing an interface for a task, much can be gained in user efficiency by allowing him/her to directly tailor the view of the information itself (rather than select from a set of views) to suit personal preferences. In this chapter, we discuss this idea further, present a design pattern to support it in various domains within Haystack, and discuss a simple capability we developed to customize views of all entities, regardless of domain.

## 6.1 View Designers

When working with information, it is critical that users be able to customize views not just in terms of cosmetic properties (e.g., color and font) or pre-determined/fixed, domain-specific properties (e.g., sorted e-mails by sender) deemed to be useful by the developer, but also by controlling which set of properties are accessible and how they can be visualized and manipulated in conformance with the semantics of the underlying information entity.

Not everyone is interested in all facets of the information being used. For example, in the case of contact information for friends, even though the user may have information on the job title and place of work of a friend, she may not be interested in seeing it listed in her address book; she may only want the phone number and e-mail address listed. Thus, the user may want to specify a particular facet of the underlying information to work with. This idea can be employed, for example, to keep information synchronized, where different parties view and/or modify different facets of the information in a manner similar to database views, e.g., the software project manager can update just the skill set of her team member, while the HR department may focus just on keeping the team member's employment status updated.

Given the same information, not everyone manipulates it or visualizes it in the same way either. For example, different employees in an enterprise may access the same information, but only one may have the ability to modify it. Similarly, a person in the accounting department may be interested in the spread sheet view of some sales figures, whereas a higher level executive would prefer a chart based on this data.

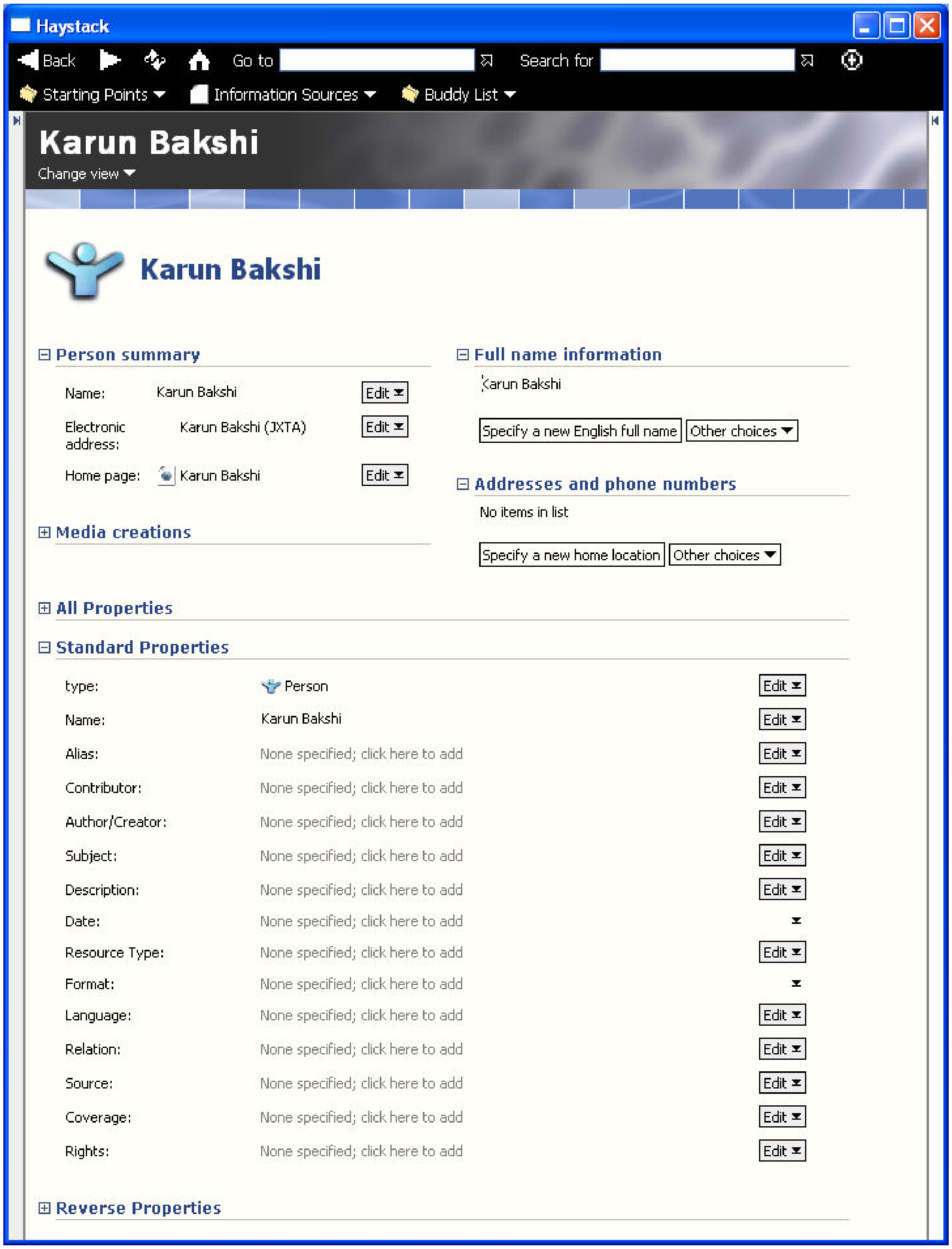
The ideas regarding fine grained control over information properties and presentation used in a view of the information are not new. For example, Microsoft Outlook allows users to select fields of interest for items in the contacts list [9]. In addition, to allowing the user to select fields, it allows the user to control the visualization by specifying the order of the fields and font/color preferences for contacts. This could easily be extended to making certain fields editable vs. read-only.

Similar notions exist in Haystack; Aspects are a means of inspecting an information entity [21]. One type of aspect (metadata aspect) allows users to view a set of properties, e.g., “All Properties” aspect, “Standard Properties” aspect, etc. (see Figure 14).

In fact, as Quan points out, the notion of aspects need not be restricted to a subset of reified properties of the underlying information entity; they can be the result of any computation on the entity, e.g. the age of the underlying entity based on its date of birth and the current date, or the size of the underlying collection of items.

Related to the problem of customizing views of information is that of allowing non-programmers to create user interfaces. Marquise is an example of a tool that allows users to create user interfaces for graphical editors by “demonstration, without programming.” [30] The Marquise tool “contains knowledge about palettes for creating and specifying properties of objects, and about operations such as selecting, moving and deleting objects” and hence constitutes a domain specific (applicable only to designing interfaces for graphical editors) user interface designing tool. Marquise highlights the notion of embedding domain knowledge into an interface builder tool, rather than the interface itself.

In this chapter, we advocate giving users the power to *create* their own views, using appropriate view designers that interpret the underlying information using various semantics, and can expose appropriate primitives for creating views. Semantics can be leveraged in various ways by view designers to allow creation of powerful views by users, while preserving valid data. For example, a simple understanding of the properties the user wishes to view/manipulate would allow the view designer to present appropriate widgets, e.g. a checkbox for a Boolean property, a slider control for a property having a fixed continuous range or enumerated values as valid values. Semantics used to interpret information can also be used to allow designers to correspond to styles of views, e.g. a list of numerical pairs can be interpreted as coordinates for a curve that is drawn. As a result, different view designers can make available different types of rendering preferences, e.g. text font vs. line thickness. Similarly, a list of genetic bases (Adenine, Guanine, Thymine, Cytosine) can be interpreted appropriately by a designer, and allow the user to specify a preference of whether just the sequence, or its complement is also to be rendered.



Karun Bakshi

Karun Bakshi

Person summary

| | | |
|---|---|---|
| Name: | Karun Bakshi | Edit |
| Electronic address: | Karun Bakshi (JXTA) | Edit |
| Home page: | Karun Bakshi | Edit |

Media creations

Full name information

Karun Bakshi

Specify a new English full name | Other choices

Addresses and phone numbers

No items in list

Specify a new home location | Other choices

All Properties

Standard Properties

| | | |
|---|---|---|
| type: | Person | Edit |
| Name: | Karun Bakshi | Edit |
| Alias: | None specified; click here to add | Edit |
| Contributor: | None specified; click here to add | Edit |
| Author/Creator: | None specified; click here to add | Edit |
| Subject: | None specified; click here to add | Edit |
| Description: | None specified; click here to add | Edit |
| Date: | None specified; click here to add | |
| Resource Type: | None specified; click here to add | Edit |
| Format: | None specified; click here to add | |
| Language: | None specified; click here to add | Edit |
| Relation: | None specified; click here to add | Edit |
| Source: | None specified; click here to add | Edit |
| Coverage: | None specified; click here to add | Edit |
| Rights: | None specified; click here to add | Edit |

Reverse Properties

**Figure 14 Screenshot of Haystack showing various aspects of the entity Karun Bakshi**

Since Haystack can represent information from arbitrary domains, it should support the addition of new view designers. As a baseline however, it is important that Haystack allow users to inspect arbitrary sets of properties of any entity, until a domain specific designer becomes available. Thus, we developed a simple, generic view designer that extends the power of metadata aspects in Haystack to users by allowing them to create views (henceforth referred to as metadata lens views or view parts) that select and view properties of interest from the underlying entity. Much like Magic Lenses, the user can specify that the view be used to examine a particular entity; the view then exposes the specified properties [11]. The only semantics the designer understands are that property values can be either literals or resources. However, even this simple capability becomes powerful, when combined with the power to reuse existing views in creating new views.

## 6.2 Design

The aforementioned user creatable views are designed analogously to information spaces. They allow users to specify "portals" for properties of the underlying entity, lay them out in two dimensions and stipulate various preferences for rendering them. Since the user creates views, he/she must be able to inspect and manipulate them. Similar to information spaces, the views themselves (and the property portals they embed) support two views: a designer view that allows users to layout the property portals and specify preferences, and a usage view that is rendered when the user has selected the view for examining some entity. Unlike information spaces however, it is not possible to switch from design view to usage view since the view needs to be applied to some information entity; unlike information spaces, such an entity is not specified in a fixed manner. (Note, the design and implementation of the metadata lens view part, and its designer and associated infrastructure require a thorough understanding of the Haystack view/part architecture and UI rendering ,which is beyond the scope of this document. Please consult [31] for further details.)

Finally, like Information Spaces, a "Metadata Lens View Parts Manager" has been implemented to allow users a convenient interface for creating and managing various views. We discuss the supporting ontologies for this framework, followed by a discussion of the user interfaces for the views (designer vs. usage) and the manager.

### 6.2.1 Underlying Ontologies

As in the case of information spaces, metadata lens views are implemented using three primary ontologies: Metadata Lens View Part Ontology, Property Lens Ontology and the Two Dimensional Layout Ontology. We do not discuss the Two Dimensional Layout Ontology as it has already been covered in the previous chapter.

The **`mlvp:MetadataLensViewPart`** class is declared in the Metadata Lens View Part Ontology, and is a subclass of **`ozone:ViewPart`**, i.e. it declares a particular type of view part. (This class is analogous to the **`is:InformationSpace`** class.) It is with this type, that all metadata lens views the user creates are annotated. In addition to supporting the various properties that views in Haystack support as well as a user-specified title, this class also has two additional properties: **`mlvp:metadataLensViewPartLayout`**, and **`mlvp:metadataLensViewPartClass`**. The first property captures the two dimensional layout of the properties to be displayed. The second property points to a unique view part class that the user can name, and which then becomes available for use in Haystack (e.g., in the information space designer to specify the view part class for the entity underlying an information portal).

The Property Lens Ontology declares the **`pl:PropertyLens`** class. (This class is analogous to the **`ip:InformationPortal`** class.) The view that the user creates, embeds a set of property lenses that are to be applied to the underlying entity. Property Lenses support five attributes: **`pl:property`**, **`pl:editable`**, **`pl:showTitle`**, **`pl:propertyTitle`** and **`pl:viewPartClassToUse`**. The **`pl:property`** attribute points to the property that is to be extracted for the underlying entity. The **`pl:editable`** attribute is only applicable if the property that is to be examined has literal values, and specifies if the user would like to be able to edit the property, or just view it. The **`pl:showTitle`** attribute controls whether or not the title indicated by **`pl:propertyTitle`** is to be shown when the property lens is rendered in its usage view. Note, **`pl:propertyTitle`** points to a copy of the title of the property in order to allow users to change it in the context of the view being created (for visualization purposes), without actually changing the title of the property being inspected. As a result, the user has the flexibility to use a meaningful title for the view, yet still retain the ontologically meaningful title of the property. Finally, the **`pl:viewPartClassToUse`** property is only applicable if the property being examined results in a resource. It provides the vehicle for reuse of existing views and specifies which view part class to use in order to render the resource that results when the specified property of the underlying entity is queried.

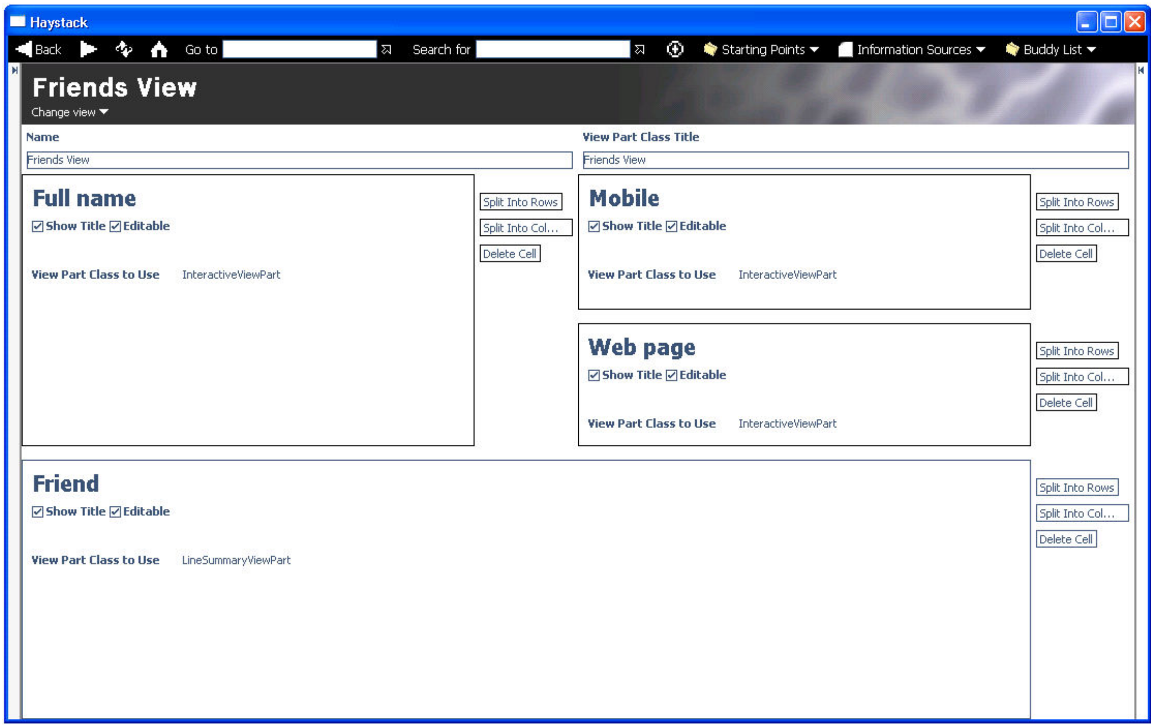
### 6.2.2 Metadata Lens View Part Designer View

The Metadata Lens View Part Designer View of **`mlvp:MetadataLensViewPart`** can be used to design the view being created by laying out property lenses. A new metadata lens view is initialized with a single property lens. The figure below shows the layout for the designer.

Similar to Information spaces, the designer is split into two sections: the top pane lets the user edit the title of the view and the associated view part class. The bottom pane allows users to layout the various property lenses, using the same mechanism for editing the two dimensional information portal layout described in Chapter 5.

Each of the property lenses allows all five attributes of property lenses to be edited. The **`pl:property`** can be specified by using drag and drop to specify a property to inspect. The **`pl:editable`** and **`pl:showTitle`** Boolean properties can be edited by the corresponding checkboxes. The title to display for the property being inspected can also be edited and defaults to its current title. Finally, the **`pl:viewPartClassToUse`** property can be specified using drag and drop.

An important feature of the current interface is that a user is required to specify a title (initially blank) for the view part class that is associated with the view that the user has created. Albeit confusing, this distinction between the view part class and the view (also known as a view part) is an artifact of the current Haystack view implementation infrastructure, since view part classes (rather than views) are specified when the user wants to see some entity in a particular manner. Generally, the user will want both titles to be the same for ease in remembering, and since the view part class (a concept understood by Haystack) is used to refer to the view (a concept understood by the user) when finding and creating a specified view in the Haystack implementation. However, the interface does not ask for just a single title and update both, since the two titles are in fact titles for distinct concepts (they cannot be used interchangeably).



Name: Friends View
View Part Class Title: Friends View

Full name
☑ Show Title ☑ Editable
View Part Class to Use: InteractiveViewPart

Mobile
☑ Show Title ☑ Editable
View Part Class to Use: InteractiveViewPart

Web page
☑ Show Title ☑ Editable
View Part Class to Use: InteractiveViewPart

Friend
☑ Show Title ☑ Editable
View Part Class to Use: LineSummaryViewPart

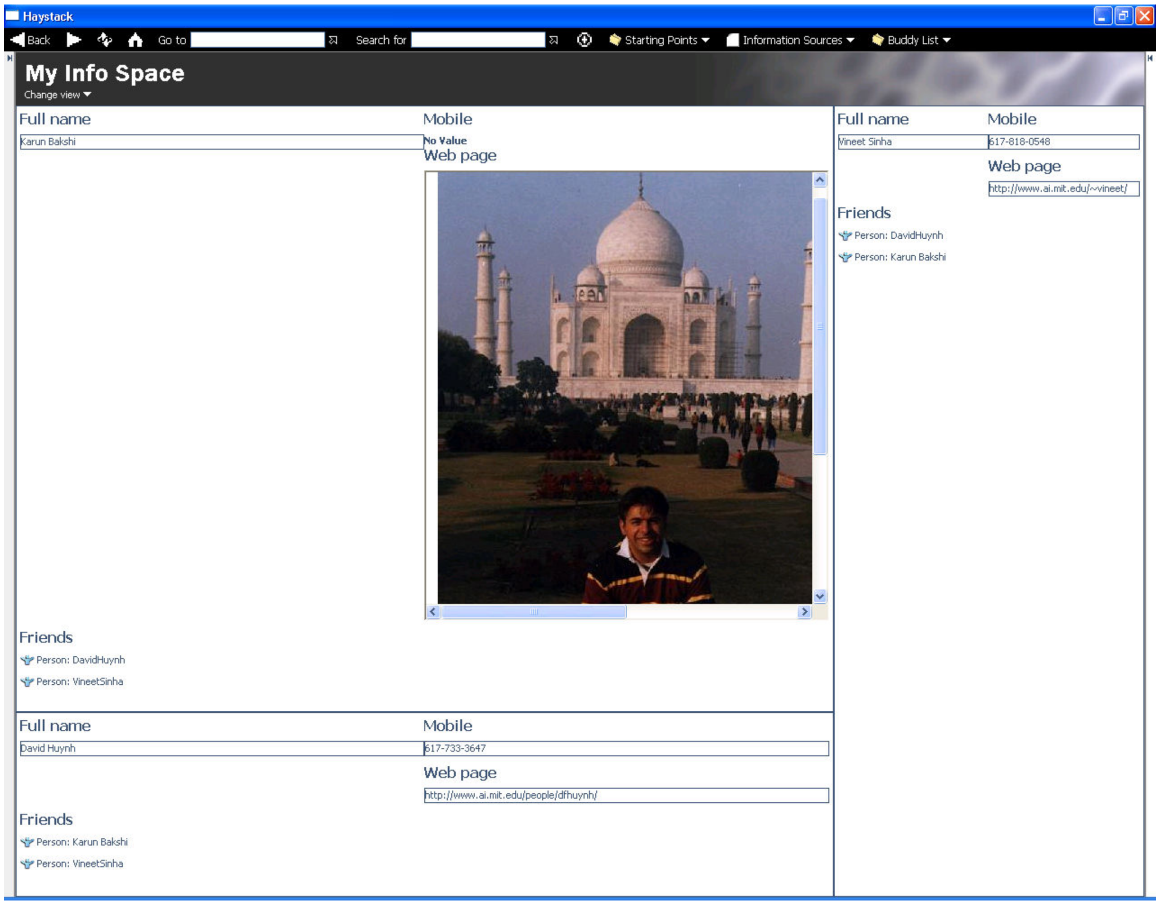
**Figure 15 A Metadata Lens View Part shown using the Designer View**

A second feature to note about the current design interface for property lenses is that even though `pl:editable` and `pl:viewPartClassToUse` may be deemed mutually exclusive (`pl:editable` is applicable to literal values, and `pl:viewPartClassToUse` is applicable to resource values), they are both available simultaneously for editing. This is done since Haystack supports a semi-structured data model, where the declared ontologies are not enforced, and hence there are no guarantees on the nature of the values of properties. Furthermore, depending on the data model, some ontologies may actually model properties as having either literal or resource values. Thus, the property lens collects the relevant information "just in case", and applies the preferences dynamically, based on the results of querying the property value.

### 6.2.3 Metadata Lens View Part Usage View

The Metadata Lens View Part Usage View is the view that is rendered when the view part is being used to inspect the properties of some information entity. The figure below shows an example of this view: an information space is used to show three people (Karun

Bakshi, David Huynh and Vineet Sinha) that are mutual friends, using the Friends view part that the user created above to see who is a friend of whom.



My Info Space

Full name
Karun Bakshi
Mobile
No Value
Web page
Friends
Person: DavidHuynh
Person: VineetSinha

Full name
David Huynh
Mobile
617-733-3647
Web page
http://www.ai.mit.edu/people/dfhuynh/
Friends
Person: Karun Bakshi
Person: VineetSinha

Full name
Vineet Sinha
Mobile
617-818-0548
Web page
http://www.ai.mit.edu/~vineet/
Friends
Person: DavidHuynh
Person: Karun Bakshi

**Figure 16 An information space that allows the user to inspect three people using the Friends View**

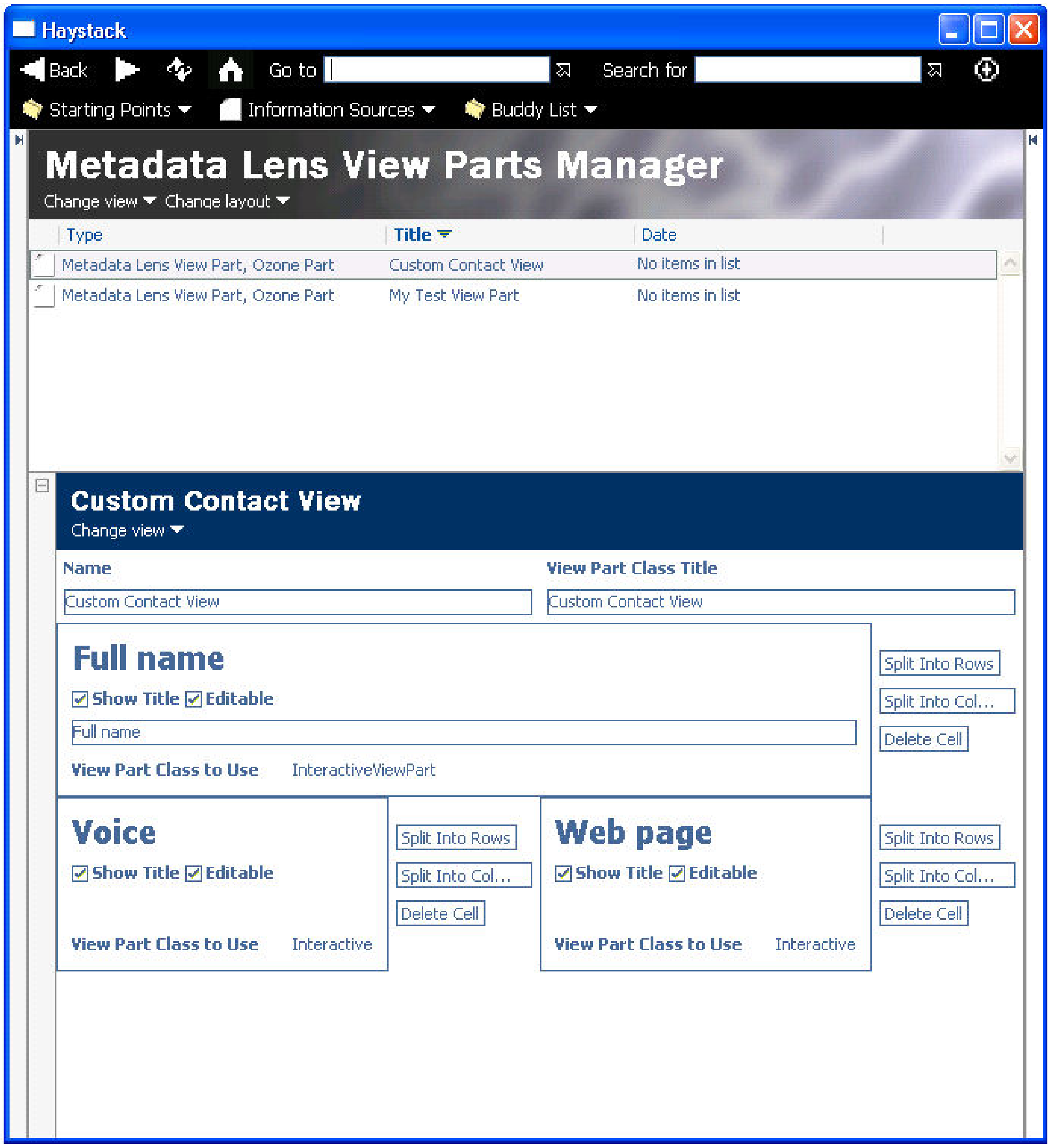
As in the case of information spaces, the metadata lens view part lays out the usage views of the embedded property lenses according to the specification of the two dimensional layout. Each of the property lenses in turn shows a title for the property (if selected by the user), followed by a list of values for the property. (Just as the ranges of properties are not enforced in the semi-structured store, neither is the cardinality of the property. Thus, we show multiple values if they exist, for completeness.) If the property was selected to be editable (and is a literal), then the user is presented with an edit box; otherwise, just the value of the literal is presented. If the value is a resource, the appropriate view part class is used to display it.

In the view above, each person has a list of friends, but only single values for the other properties. Notice, that the homepage for the person Karun Bakshi is displayed rather than an editable URL. In this case, the URL that was given was a resource (not a literal). Prior to looking for a view part locally, Haystack automatically tries to dereference all resources that appear (based on some heuristic, e.g., begins with "http:…") to be URLs on the Web, when attempting to use an InteractiveViewPart to display the resource.  In this case, it succeeds in finding and embedding it. Thus, the value of the same property is dynamically displayed, based on whether it is a literal or a resource.

Several features of this view are noteworthy. First, the type of interaction for a property with a resource value depends on the view selected to render it. Thus, for example, the interaction with the friends depends on what the underlying view (LineSummaryViewPart) supports for resources of type `hs:Person`. Also, since the underlying layout engine is the same as information spaces, the resulting metadata lens view supports resizing the lenses. However, we do not show borders between properties to hide any notion of a resize handle.

### 6.2.4 Metadata Lens View Parts Manager

The Metadata Lens View Parts Manager is implemented in exactly the same manner as the Channel and Information Space Manager, with the correct view for an appropriately defined channel. Relevant operations are made available for creating, copying and deleting metadata lens view parts, as with the other channels. The only difference is that the preview pane only allows interacting with the view using its designer view since there is no underlying information entity to which it can be applied. A screenshot of this user interface is presented below.



## Metadata Lens View Parts Manager

| Type | Title | Date |
|---|---|---|
| Metadata Lens View Part, Ozone Part | Custom Contact View | No items in list |
| Metadata Lens View Part, Ozone Part | My Test View Part | No items in list |

### Custom Contact View

| Name | View Part Class Title |
|---|---|
| Custom Contact View | Custom Contact View |

**Full name**
☑ Show Title ☑ Editable
Full name
View Part Class to Use: InteractiveViewPart

**Voice**
☑ Show Title ☑ Editable
View Part Class to Use: Interactive

**Web page**
☑ Show Title ☑ Editable
View Part Class to Use: Interactive

**Figure 17 Screenshot of Metadata Lens View Parts Manager**

## *6.3 Implementation*

In this section, we discuss the implementation of the design and usage views of the Metadata Lens View Parts. We do not discuss the implementation of the Metadata Lens View Parts Manager as it is implemented in the same manner as previous such interfaces. The implementation files for the Metadata Lens View Part class are schemata/MetadataLensViewPart.ad and ui/MetadataLensViewPartDesigner.ad. The implementation files for Property Lens are schemata/PropertyLens.ad and ui/PropertyLens.ad.

In order to understand how Metadata Lens View Part's views are implemented, we briefly examine a portion of the declaration for a new view of type **`mlvp:MetadataLensViewPart`** below. (Note, identifiers without : below indicate variables having a unique URI, which changes for each view part the user creates. The other URIs remain the same for each view part.)

```
add {daml:Thing   hs:classView      :MetadataLensViewPartClassView}

add {newViewPart
      mlvp:metadataLensViewPartLayout     viewLayout;
      mlvp:metadataLensViewPartClass      metadataLensViewPartClass;
      dc:title                           "Untitled Metadata Lens View
Part";

      rdf:type               ozone:Part;
      rdf:type               ozone:ViewPart;
      rdf:type               mlvp:MetadataLensViewPart;
      rdf:type               metadataLensViewPartClass;
      ozone:viewDomain        :MetadataLensViewPartClassView;
      ozone:partDataGenerator :presentMetadataLensViewPart;

}
```

Several attributes are noteworthy. First, all items (**`daml:Thing`**) are designated as having class view :MetadataLensViewPartClassView, that the new view part implements (as the **`ozone:viewDomain`** property indicates). Second, the view part has also been annotated as being of a unique view part class type (**`metadataLensViewPartClass`**). These two annotations are necessary for Haystack to locate this view part for any item that specifies the unique view part class for rendering. Third, the new view part has been annotated as being type **`mlvp:MetadataLensViewPart`** to indicate that it is a view part of a particular type. Finally, the view part has attributes that are specific to the new type of view part (**`mlvp:MetadataLensViewPart`**), e.g., **`mlvp:metadataLensViewPartLayout`**.

The designer view for the view part is implemented simply by taking the value of the **`mlvp:metadataLensViewPartLayout`** property that points to an object of type **`tdl:TwoDimensionalLayout`**, and viewing it with the TwoDimensionalLayoutView (as explained in Chapter 5) that serves as a layout engine for the property lenses. The property lenses themselves are viewed using their own design view by specifying the desired view that is to be used by the two dimensional layout engine for its cell data as TwoDimensionalLayoutCellEditorView.

The usage view that is created by the user (when the view part is being used to view the properties of an underlying entity), is rendered by calling an adenine method that generates the part data (UI widget description of the view), as specified by the **`ozone:partDataGenerator`** property. This method also queries for the **`mlvp:metadataLensViewPartLayout`** property of the view, and also invokes the two dimensional layout engine, but specifies the TwoDimensionalLayoutCellUsageView for the property lenses by registering an appropriate context property (see Chapter 5). In addition, the underlying entity to which this view is being applied (i.e., the entity whose properties are being inspected) is also registered in the context so that the property lenses can be applied to the right underlying entity.

The implementation of the usage view above can be used to model other view part designers that may be created in the future. A new view part designer will in general allow the user to create a particular type of view part that is a subclass of `ozone:ViewPart`. Such a view part should have one or more properties to capture user preferences for the view (in our case, this predicate was `mlvp:metadataLensViewPartLayout`). Then, the view should be rendered by a view container implemented that will invoke the method specified by `ozone:partDataGenerator` (`:presentMetadataLensViewPart` in our case). This method interprets the user preferences captured by the view, and translates them to part data (UI widget descriptions) understood by Haystack. Finally, the designer should make the appropriate annotations for the view part class and class views (`metadataLensViewPartClass` and `:MetadataLensViewPartClassView`) to ensure that the view part gets invoked in the right circumstances, i.e., for objects of the right type, that want this particular view.

The Property Lens designer view is fairly straightforward, as it simply exposes all the property lens properties for editing by the user. The Property Lens Usage view interprets the user preferences for the property lens, and generates appropriate part data to show the values of the property.

## 6.4 Conclusion

In this chapter we deliberated the need for users to be able to *create*, not just reuse, views for a given information entity. Thus, we proposed the need for view designers that understand the semantics of various types of information and expose relevant capabilities for creating views. Given that Haystack's data model can support various types of information, it should support various types of view designers. As a result, we developed a domain-independent, baseline view designer that lets users inspect any properties of an information entity in Haystack, regardless of its domain and taking advantage of minimal information semantics. Nevertheless, we recognized that more powerful views can be designed by view designers that better understand the semantics of the underlying information that the views they generate will be manipulating.

# Chapter 7 Conclusion and Future Work

Having considered the architecture and the design and implementation of tools required for users to create and customize their own information spaces, this final chapter is devoted to understanding some of the merits of our approach and the resulting contributions, new ideas worth exploring in the future as well as areas deserving additional improvement.

## 7.1 Contributions

Although we currently lack an evaluation of our system, and an understanding of its efficacy in increasing user productivity in arbitrary information based tasks, we feel that our work nevertheless contributes some important ideas.

The most important (and simplest) idea we highlight is the existence of a problem (and thus, research topic) in the way user's currently perform information based tasks; the tasks are generally unique based on various factors, and require information from arbitrary domains. As a result, traditional approaches to increasing user efficiency using multiple applications or more user-friendly, task-centric, all-encompassing interfaces built by developers will inherently be rigid and lacking in some respect from the user's perspective; users are different and tasks continuously change in manners developers simply cannot foretell. Instead, we propose allowing users to build their own interfaces for their tasks rather than be fettered by the constraints of information locked in application specific islands, using various tools for creating customized task-based interfaces: information spaces. As a result, we allow users to create their own "applications," tailored to a particular task.

In support of this approach, we developed the notion of Channels of information, a general means of allowing the user to specify a persistent query/description of how to compute a collection of information that is related in some manner. As a result, a channel can be used as a unit of specifying content that can be "re-directed" into any/multiple user created information spaces.

We also proposed an architecture for a solution to the above problem that is open in that it allows adding more components that enhance the toolkit available to users to customize their interfaces. New query and computational primitives may be added to enhance the expressive power available to the user in specifying channels via the Channel Manager.

Building on top of the Haystack philosophy of having multiple means of viewing an information entity, we proposed the notion of view designers that understand the same information using different semantics, and hence allow building views that interact with that information in various ways (in addition to ones that are implemented by developers). We also outlined a general design pattern for implementing other view designers in Haystack. Developers may implement domain or type specific view designers, which allow the user to create views that provide a better interaction and visualization experience. For example, whereas the user may create a view to look at the properties of appointments via the current generic view designer, a specialized view designer for a calendar can allow the user to render appointments in a timeline format or a week at-a-glance format.

Finally, Haystack already supports operations as first class entities, and hence additional operations can be written and exposed to the user for availability in particular contexts.

## 7.2 Future Work

Our current system as it stands leaves plenty of room for future work, which we consider in this section. For simplicity, we categorize the work into three categories: evaluation of current system, new ideas for supporting information customization within the current framework, new features that would make the current set of functionality more useful.

### 7.2.1 Evaluation

Since we have not evaluated our system, or any of our claims, a major portion of our future work will focus on obtaining feedback on the relevance of our ideas and usability of our implementation via a user study. Some of the questions that the user study should seek to find answers to include:

- Is the idea of user creation and customization of information spaces compelling? That is, do people actually have the problems we argued in our motivation? If so, is the power they are given in this respect, worth the accompanying burden of creating their own channels and information spaces? If not, why not? How can the burden be minimized or alleviated. How useful are domain specific view designers in this respect, or is the ability to select (rather than create) from a pre-defined set of views sufficient?
- What are some of the use cases for information spaces that are important from users' perspective? Which features/ideas in our current implementation support them? Which features are missing? Which features/ideas/abstractions need to be modified to enhance usability?
- How did the individual design decisions for the various tools fare? For example, does projection operations by creating buttons, rather than modifying the context

menu with the user specified operations help, or is the accompanying loss of uniformity disorienting to the user? Should users be able to resize information portals, or simply collapse them to make more space for other parts of the interface?

We are already aware of various improvements to the current implementation that are required to enhance usability, before a user study can be conducted:

- The current means of piping the results from one set transform to another, requires users to build the channel definition in reverse order: the final set transform must be specified first, and its inputs can then be specified as being the results of other set transforms. A graph editor that does not force this "tree-like" declaration method and that allows users to link the results of set transforms to the inputs of other set transforms in any order would be more intuitive.
- The current query interface is complex and requires users to remember individual query primitives that can only perform one type of query, and specify arguments for them. A query interface that provides an intuitive interface for multiple types of queries, and maps the user specified information to the appropriate underlying query primitives is more desirable.
- The names of various set transforms and primitives are cryptic, and should be more user-friendly and descriptive.
- The current means of changing the layout of information spaces results in loss of information portal preferences if they are split further. Also, within a given layout, it is difficult to switch the locations of information portals. Users should not have to do significant re-creation of the information space, just because the layout has changed.
- Due to the way in which Haystack is implemented, users are forced to contend with implementation complexities such as view part classes. The current Haystack infrastructure should be changed to simplify and minimize the abstractions and concepts the user must understand.
- In order to have a good use case for the user study, one or more domain specific view designers or views may need to be created.

### 7.2.2 New Avenues for Research

In this section, we discuss new ideas that will enhance the power of users to customize their information spaces.

- **Channel Triggers** – It would be useful for the user to specify certain actions that are to be taken by the system, when items are added to or removed from the channel.
- **Dynamic Currying** – The operations shown in an information portal should be sensitive to the channel item currently selected; operations that are not relevant to it should not be visible, and operations that require an argument type matching that of the selected item should dynamically bind to it, as is done in context menus.

- **Information Space Wiring** – Information portals showing channel items should expose the currently selected item, such that it can be used elsewhere in the interface, e.g., a preview pane can be shared by the most recently selected item from multiple channels. Thus, users should be able to "wire-up" their interface.
- **Channel Learning** – Haystack should be able to learn and generate the channel description based on set transforms from a collection of items specified by the user.
- **Virtual Property Lenses** – A new type of property lens that *computes* a property value (rather than extracting it) would significantly enhance the power of views users can create.
- **Viewing Heterogeneous Channels** – The items in a channel need not necessarily be of the same type, e.g., "Messages from David Karger" may include instant messages as well as e-mail. Users may want to see different types of information using different types of views, and the current information portal implementation would need to be enhanced to support this functionality.
- **Reuse of Tools** – The tools we developed for information spaces allow users to bootstrap and can be used to actually create some of the tools themselves. For example, information spaces can be used to create an Ontology Browser, and property lenses can be used to actually implement the Design View of property lenses themselves! These exercises should be performed to see where limitations exist that do not allow such recreation of functionality, and what new abstractions are needed to overcome them and further enhance the power available to users.

### 7.2.3 New Features for Current Implementation

The following feature enhancements would tremendously increase the usability of our tools:

- **Polished Information Space User Interface** – The current user interface for information spaces is not space efficient and forces significant user interaction. For example, it requires the user to manually resize various information portals, and does not allow hiding the pane showing the buttons for relevant operations. It would be nice to be able to collapse portals and sub-panes with a single click. Also, space is currently wasted by vertically placing the buttons that allow users to split information portals in design mode. If they were horizontally laid out (e.g., below the portal), users would have more space to work with the portals in designer view.
- **Pervasive Style Annotations** – Currently, users cannot specify any style specifications, e.g. background color, border widths, text fonts, etc. for any part of their information space. Such a capability is provided by most applications, and should be supported in Haystack as well.
- **Channel Update Specifications** – Users should be able to specify when individual channels are updated.
- **Additional Primitives** – New set transforms that implement additional primitive operations would increase the expressive power available to users in defining channels. For example, a set transform that finds other items "similar" to a specified item using machine learning would be incredibly useful. Similarly,

additional condition tests, e.g., “starting with string xyz” or “having 2 or more instances of a property” would also increase expressive power available to users.

- **Reusable Component Library** – Currently, the only means of reusing a portion of another channel’s description (i.e. some set transform instance that it encapsulates) is to declare a channel that has that set transform instance as its description. The channel is declared simply so that it can be re-used in some other channel, and has no semantic value of its own. Condition Tests cannot be re-used at all. Thus, it would be useful for the user to be able to store set transform instances (that are not actually computed by the channel manager) and condition test closures that can be duplicated and used when defining channels.
- **Property Lens Library** – Given that many properties are common across ontologies, such as title, date, etc., the user may want to inspect them in multiple views. Thus, the ability to add existing property lenses to new views without having to define them each time would be very useful. Perhaps, our system should increase the granularity at which users can control visualization of information by allowing them to create and manage individual property lenses, and then reuse them in different layouts and combinations to create new views rapidly.
- **Metadata Lens View Enhancement** – The current implementation of the metadata lens view parts that users can create, do not allow them to add or delete property values; property values can only be inspected or modified. Such a capability would allow users complete CRUD (Create, Replace, Update, Delete) control over their information store. An interesting design question here would be whether or not to enforce the schema for the underlying object (i.e. not allow additional values, if the property is unique, e.g. age). Users may desire both structured and semi-structured access in different circumstances.

# Appendix A – Available Set Transforms

**Table 1 Available Set Transforms**

| Name | Arguments | Description |
|---|---|---|
| **Set Operators** | | |
| **SetUnion** | **Set1, Set2**: The sets to be combined via a set union. | A transform that allows a set union to be computed of its constituent arguments. It is used as a substitute for boolean disjunction (Inclusive OR). |
| **SetIntersection** | **Set1, Set2**: The sets to be combined via a set intersection. | A transform that allows a set intersection to be computed of its constituent arguments. It is used as a substitute for boolean conjunction (AND). |
| **SetDifference** | **Minuend Set**: The "universal" set.<br><br>**Subtrahend Set**: The set to be subtracted. | A transform that returns the set difference of two sets. It is used as a substitute for boolean negation when supplied with an appropriate "universal" set. |
| **Query Primitives** | | |
| **Subject Query (Fixed** | **Predicate**: The predicate to | A transform that returns all |

| | | |
|---|---|---|
| **Predicate, Literal Object)** | fix in the subject query.<br><br>**Condition Test**: A closure for the condition test to apply. | resources that are subjects having the specified predicate, and an object value that satisfied the specified condition test closure. |
| **Subject Query (Fixed Predicate, Resource Object)** | **Predicate**: The predicate to fix in the subject query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are subjects having the specified predicate, and an object value that satisfies the specified condition test closure. |
| **Subject Query (Fixed Literal Object)** | **Object (Literal)**: The (literal) object to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are subjects having the specified object, and a predicate that satisfies the specified condition test closure. |
| **Subject Query (Fixed Resource Object)** | **Object (Resource)**: The (resource) object to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are subjects having the specified object, and a predicate that satisfies the specified condition test closure. |
| **Predicate Query (Fixed Subject, Literal Object)** | **Subject**: The subject to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are predicates having the specified subject, and a corresponding object value that satisfies the specified condition test closure. |
| **Predicate Query (Fixed Subject, Resource Object)** | **Subject**: The subject to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are predicates having the specified subject, and a corresponding object value that satisfies the specified condition test closure. |
| **Predicate Query (Fixed Literal Object)** | **Object (Literal)**: The object to fix in the query. | A transform that returns all resources that are predicates having the specified object, |

| | | |
|---|---|---|
| | **Condition Test**: A closure for the condition test to apply. | and a corresponding subject value that satisfies the specified condition test closure. |
| **Predicate Query (Fixed Resource Object)** | **Object (Resource)**: The object to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are predicates having the specified object, and a corresponding subject value that satisfies the specified condition test closure. |
| **Object Query (Fixed Subject)** | **Subject**: The subject to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are objects having the specified subject, and a corresponding predicate value that satisfies the specified condition test closure. |
| **Object Query (Fixed Predicate)** | **Predicate**: The predicate to fix in the query.<br><br>**Condition Test**: A closure for the condition test to apply. | A transform that returns all resources that are objects having the specified predicate, and a corresponding subject value that satisfies the specified condition test closure. |
| | | |
| **Identity/Null Primitives** | | |
| **Null** | None | A transform that returns an empty set. |
| **Channel Duplicator** | **Channel**: The channel whose contents are to be duplicated. | A transform that copies and returns the contents of another channel. |
| **Collection Wrapper** | **Collection**: The collection that is to be wrapped. | A transform that copies and returns the contents of another collection. |
| **Others** | | |
| **Items of Type** | **Item Type**: The type of the items to be selected. | A transform that allows selection of items of a particular type. |

# Appendix B – Available Condition Tests

**Table 2 Available Condition Tests**

| Name | Arguments | Description |
|---|---|---|
| **Null Condition** | | |
| Any Value | None | A condition test that always returns true, i.e. any value is acceptable. |
| **Conditions on Literals** | | |
| **== (Literal)** | **Variable Value (Literal)**: The item to which the condition test is applied.<br><br>**Fixed Value (Literal)**: The item used to compare to the variable value in the relational condition test. | A condition test that returns true if the variable argument is equal to the fixed argument (in a string match sense). |
| **!= (Literal)** | **Variable Value (Literal)**: The item to which the condition test is applied.<br><br>**Fixed Value (Literal)**: The item used to compare to the variable value in the relational condition test. | A condition test that returns true if the variable argument is not equal to the fixed argument (in a string match sense). |

|  |  |  |
|---|---|---|
| **Conditions on Resources** |  |  |
| **== (Resource)** | **Variable Value (Resource)**: The item to which the condition test is applied.<br><br>**Fixed Value (Resource)**: The item used to compare to the variable value in the relational condition test. | A condition test that returns true if the variable argument is equal to the fixed argument (in a string match sense). |
| **!= (Resource)** | **Variable Value (Resource)**: The item to which the condition test is applied.<br><br>**Fixed Value (Resource)**: The item used to compare to the variable value in the relational condition test. | A condition test that returns true if the variable argument is not equal to the fixed argument (in a string match sense). |
| **Conditions on Numeric Values (Literals)** |  |  |
| > | **Fixed Value (Double)**: The item used to compare to the variable value in the relational condition test.<br><br>**Variable Value (Literal)**: The item to which the condition test is applied. | A condition test that returns true if the variable argument is greater than the fixed argument. Both operands must be numeric. |
| < | **Fixed Value (Double)**: The item used to compare to the variable value in the relational condition test.<br><br>**Variable Value (Literal)**: The item to which the condition test is applied. | A condition test that returns true if the variable argument is less than the fixed argument. Both operands must be numeric. |
| >= | **Fixed Value (Double)**: The item used to compare to the variable value in the relational condition test.<br><br>**Variable Value (Literal)**: | A condition test that returns true if the variable argument is greater than or equal to the fixed argument. Both operands must be numeric. |

| | | |
|---|---|---|
| | The item to which the condition test is applied. | |
| <= | **Fixed Value (Double)**: The item used to compare to the variable value in the relational condition test.<br><br>**Variable Value (Literal)**: The item to which the condition test is applied. | A condition test that returns true if the variable argument is less than or equal to the fixed argument. Both operands must be numeric. |
| **== (Numeric)** | **Fixed Value (Double)**: The item used to compare to the variable value in the relational condition test.<br><br>**Variable Value (Literal)**: The item to which the condition test is applied. | A condition test that returns true if the variable argument is equal to the fixed argument. Both operands must be numeric. |

# References

1. Extensible Markup Language (XML). http://www.w3.org/XML/.
2. Web Services Activity. http://www.w3.org/2002/ws/.
3. Semantic Web. http://www.w3.org/2001/sw/.
4. Bellotti, V., Ducheneaut, N., Howard, M. and Smith, I. Taking Email to Task: The Design and Evaluation of a Task Management Centered Email Tool. Proceedings of the Conference on Human Factors in Computing Systems 2003.
5. Kubi Software. www.kubisoft.com.
6. Anderson, C. and Horvitz, E. Web Montage: A Dynamic Personalized Start Page. Proceedings of the eleventh international conference on the World Wide Web, 2002.
7. Bauer, D. Personal Information Geographies. Extended Abstracts of the Conference on Human Factors in Computing Systems 2002.
8. Microsoft Office Online. http://www.office.microsoft.com/home/.
9. Microsoft Office Online, Outlook. http://www.microsoft.com/outlook/.
10. Storey, M., Best, C., Michaud, J., Rayside, D., Litoiu, M. and Musen, M. SHriMP Views: An Interactive Environment for Information Visualization and Navigation. Extended Abstracts of the Conference on Human Factors in Computing Systems 2002.
11. Bier, E., Stone, M., Pier, K., Buxton, W. and DeRose, T. Toolglass and Magic Lenses: The See-Through Interface. Proceedings of International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH) 1993.
12. Hutchings, D. and Stasko, J. QuickSpace: New Operations for the Desktop Metaphor. Extended Abstracts of the Conference on Human Factors in Computing Systems 2002.
13. Berners-Lee, T., Hendler, J. and Lassila, O. The Semantic Web. *Scientific American*, May 2001.
14. Hogue, A. Tree Pattern Inference and Matching for Wrapper Induction on the World Wide Web. MEng. Thesis. MIT, 2004.
15. *MyYahoo!* http://my.yahoo.com.

16. Card, S., Henderson, D. Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. *ACM Transactions on Graphics* Vol. 5, Issue 3, July 1986, pp. 211 – 243.
17. Cruz, I., and Lucas, W. A Visual Approach to Multimedia Querying and Presentation. Proceedings of the fifth ACM International Conference on Multimedia, 1997.
18. Schraefel, M., and Zhu, Y. Hunter Gatherer: A Collection Making Tool for the Web. Extended Abstracts of the Conference on Human Factors in Computing Systems 2002.
19. Quan, D., Huynh, D. and Karger, D. Haystack: A Platform for Authoring End User Semantic Web Applications. 2nd International Semantic Web Conference, 2003.
20. Resource Description Framework (RDF). http://www.w3.org/RDF/.
21. Quan, D. Designing End User Information Environments Based On Semistructured Data Models. PhD Thesis. MIT, 2003.
22. RSS 2.0 Specification. http://blogs.law.harvard.edu/tech/rss.
23. Goldman, R. and Widom, J. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, Proceedings of 23rd International Conference on Very Large Data Bases, 1997.
24. START Natural Language Question Answering System. http://www.ai.mit.edu/projects/infolab/start-system.html.
25. Popescu, A., Etzioni, O. and Kautz, H. Towards a Theory of Natural Language Interfaces to Databases, Proceedings of the 8th International Conference on Intelligent User Interfaces, 2003.
26. Banko, M., Brill, E., Dumais, S. and Lin, J. AskMSR: Question Answering Using the Worldwide Web. Proceedings of 2002 AAAI Spring Symposium on Mining Answers from Texts and Knowledge Bases, March 2002.
27. Zloof, M. "Design Aspects of the Query-By-Example Data Base Manipulation Language," in B. Shneiderman (Ed.) *Databases: Improving Usability and Responsiveness*. Academic Press. New York, USA.1978.
28. Microsoft Office Online, Access. http://www.microsoft.com/office/access/.
29. FileMaker. http://www.filemaker.com/.
30. Myers, B., McDaniel, R. and Kosbie, D. Marquise: Creating Complete User Interfaces By Demonstration. Proceedings of the Conference on Human Factors in Computing Systems 1993.
31. Huynh, D. Haystack's User Interface Framework: Tutorial and Reference. http://haystack.lcs.mit.edu/documentation/ui.pdf.