Designing End User Information Environments Built on Semistructured Data Models

by

Dennis A. Quan, Jr.
S.M. Computer Science
Massachusetts Institute of Technology, 2002

S.B. Mathematics
Massachusetts Institute of Technology, 2002

S.B. Chemistry
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

June 2003

Signature of Author.....
Department of Electrical Engineering and Computer Science
May 29, 2003

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

Accepted by.....
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

DESIGNING END USER INFORMATION ENVIRONMENTS BUILT ON SEMISTRUCTURED DATA MODELS

by

DENNIS ARTHUR QUAN JR.

Submitted to the Department of Electrical Engineering and Computer Science on May 29, 2003 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

ABSTRACT

Today's information systems were not built to adapt themselves to personal needs. For example, assigning properties not envisioned by the database administrator or the software engineer such as "good music to listen to when I am in a bad mood" or "excellent sushi place for taking foreign guests" is difficult in most programs because schemas are often cast in stone by a compiler or database management system. Relationships between objects in the same program or different programs, such as "Bob is Mary's brother" or "this report is relevant to next week's meeting", are similarly difficult to specify, since programs either fail to expose their object models to each other, or their object models are not fine-grained enough. If computers cannot record connective meta-data—the "glue" we use for keeping track of things—information overload will eventually prevent users from taking full advantage of information technology.

We claim that by using a semistructured data model, the system can capture the context and circumstances underlying information and not simply the information itself and thus help to elucidate the relevance of the information for the user and others. To demonstrate the efficacy of our semistructured data model (based on semantic networks) and explore its consequences to the user interface, a system named Haystack has been built that permits users to store their information in a flexible fashion that can adapt to their needs. Haystack also coalesces previously segregated sources of the user's information, describes the data found in these sources using our rich description framework, and explores new means and synergies for creating, displaying, and sharing this information. The thesis begins with a motivation of the problem space and a tour of our system and then delves into the details of our user interface paradigm and other higher level concepts we believe will be helpful for those designing future information environments.

Thesis Supervisor: David R. Karger

Title: Associate Professor of Computer Science

Table of Contents

Acknowledgements

First and foremost, I would like to thank my mom for her support and inspiration throughout the construction of this thesis.

I wish to thank my thesis advisor, Professor David R. Karger, for endlessly pushing me to find better ways to tackle the problems in this thesis, for his dedication in spending countless hours in discussions with me, and for putting up with me over the past three years.

I want to thank my thesis committee members Professor Robert Miller, Professor Peter Szolovits, and Dr. Howard Shrobe for their countless hours in helping me to clarify the ideas in my thesis and for their invaluable insights. Thanks also to Professor Mark Ackerman for his insights and advise during my research.

Thanks to my colleagues Sean Martin and Dr. Chetan Murthy and to my managers David Grossman, John Patrick, Dr. Stuart Feldman, and Dr. Irving Wladawsky-Berger for their support and insights into the work embodied by this thesis.

I would like to thank my fellow graduate students David Huynh, Jimmy Lin, and Karun Bakshi for their friendship, the discussions we had, and for coauthoring papers with me.

I want to thank Walter Bender and Professor David Gifford for their stimulating conversations, inspiration, and advice during the completion of my thesis. Thanks also to members of the W3C, in particular Tim Berners-Lee, Ralph Swick, and Eric Miller, for their support and introduction to the Semantic Web.

Special thanks go to all of our user study participants who gave us their time in an effort to reveal the usefulness of our ideas.

This work was supported in part by IBM and the MIT Oxygen Project.

Chapter 1 Introduction

The world is becoming increasingly information-centric: daily life confronts us with more and more information to handle, in the form of financial reports, interoffice memoranda, special offers, travel itineraries, and the like. We somehow manage to keep tabs on much of it, organizing information according to a combination of the information's factual and subjective characteristics. Human memory can hold some of our metadata; most, however, is recorded physically, in the form of filing cabinets, piles, sticky notes, and binders. The act of putting information into context plays such an important role in keeping track of information that we often involve others in this endeavor. Information that can be more formally organized is placed in libraries, where librarians use their judgment to bring order to millions of published books and magazines. Similarly, much of the metadata we use for judging information comes from others' subjective experiences, such as recommendations from colleagues and magazine reviews.

The Internet has not changed our information-centric view on life, but it has changed the scale of the game. The efficiencies of electronic publication have caused a corresponding catalytic acceleration in the production of information, and as a result, traditional techniques for finding important information are gradually becoming less and less effective. Libraries maintain a fairly small set of characteristics for the books they keep, and librarians are barely able to keep up with the current influx of literature. Even if only a few books would have matched a query with a limited set of characteristics in the past, now thousands of Web pages or e-mails might result. Furthermore, some old techniques for taking advantage of subjective information do not scale on the Internet. Because of its quantity and nonphysical nature, filing away or putting sticky notes on information found on the Internet or in e-mails is a daunting task. Similarly, recommendations must be written and consumed one at a time; there are few automated mechanisms for surveying or distributing them.

Because of this change in scale, we are faced with redefining information management; our approach is to do so in terms of what is *personally* salient instead of simply what is available for any given topic. Information retrieval—finding information of relevance—depends on what characteristics are remembered and considered important, which in turn depends on whom you ask [18] [79]. Intertwined in the traditional metrics of preci-

sion and recall [79] is the personal nature of the definition of relevance, which we claim needs to be captured both when information is filed away as well as when it is retrieved. Furthermore, computers need a better sense of the context of information requests in order to determine relevance [18], and the way in which context is characterized is also dependent on the user. To achieve these goals, computers must be made to manage metadata of all different forms and in person-specific ways.

Unfortunately, today's systems were not built to adapt themselves to personal needs; the rigid specifications to which information management software is designed today is a result of decades of engineering practices and brings about a handful of different kinds of problems. For example, assigning person-specific properties not envisioned by the database administrator or the software engineer such as "good music to listen to when I am in a bad mood" or "excellent sushi place for taking foreign guests" is difficult in most programs because schemas are often cast in stone by a compiler or database management system. Even when the needed relationships are defined ahead of time, metadata that specifies individually-relevant connections between objects in the same program or different programs, such as "Bob is Mary's brother" or "this report is relevant to next week's meeting", are similarly difficult to specify, since programs either fail to expose their object models to each other, or their object models are not fine grained enough. If computers cannot record connective metadata, information overload will eventually prevent users from taking full advantage of information technology [81].

In order to continue to advance our capacity to process information, we will argue that our computers must be able to store individually-relevant contextual and connective metadata. User interfaces must be created that take advantage of the user's context to present salient information to the user. Contextual information must be distributable to allow a user to benefit from the experience of his or her collaborators. **We claim that by using a semistructured data model, the system can capture the context and circumstances underlying information and not simply the information itself and thus help to elucidate the relevance of the information for the user and others.**

1.1 Approach

In this thesis, we advocate and justify several fundamental changes to the way information management systems and data repositories are designed. Programs must be able to

extend their schemas and abstractions at runtime in order to adapt to users' changing and unpredictable data modeling requirements. Data abstractions must be exposed at a fine enough level of granularity to allow users to express relationships between objects that exist in the same program or between programs. Moreover, programs must work with the abstractions of other programs and expose these abstractions to users by means of an intuitive user interface. In this thesis we advocate a semistructured, directed graph-based data model that is designed to more easily adapt to the ways in which users work with their information.

It is important to ensure that user interfaces expose this power properly. First, programs must make the processes of creating and revising semistructured information simple and intuitive. Similarly, users need tools that will allow them to characterize the significance of information to help them to retrieve it later [18]. When new forms of information are encountered, programs must be able to incorporate new functionality that will help users deal with such information. In addition, in today's networked world, knowledge workers must be able to utilize information originating from a variety of different sources and be able to distribute new information easily [82]. While today's operating environments provide tools such as file systems, e-mail, and web browser plug-ins for addressing these needs, the semistructured model enables the developer to expose a uniform experience to users because the solutions to the organization, distributed retrieval, and sharing problems can be incorporated into the base environment.

This radical change in the way information is described disrupts the traditional pathways by which user interfaces and data models are constructed. However, this new data model offers an opportunity to create new generalized abstractions that allow users to describe, navigate, and share their information in new ways. In addition to addressing the problems of expressiveness described earlier, these abstractions will enable a number of powerful, synergistic capabilities, such as the ability to model users' activities and to recommend information, organization, or other activities based on context, as we further demonstrate in this thesis.

In the process of supporting information management, the system is building a model of how users think about their information. As users customize their environments to express what information is important in different contexts or for different kinds of objects

or when users create custom relationship types or borrow existing ones from other areas of the system, they are implicitly recording *information expertise*. In many ways, information expertise represents the embodiment of generalized connective knowledge binding concepts together in a domain and can take many forms, including *ontology*, *contextual knowledge*, and *presentation knowledge*. Information expertise is an invaluable intangible asset that is difficult to capture in today's systems and deserves to be treated as a first class entity because of the role it plays in helping users to overcome the learning curve associated with entering new domains and expressing connective information in these domains.

1.2 Implementation

To test the efficacy of the semistructured data model and explore its consequences to the user interface, a system named Haystack has been built that employs a semistructured data model in order to permit users to store their information in a flexible fashion that adapts to their needs. Haystack coalesces previously segregated sources of the user's information and describes the data found in these sources using our rich description framework. Information on the Haystack project can be found online at http://haystack.lcs.mit.edu/.

Haystack was designed to be of practical use to the non-technical user. During construction we took advantage of standard components and languages in order to incorporate as many sources of information as possible. Haystack's semistructured data model is implemented using the Resource Description Framework [29], a descriptive logic framework for encoding metadata on the World Wide Web. RDF is a World Wide Web Consortium (W3C) recommendation, and its status as a standard has encouraged the development of a variety of different Internet-based metadata systems, including the widely-used RDF Site Summary (RSS), a news syndication format [33]. The key distinguishing feature of RDF is its use of Uniform Resource Identifiers (URIs) for identifying objects. These identifiers are globally unique and have well-defined semantics that persist when they are exchanged between systems. As Haystack is designed to support collaboration, this feature makes RDF a natural choice for Haystack's semistructured data model.

In this scenario, three fictitious colleagues, Charles, John, and Mary, are preparing to attend the HIV 2003 conference in San Francisco. In preparation for the trip, Charles begins by taking care of some e-mail, which appears on his Haystack home page in Figure 1.

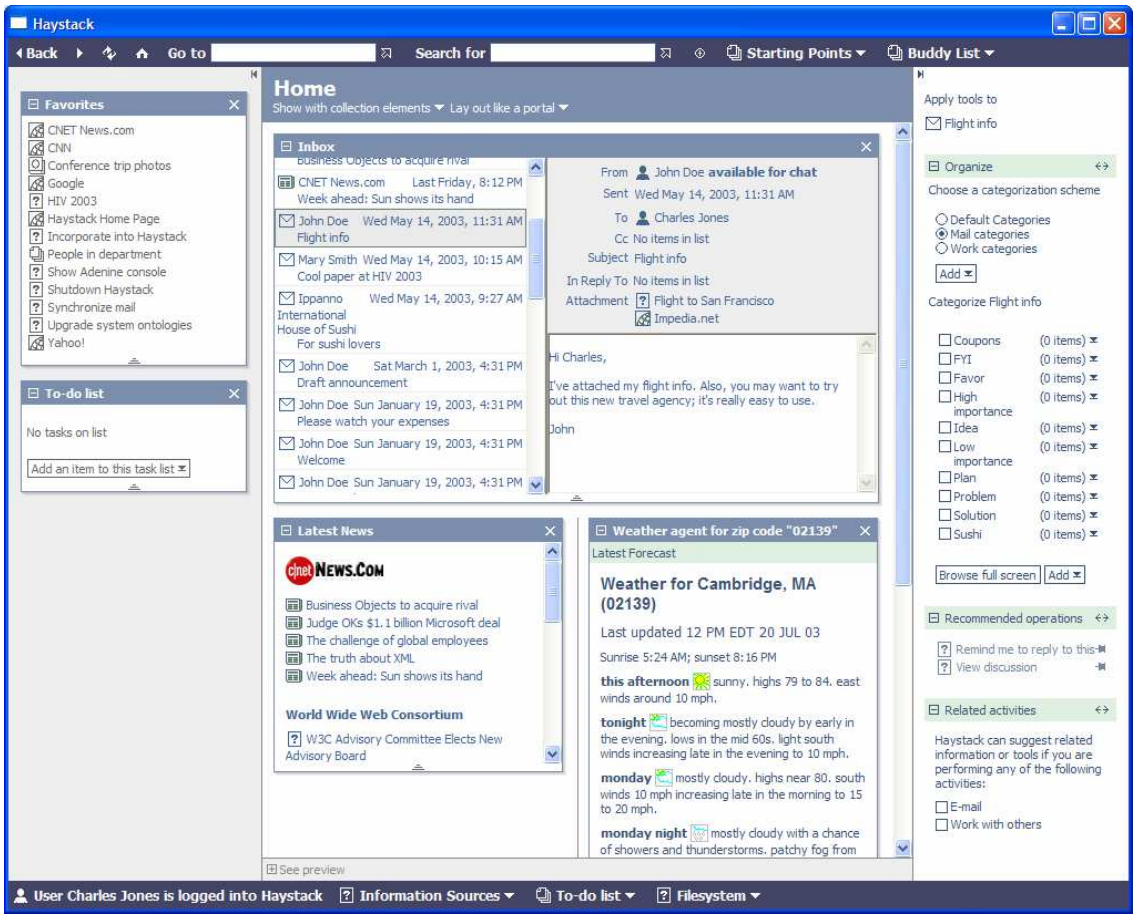

Figure 1: Haystack home page

The first message is from John, who has e-mailed Charles his flight itinerary and a bookmark to a travel agency he found useful. Charles drags and drops the itinerary to his to-do list on the left and the bookmark into his Favorites collection, also on the left. He does this without regard to the type of object he is dragging or the fact that the flight itinerary is not technically a "to-do item" to the computer; Haystack's data model is designed to be flexible enough for users' varying organizational patterns.

Next, Charles proceeds to look at an e-mail from Mary, who has recommended a paper to Charles (see Figure 2).

Figure 2: Message from Mary

Charles clicks on the paper's link, and a description of a research paper appears on the screen, as shown in Figure 3. The data for the research paper was retrieved automatically from a publicly-available LSID server (LSIDs will be discussed in Chapter 15). Haystack blurs the distinction between viewing locally-stored information and information from elsewhere on the Internet. Charles reads the abstract and thinks it would be interesting to see, so he files it into the "Stuff to see and do" category.

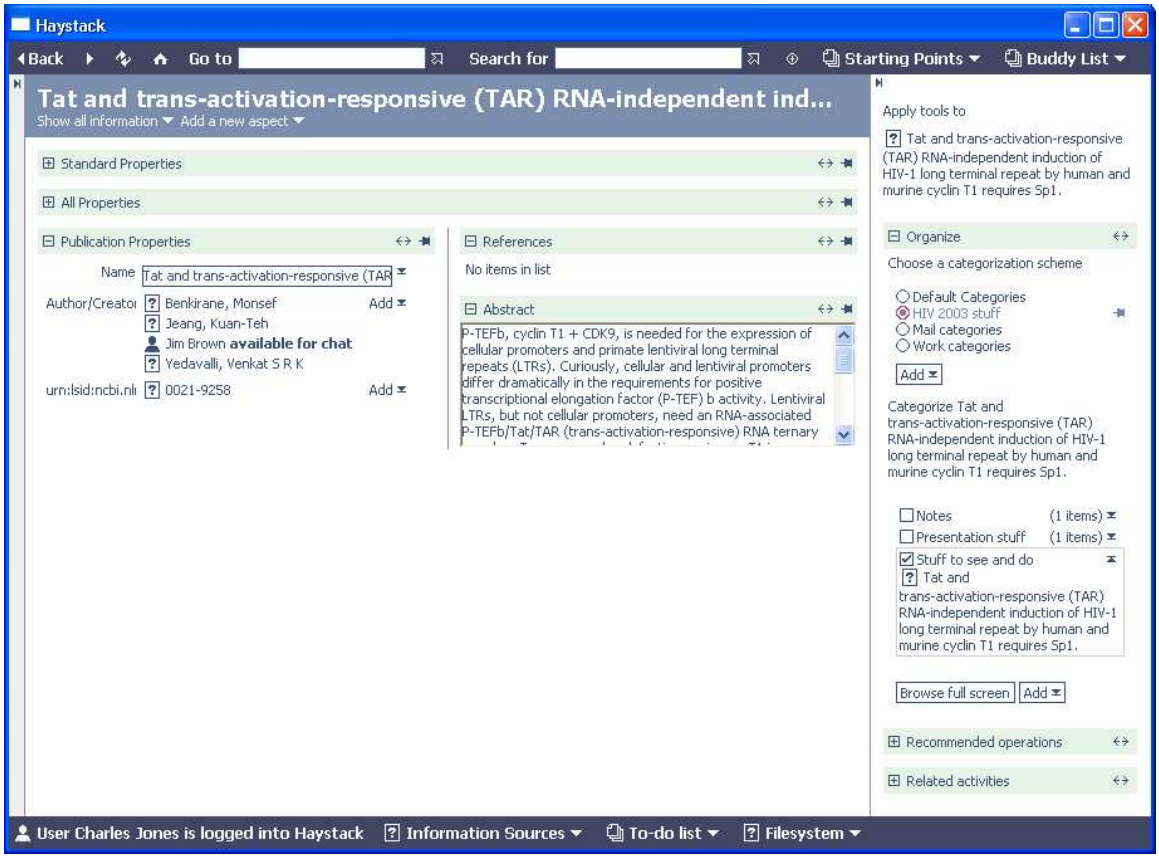

Figure 3: LSID-backed research paper description

Charles wants to see how this paper fits into the bigger picture, so he clicks on "Show all information" and switches the display into the "Explore relationships" view, as shown in Figure 4. Here, Charles can see not only the authors of the paper but also the collections into which this paper has been filed. This is an example of Haystack providing multiple views of the same underlying information, each useful for a different purpose. The data used to populate this display comes from a mixture of data on Charles's local Haystack and data from an LSID server.

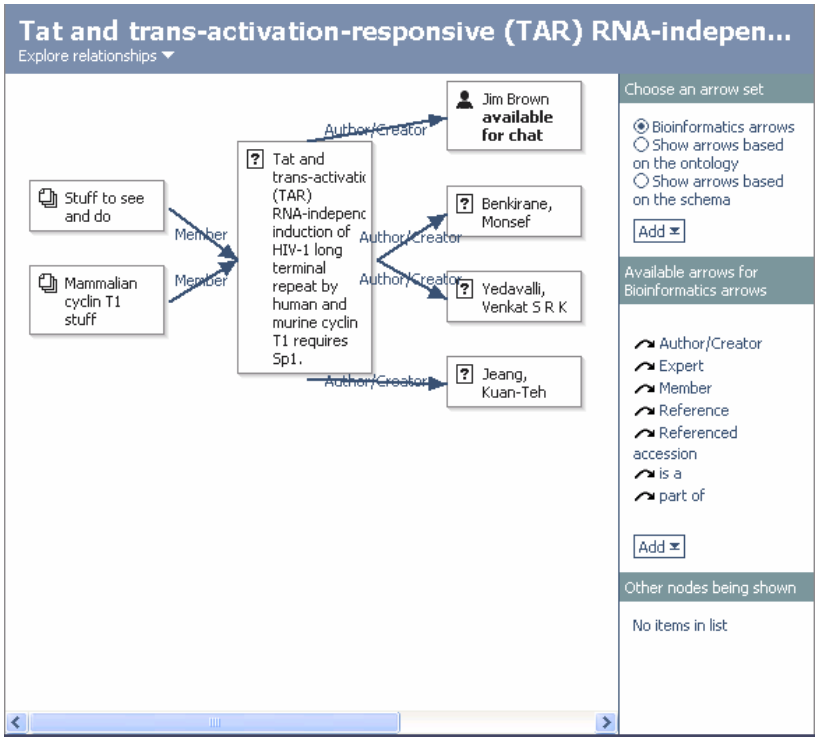

Figure 4: Research paper in relationship browsing view

The name Jim Brown sounds familiar to Charles. He clicks on Jim's icon to browse the display to a description of Jim, which is shown in Figure 5. Charles notices two other papers, named by LSID (but whose information has not been retrieved from the LSID server), authored by Jim. He also notices that Jim is in his buddy list.

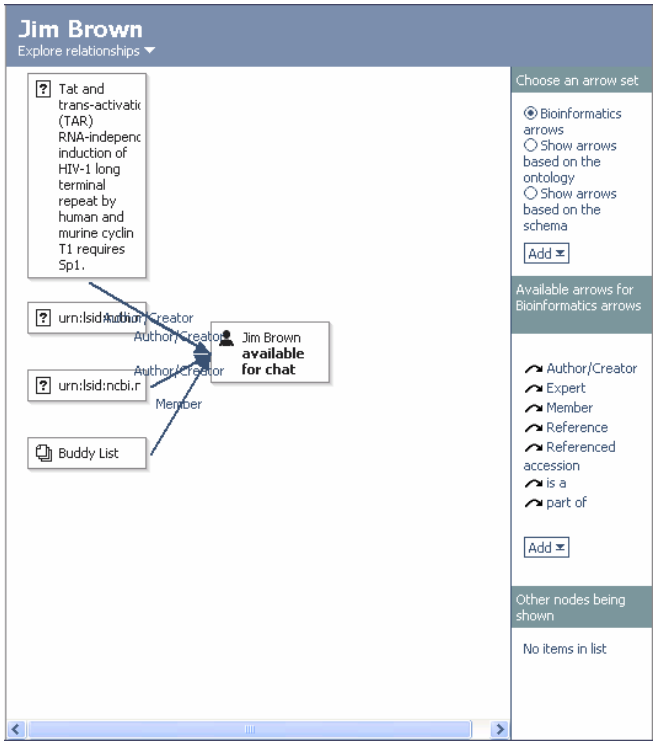

Figure 5: Person in relationship browsing view

Charles suddenly remembers that Jim was an old acquaintance of his and decides to get in touch with Jim. He right-clicks on Jim's icon and selects "Remind me to contact this party" from the context menu (see Figure 6). Charles goes back to finishing going through his e-mail.

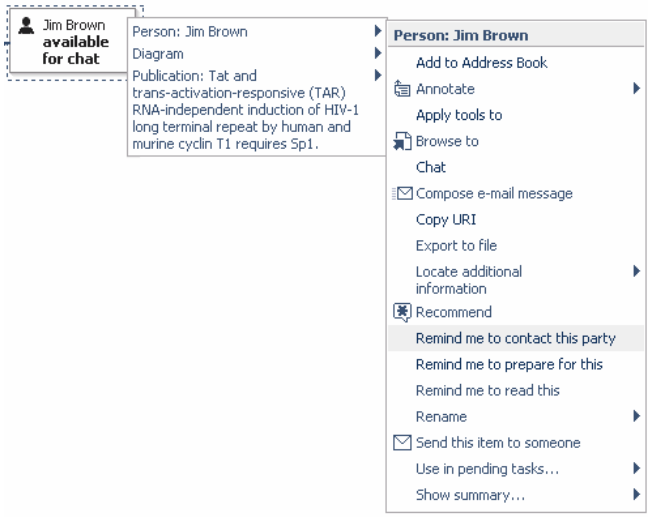

Figure 6: Adding a reminder to the to-do list

The next piece of e-mail turns out to be spam from a sushi restaurant chain. Charles is a sushi addict and files away the coupon attached to the message under both the "Coupon" category and the "Sushi" category (see Figure 7).

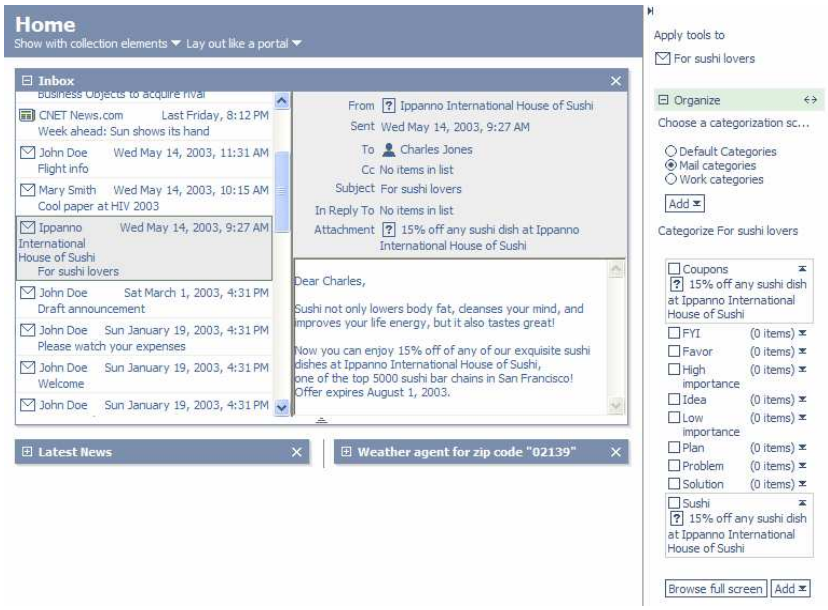

Figure 7: Spam

Here we can see that Haystack does not restrict the kinds of items that can be filed away or the number of places into which an item can be filed.

Charles now has a few things on his to-do list to take care of. He opens the flight itinerary on the to-do list to see what John has booked for himself (see Figure 8). Haystack displays basic information concerning the flight immediately; no custom program is needed to display this information, since Haystack can render arbitrary forms of information to the screen if no custom visualizations are present.

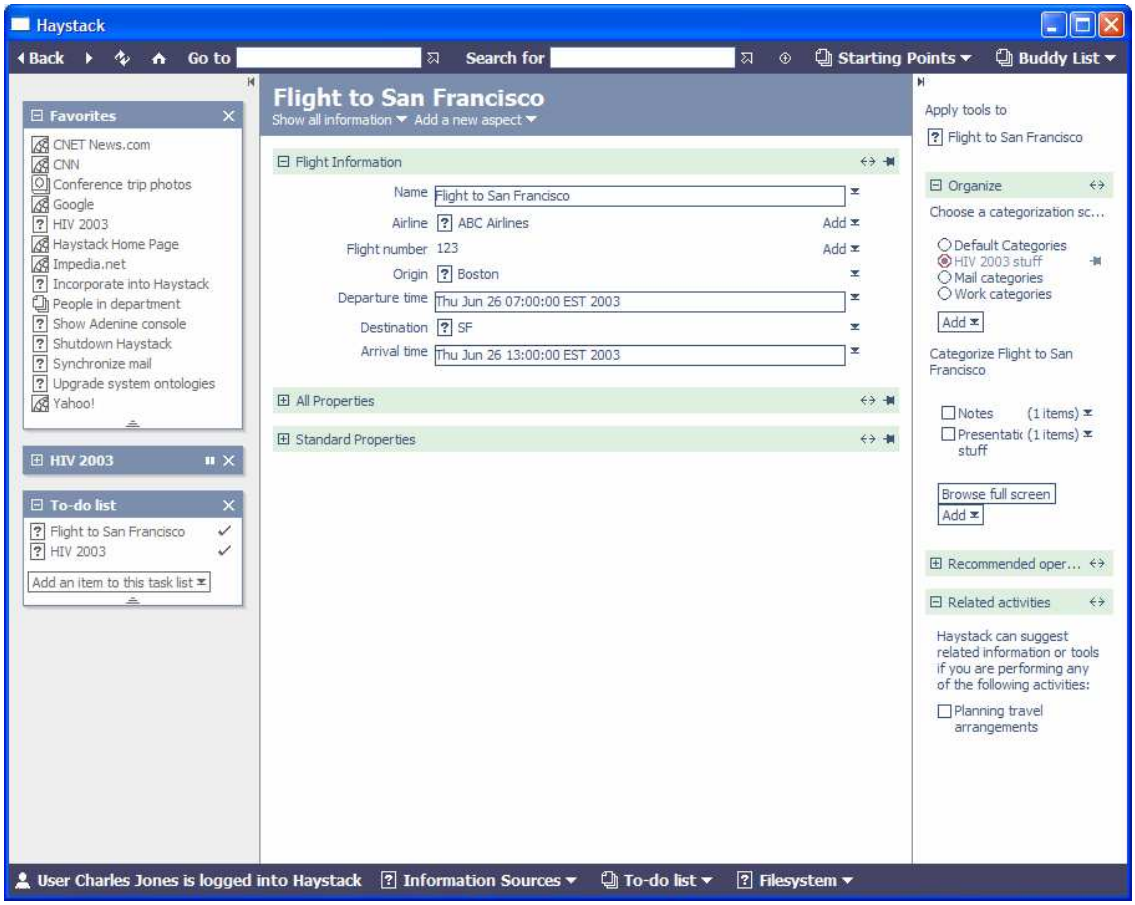

Figure 8: Flight itinerary

Haystack also informs Charles that it can present further information if he happens to be planning travel arrangements, which he is. Charles checks the box in the lower right hand corner, and Haystack displays two further pieces of information: the weather forecast for San Francisco, and a list of acquaintances that live in San Francisco. Note that by prompting the user for more contextual information, Haystack can reduce clutter on the screen and display more relevant, task-focused information and commands. For example, a weather forecast is useful for planning a trip but set of tools not when doing one's expense account report.

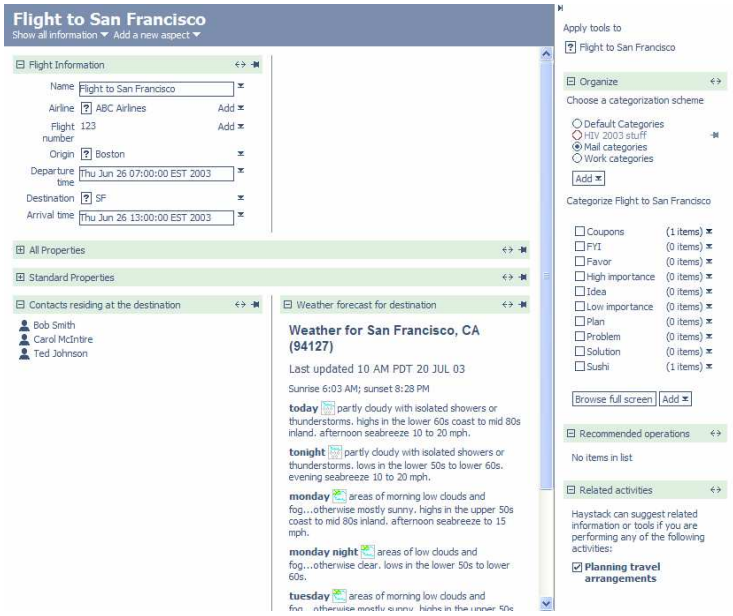

Figure 9: Flight itinerary with contextual information shown

Charles starts to look at how to book his own airplane ticket and clicks on the bookmark sent to him by John. The impedia.net web site appears (see Figure 10)[1].

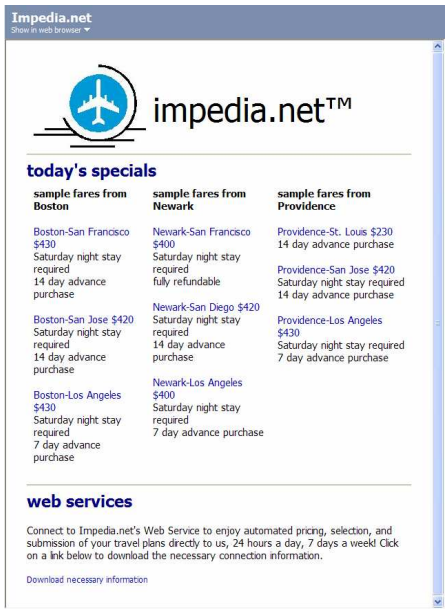

Figure 10: impedia.net web site

[1] impedia.net is completely fictitious and is a simulation of what a Web Services-powered travel agency might look like in the future.

Charles is not looking for a flight; he would prefer to be on the same flight as John. Nevertheless, many travel web sites would require John to copy and paste the flight information from John's itinerary into a new itinerary. Luckily, impedia.net has a Web Service that allows impedia.net to communicate directly with Charles's machine. He clicks on the "Download necessary information" link at the bottom of the page, and Haystack incorporates the connection information. Now, when he goes back to John's flight itinerary, a new command appears, "Book same flight as companion on impedia.net". John clicks on it and a dialog box appears on the left hand side.

Unlike dialog boxes in most applications, which are modal and restrict users to a limited for filling out the dialog box, Haystack treats the dialog box as a to-do item, which the user can fill out at his or her convenience using whatever tools are available in the system. The first piece of information needed is the person to bill. Charles opens his Favorites collection and clicks on "People in department", which brings up a list of people in the department (see Figure 11).

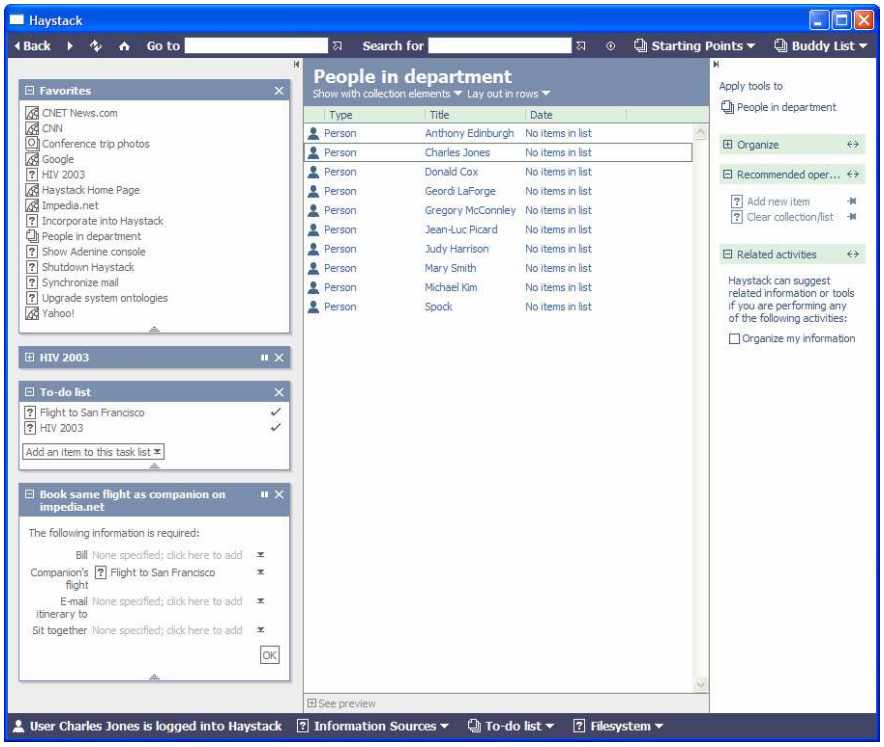

Figure 11: Listing of people in department

Charles needs to bill the ticket to his boss, but who his boss is is not evident from the list view, so Charles clicks on "Lay out in rows" (see Figure 12) and switches to an organizational chart view.

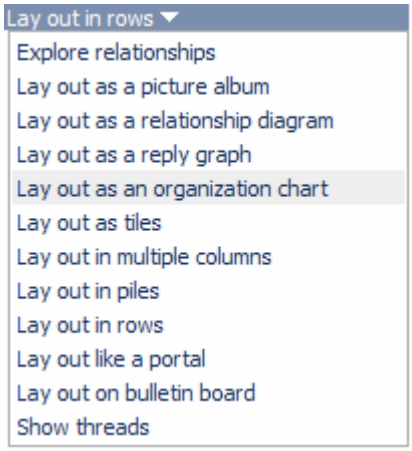

Figure 12: UI for changing view of collection

From the organizational chart view, it is evident that Judy Harrison is his boss; he then drags and drops her into the Bill field (see Figure 13).

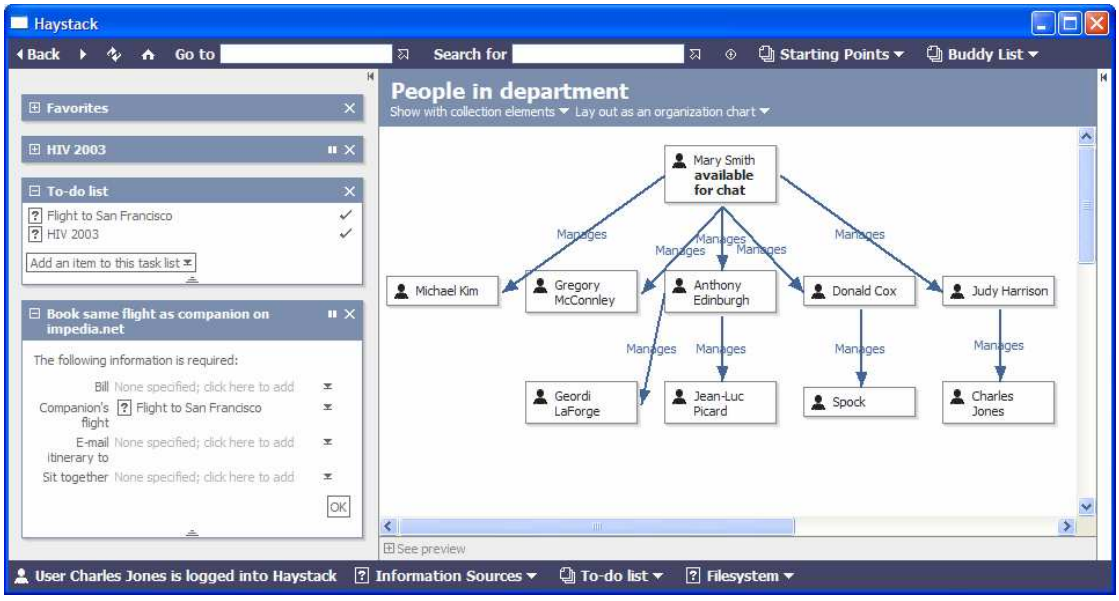

Figure 13: Department shown as an organizational chart

He types in his e-mail address into the "E-mail itinerary to" field and clicks OK. Impedia.net processes the request and sends Charles a receipt. Charles files this away in his "Stuff to see and do" category. He also creates a new category for "Expenses" and files the receipt away there too (see Figure 14).

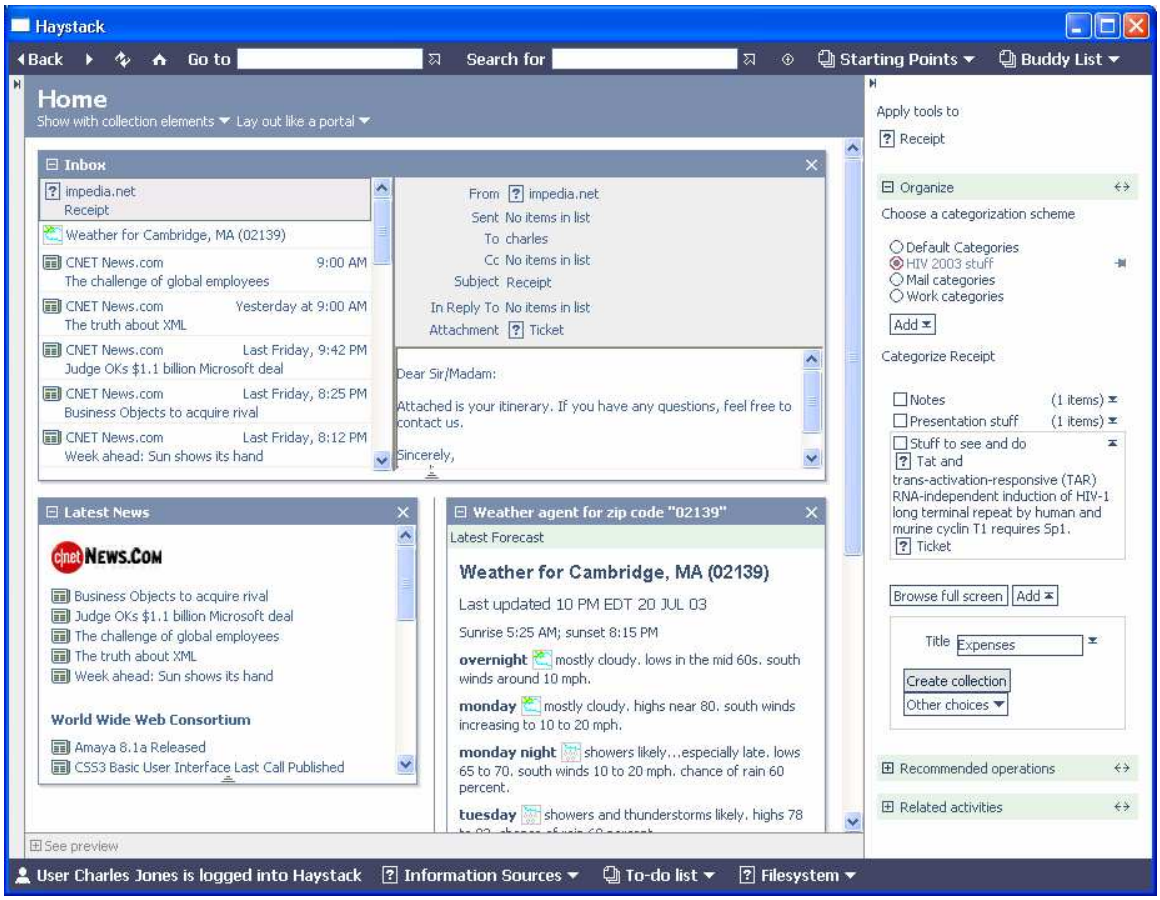

Figure 14: Filing away the ticket receipt

Charles has finished his travel arrangements without any reentry of existing data into the system—mostly via drag and drop. Now he turns his attention to contacting Jim Brown, which is his next task on the to-do list. When he clicks on the reminder task, Jim's information appears (see Figure 15). Besides showing his basic information, Haystack also indicates that Jim is online and available for chat. Charles clicks on the "Chat" button in the "Recommended operations" pane on the right.

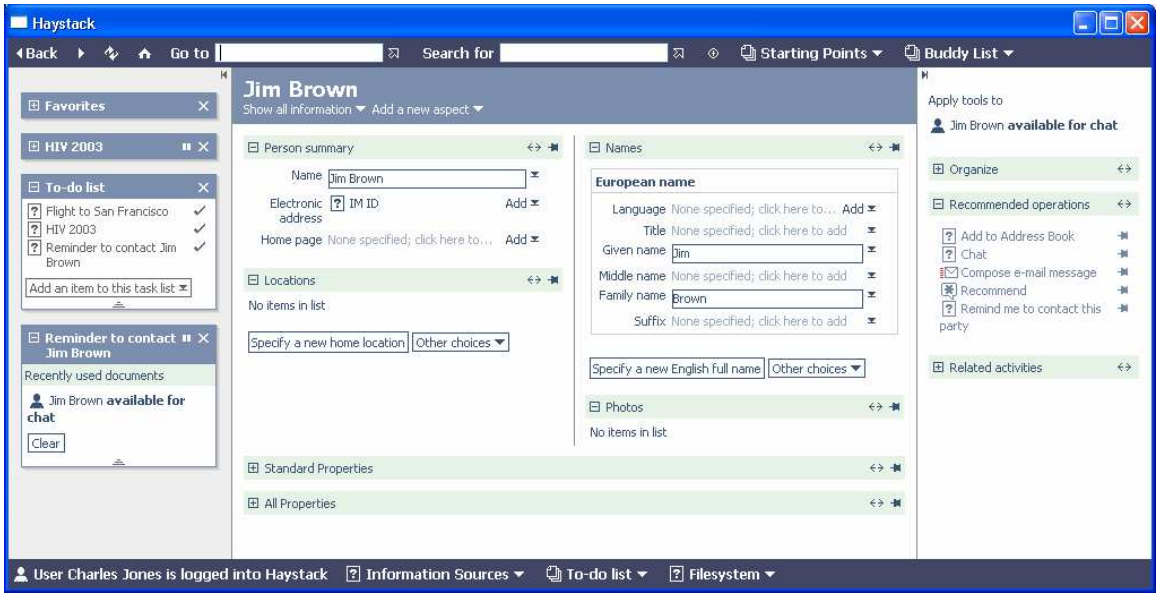

Figure 15: Information about Jim Brown

Charles says hi to Jim and asks him if he would like to have lunch together. Jim replies with a suggested time encapsulated as an appointment. To find out whether the time is good, Charles has to see what else he has planned to do. Luckily, he has filed these things away into the "Stuff to see and do" category. Charles right-clicks on the category and selects "Browse to"; this brings up the collection full screen as a list. He changes the view to "Show as a calendar" and sees his flight in the morning and the paper talk in the afternoon. Notice that viewing a collection as a calendar is simply a matter of changing the view; no special conversion is required. When he drags and drops the lunch appointment into the calendar, he sees it fits right in the middle (see Figure 16).

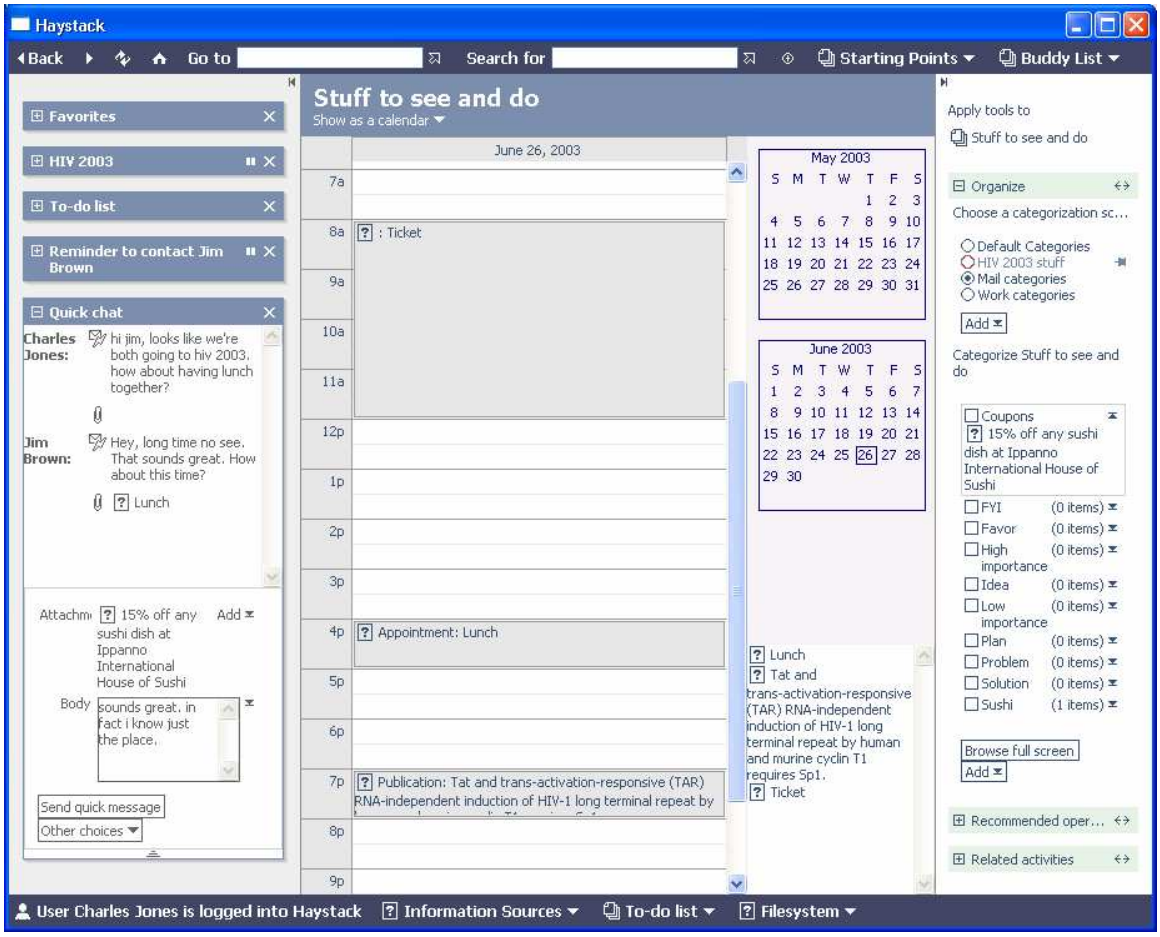

Figure 16: Coordinating lunch with Jim Brown

Charles replies that the time sounds good. Then it occurs to him that he has a coupon for a sushi place he is dying to try in San Francisco. He opens his coupons collection and drags and drops it into his chat window. Jim agrees but tells Charles that he has to run, so they say goodbye.

The key elements of the scenario—rich interaction with information, lack of application boundaries, putting things in context, and collaboration—are the themes that we have attempted to emphasize in creating the Haystack user experience. Haystack's ability to actualize these themes is a direct result of the many layers of infrastructure that are described in this thesis.

1.3 Contributions

The thesis will present eight major contributions and demonstrate their applications to Haystack.

First, we give an analysis of the infrastructure required to enable the aggregation of various information sources into one unified environment. The data layer must be capable of efficiently managing and exposing data in a variety of formats, styles, and schemas. Support for retrieving information from and sending information to other systems is needed, as are mechanisms for performing rudimentary data conversions, in order to maintain the virtualized data abstraction upon which the rest of the system is built.

Second, we present a pair of languages for describing the processing of semistructured data and presentation thereof onto the screen or another output device. The Adenine language reexamines the Lisp-like concepts of mixing data and *imperative* code in the context of the semistructured data model presented in the thesis. Ozone is an extensible *declarative* language that allows the developer to specify user interface presentation based on information in the underlying data model. Our programming model is based on a mixture of these two languages, which encourages paradigmatic use of the data model that abides by the three principles of schema extensibility, fine granularity, and shared object space discussed in Chapter 3.

Third, we describe a user interface paradigm that enables a system to dynamically present information based on the underlying semistructured data model. The Haystack user interface is designed to address the problem of adapting to customizable schemata and the user's need to view information in various presentation modes depending on the current context. At the heart of this component model is metadata, which is used to describe these components in terms of the types of information they can present, and the rendering engine, which assembles components at runtime.

Fourth, we specify a unified model for user interface operations, such as those accessible via menu items and toolbar buttons. We discuss the applications of Adenine to the mixing of code and metadata in the implementation of operations. In particular, we show how adding specific forms of metadata to Adenine code can enable developer-written functionality embedded within the code to be directly exposed to users in the user interface.

Fifth, we discuss several modalities for allowing the user to input semistructured information into and retrieve semistructured information from the system. These specific modalities serve specific needs of the user in terms of how he or she organizes, describes, and composes information, and they are meant to replace the simple graph- or tree-based approaches to managing semistructured data commonly found today. These modalities are also responsible for helping the user maintain his or her ontology.

Sixth, we discuss one specific but increasingly important application of this semistructured framework: modeling messaging. We describe an ontology and a framework by which e-mail, instant messaging, and newsgroups can be integrated and uniformly presented to the user.

Seventh, we give an ontology for describing tasks performed by the user that models concepts such as projects, activities, to-do items, and appointments. Tasks are associated with contexts, which are resources that characterize some state of the user, such as writing a paper, looking for information, or reading e-mail. We discuss how the user interface in Haystack can use knowledge of the current task and context to choose the most relevant information and commands to present to the user.

Eighth, we propose a mechanism for capturing information expertise from users by means of ontological proxies. These proxies record implicit knowledge about a domain from the user and materialize it in the data layer. Information expertise can then be exchanged between different users, allowing users who are more familiar with the processes of modeling information ontologically to share this knowledge with others. This functionality will build upon the messaging layer to help transport ontological and user interface metadata to other machines.

1.4 Thesis organization

The thesis discusses the principles underlying Haystack's design, starting with basic data modeling principles and user interface primitives and working towards higher level concepts of organization and collaboration. Finally, we present several applications of Haystack that demonstrate the power of the framework to solving problems ranging from bioinformatics data federation to organizing personal information.

Chapter 2 frames the problems discussed in this thesis and examines key related work in the literature, highlighting the differences between these previous approaches and Haystack as presented in this thesis. We also point out the previous research upon which Haystack is built.

Chapter 3 presents our semantic network-based data model, which embodies the three characteristics of common object space, fine-grained object model, and runtime schema customizability. We describe how RDF suits these requirements and how it is employed in Haystack. We also introduce some basic RDF vocabulary that is used throughout the system and this thesis.

Chapter 4 describes our implementation of the data model. We characterize the Haystack backend, which consists of a multi-agent environment with a shared RDF-based repository acting as a blackboard. Haystack supports a basic forward chaining mechanism, which can be used for limited forms of inference. We also discuss how the system manages a virtualized, distributed RDF data model in a practical sense with mechanisms for retrieving information about unknown resources and for extracting graph fragments for transmission to other systems. This work lays the foundation for permitting uniform access to all of a user's information.

Chapter 5 presents the environment used by developers for implementing functionality in the Haystack framework. We describe Adenine, Haystack's RDF-enabled imperative programming language, and Ozone, Haystack's RDF-enabled declarative user interface language.

Chapter 6 describes the Haystack user interface engine and the ontologies we have developed for describing presentation and aggregation of user interface components. Herein we make the case for dynamically producing the user interface at runtime in order to respond to changes in the user's data model.

Chapter 7 explains how we model information manipulation primitives within Haystack using a combination of RDF metadata and Adenine code. We present means for allowing the system to automatically generate commonly-used user interface elements, such as dialog boxes, menus, and toolbars.

Chapter 8 describes techniques for permitting users to manage collections—in essence, unary predicates—and how they play an important role in enabling users to aggregate objects, navigate through their information and manage taxonomies.

Chapter 9 describes techniques for managing more general forms of information. These methods are based on well-established paradigms for accepting information from users adapted to our semistructured data model, including form-based input and diagram editing. Also, annotation is presented as a pervasive feature in Haystack; this pervasiveness results from the granularity of our data model.

Chapter 10 discusses basic retrieval techniques, including metadata-based retrieval, natural language question answering, and associative recall.

Chapter 11 characterizes the concept of context, which allows information and operations to be aggregated according to the task at hand. This method of grouping functionality is in direct contrast to the application-based centralization that frequently occurs today. We discuss a basic task model and a means for exposing it to the user in terms of a to-do list paradigm.

Chapter 12 explores our messaging ontology, which enables collaboration by abstracting communication into discrete units called messages. We describe how existing messaging paradigms, whose usability and social characteristics have been well established, can be brought together into a common paradigm, typifying the transformation that occurs during the application of our data model to an information environment.

Chapter 13 assembles the various user interface-driven proxies for information expertise described throughout the thesis and shows how they, in conjunction with the messaging framework, can be applied to enable users to easily manage and exchange ways of presenting information, organizational schemes, contexts, and ontologies.

Chapter 14 demonstrates the application of Haystack's framework to the problem of managing personal information. Using Haystack's ability to handle custom schemas and taking collections as the fundamental tools for organizing information, we explore how different forms of information, ranging from calendars and travel itineraries to photo galleries and address books, can be modeled and managed in the same framework. We

also describe the results of a user study we have conducted to explore the effectiveness of our designs for actual users.

Chapter 15 gives one final demonstration of Haystack's framework to the problem of aggregating biological information. Bioinformatics is a particularly motivating example because of the abundance of disconnected, disparate information sources rich with ontologically-encoded, relationship-rich information in this space. We show how Haystack can provide seamless navigation of bioinformatics information sources.

Chapter 16 gives some concluding remarks on the limitations of our data model and suggestions for future work in the areas of security, privacy, automation, and collaboration.

Chapter 2 Problem space

In this chapter we wish to set the stage for a discussion of the fundamental principles underlying Haystack that follows in the ensuing fourteen chapters. We begin by reviewing evidence of the problems of information management that exist in today's systems as reported by the HCI literature, including the difficulties of archiving and remembering to do important tasks. We then explore some of the bigger picture psychological issues that have been used to characterize our problem space, which point to a fundamental need for associative, attribute-driven approaches. Several systems have attacked these issues by introducing new paradigms at the user interface and data model levels, and we compare these approaches to Haystack's. Finally, we discuss recent work that has served as the basis for the current Haystack implementation.

2.1 Evidence of the information management problem

The problems users face when managing their information on computers have been explored extensively in previous research. Because of the way in which information is split between multiple, disjoint applications today, studies have tended to focus on the problems experienced by users of each application separately. Despite the separation of the results of this research by system, common themes have, not surprisingly, arisen and point to common difficulties harbored by computer-based information management in general.

2.1.1 Managing documents on a file system

As the storage mechanism and rudimentary organization system used by most desktop productivity applications, the file system is an important environment to study in order to gain an understanding of what is not working for users in the area of desktop computing. One commonly cited study was performed by Barreau and Nardi [68] to determine the patterns of usage among DOS/Windows and Macintosh users. Their findings point to a tendency of users to adopt location-based finding, such as placement of documents on the desktop, as a reminding mechanism and to avoid archiving of information. While not disagreeing with their superficial findings, Fertig et al. took issue with drawing general conclusions from these observations on the topic of users' preferred information management paradigms [69]. They concluded that users avoid archiving because of its prohibitively difficult user interface and use location-based finding as a

best of evils—noting that location is simply one attribute that could be used when conceptually considering the problem to be one of generalized attribute-based search. Both papers emphasize the need of a system to provide reminding functionality, a form of task management, and better support for archiving.

2.1.2 E-mail

An increasingly important and consuming space of information management is e-mail, one of the most successful applications of Internet technology and the primary means for electronic communication today. However, two important studies have shown that e-mail has grown beyond being merely a form of communication and has gained unintended characteristics that make it similar to the file system, emphasizing the artificiality of the separation between e-mail and the file system in most computers.

A study by Whittaker et al. discussed how e-mail has grown in purpose to include functions such as task management and personal archiving or filing [32]. As was the case with the file system, e-mail inboxes are being used not only to remind users of important tasks but also to file things away for later retrieval. Whittaker et al. more explicitly found that users could be separated into three classes based on their filing patterns: those who filed frequently, those who filed occasionally ("spring cleaners"), and those who never filed. Despite the difference in behavior, all three kinds were identified as wanting to have more systemization but perceiving differing levels of impedance towards filing. Along these lines was the observation that folder creation was not regarded as a lightweight process.

Ducheneaut et al. more explicitly studied the use of e-mail as a personal information management tool and arrived at similar results [67]. They found that filing tended to occur by criteria such as sender, organization, project, and personal interests, i.e., important attributes of messages by which users found it useful to search. Filing structures (i.e. archives) were found to be relatively shallow, perhaps for the same reasons found by Barreau et al.

2.1.3 Bookmark collections

With the advent of the World Wide Web, publicly available information viewable through a web browser constitutes a great deal of the information we deal with on a day-to-day basis. The most common mechanism for filing online information is the

bookmarks support embedded within web browsers, which in turn makes it another place in which to organize personally relevant information. A study performed by Abrams et al. found a number of similarities between the problems experienced by web browser users and users of e-mail and the file system [1]. Bookmark organization was found to be restricted by usability problems, such as the inability to see all of the bookmarks of importance on the screen at once. Participants of the study noted the desire to create and file bookmarks with considerably less effort than what is required now as well as to expose bookmark collections to other machines.

2.2 Underlying psychological problems

Amidst the common themes of archiving, task management, and collaboration are the more fundamental psychological needs of users in working with information. In some sense, our current information tools were designed to mirror common paradigms that exist in the physical world, such as filing cabinets and desktops. However, these paradigms bring with them limitations for supporting our ever increasing information base. Vannevar Bush was one of the first to propose that technology—one of the causes of information explosion—should be used to address this very problem [81]. Bush posited that the growth of information was, even in his day, easily outstripping people's ability to consume it. He suggested that since the human mind processes information associatively, machines should be designed to store and retrieve information in terms of associations.

Forty years later, despite the fact that the technology needed to implement Bush's "memex" machine had come into existence, information technology had yet to bring his vision to life. Lansdale examined the failures of information technology from the perspective of the psychological processes in play during the archiving and retrieval of information [18]. He, like Bush, noted the failure of arbitrary indexing schemes (such as filenames) to effectively assist those in search of information. Fundamentally, retrieval begins with identifying remembered qualities of the items being sought, which may not coincide with the particular attributes designated to be indexed.

Furthermore, the decision processes embedded within the use of tools such as folder hierarchies (e.g., the file system, e-mail folders, and bookmark collections) were noted as being inconsistent with the human thought process. Good examples of where this prob-

lem arises are automated telephone menu systems or complex taxonomies in which items exist in only one place in the hierarchy. Lansdale reported that the fact that people were choosing the wrong path navigating down a hierarchy is a direct result of the need for context to be present for disambiguation (e.g. table is easier to define when put in the context of the phrase "he put the plate on the table"). This observation may explain why the participants of the studies given in the previous sections avoided deep hierarchies whenever possible.

Finally, Shneiderman reports on another fundamental endeavor of information management—fostering creativity [82]. He asserts that creativity is a concept that directly affects the quality of work in fields ranging from education and research to professional work done by lawyers and designers. In his paper Shneiderman reflects on three kinds of creativity, inspirationalism (creativity derived from sparks of insight), structuralism (exhaustive and orderly searches), and situationalism (creativity derived from intellectual and social surroundings), and from these he discusses ways in which information technology can help foster each kind. Of note are his thoughts on the usefulness of data visualization and exploration tools (e.g. Spotfire, http://www.spotfire.com/), free association brainstorming (e.g. MindManager, http://www.mindman.com/), and templates that guide users through synthetic processes (e.g. Microsoft PowerPoint AutoContent Wizard) for enhancing creative processes. He also proposed a framework called "genex" composed of four steps: collecting, relating, creating, and donating. If we evaluate the progress of information technology today against these steps, we find that collecting and relating echo the themes of archiving and task context management, but creating and donating are less prevalently explored.

2.3 Previous information management environments

The research shows that users should benefit from more flexible tools for connecting, organizing, browsing, and searching their information. Past research has produced a number of prototype systems that attempt to address some or all of these issues in terms of how information management should be supported on computers. In this section we identify the strategies that were successful for these systems and compare them to the approach we present in the remainder of the thesis.

2.3.1 Presto

Presto is a system developed at Xerox PARC for managing documents in terms of a set of user-customizable attributes [9]. The basic idea is that attributes such as "Word file", "published paper", or "currently in progress" carry user-specified meanings and can be used to capture the notion of documents playing different roles. The Presto data layer was designed to efficiently manage document attribute sets and to allow incorporation of attributes from other sources such as the file system through an extensible service framework. Furthermore, in contrast to the hierarchical folder system, Presto supports objects called collections that aggregate documents together based on common attribute values. A user interface called Vista was constructed on top of Presto to explore the paradigms needed to expose attribute-based document management to users. Overall, Presto represents one of the biggest and influential efforts in the study of allowing users to customize their organization beyond what is possible with strict folder hierarchies and has significantly influenced the design of the Haystack system.

While overcoming many of the problems with folder-based management of documents, a number of issues relating to the management of information in general are not addressed. First, one could argue that documents are merely encapsulations of information and that the entities being described by such information are relevant and useful to expose as well. For example, one way to describe an encyclopedia is as a collection of articles (articles are documents). When a user incorporates a second or third encyclopedia or other information source, he or she simply gains more articles, some of which may be about the same topic. On the other hand, if other kinds of objects (e.g., topics, places, or concepts) were considered first class, then the user could have an object (e.g. the place "Stonehenge") that would act as a centralized holding place for information on that object, including articles on the subject, custom annotations, and noted historians.

Second, concepts such as attributes or user interface displays are not first class, limiting the kinds of collections that can be created. A set of attributes that a user feels can be used to characterize a certain kind of object with some level of completeness is a valuable piece of information that would be useful to share with others. This idea is further motivated and explored in the context of Haystack in Chapter 13.

Finally, we argue from a usability perspective that a custom application separate from the applications used for creating information (e.g., Microsoft Word) is not sufficient for encouraging users to organize their information. As was pointed out earlier, if organization tools are not conveniently present when a user wishes to file something away (i.e., in this environment, adding attributes), the user will be discouraged from doing so. With this issue comes the problem of adding more sophisticated manipulation capabilities, such as those present in Microsoft Office, although it is clear that such work was beyond the scope of the Presto research.

2.3.2 Lotus Agenda

Lotus Agenda was a text-mode product developed by Lotus over a decade ago for managing semistructured databases [84]. The underlying philosophy governing the system's design was that the concepts of freeform textual data, structural evolution, and multiple user interface views were important features missing from conventional database management systems. Agenda was ahead of its time in its purpose of adapting database management techniques to the more fluid, changing needs of real users. The system managed what was referred to as "item/category" databases, in which the concepts of database row and column could be interchanged depending on what is needed when the data is being viewed. Items were fragments of text, which could be aggregated in a tabular fashion into databases. Multiple views, consisting of selections of what "categories" were relevant, could be used to browse and input new information. Databases could be easily searched and sent between users.

One conscious decision embodied by Agenda's design is the choice of not representing first class objects. In other words, Agenda manages semistructured collections of text, not objects. The fact that multiple text notes may reference the same object is apparent only upon noticing that the same or similar sequence of characters appears in the notes. While abstracting away the problem of mapping names onto objects, this approach affects expressiveness in the following way: The connections between objects cannot be explicitly specified; these connections can only be made through the existence of common fragments of text.

There are also interesting questions in the area of user interface customization, which were not possible to address in Agenda because of its text-mode presentation. Similarly, collaboration over the Internet was not as prevalent at that time as it is currently.

2.3.3 Lifestreams

The Lifestreams project produced an information management environment designed to automatically archive documents in time-demarcated structures called "lifestreams" [88]. The system supported custom organization by allowing users to create their own streams, and the reminding problem was addressed by exposing functionality to send a message to yourself in the future, i.e., placing a notification into a stream with a future time stamp. A substream can be created by specifying a query that filters elements from another lifestream. Furthermore, the system supported the notion of multiple viewers, abstracting presentation and platform from the underlying data model. The core notion was that a lifestream was a better abstraction for grouping objects that fit the needs of users.

Comparing Lifestreams to the approach we have adopted, we acknowledge that time is a useful metaphor for working with sets of documents, but we find no particular reason for focusing our system around it. Indeed, other metadata, as Lansdale pointed out, may be equally suitable for use in structuring a user's system of organization. We feel that this choice should be left to the user, and the system should be flexible enough to allow multiple paradigms.

Additionally, like Presto and Agenda, Lifestreams is a separate application for archiving, and as we noted earlier, could inhibit its use if Lifestreams is to be employed for specifying classification metadata in addition to simply retrieving documents based on existing metadata.

2.3.4 Dynamic Windows

Dynamic Windows is the user interface architecture of the Symbolics Genera operating system [24]. This architecture possesses a visual component model and builds presentations recursively from components designated as capable of displaying objects of specific types and in context-dependent ways. Presentations of objects on screen are interactive and can be used as proxies for performing manipulations to those objects. In particular, Dynamic Windows models user interface operations that take parameters of spe-

cific types, and on screen presentations can be selected for use as parameters. Because it is embedded within Common Lisp, which possesses extremely flexible data representation capabilities and support for first class procedures, components are represented as procedures that assemble primitives (e.g., text, rectangles, etc.) and retrieve data via adapters, which are similarly procedures that provide access to application-specific data models. As a result, it is easy to create a user interface on top of any Lisp-based data model using this environment. Haystack's user interface framework shares many of the notions discussed above.

The fact that Dynamic Windows makes few assumptions about the underlying data model has both advantages and disadvantages. While adapters provide the ability to abstracting user interface from presentation, we feel that they also promote heterogeneity in the representation of information, which may serve to create artificial boundaries within a user's information corpus. Also, the dependence of Dynamic Windows on Lisp, while theoretically providing a solid foundation for any information environment, may turn out to be impractical in a world of applications built on top of conventional object-oriented languages such as Java. Interfacing a flexible data model with popular languages less expressive than Lisp, such as Java, is a key challenge we address in this thesis.

2.3.5 Oval

Oval, an experimental collaborative system, was built upon the notion that the basic elements of objects, views, agents, and links (hence the acronym "oval") can be assembled by end users and are sufficient for building rich information environments [20]. Oval's object model embodies the familiar characteristics of classes and inheritance, and this object model can be customized by the end user without programming. Views, such as forms and graph diagrams, are automatically constructed based on the data in the system and the object model. Agents are sets of if-then rules that respond to changes in the data model. Finally, links allow objects to be connected together in arbitrary ways.

In some sense, while demonstrating the power of exposing an object-oriented model to users, the Oval interface exposes the data model in a somewhat raw fashion. One of our objectives is to provide both developer- and user-level support for encapsulating details of the object model into user interface elements, enabling the underling data abstraction

to grow arbitrarily complicated without affecting usability; we discuss this concept further in Section 6.3. By providing a framework for building more sophisticated forms of user interface "scaffolding" on top of an Oval-like data model, we believe that users can gain the power of the data model while enjoying the level of usability exposed by modern desktop productivity applications.

2.3.6 Microsoft Outlook and Lotus Notes

Two systems, Microsoft Outlook and Lotus Notes, are popular personal information management tools in use today and deserve mention here. The basic idea behind both systems is a flexible, object-oriented database with support for constructing forms and other user interfaces provided on top. Their strengths lie in the fact that both have created highly specialized ontologies and user interfaces for describing commonly-used desktop productivity scenarios, such as address book management, calendaring, and e-mail. Furthermore, paradigms such as drag and drop and context menus are used to provide a high level of interactivity to users of these systems.

Despite the flexible nature of their underlying repositories, both systems fail to expose enough of this power to the user. Creating first class links between objects, as can be done in Oval, is not well supported in the user interfaces of these systems. Links tend to exist as embedded objects within text fields rather than as properties of documents; this is borne out by the fact that fields such as "Manager's name" exist in Outlook as a place to hold the string representing the manager's name rather than "Manager", a first class slot for a person object. Additionally, user interface customization is left mainly to administrators; the functionality for customizing a template to include custom fields is not exposed in the same intuitive fashion as that for general use of the system. Finally, although both systems support more flexible forms of organization, they expose what are essentially folder-based mechanisms for filing documents. Even though a conscientious effort was made to expose improved archiving functionality in Lotus Notes (such as documents being filed into more than one collection), the e-mail study performed Whittaker et al. discussed above (which was performed on Lotus Notes users) still shows deficiencies in the area of organization.

2.4 Foundations for Haystack

In developing Haystack, we have borrowed and combined ideas from many of the systems discussed above. In addition, the implementation of Haystack as described in this thesis builds explicitly upon the Semantic Web project as well as on previous incarnations of Haystack.

2.4.1 Semantic Web

The Semantic Web project at W3C [4], like Haystack, is attacking the data model problem from the viewpoint of interchangeability. The focus of the Semantic Web effort is to proliferate RDF-formatted metadata throughout the Internet in much the same fashion that HTML has been proliferated by the popularity of web browsers. By building agents that are capable of consuming RDF, data from multiple sources can be combined in ways that are presently impractical. The simplest examples involve resolving scheduling problems between different systems running different calendaring servers but both speaking RDF. A more complex example is one where a potential car buyer can make automated comparisons of different cars showcased on vendors' web sites because the car data is in RDF. Haystack is designed to work within the framework of the Semantic Web. However, our focus is on aggregating data from users' lives as well as from the Semantic Web into a personalized repository, whereas the Semantic Web focuses more on agent-to-agent interaction.

RDF itself derives from research done on semantic networks dating back almost 40 years [27]. Semantic networks model information by labeling objects of interest and the relationships between them with English words [31]. The end goal was to create a representation that could model natural language-expressible concepts. The network model has been used often because of the way in which incrementally-accumulated knowledge can be represented in a manageable form.

2.4.2 Previous version of Haystack

The Haystack project dates back to 1997. The original ideas of aggregating document metadata into a common information repository can be traced through several generations of prototypes. To manage and populate this repository, the basic notions of a semantic network data representation and agents still persist into our current version [2].

The version of Haystack described in this thesis represents an almost complete reimplementation. We have abandoned our custom semantic network data model in favor of RDF in an effort to broaden our base of possible sources of metadata and to be more compatible with other systems. Also, the previous version sported a web-based interface with hyperlinks for traversing relationships between objects. However, we found this interface too impoverished to support the forms of navigation and manipulation explored in this these. As a result, we have for the first time in the history of the project incorporated a sophisticated user interface for creating and manipulating the metadata managed by the system.

Chapter 3 Data model

In this chapter we motivate and explain the three characteristics of the Haystack data model: extensible schemas, fine levels of granularity, and a shared object space. We show how these three characteristics can address many of the expressiveness problems inherent in current systems. During our explanation we make several comparisons to commonly used data modeling paradigms such as the object-oriented paradigm and the relational database model. Fundamentally, creating a clear partition between our hybrid data model and other data modeling paradigms is difficult; instead, we wish to emphasize our rationale for selecting the successful aspects of these various paradigms. We also address several issues that arise when fitting our semistructured model into real systems that include extant structured and unstructured information originating from different sources. Finally, we describe our adoption of RDF, a standard semistructured representation language that embodies our three key characteristics.

3.1 Extensible schemas

In Chapter 1 we posited that a huge inhibitor to the advancement of computers as repositories of human-generated information was a lack of expressiveness in the user interface. Being able to record different relationships between objects that exist at arbitrary layers of abstraction not only allows users to use these relationships to find information but also provides valuable contextual information that may be of use to other areas of the system, allowing different parts of the system to benefit from knowledge gained or maintained by others.

The inability to capture various forms of associative knowledge stems from the fact that most programs' user-level data structures are ill-suited to storing custom annotations and relationships expressed by the user. Under the covers, most software is written to a fixed ontology, i.e., the relationships between objects are predetermined. This is partially the result of the fact that most programming environments, such as C++, Java and relational databases, were designed to encourage developers to write programs in this fashion. The concepts of abstraction barriers and encapsulation are useful for implementation-level object model development but can be a hindrance to user-level data model customization when these concepts are used too freely to develop user-level object models. While it is possible to write software that supports end user customization of the

user-level ontology, doing so requires custom abstractions to be designed with little or no assistance from the programming environment[2]. As a result, the implementation-level and user-level object models both end up being rigid and not customizable at run-time.

Instead, we propose that programs maintain a *semistructured* data model, i.e., a data model in which the ontology in use can be extended dynamically. In other words, new relationships can be defined and established between objects at runtime. A labeled directed graph representation can be used to describe and in fact also to naïvely visualize a semistructured data model. Herein, nodes in the graph represent objects in the data model, and arcs represent directed relationships between objects. In contrast, traditional, fixed-ontology data models correspond to a picture in which only certain arcs can appear between certain objects, depending on the predefined schema in play.

Furthermore, we specify that our data model allows the ontology to remain implicit, i.e., no explicit specification of the associations supported by the ontology needs to be created. This is an important difference between our data model and what would otherwise be considered a relational database model [89], which mandates that the schema at least be explicit and be updated any time a change occurs.

Indeed, users can use associations to model a variety of different kinds of information. Traditional notions of fields in classes in the object-oriented paradigm correspond to names of arcs. Classifications of objects into categories are examples of a membership relationship that exists between objects and categories. Of course relationships such as "parent/child" or "manager/employee" are expressed naturally as arcs between nodes.

In addition, custom associations can be coined (but do not need to be documented) to suit the purposes of the user. Some properties will be defined *ad hoc* to describe user-specified characteristics. Of course, when the average user defines a property, he or she may not know how to create the necessary type specification required for a schema, and the system will need to work properly without such information. On the other hand, experts will be able to create property vocabularies and distribute them to other users.

---

[2] Haystack is an example of such a system, but the customizability that arises from carefully-tailored abstractions built on a traditional programming environment is taken as baseline support for programs developed on top.

Our data model enables systems to incorporate such vocabularies easily without requiring the data model to be restructured to accommodate the changes.

Schema extensibility means that the list of available associations can also be extended by programs or components of the environment for their own use *de facto*. For example, a newly-added user interface component may require special attributes to be placed on certain objects to enable them to be rendered to the screen in a new way. New components and the additional metadata they require can be added to the data model at any point.

3.2 Fine granularity

Being able to describe custom user-specified or program-specified relationships is only feasible if the objects involved in the description can be referenced. As a result, it is important that all annotatable objects in the system exist as nodes in the graph in order to fulfill the goal of making systems capable of capturing associative information. As opposed to the object-oriented paradigm, which encourages objects to hide sub-objects within their abstraction barriers, the semistructured data model encourages exposing these sub-objects if users or programs will be interested in annotating them. In other words, objects' substructure should be described in terms of nodes and arcs in the greater directed graph. In fact, entities that are not typically addressable, such as class declarations, property declarations, or even the basic unit of information—the arc connecting two nodes—are describable in our model; this notion allows descriptions of both normal resources and "meta" resources to share a common language.

Every knowledge representation makes certain assumptions about the level of granularity of the resources to be represented [55]. For example, for the purposes of creating a personal inventory of the books one owns, one could assign one unique identifier to each book. On the other hand, for a large library, such a limited representation may be insufficient. A librarian may need to model books with a higher level of granularity: different identifiers may be given to a book (the general concept), a specific edition of that book (a specific version), and a specific copy of a specific edition of that book (a rendition or instance of a general concept). The reason for giving names to such specific notions is so that we can be more precise in describing the relationships between these notions. Of course, it is possible to over-represent any system; in the book example, we

could model the individual atoms comprising the pages of the books. We adopt the principle that only when individual notions need to be referred to should they be isolated. (The limitations of this principle are discussed in Section 16.2.2.)

Obviously, abstraction barriers have been used in the past with good reasons in mind. Maintaining a proper abstraction allows the implementation to be changed without affecting clients of an object. While abstractions in our data model cannot be enforced, they can be encouraged or "implied" in the definition of custom predicates and types, i.e., use of a certain custom predicate defined some party can be implicitly specified to have certain semantics (e.g., by human-readable documentation) and be enforced by convention. The namespace mechanism used for separating different sets of custom predicates will be discussed later. Furthermore, methods can still be defined for accessing objects; this will be discussed in Section 5.3. In general, we believe that the benefits of the higher level of expressiveness of an open data model exceed the loss of protection afforded by traditional abstraction-based models, as will be demonstrated throughout this thesis. (On the user's end, views, which are discussed in Section 6.3, in some sense act as abstraction barriers to the underlying database.)

3.3 Shared object space

Because information is represented in terms of its basic elements, the possibility for reuse is improved. Key objects in a program's representation such as people and appointments can be built up from smaller objects, such as names and times. While programs already tend to reuse these basic elements individually, there is very little systemwide reuse of components. We argue that the best way to truly achieve object reuse throughout the entire system is to remove the barriers between objects of different programs.

Our semistructured data model allows information managed by several programs to be stored in a common representation. In the current data paradigm, the basic unit of information is the file, stored on a file system. These files exist in a variety of different formats and are created by myriad of different programs. Segmentation of data into files of different formats makes it difficult, if not impossible, to search for the answers to questions where the necessary information spans multiple sources. The information needed to answer the question "how do I get to the meeting set up by my secretary on the request of the person in HR I met with last Tuesday?" is currently storable in today's

computer systems: information on meetings and their coordinators is stored in group calendaring systems; the identity of people's secretaries and the knowledge of who is in Human Resources is contained in enterprise directories; map information and directions are readily available on the Internet. The problem with answering the request is the need of the computer agent to query essentially isolated systems that speak a myriad of distinct languages and protocols.

As a result of the segregation of data into different formats and inconsistent object models, programs have become "ivory towers", storing information in ways incomprehensible—not to mention unparseable—to other programs. Expressing information in semistructured form assuages this problem because our data model contains enough expressiveness to allow objects to be described in whatever form is necessary for each application, making it easier for applications to adopt common representations for shared concepts. Instead of casting information into discrete buckets of data conforming to a fixed format, our paradigm encourages programmers to holistically recognize that there are a number of objects in the entire system (e.g., "meeting", "my secretary", "person in HR", etc.) with relationships between them (e.g., "set up by", "on the request of", etc.), where information about each relationship is provided by a separate program. In other words, our data model encourages a certain level of "syntactic" consistency that defines the underlying concepts of object and relationship at the ground level, providing a firmer basis for inter-application interoperability.

This concept of permitting distributed queries across multiple data sources has been the subject of recent research in the database community [83]. However, unlike traditional relational database environments, where users set up separate tables to describe new relationships and attributes, semistructured data stores can hold information without needing a schema to be specified first; users work with the "sloppy", federated representation the whole time.

3.4 Resource Description Framework

Haystack has adopted the Resource Description Framework (RDF) as a means for describing information according to our data model. RDF was designed in order to enable programs and agents on the Semantic Web to exchange information concerning a variety of different topics smoothly and efficiently. Because of its status as an Internet standard,

RDF has been adopted as the basis for Haystack's data model. From a practical standpoint, our adoption of RDF enables us to take advantage of the numerous forms of information encoded in RDF at present and in the future.

Below we briefly introduce the RDF concepts important to Haystack and describe the notation and vocabulary that will be employed in the remainder of the thesis as well as the correspondence between RDF concepts and the principles in our data model.

3.4.1 Resources and literals

There are two kinds of nodes in an RDF graph: resources and literals. A **resource** is a thing to which we want to refer. Examples include the person John F. Kennedy, the act of eating, *A Tale of Two Cities*, and the web site http://www.yahoo.com/. Resources are named by uniform resource identifiers (URIs). By convention, we will refer to resources in this thesis by enclosing their URIs in angle brackets (<>).

A **literal** is a string or fragment of XML. Literals are used to express basic properties of resources, such as names, ages, or anything that requires a human-readable description. Literals will be enclosed in double quotation marks ("").

3.4.2 Statements

Using resources and literals, we can express information in RDF by composing statements, which correspond to arcs connecting nodes in a directed graph representation. An RDF statement consists of three parts:

- A **subject**: the resource being talked about (the originating node).
- A **predicate**: a resource that describes the relationship between the subject and the object (the arc).
- An **object**: a resource or literal whose interpretation depends on the predicate (the target node). (N.B.: This use of the word "object" is distinct from the use of "object" in object-oriented programming. The term "object" will only be used to refer to the third element of a statement for the remainder of this thesis.)

Because of their tripartite construction, statements are sometimes referred to as **triples**. Some examples of statements are given below.

```
<urn:examples:john> <urn:examples:hairColor> <urn:examples:black>
<urn:examples:john> <urn:examples:lastName> "Smith"
```

3.4.3 Vocabularies

We can infer from the URIs being used that the two statements above state that John's hair color is black and that his last name is spelled "Smith". However, a computer would not be able to make any such inference; all that could be gathered is that two properties are known concerning `<urn:examples:john>`. Indeed, to the computer, `<urn:examples:lastName>` could mean "was born in".

It is up to the author of a URI to determine its usage or *semantics* such that when others use the URI they will be using it in accordance with these semantics. To this end, vocabularies are created to give meanings to sets of URIs. One common vocabulary is called the Dublin Core. The Dublin Core vocabulary provides a means for using RDF to state standard properties of publications with URIs such as `<http://purl.org/dc/elements/1.1/publisher>` and `<http://purl.org/dc/elements/1.1/creator>`.

To help prevent two different parties from using the same URI, the RDF standard encourages the use of namespaces. In the case of Dublin Core, the namespace is defined to be `http://purl.org/dc/elements/1.1/`; URIs of the resources in the Dublin Core vocabulary all start with this prefix. This technique relies upon the relatively safe assumption that domain names are unique, and the owner of a domain name would have control over the definition of URIs falling within that space. For convenience, we will refer to resources in commonly-used ontologies using prefix notation. For example, we will define `dc:` to refer to `http://purl.org/dc/elements/1.1/` and refer to `<http://purl.org/dc/elements/1.1/creator>` as simply `dc:creator` (the lack of angle brackets around this expression is used to indicate that this URI is expressed in prefix notation; disambiguation is necessary since `<dc:creator>` is itself a valid URI).

3.4.4 Some basic RDF vocabulary

The RDF standard provides rudimentary vocabulary for talking about resources; these resources reside within the `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` namespace, and we will use the `rdf:` prefix to refer to RDF standard vocabulary.

RDF defines one basic predicate called `rdf:type` that allows us to specify the **type** of a resource. For example, the following statements provide basic metadata on MIT's home page.

```
<http://web.mit.edu/> rdf:type <http://haystack.lcs.mit.edu/schemata/web#WebPage>
<http://web.mit.edu/> dc:title "MIT's Home Page"
```

Even resources that normally occur in the predicate position, referred to as either **predicates** or **properties**, can have statements written about them—including `rdf:type`.

```
rdf:type rdf:type rdf:Property
```

The above statement introduces one of the standard classes defined by the RDF standard. `rdf:Property` is the type given to resources that are used in the predicate position.

3.4.5 RDF Schema

A sister standard to the RDF specification, the RDF Schema specification defines standard vocabulary for talking about RDF properties. An **RDF schema** is a specification of a set of **RDF classes** and properties. The RDF Schema ontology is defined within the `<http://www.w3.org/2000/01/rdf-schema#>` namespace and is commonly referred to by the rdfs: prefix. We will illustrate the RDF Schema ontology with an example schema called `:mySchema` (the blank prefix will be used for illustration):

```
:Person             rdf:type            rdfs:Class
:Person             rdfs:label          "Person"
:Person             rdfs:comment        "Represents a person."
:Person             rdfs:isDefinedBy    :mySchema

:Man                rdf:type            rdfs:Class
:Man                rdfs:label          "Man"
:Man                rdfs:comment        "Represents a male person."
:Man                rdfs:subClassOf     :Person
:Man                rdfs:isDefinedBy    :mySchema

:hasBrother         rdf:type            rdf:Property
:hasBrother         rdfs:isDefinedBy    :mySchema
:hasBrother         rdfs:label          "Has brother"
:hasBrother         rdfs:comment        "Specifies a person's brother."
:hasBrother         rdfs:domain         :Person
:hasBrother         rdfs:range          :Man
```

RDF classes have type `rdfs:Class`, and properties can be defined as being available to instances of these classes. In the example, the `:hasBrother` property is available for re-

sources of type :Person; this connection is established by the rdfs:domain property of the :hasBrother property. Furthermore, classes can be subclassed; here, :Man is said to be a subclass of :Person. Finally, a property's type can be established by means of the rdfs:range property. The following is a concrete example of a set of statements that conforms to the schema.

```
:johnFKennedy              rdf:type      :Person
:johnFKennedy              rdf:type      :Man

:euniceMaryKennedy         rdf:type      :Person
:euniceMaryKennedy         :hasBrother   :johnFKennedy
```

Most of the other RDF Schema properties are available to improve readability. The rdfs:label property defines a human-readable name for a property or a class. The rdfs:comment property gives a human-readable description of how to use a property or a class. Finally, rdfs:isDefinedBy indicates the schema resource that defines the property or class in question. (The schema resource—here :mySchema–abstractly groups related class and property definitions together. The self-referential nature of rdfs:isDefinedBy is useful because, due to the statement-level granularity of RDF, the use of a predicate can be accompanied by a rdfs:isDefinedBy statement to tell a system where to find the relevant schema definition.)

3.5 The role of unstructured information

We have described a data model in which a variety of different forms of a user's information can be encoded. However, in reality, information exists in many forms and originates from a range of different sources. Part of demonstrating how our data model effectively addresses users' information needs will be to show how such information can be incorporated and handled in a first class fashion.

Relatively little information is currently present in semistructured form; most information is still stored in unstructured form. For example, textual content—be it in word processing documents, web pages, or e-mails—is recorded today with very little organization. Analysis of these documents through either statistical approaches or natural language processing can yield rich amounts of metadata (e.g., automatically-generated summaries, keywords, etc.), much of which can be useful for later retrieval of these documents and can be stored in semistructured form. In other words, unstructured documents can be thought of as "presentations" of semistructured information.

There are numerous practical reasons for keeping information in a condensed form. It may be more efficient from a computational point of view to have data structures designed in ways that suit the problem at hand. Additionally, programs may be able to more efficiently check the integrity of such data structures when they are cast in a specific binary format. Taken to its extreme, one will observe that the transmission of any information over a network ultimately requires information to be rendered in some unstructured, condensed sequence of bits. Realizing that information is frequently translated between such forms is key to addressing the ostensible inconsistency of incorporating unstructured information into a semistructured system such as Haystack. In the next chapter we discuss means of converting between these forms.

Chapter 4 Implementing the data model

Chapter 3 described a virtualized environment in which an extensible set of information regarding a growing collection of resources could be described. In this chapter, we address the implementation of this data model, which brings about several challenges that must be addressed in order for development and operation of a system to proceed efficiently. The database used for hosting our data model differs substantially in purpose and use from more conventional relational databases, which results in certain opportunities for optimization that were taken for granted in relational databases becoming unavailable; our database design reflects this change. A large amount of information of importance to the user will not initially reside in the data model; "agents" are responsible for importing relevant information from other sources in order to maintain the user's virtual data abstraction. Agents are also employed to expose "deduced" data from existing data in order to improve its usefulness. Finally, the concept of belief is discussed in order to address the problem of multiple parties—consisting of people, organizations, and agents—producing conflicting information; we propose (but do not demonstrate) an intermediate layer for handling belief.

4.1 RDF stores

The way in which an application's data layer is implemented is highly dependent upon the data model chosen. Applications that use a relational model benefit from the use of a relational database, which is specifically optimized for handling tabular, fixed-schema data. Object-oriented applications are often implemented in languages such as Java or C++ that provide inheritance and type hierarchy management facilities suited to programming in the object-oriented paradigm. In contrast, the distinguishing characteristics of our data model described in Chapter 3 require that the data layer make no assumptions about the structure of information and the type hierarchy in play at runtime. The data layer manifests itself in the form of RDF stores (also called RDF containers), which are described in this section.

4.1.1 Statement representation

Our data model supports the notion that custom relationships can be used at any time. As a result, it is not convenient to require the developer or user to update table definitions each time an unknown predicate is encountered. In addition, while traditional rela-

tional models take advantage of table definitions to optimize indexing and to enforce constraints, such precise information about the semantics of predicates may not be known. Indeed, inconsistencies are expected to appear, as the user is not assumed to be perfectly rational. Therefore, a natural abstraction is a three-column table, a normal form for a database [89], consisting of columns for subject, predicate, and object. Each row in such a table corresponds to an RDF statement.

While this setup appears to be capable of encoding our directed graph model, it does not easily facilitate the use of statement-level metadata. In RDF, something can only be described if there is a name by which that thing can be identified. Reification, the process by which a statement is given a name, requires the addition of at least three new statements (to assert the subject, predicate, and object of the original statement). From a practical standpoint, using reification to support statement-level metadata requires a threefold increase in the number of statements being recorded. Because of the frequency with which such metadata is used in Haystack, it makes sense to provide database-level support for naming statements; we accomplish this by adding a fourth column to our table setup, used for recording a statement's identifier. To maintain the uniqueness of this identifier, a statement's URI is derived from an MD5 hash of its subject, predicate, and object.

4.1.2 RDF API

Access to RDF stores in Haystack is accomplished by means of Haystack's Java RDF API, which exists in a package called `edu.mit.lcs.haystack.rdf`. Classes representing RDF data model elements are provided in this package, including `Statement`, `Resource`, and `Literal`. `Resource` and `Literal` derive from the `RDFNode` class and are immutable (in the Java sense of the word, like `java.lang.String`). `Statement` has three fields corresponding to a statement's subject, predicate, and object and is similarly immutable.

RDF stores implement the `IRDFContainer` interface, which exposes standard methods for adding (`add`), removing (`remove`), and querying statements (discussed below). Most Haystack components are designed to work with any object exposing this abstraction so that multiple implementations can be used depending on the circumstances. The simplest of these implementations is called a `LocalRDFContainer`, which is meant for transient storage and processing of RDF statements. Persistent storage is supported by a more sophisti-

cated implementation called Cholesterol, which is written in C++ for efficiency reasons and is described below.

4.1.3 Support for queries

Because our data model is so expressive, program state that used to be stored in local variables or in-memory structures can now be stored in the database. Programs benefit from the persistence and powerful extensibility qualities that result. However, encouraging programs to make heavier use of the database makes being able to process numerous, frequent queries quickly from multiple threads important. Furthermore, while expressive query languages such as SQL can be used to satisfactorily access program data, we find that certain access patterns are so common that they warrant special support from the database itself.

The simplest of these queries is called `contains`. The `contains` function queries ask the database whether a certain statement is present. Queries such as "does this resource have type Person" and "is this book written by John Doe" can be satisfied in this way. Another simple query is called `extract`. Here, two of the three elements of a statement are given to the database, and the database is asked to return the third resource or literal that corresponds to some statement in the database; if more than one statement matches the pattern, one statement is picked nondeterministically. For example, `extract` can be used to ask the database questions such as "what is this resource's title", where perhaps in the user interface, only one title is needed. `extract` is also frequently used to ask the database queries that would correspond to field accesses in an object-oriented language.

More complex queries can be performed with the `query` and `queryExtract` methods. These functions are designed to support **graph matching**, whereby a client supplies a fragment of a graph containing "blank" nodes and expects the store to find assignments of these blank nodes that when inserted into the graph fragment forms a subgraph of the database graph. A graph fragment can be specified as a series of statements with certain unique, distinguished resources representing blank nodes interspersed therein. In other words, this form of query supports conjunctive joins; disjunctions and other conditions are supported by other means discussed in Section 10.1. The difference between the two methods is that `query` returns all possible assignments, whereas `queryExtract`, like `extract`, nondeterministically returns one assignment.

The reason for providing `queryExtract` and `extract` is that there are efficiency benefits for when queries are being distributed across multiple stores. A distributed store can expose the same interface as other stores even though queries are being resolved across a set of individual stores. In the simplest implementation, in order to perform a join across multiple stores, the query must be evaluated one line at a time, and each line must be performed separately on each store. The results for each line of the query from each store are then combined before the next line is performed. In contrast, `queryExtract` and `extract` can simply iterate through each store until a match is found, because `queryExtract` and `extract` make no guarantees about which result will be returned if there are multiple possible results. If no match is arrived at, `queryExtract` must then distribute the entire query as outlined above; `extract` can simply return `null`.

4.1.4 Scalability

Because the possible amount of information that one may wish to store in a system is unbounded and because of the higher level of granularity of our data representation, it is important that the database be able to efficiently support frequent queries and mutations in the presence of millions of statements. We have identified several techniques for managing large databases of triples, which are discussed below.

Concurrent access from multiple threads is important for the construction of Haystack's multi-agent system, which is described later in Section 4.2. To support concurrency, multiple simultaneous reads should be permitted on the same table with minimal latency, and writes in one part of the database should not affect writes in another part. To achieve these goals, Cholesterol structures data into nodes (as in graph nodes), which correspond to resources or literals. Statement tables are stored only in two indices. Given the subject or the object of a statement, one can easily find the objects or subjects hanging off of any given predicate, respectively. In other words, two dictionaries hang off of each node, one mapping predicates to objects (the node being the subject) and one mapping predicates to subjects (the node being the object). Each index or dictionary is locked separately for supporting multiple reads/single write (writing and reading are exclusive activities).

Obviously, a database that holds all of its nodes in memory will reach its limit on the number of nodes it can support more quickly than one that can store nodes on disk.

Cholesterol adopts a most recently used (MRU) scheme and keeps only a fixed number of node tables in an in-memory cache. Helper threads monitor the cache and flush nodes to disk in large batches from the bottom of the MRU queue. Each flush creates a new file, avoiding the need to maintain read/write locks on existing node files. Cholesterol maintains an in-memory node allocation table in order to maintain what nodes are present in memory and what their locations are on disk. Given that it requires 20 bytes to describe the state of a node, the memory usage of the system scales decently with the size of the database.

To protect database integrity in the event of a crash, all statement adds and removes are logged in a journal that resides on disk. A clean bit is used to detect whether the database was shut down properly the last time the database was active. In the event of an unclean shutdown, Cholesterol reloads itself from its journal.

Because node entries are assigned in linear order and are not freed when they become unused, fragmentation can begin to artificially reduce the performance of the system. Also, the flushing mechanism employed by Cholesterol will eventually consume all disk space. To address these issues, Cholesterol is designed to be defragmented on a regular basis. This is accomplished by writing all of the statements present in the store into a freshly created journal and then reconstructing the database from this journal. After defragmentation, unused node indices no longer exist and the size of the node index corresponds to the true number of nodes present.

Our table structure makes it efficient to process queries where either the subject or object of a needed statement is known, but is terribly inefficient for the case when only the predicate is known. Previous versions of Cholesterol tested other indexing approaches, including one that did all possible indexing and one that indexed by predicate first then subject to object and object to subject. The first of these approaches did not scale well with respect to memory usage because the number of predicates is relatively low with respect to the number of subjects and objects in the system, so popular predicates would remain in memory, including statements involving rarely-used subjects and objects. The second approach, while optimizing retrieval, was three times slower during adds (because of the additional indexing required) and severalfold more memory intensive. On

current systems with only 512 megabytes of memory, this configuration did not perform well.

Finally, joins—the mechanism by which graph matching is performed—are optimized to perform the lines of queries that have known subjects and objects first in order to avoid having to iterate through the entire database if only the predicate position is known.

4.1.5 Events

Cholesterol supports an event mechanism that allows clients to be notified when statements satisfying a given pattern are either added or removed. Unlike the case with traditional database views, in which any query can be used to define an alternative way of looking at tables, for performance reasons Cholesterol restricts the query pattern to one statement[3]. Since every statement that is added or removed from the database must be checked to see if it satisfies a registered event pattern, our interface has very few quality of service guarantees. First, the database updates its list of event listeners on a periodic basis, so events will begin firing after a slight delay following event registration. Second, events are not promised to be fired in order; in fact, a statement add event may be thrown even after that same statement was just removed. It is up to the client to handle such situations. Improving the quality of the event service would require more locking and more latency in database accesses, unacceptably reducing the performance of the system, which is highly dependent on constant access to the database.

RDF database events have many uses for different components of the system, especially the user interface. Motivating examples are given throughout this thesis.

4.2 Multi-agent blackboard model

The RDF store serves as the foundation of a system that utilizes our data model. On top of the RDF store sit components that process information residing in the RDF store. In Haystack, agents account for the biggest class of these components. An **agent** is simply a resource in Haystack (named by URIs) that stores its state (if any) in the RDF store and has pieces of executable code called **methods** that can be invoked. Agents play a similar role to daemons in a UNIX environment or to Web Services in a web server, providing

[3] Further testing is needed to show whether it is more efficient for the database or the client to implement support for more complex patterns. Another motivation for the choice we have made is ease of implementation.

basic pieces of functionality to the system such as scheduling, e-mail connectivity, database abstraction, and even storage. In Haystack, the RDF store itself is an agent, and multiple RDF stores can reside within the system. This setup is analogous to (human) project teams in which one member is designated as a recorder or secretary; being a secretary does not preclude a person from being considered a member of the team.

Sitting at the core of the Haystack system is the service manager, a bootstrap process that is responsible for starting up the agents it hosts. At system startup the service manager reads an RDF configuration file to determine where the root RDF store is. The service manager then instantiates this root store, much as a UNIX system mounts its root file system at startup, and determines what agents should be started based on the values of the `config:hostsService` property of the service manager's resource (all service managers are named by URIs). As a result, with service managers that share a root store but run on different machines (akin to running a system off of a network file system; connecting to remote agents is described below), a simple switch of a `config:hostsService` property can cause an agent to move from one machine to another. When agents are instantiated, they are given a connection to the root RDF store and the service manager. As all persistent system state is described in RDF, Haystack uses RDF stores much as modern software uses the file system.

Agents can be written in a variety of languages, including Java. The list of what methods the agent supports, however, is described in RDF using an ontology derived from WSDL [90]. The extensibility characteristics of our data model make it very amenable to description of interface metadata. For example, custom properties can be applied to method and agent definitions to classify them against some taxonomy or to describe performance characteristics.

The service manager is also responsible for allowing agents to connect to one another. If an agent requests to connect to an agent running on the same service manager, the service manager can return a pointer (i.e., object reference) to the other agent directly; otherwise, the service manager uses the information about the agent encoded with the WSDL ontology to construct a proxy.

Because agents in Haystack share an underlying store, they can interoperate with each other by treating the store as a "blackboard". Blackboard architectures permit multiple

agents to attack a problem by allowing services to use information on the blackboard to perform some specific analysis and to pose new information that is derived from that analysis. Agents can learn when new information (i.e., RDF statements) has been posted to the store by registering for events with the RDF store. New, synergistic functionality can be introduced into the system by adding agents that perform certain tasks when specific forms of information enter the system.

4.3 Importing information from other systems

As mentioned earlier, agents play an important role in maintaining the data abstraction shown to the user. Information that does not originate from Haystack must be incorporated into the system in one of several ways. There are two extremes to consider in this situation. At one extreme, the information may already exist in an RDF store, and the problem of incorporation reduces to one of federation. At the other extreme, the information is in some non-RDF format and must be converted into RDF and imported into an RDF store.

For the cases in between the two extremes, a number of factors must be considered. For information sources that use an encoding that is isomorphic to RDF, a possible solution is to create an agent that *emulates* the RDF store interface. For a system to be isomorphic, the system must employ a distributed identifier scheme for naming objects and have the same, flexible triples-based representation for metadata. However, apart from RDF stores written to interfaces that are a slight variation on Haystack's standard, few systems meet this requirement.

Some systems, such relational databases, utilize unique identifiers (i.e., primary keys) but cannot encode arbitrary metadata. In these cases, read-only RDF store-like agents can be produced. Others, like IMAP servers, can store arbitrary metadata (to some extent—only metadata attached to messages) but do not have a unique identifier scheme (IMAP paths change when users move messages between IMAP folders). Also, unlike databases, IMAP servers cannot process arbitrary graph matching queries, making interfacing with IMAP servers fit more suitably with one of the solutions outlined below.

For systems without unique identifiers, one must resort to synchronization. Synchronization agents range in functionality from dumping (i.e., a one-way transfer from the originating system into an RDF store) to true synchronization, of the kind performed by

personal digital assistant (PDA) synchronization software. An example of a dumping agent is the POP3 agent in Haystack, which downloads mail from a POP3 server and places mail header metadata into the RDF store. Message IDs serve as makeshift unique identifiers when they exist. Haystack does not have any true synchronizing agents, but ones could be written to interface with systems such as Microsoft Outlook and Lotus Notes.

Even when unique identifiers are used, it may be the case that processing arbitrary graph queries can be unacceptably inefficient. In this situation, it makes sense to create a caching RDF store to hold an RDF-encoded copy of the metadata and to use synchronization or write-through to back-propagate changes if necessary. The file system RDF store is an example of a read-only RDF store emulator that is best used with a caching RDF store.

One final situation—given a URI, retrieving relevant metadata—is considered here. The motivation for using HTTP URIs is that absent other information on how to locate relevant metadata, a client can at least try using HTTP to retrieve what is available [3]. Some systems return XML-encoded RDF metadata in response to a requested MIME type of "text/xml" [99]. Some classes of URIs have more well-defined resolution mechanisms. For example, the Life Science Identifier (LSID) standard for naming biological information on the Internet uses a DNS name embedded within the identifier (termed an LSID) that names a server that, when contacted with the correct SOAP request, returns metadata associated with that LSID [91]. Further discussion appears in Section 15.2. For general URIs that do not fall into one of these categories, Haystack can also take advantage of the `info:knowsAbout` predicate, which indicates a specific server to contact to obtain related metadata. (The way in which Haystack responds to such a request is described in the next section.) In any event, one can imagine a system choosing to contact "metadata search engines" that would be created to respond to queries about URIs with no resolution hints embedded within.

4.4 Exchanging graph fragments

When Haystack is requested to send out information relevant to a resource, Haystack must identify some portion of the graph to transmit. (Alternatively, one could assume

that the receiving end will "pull" the metadata it wants from the sending Haystack; this case is a special case of one considered below.)

The naïve approach is to perform a search of all reachable resources leading out (i.e., going in the subject to object direction in the directed graph) from the resource being sent and to collect the statements that describe the traced connections. This corresponds directly to the approach taken by most systems today, including Java object serialization. However, when applied to RDF a number of problems arise. First, type—which used to be a meta-property managed by the system—is exposed in our data model; as a result, when a resource is serialized in this fashion, a huge portion of the class hierarchy may come with it! Second, sometimes the directionality of a predicate is arbitrarily defined, such as with the reflexive sibling relationship. To cover such cases, we would need to extend the algorithm to trace not only outgoing predicate arcs but also incoming arcs. Given the connected nature of a personal RDF graph, one may end up with the entire graph as the serialization for a resource—likely not the desired outcome.

To avoid pulling the entire graph when serializing a resource, we specify that only certain predicate arcs should be followed to determine the relevant subgraph. Predicates must be annotated as being of certain types to indicate whether they should be traced during serialization. The first of these types is called a **relational property**. Relational properties are ones that should not be followed when serialized; in other words, the two resources on either side of the predicate are considered to be at the same abstraction level. Examples include most interpersonal and type relationships. When a user requests a description of a contact to be sent to an associate, it is unlikely that he will expect descriptions of that contact's siblings, parents, grandparents, secretary, and manager sent too; it is sufficient to simply mention the presence of these resources. In contrast, a **proprietal property** indicates a part-of relationship in which the object of such a property should be traced and considered part of the relevant subgraph. Naturally these distinctions apply only to object properties, as there is no subgraph to follow for a datatype property. (By default, a property is considered relational if not otherwise specified.) Examples include the `msg:body` property (described in Section 12.3), as sending a message without its body is not terribly useful.

The resulting subgraph may refer to resources that are foreign to the receiving party, but because those resources were objects of relational properties, the subgraph contains no information on them. To address this problem, the system can assert that the user producing the graph fragment `info:knowsAbout` these terminal leaf nodes in the subgraph; if more information is needed, the recipient may contact the sender.

The graph extraction technique is generally useful for the creation of **information resources**, which are sets of RDF statements represented in some format, such as RDF/XML. Information resources act as containers for RDF graphs, which can be easily transported over the wire.

4.5 Forward chaining

Semistructured data models are amenable to declarative forms of programming, such as first order logic. As with a predicate calculus, rules can be applied to the relationships between resources in our data model to permit inferences such as derived class properties or transitive deductions [6].

As is the case whenever logic is to be used, care must be taken to precisely define the concepts and relationships to be represented. Because of this need for precision, inference-based semantic network applications have had limited success for traditionally keyword-based, statistical information retrieval techniques [7]. Furthermore, our data model is designed to be populated by end users, who are not often characterized as being highly logically consistent. Our use of semantic network research focuses on the epistemological representation of knowledge and not on the level of precision needed to support inference.

As a result, our system makes extremely limited use of any form of inference. That being said, there are many instances in which declarative patterns are a convenient way to express means for generating new data from old. Simple ontological mappings can be achieved with naïve forward chaining, whereby target graph patterns can be identified and new graph patterns can be generated.

For example, when an American user buys a book from an online French bookstore, the system should be able to utilize an ontological mapping to produce “book”, “name”, and “author” attributes from “livre”, “nom”, and “auteur” attributes, respectively. An-

other use is for extracting the significant amount of information buried in today's computer systems in hundreds of file formats, including document titles, authorship information, and physical dimension specifications. This metadata is also scattered across several different types of repositories, such as databases, file systems, and web servers. In these cases, patterns can be used as a simple event triggering mechanism, and code can be used to express the extraction of certain metadata buried within custom file formats.

The Serine agent in Haystack is responsible for supporting forward chaining functionality. Serine registers for events from Cholesterol that correspond to graph matching patterns given in forward chaining pattern specifications and adds the resultant statements into the database. Sometimes, the relationship between the original statements and resultant statements is not easily expressed, and in these cases code fragments can be invoked to create the resultant statements. In this way, Serine becomes a convenient vehicle for registering lightweight RDF store event patterns.

4.6 Belief

Sending and receiving messages and other information over the Internet has become a commonplace activity. Taken in the context of Haystack and its data model, collaboration entails that information present in the repository may have originated from multiple sources. Considering that agents are also responsible for populating the repository with deduced or imported information, one can conclude from a practical standpoint that it is important to keep track of who said what. Naturally, one cannot believe everything one reads; information originating from certain sources must be taken with a grain of salt. However, the discussion of our data model heretofore has assumed that statements are all given the same truth value. We introduce the notion of belief to account for these differences in trustworthiness.

In order to determine whether a statement is to be believed, one must know a few properties, or pieces of metadata, concerning the statement, such as who uttered it. Such metadata is describable in our data model but requires special treatment: If we assume that by placing a statement into a graph one is asserting the validity of that statement, then metadata affecting the validity of such a statement cannot be encoded at the same conceptual level as the statement itself. This is because the statements that assert the va-

lidity of other statements would have the same validity as the statements whose validity was being described.

A simple solution is to extend the data model to include fields that affect the validity of the statement in addition to subject, predicate, and object. However, the fundamental notion that our triples-based model can extensibly describe data is somehow in conflict with the idea of arbitrarily extending the size of the "triple" to incorporate additional statement-level metadata. Statements themselves are resources, about which utterances should be possible. (A statement is a resource of type `rdf:Statement` with three properties, `rdf:subject`, `rdf:predicate`, and `rdf:object`, denoting the statement's subject, predicate, and object, respectively. The process of naming a statement with a resource is called **reification**.)

Another solution is to use a triples model to describe only statements at the same level of truth. One possibility would be to take the user's utterances as the base level of truth; other statements would be "reified" into resources, and they would appear in the graph only indirectly in the form "<someone> said <something>". On the other hand, one could also take the "system" to be the base level of truth. All statements would then be reified, and the system's utterances would only be those concerning the reliability of other statements. This solution provides a certain level of consistency that is desirable, in that the user's statements are no longer taken to be a special case. Additionally, the system could use digital signatures to guarantee that an utterance claimed to have been made by some party was indeed made by that party.

Because all statements are now first class resources in this representation, whatever properties are required for any particular belief system can be added freely. One clear candidate for such a property would be "asserted by", as the trustworthiness of a statement is often a function of who uttered it. Another is "denied by", whereby a party may assert disagreement with a statement. Other properties can be used to describe the relative trustworthiness of different parties with respect to the user. Using these forms of statement-level metadata as a basis for characterizing trustworthiness, a system would then be able to determine which statements are trusted and which are not.

While not currently implemented in the version of Haystack discussed in this thesis, previous work has investigated the incorporation of the ideas given here into Haystack [61].

Chapter 5 Programming environment

Our data model embodies a high level of expressive power, which we show in later chapters to be necessary for capturing important metadata from the user. In this chapter we demonstrate how this expressive power can also be harnessed to describe programmatic abstractions that have proven useful for building infrastructure to support implementation of the Haystack system. As is the case with other knowledge representations, the goal of the abstractions we present here is to capture the intention of the programmer succinctly with respect to the relevant target domain—namely, manipulation of an RDF data model and interfacing with the user and other systems. Programs are themselves recorded in the same RDF data model as the data they manage, reincarnating the classic data-code duality explored by Lisp systems. One immediate benefit is that code needed to properly manage data recorded with respect to an ontology can be packaged with the ontology itself.

In this chapter we explore the details of two programming "ontologies" whose lexical structure is embedded in the RDF data model and that are adapted to the problems of dealing with RDF metadata. First we present a syntax that will be employed throughout this thesis for recording RDF metadata. We then talk about the details of the two ontologies, named Adenine (an imperative language) and Ozone (a functional language), and the runtime support needed to evaluate programs written in these ontologies and about where to draw the line between (RDF-based) persistence and in-memory persistence, a problem that was somewhat ignored by Lisp systems but broached in the context of Dynamic HTML (HTML with JavaScript) [97].

5.1 Mixing code and data

The common theme that runs through both Adenine and Ozone is the Lisp-like idea of mixing code and data in the same environment. In Lisp, the basic data element is the pair, and both executable expressions and data structures can be built up from pairs. The main distinction lies in the fact that executable expressions can be passed to the interpreter (i.e., the `eval` function) and can cause the computer to perform certain operations. Similarly, the RDF statement is the basic data element for both data and code in our environment, and the basic model for interpretation of an Adenine or Ozone program is the passing of the URI of the program (a resource) to the appropriate interpreter.

5.1.1 Persistent versus ephemeral runtime data

Implicit in Lisp code fragments is the notion of the runtime environment against which a code fragment is executed. This runtime environment supports the concepts of variable bindings, continuations, and object references. It is these concepts that characterize modern programming languages. Appropriately, Adenine and Ozone also expose similar concepts to programmers. What is more interesting is that, unlike data and code, the Lisp runtime environment is not represented as pairs but is rather extant only behind the abstraction of the virtual machine. For example, if a variable `x` is bound to the value `2` at some point in time, nowhere in memory can one find a representation of this in terms of pairs. One important motivation for creating this fundamental split between persistent source data and transient runtime data exists: while creating a higher level of uniformity within the language, representing the runtime environment in the data model can be less efficient than representing it in a more encapsulated and hidden fashion. This split is similarly adopted by Adenine and Ozone in order to improve performance. It is, however, interesting to note that RDF is actually expressive enough to model the runtime environment, and perhaps this idea will be worthy of future investigation when systems are performant enough to sustain such a representation.

One important difference between Lisp and Adenine or Ozone is the representation of the basic data element. In Lisp, pairs exist both syntactically in source files and as living objects in the runtime environment. Persistence is achieved by converting data in the runtime environment to a pair representation and serializing this representation. In contrast, Adenine and Ozone are based on the notion that RDF statements exist primarily in an RDF store, not the in-memory runtime environment. This distinction implies that when data is represented in terms of RDF statements—which were intentionally structured to make representation of general data easy—persistence is automatic, and as a result, less marshalling between persistent forms and in-memory forms is needed. In other words, Adenine and Ozone treat both persistent and in-memory state in a first class fashion.

5.1.2 Annotating code

Representing code in the data model can also be considered with respect to bytecode-based virtual machine-driven languages such as Java. The key benefit of bytecode representation has been portability, enabling interpretation of software written for these plat-

forms on vastly different computing environments. In essence, bytecode is a set of instructions written to a portable, predetermined, and byte-encoded ontology. One immediate benefit of RDF representation is that creating new instructions in the language is simply a matter of defining an extension to the language's ontology. Both Adenine and Ozone allow handlers for new language elements to be declared in RDF and implemented in Java. (This aspect is infrequently used in Adenine but is used extensively by Ozone, as we shall see later.)

There are several other benefits to embedding code in a data model designed to host metadata. Consider one recent example of a system that uses metadata-extensible languages, Microsoft's Common Language Runtime (CLR). In a language such as C#, *developer-defined* attributes can be placed on methods, classes, and fields to declare metadata ranging from thread safety to serializability. Compare this to Java, where serializability was introduced only through the creation of a new *language keyword* called transient. The keyword approach requires knowledge of these extensions by the compiler; the attributes approach delegates this knowledge to the runtime and makes the language more extensible.

In Adenine and Ozone, RDF assertions such as comments, classifications, authorship attributions, and information about concurrency safety can be applied to any element of the code. This fact enables a number of different features, from self-modifying code to automated object code analysis. However, most of these features remain unexplored at present, and further investigation into the benefits of modeling programs in RDF is warranted. One feature that has proven to be highly useful is the ability to annotate functions or components with specialized types such as "asynchronous constructor" or "query operator". This annotation allows a developer to easily embed flexibility into his or her program by allowing the user to choose from among the functions or components that exist in the data store with a given annotation. These annotations play an important role in exposing Adenine code as user-callable functions, as we will see in Section 7.2.

5.2 Adenine RDF syntax

Before delving into the details of the Adenine and Ozone ontologies, we must establish a syntax by which we can express programs in these ontologies. Despite the ostensible separation that exists between syntax and semantics, it is quite common for program-

ming languages to specify both within the same specification. Adenine, like C++, Java, Python, and most other languages, specifies both a model and a syntax. This combination allows an Adenine interpreter to serve as a complete environment for interacting with our runtime environment. In this section we will discuss the syntactic features of the Adenine language, which serves as a substitute for RDF/XML and is better suited for the recording of RDF-encoded metadata by humans.

5.2.1 The structure of an Adenine source file

Adenine source files hold Adenine source code and serve as the unit of compilation. The Adenine compiler takes Adenine source code and produces a collection of RDF statements. This is analogous to the process by which a Lisp or Scheme parser takes a series of s-expressions and produces a collection of pairs. It is also analogous to the process by which an assembler takes an assembly source file and produces a collection of bytes. That is to say that an Adenine source file is a syntactic representation for a set of RDF statements. (Why this is referred to as compilation and not parsing is explained later.) An example source file is given below in which a series of RDF statements have been declared.

```
# An example Adenine source file
@base <http://example.org/sample>

@prefix dc:         <http://purl.org/dc/elements/1.1/>
@prefix sample:     <http://example.org/sample#>

add { ^
       rdf:type     daml:Ontology ;
       dc:title     "Sample Ontology"
}

add { sample:age
       rdf:type            daml:DatatypeProperty ;
       rdfs:isDefinedBy    ^ ;
       rdfs:label          "Age" ;
       rdfs:range          xsd:int
}
```

The first line of the file demonstrates how comments are expressed in Adenine: with a hash mark (#) as the first non-space character on a line. The comment area extends to the end of the line.

The second line of the file gives an `@base` declaration. This directive is optional and specifies the URI of the Adenine source file, much like the `<base>` tag does in HTML.

Following the `@base` declaration is a series of `@prefix` specifications. They instruct the compiler to define prefixes such as `dc:`, whose URI is prepended to the text following the colon in identifiers such as `dc:title`. By default, the following prefixes are also defined:

Prefix	URI	Use
`rdf:`	`<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`	RDF standard ontology
`rdfs:`	`<http://www.w3.org/2000/01/rdf-schema#>`	RDF Schema ontology
`daml:`	`<http://www.daml.org/2001/03/daml+oil#>`	DAML+OIL ontology
`xsd:`	`<http://www.w3.org/2001/XMLSchema#>`	XML Schema Datatypes
`adenine:`	`<http://haystack.lcs.mit.edu/schemata/adenine#>`	Adenine ontology
`random:`	A random URI generated each time the file is compiled	Varies
`:`	The `@base` URI plus a colon (`:`) (ensures local uniqueness), or a random URI distinct from the one assigned to `random:` if `@base` is not specified	Local definitions

While `@base` and `@prefix` declarations can appear anywhere in a file, they only affect the text following the declarations and are best made at the top of the file.

5.2.2 Defining RDF statements

The remainder of a file consists of a series of `add {}` blocks. In Adenine curly braces (`{}`) are used to enclose RDF statements. The elements within the curly braces come three elements at a time: subject, then predicate, then object, in that order. No delimiter is used between statements. `add {}` is used to directly denote RDF statements that are in the source file. Despite appearing like an imperative instruction, there is no executable aspect to `add {}`; it is like the assembly language directives for inserting immediate byte

values into the resulting binary image. (The rationale behind this choice of keyword is discussed later.)

Certain special symbols are permitted within statement blocks. The caret (`^`) refers to the URI of the file given by an `@base` declaration. In the example, the source file is declared to have type `daml:Ontology`. A semicolon (`;`) may be used to link together predicate-object pairs to the same subject.

Additionally, Adenine supports the notion of **existential nodes**, declared by the `${}` syntax. Existential nodes are resources whose properties are known but whose URI is not known or relevant. They can be used anywhere a resource is needed. For example:

```
add { :john
       :hasChild ${
              rdf:type      :Person ;
              dc:title      "Joe"
       }
}
```

Here, `:john` is asserted to have a child whose name is `"Joe"` and who is a `:Person`, but the URI of this child is randomly generated at compile time. Like normal curly braces, existential node expressions can have RDF statements embedded within them, except that the subject is never specified as it is implied to be the existential node; furthermore, semicolons must be used to separate property/object pairs.

Finally, Adenine supports a convenient syntax for specifying DAML+OIL lists: `@()`. For example:

```
add { :john
       :favoriteColorsInOrder     @( "blue" "black" "green" "red" )
}
```

`@()` can contain any combination of resources, literals, existential nodes, or even other lists. While `@()` can be used anywhere resources can, it does not allow you to specify the name of the `daml:List` resource, much like `${}`. In fact, the preceding sample is identical to the following:

```
add { :john
       :favoriteColorsInOrder ${
              rdf:type      daml:List ;
              daml:first    "blue" ;
```

```
            daml:rest ${
                  rdf:type     daml:List ;
                  daml:first   "black" ;
                  daml:rest ${
                        rdf:type     daml:List ;
                        daml:first   "green" ;
                        daml:rest ${
                              rdf:type     daml:List ;
                              daml:first   "red" ;
                              daml:rest    daml:nil
                        }
                  }
            }
      }
}
```

5.3 Adenine ontology

The Adenine ontology is used in Haystack to express programs written in an imperative style. Considering that programs in Haystack are in some sense algorithms that effect transformations on the data store, perhaps the most important feature of Adenine is the convenience with which one can express additions to, queries against, and removals from the semistructured data store. Because components will need to store their abstractions in the data store, these elementary database operations become the building blocks for writing components. The data models provided by most object-oriented programming languages, on the other hand, do not readily facilitate database-backed persistence. In general, marshalling between in-memory state and serialized state occurs today only when a user saves his or her file to disk; in a semistructured model, user data lives almost entirely in the store. One of the few exceptions is the Enterprise JavaBeans (EJB) model, which requires an extensive number of proxies and marshaling classes to be generated at compile time to support a limited form of database persistence.

One of the main motivations for creating Adenine is having the language's syntax support the data model. Introducing the RDF data model into a standard object-oriented language is fairly straightforward; after all, object-oriented languages were designed specifically to be extensible in this fashion. Normally, one creates a class library to support the required objects. However, more advanced manipulation paradigms specific to an object model begin to tax the syntax of the language. In languages such as C++, C#, and Python, operator overloading allows programmers to reuse built-in operators for manipulating objects, but one is restricted to the existing syntax of the language; one

cannot easily construct new syntactic structures. In Java, operator overloading is not supported, and this results in verbose APIs being created for any object-oriented system.

Arguably, this verbosity can be said to improve the readability of code. On the other hand, lack of syntactic support for a specific object model can be a hindrance to rapid development. Programs can end up being much longer than necessary because of the verbose syntactic structures used. This is the reason behind the popularity of domain-specific programming languages, such as those used in Matlab, Macromedia Director, etc. Adenine is such a language. It includes native support for RDF data types and makes it easy to interact with RDF stores and RDF-based services.

Furthermore, the concepts of in memory classes and properties, whose inexpressiveness in other languages was mentioned above as inhibiting efficient programming with the RDF data model, can be merged with the notions of classes and properties used in RDF Schema (i.e., in the persistent data model; see Section 3.4.5) and augmented with Adenine code when representing concepts being exposed to the user, such as documents and commands. This is of interest given that object-oriented notions of classes are generally rigidly typed, whereas our data model encourages only suggestive, unenforced typing.

Adenine's RDF representation and its treatment of the RDF triple as a native data type emphasizes Adenine's similarity to Lisp, in that both support open-ended data models and both blur the distinction between data and code. However, there are some significant differences. The most superficial difference is that Adenine's syntax and semantics are especially well-suited to manipulating RDF data. Adenine is mostly statically scoped, but exposes dynamic variables that address the current RDF containers from which existing statements are queried and to which new statements are written. Adenine's runtime model is also better adapted to being run off of an RDF container.

Adenine supports two types of program state: in-memory, as is with most programming languages, and RDF container-based. Adenine in effect supports two kinds of closures, one being an in-memory closure as is in Lisp, and the other being persistent in an RDF container. This affords the developer more explicit control over the persistence model for Adenine programs and makes it possible for agents written in Adenine to be distributed.

5.3.1 Writing executable code

The Adenine syntax provides special syntactic sugar for writing program code with respect to the Adenine ontology. This support revolves around defining pieces of executable code called **methods**—resources described in RDF according to the Adenine ontology.

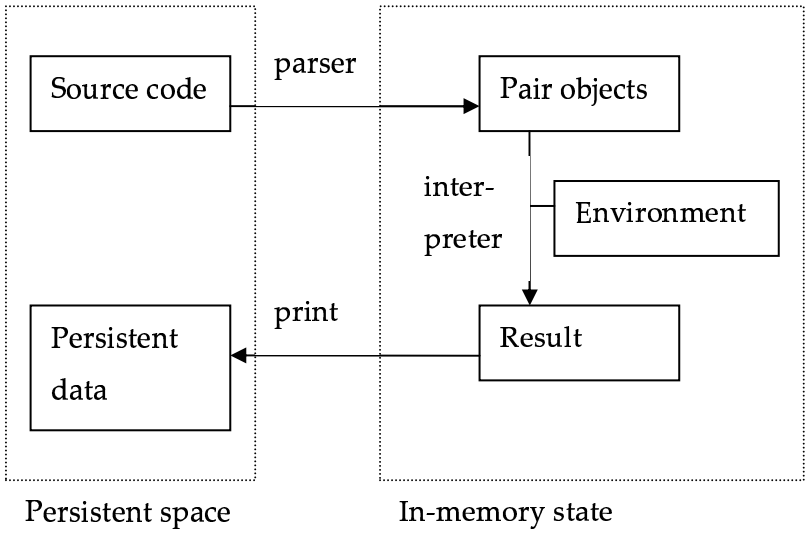

Figure 17: Lifecycle of Lisp code

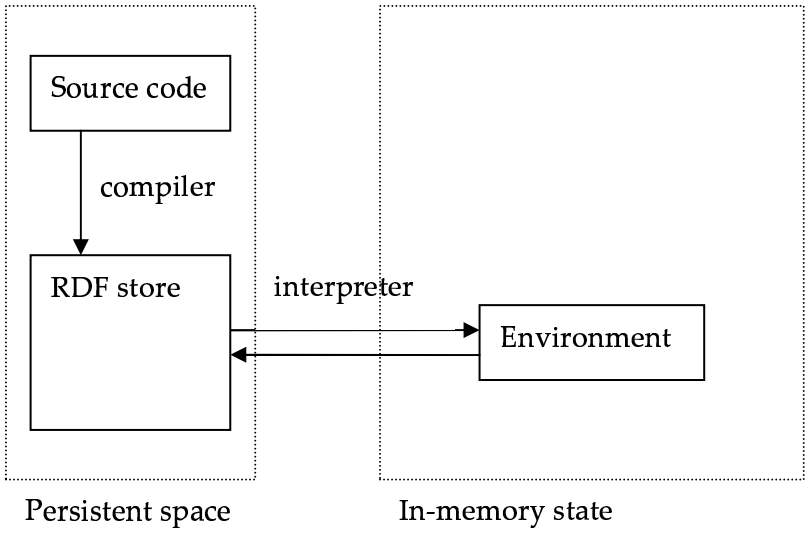

Figure 18: Lifecycle of Adenine code

To understand how code and data are mixed within an Adenine source file, it is useful to make another comparison to Lisp. In Lisp, a source file contains text that when parsed produces a series of pairs, which often contain other pairs nested within. It is assumed that these pairs have a certain structure so that they can be interpreted successfully by the system. For example, these pairs nest to form lists, and the car (first element of a pair) of the top-level pairs are Lisp keywords. It is assumed that the top-level pairs are commands that can be evaluated by the Lisp interpreter in sequence. In contrast, Adenine has no such "top-level" starting points. When an Adenine file is compiled, a large directed graph is produced with no distinguished "starting points"; in other words, statements and resources are not nested within each other but are all at the same level.

To execute Adenine code, the interpreter is explicitly passed a starting point—a resource of type `adenine:Method`. The interpreter then follows the supplied chain of instruction resources (resources connected by the `adenine:next` predicate) and executes them with respect to a runtime environment. As mentioned earlier, the environment is retained in memory although the program and persistent data reside in the RDF store. Refer to Figure 17 and Figure 18 for a comparison of the execution models of Adenine and Lisp. A prototype interpreter has been implemented in Java and is used to run much of Haystack. However, to improve performance, a translator is available, which can compile Adenine methods into Java Virtual Machine bytecode. While eliminating some of the dynamic nature of Adenine, Java translation is semantics-preserving and does provide a significant performance increase.

An example of a method definition is given below:

```
method :helloWorld name
       print 'hello world,' name
```

While it is possible to define Adenine methods completely using the syntax given in the previous section, doing so is extremely inconvenient. Source code is inherently hierarchical (hence the term "parse tree"), and it is easy to describe parse trees in terms of nested cons's in Lisp. In contrast, the directed graph nature of RDF is not inherently hierarchical; trees must be embedded within the data model by means of custom predicates. One can view an s-expression in terms of a directed graph model with two predicates: car and cdr (the second element of a pair). As a result, the parenthesized represen-

tation for cons's can be used to describe both data and code. However, in RDF, other predicates are possible and therefore must be explicitly specified in a triple. This makes an Adenine instruction "tree" extremely cluttered to write and warrants syntactic sugar. For the purposes of comparison, here is example method from earlier written without syntactic sugar:

```
add { :helloWorld
       rdf:type            adenine:Method ;
       adenine:parameters @(
              ${     rdf:type     adenine:Identifier ;
                     adenine:name  "name"
              }
       ) ;
       adenine:start ${
              rdf:type     adenine:FunctionCall ;
              adenine:function ${
                     rdf:type     adenine:Identifier ;
                     adenine:name  "print"
              } ;
              adenine:parameters @(
                     ${     rdf:type            adenine:String ;
                            adenine:string      "hello world,"
                     }
                     ${     rdf:type     adenine:Identifier ;
                            adenine:name  "name"
                     }
              )
       }
}
```

Like other languages, Adenine supports commonly-used constructs such as for and while loops, if/else blocks, and functions. A complete reference of Adenine's syntax and semantics can be found elsewhere [96]. In Lisp, these constructs are identifiable by the presence of a keyword in the car position of a pair. Similarly, such a construct in Adenine is identified by an `rdf:type` assertion with types such as `adenine:WhileInstruction`. No

further decomposition of instructions is made from the source file in the Adenine compiler (e.g., into jump instructions or register manipulations). The transformation from source code to object code is referred to as compilation and is analogous to the parsing stage of Lisp because Adenine's code constructs are like macros and are desugared by the compiler into their equivalent RDF representations.

The syntax of Adenine code resembles a combination of Python and Lisp. As in Python, indentation levels denote lexical block structure. Adenine is an imperative language, and as such contains standard constructs such as functions, for loops, arrays, and objects. Function calls resemble Lisp syntax in that they are enclosed in parentheses and do not use commas to separate parameters. Arrays are indexed with square brackets as they are in Python or Java. Also, because the Adenine interpreter is written in Java, Adenine code can access Java objects (via the `importjava` command), call methods and access fields of Java objects using the dot operator, as is done in Java or Python. The execution model is quite similar to that of Java, Lisp, and Python in that an in-memory environment is used to store variables; in particular, execution state is *not* represented in RDF. Values in Adenine are represented as Java objects in the underlying system.

Because methods are simply resources of type `adenine:Method`, one can also specify other metadata for methods, as was mentioned earlier. Methods can be declared to have custom types and properties for use in a variety of situations. For example, in a statistical analysis application, the system can prompt the user to choose from the available Adenine methods marked as being suitable for the kind of analysis that needs to be performed. Similarly, methods, being resources, can also be used in the metadata of other resources, such as RDF classes. For example, reproducing the virtual table-style method resolution system present in object-oriented languages (i.e., given an instance, determine which class's implementation of a method to call) can be performed by associating methods with classes and using a query to resolve methods given an instance resource. Such a query can be set up to suit the particular object structure of the application, a runtime customization that normally requires techniques such as aspect-oriented programming.

5.3.2 RDF data types

The Adenine programming language natively supports RDF containers and other objects from the RDF API. A `LocalRDFContainer` can be created and initialized with a series of RDF statements by using curly braces (`{}`) (`=` is the Adenine assignment command and `x` is a variable name):

```
= x { :john rdf:type :Person ; dc:title "John" }
```

The same notation that was introduced in Section 5.2.2 can be used within the `{}` operator. In other words, resources can be named with angle brackets (`<>`), literals with double quotes (`""`), and existential nodes and lists with `${}` and `@()`, respectively. In fact these objects can even be used outside of `{}`:

```
= person1 ${
       rdf:type      :Person ;
       dc:title      "Mary"
}
= person2 <urn:sample:john>
= person3 :joe
= myNameAsALiteral "Dennis"
```

In other words, resources and literals are treated as built-in Adenine data types that can be used anywhere in Adenine. Statements generated as a result of `${}` declarations are sent into the target RDF container, as described in the next section. Furthermore, variables and expressions can be used in `{}`, and the values to which they evaluate will be used in constructing statements:

```
= peopleInformation { :dennis
       :knows @(
              person1
              person2
              person3
       ) ;
       dc:title      myNameAsALiteral ;
       :idNumber     (+ 45 50)
}
```

When an expression that appears in `{}` does not evaluate to a resource or a literal, a literal is constructed consisting of the expression's string representation. In the preceding example, the `:idNumber` property is set to be the literal `"90"`. Also, the block indentation rules do not apply within `{}` or `${}`; one is free to add whitespace to the expression as needed.

One may wonder why expressions can be used within `{}` in methods but not at the top level. Making another analogy to Lisp, `{}` expressions in methods can be described as being "quoted" with every element inside being "backquoted". Because backquoting occurs so frequently in Adenine and identifiers (e.g. `xyz`) do not conflict syntactically with resource and literal declarations (e.g. `<xyz>` and `"xyz"`), the backquoting is made to be implicit. In contrast, symbols, the Lisp analog to resource and literal values, are used to express both data in s-expressions and Lisp identifiers and therefore must be disambiguated. Another way to look at this difference is to say that in methods, `{}` is sugar for a large set of `add` function calls (introduced below), which naturally can accept Adenine expressions, whereas at the top level, statements in `{}` are treated as immediate values.

The `<>`, `""`, `@()` and `${}` syntaxes are preserved between the top level syntax and the method syntax for convenience. Resources and literals can also be constructed directly via the `Resource` and `Literal` functions:

```
# The following two expressions will evaluate to true
== <urn:test> (Resource 'urn:test')
== "Test" (Literal 'Test')
```

5.3.3 Dynamic variables

In addition to statically scoped variables, Adenine supports variables that are **dynamically scoped**. That is, the value of a dynamic variable is determined based on whatever value was last assigned to it in a given call stack, regardless of block structure. Dynamic variables are assigned using the `with` command and retain their value for the body of the `with` command, as well as for whatever functions are called within. For example, the following code prints out the number 25:

```
# __x__ is not defined at this point
function addToX y
       return (+ __x__ y)

with __x__ 10
       # __x__ is defined for the duration of this block,
       # as well as for whatever functions that are called within
       print (addToX 15)

# __x__ is now no longer defined
```

By convention, dynamic variables are prefixed and suffixed with two underscore characters to distinguish them from statically-scoped variables. Otherwise, if the same identi-

fier could refer to a statically-scoped and a dynamically-scoped variable, the statically-scoped variable will take precedence.

5.3.4 Manipulating RDF containers

We now introduce two standard dynamic variables that are always defined in an Adenine program: `__source__` and `__target__`. These variables refer to RDF containers and are used by the functions `add`, `remove`, `query`, `extract`, `contains`, and others. (These functions correspond to those exposed by the `IRDFContainer` interface discussed in Section 4.1.2.) In fact, the reason behind Adenine's support for dynamic variables can be seen if one compares the use of `__source__` and `__target__` with standard input and output in C programs: these standard file handles, when redirected, retain their redirection regardless of functional boundaries.

The `add` function adds the statements in the given RDF container into `__target__`. Consider the example below:

```
add { :john
      :hasChild ${
            rdf:type     :Person ;
            dc:title     "Joe"
      }
}
```

This example is in fact repeated from Section 5.2.2. The syntax of the `add` function was designed to duplicate the syntax used in the data definition language, except that Adenine expressions can also be incorporated into `{}`. Also, within `{}`, the `LocalRDFContainer` constructor, the `__target__` is set to be the `LocalRDFContainer` being created. The syntactic similarity between the data definition language and Adenine executable code makes it easy to add executable code expressions to data declarations that previously existed outside of methods, just as one can construct JavaServer Pages or Active Server Pages code by copying and pasting HTML and adding executable Java or JavaScript code expressions.

The `contains` function checks for the existence of a statement in `__source__`:

```
contains :john rdf:type :Person
```

The remaining functions support the use of **existential variables**, which serve as placeholders or wildcards. Existential variables are composed of a question mark (`?`) and any

sequence of alphanumeric characters. For example, the remove function deletes statements of the form given by the three parameters to remove from `__target__`, treating any existential variables as wildcards:

```
# Remove all information about :john
remove :john ?x ?y
```

The `extract` function, like the `contains` and `remove` functions, takes three parameters, a subject-predicate-object triple. However, exactly one of these three parameters must be an existential variable, and extract finds some statement that satisfies this pattern and returns a resource or literal that fills that position. For example:

```
= database { :john :nickname "Johnny" ; :nickname "Jonathan" }
with __source__ database
       print 'John likes to be called' (extract :john :nickname ?x)
```

This example, when run, would print either "Johnny" or "Jonathan". Notice the use of `with` for overriding the dynamic variable `__source__` for use with the `extract` function.

The `query` function allows you to express complex queries against the `__source__` RDF container. `query` usually takes either one or two parameters. The first parameter is a set of RDF statements that incorporate a series of existential variables. `query` returns a set of arrays that correspond to assignments of existential variables. When these assignments are substituted into the query specification, statements are formed that match statements in `__source__`. The second parameter allows you to specify which and in what order the existential variable matches will appear in the result set arrays.

The following code snippet shows some example queries. Note that the Adenine Console has a built-in RDF container that is attached to both `__source__` and `__target__`. Also, the `printset` command is used to print out the `Set` returned by `query`.

```
adenine> @prefix : <urn:sample:>
add { :bob :age "30" ; :hasSister :mary ; :hasSister :jane }
add { :mary :age "30" }
add { :jane :age "25" }
Result: null
adenine> printset (query { ?x :age "30" })
<urn:sample:bob>
<urn:sample:mary>
Result: null
adenine> printset (query { ?x :hasSister ?y } @(?x ?y))
```

```
<urn:sample:bob>    <urn:sample:mary>
<urn:sample:bob>    <urn:sample:jane>
Result: null
adenine> printset (query { ?x :hasSister ?y ?y :age ?z } @(?x ?z))
<urn:sample:bob>    "30"
<urn:sample:bob>    "25"
Result: null
adenine>
```

The first example asks for all `?x` whose `:age` property is `"30"`. The second example is more complex: it searches the `__source__` for all pairs `?x` and `?y` that satisfy the relationship `?x :hasSister :y` and asks that the rows in the result set be ordered `?x` then `?y`. The last example is more complex yet; here, a list of resources and their sisters' ages is requested.

5.3.5 Generating new code

In addition to adding statements one at a time, Adenine also supports adding new methods into an RDF store at runtime. The `method` and `uniqueMethod` commands can be used from within methods to define Adenine methods. (These commands are shorthand for the equivalent `add {}` and `${}` expressions to which they desugar.) The bodies of `method` and `uniqueMethod` commands are, unlike other blocks, scoped separately from their parent blocks. In order to access variables, the backquote operator (`` ` ``) must be prefixed to an expression. These expressions are then inserted into the resultant method according to the rules used to evaluate expressions in `{}` since these values must be serialized.

The `method` command resembles the `method` declaration used at the top level of the file, except that the first parameter may be any expression that evaluates to a resource, not just a URI. `uniqueMethod` takes a variable name as its first parameter and binds a randomly-generated URI to it, which also names the method that is defined.

5.4 Ozone ontology

The Adenine ontology was designed to allow programmers to easily express imperative programs that operate on the RDF data model using concepts similar to those that exist in language such as Java and Python. In contrast, the purpose of the Ozone ontology is to enable the declarative construction of functional programs. The contrast to draw here is this: imperative programming languages allow description of a program as a sequence of operations that affect the state of some virtual machine, precisely explaining how the

machine will end up in the desired result state. In contrast, declarative languages such as Ozone allow programs to be described directly in terms of the desired result, with respect to some target domain. Here the target domain is presentation: Ozone programs are primarily used to express ways of presenting information on the screen or other output device. Unlike the basic commands in the Adenine language, e.g., for loops and variable assignments, the basic code unit in Ozone is the primitive presentational element, such as a piece of text, an image, a horizontal line, etc. Containers also exist for the purposes of aggregation, such as paragraphs and row sets. Finally, like Adenine, Ozone also supports the notions of runtime environment and variable bindings, which we explore in this section.

5.4.1 Generalizing existing user interface specification models

User interface specification languages can be characterized by the extent to which they permit declarative composition of the interface. On one extreme are pure imperative approaches for assembling interfaces, such as the one adopted by Java for the Abstract Windowing Toolkit (AWT). Code must be written to construct objects corresponding to widgets, and method calls are made to indicate how widgets are placed within one another. On the other extreme are languages that permit user interface elements to be described purely in terms of the connections between the elements themselves, without the use of imperative code. Examples include HTML, XUL, and Guile—all XML-based languages for describing the composition of user interface elements.

The contrast to draw here is not one of the level of support for imperative programming. Indeed, HTML, XUL, and Guile are all associated with imperative languages (JavaScript and C) that can be used to both create and modify instantiated user interface specifications. Imperative techniques provide a certain degree of expressive power that is not always present or easily expressible in typical declarative user interface languages. On the other hand, purely imperative approaches such as the one adopted by Java's AWT bury the structure and composition of the resultant user interface within Turing universal pieces of code, making it difficult for a development environment to expose the declarative intent of simple user interface specifications. On a related note, the complexity associated with imperatively programming the user interface as compared to "designing" a user interface by assembling components is unnecessarily coupled into the AWT approach.

Incorporating a declarative approach is straightforward in the context of our data model. Most user interfaces are composed hierarchically, whereby user interface elements have at most one unique parent element; this hierarchical tree structure is easily expressed in a graph. Descriptions of these elements—called **prescriptions**—are the building blocks of the Ozone language. Prescriptions are resources with attributes such as color, font, size, margins, and textual content encoded as properties. Similarly, a "child" or "children" property exists off of certain kinds of container prescriptions to indicate that child elements are to be nested within a container.

RDF Schema—a general technique for describing the structure of data—is employed to describe the properties of a prescription. In fact, prescription resources have RDF types (called **prescription classes**) such as `slide:Text` or `slide:Image`. Furthermore, prescription resources have properties such as `slide:text` (the text to display) or `slide:source` (cf. `<img src="…">` in HTML) for describing the attributes of prescriptions. The major implication is that customization of prescriptions can be performed with any RDF editor.

5.4.2 Parts

Another way to view the user interface specification problem is in terms of assembling visual components, i.e., packaged pieces of code that perform some functionality. Our analog of a JavaBean (the Java component model term for a component) or a COM class (the Windows Component Object Model term for a component) is called a **part**. Parts are resources of type `ozone:Part` and have associated properties that describe how they are implemented. Haystack uses different kinds of parts to implement support for various facets of the system. What connects the component-oriented picture to the idea of a declarative specification language is the fact that it is the job of a part to handle one or more supported types of prescriptions.

At the implementation level, when a part is instantiated the part's parent or host passes it a **context property bag**—a runtime data structure consisting of a key-value pair map that informs the part of its configuration. (This structure is implemented as a Java object instead of in RDF for performance reasons.) The most important piece of configuration information is the prescription to render, which tells the part how to initialize itself when it is instantiated. Other information encoded in the context object includes the part's URI and custom settings that are passed to the part from its parent host. In addi-

tion, a context property bag contains a pointer to its parent context property bag (if one exists). This chaining setup allows parts to be aware of the "context" in which they are embedded. Also, properties can be *inherited* from parent contexts. Context property bag-based configuration settings are used frequently throughout the framework and serve as a kind of variable scoping mechanism for Ozone.

Also, in most modern component models (including user interface component models) there exists a substantial amount of registration metadata that describes what components exist, how they may be used, and how they are to be instantiated. COM utilizes a hierarchical key-value pair system called the Windows Registry for storing such information. JavaBeans uses both key-value pair files (cf. JavaBeans Activation Framework [95]) and class metadata (e.g., patterns for naming property manipulators and derivation from certain base classes) to encode this information. It is clear that we are able to take advantage of our data model to easily describe such information: We use RDF to specify the one or more prescription classes supported by any given part through the `ozone:dataDomain` property of the part resource. By hosting both the metadata describing the capabilities of parts and prescriptions in the same data model, the problem of instantiating the right parts for a user interface is reduced to a series of graph matches:

```
function findPartToInstantiate prescription
      return (queryExtract {
            ?x          rdf:type           ozone:Part ;
                        ozone:dataDomain   ?y
            prescription rdf:type          ?y
      } @(?x))[0]
```

5.4.3 Slide ontology

The Slide ontology was developed to provide basic support for formatting text, displaying images, and laying out other parts, much like the core syntactic HTML tag vocabulary does. The two simplest parts, the text and image parts, understand the `slide:Text` and `slide:Image` prescription classes, respectively. Like their HTML counterparts, the text part displays a string and the image part displays an image.

Most other parts serve as containers for these basic building blocks; many of these parts resemble in behavior their corresponding HTML tags. The `slide:Paragraph` prescription

instructs the system to lay out other parts in a paragraph flow. Similarly, the `slide:Span` prescription indicates that children parts are to be laid out continuously but does not induce breaks before or after itself. Two prescription classes play the role of the table tag: the `slide:RowSet` prescription calls for parts to be arranged as a series of rows and the `slide:ColumnSet` prescription as a series of columns.

Slide parts also support the notion of inherited style. Font attributes can be applied to part data and are inherited through context by children parts.

5.4.4 Interfacing with Adenine

Imperative code appears most frequently in the form of event handlers to elements such as menus and buttons. Most systems adopt hybrid approaches to compensate. In Guile and XUL, languages such as JavaScript can be employed to respond to events and to manipulate the assembly of user interface components at runtime. With HTML, two approaches have been taken, one resembling the method just mentioned, and the other using imperative code to generate HTML source code on demand (e.g., Microsoft Active Server Pages and JavaServer Pages).

Like HTML, XUL, and Guile, Ozone has left the job of providing imperative programming functionality to another language, namely Adenine. This allows us to avoid resorting to first order logic (or another highly expressive declarative approach) and its associated performance penalties for enriching the expressiveness of the user interface language. In our user interface paradigm, we have adopted a hybrid approach that incorporates features from the two approaches used for HTML. Taking advantage of Adenine's powerful support for creating RDF, our system can utilize Adenine scripts to generate prescriptions:

Also, Adenine can be used to write event handlers, which can interact with live user interface widgets. More precisely, event handlers are called behavior prescriptions for parts called behavior parts. Event handlers are attached to prescriptions via predicates such as `ozone:onClick`. The `adenine:Method` class is defined to be the prescription class for the `ozone:adenineBehaviorPart` behavior part. For example:

```
add { :exampleSlide
      rdf:type     slide:Text ;
```

```
        ozone:onClick :doSomething ;
        slide:text    "Click on me"
}

method :doSomething
        print 'user clicked'
```

5.4.5 Data providers

One of the main purposes of having prescriptions generated by Adenine scripts is to have those scripts extract the proper underlying information from the database for presentation. However, this approach inherits many of the problems of the web application approach to user interface generation. Data on the client-side representation becomes stale and must be refreshed constantly, and having to rerun scripts often can be expensive. Also, the way in which the data is queried from the database is often a graph pattern or another declarative form, and benefits can be realized from maintaining the declarativeness of the data retrieval specification without depending on intervening imperative code.

To support declarative approaches to data retrieval, Ozone uses **data providers**, which are parts without user interfaces but that support prescriptions that specify the kind of data to retrieve. Examples of data providers are the predicate set data provider, which returns a list of all objects hanging off of a given subject-predicate pair, and the RDF query data provider, which (more generally) returns all possible resources associated with one blank node in a graph matching pattern. Data providers are designed to abstract certain data extraction motifs, and data providers often register with the RDF store to receive motif-dependent events that affect their exposed result sets. For example, the RDF query data provider registers to receive events for every statement of the supplied graph pattern and automatically determines when the result set of the query has changed by re-performing the query when an event is fired.

Data providers are designed to be attached to **data consumers** that can retrieve the latest data from data providers and can be notified by their data providers when the underlying information changes. Some data providers are in themselves data consumers, allowing data providers to be chained and composed to increase the expressiveness of the kinds of queries that can be performed using data providers. Certain data providers are

specifically designed to be chained, such as the set merge data provider, which merges the result sets of the data providers chained to it. Furthermore, many Slide parts also act as data consumers, such as the text and image parts, which can be set up to display the most current piece of text or image associated with the result of a query and to automatically refresh their displays when the underlying information changes. Many prescription classes employ the `ozone:dataSource` predicate to hook prescriptions to data provider prescriptions describing the source of the data for that particular user interface element.

Chapter 6 Presenting information to users

In this chapter we begin to discuss the problem of exposing our data model to the user—one of the fundamental themes of this thesis. We observed in the previous two chapters that the difference in emphasis in terms of the design of the data model, i.e., fixed versus customizable schemas, led to changes in the programming paradigm. Similarly, we find that the approach taken to creating user interfaces must be changed as old assumptions about the data model no longer hold true. However, instead of developing new user interface paradigms from scratch, we wish to pick out the principles of effective user interfaces and reinterpret them in terms of our new data model. This involves breaking down familiar user interfaces into their elements, such as inspectors for displaying specific properties of objects or general containers for displaying sets of objects, and reassembling them based on the customizations to the data model in effect at the time.

After discussing several classes of problems present in current user interfaces, we make several observations on how interfaces should be dynamically constructed. We present the key theme of this chapter, views, which is the overarching paradigm by which information in the data layer is transformed into a presentation on the screen. Views are generally implemented in a combination of Adenine and Ozone. We also discuss aspects—an encapsulation of intensional knowledge that can be used as building blocks for views.

6.1 The problems with current user interfaces

The designers of commonly-used data-driven user interfaces have relied for decades on the simplifying assumption that the data being presented conforms to a set of schemata that are established ahead of time. Tools such as database form builders and rapid application development (RAD) environments simplify the task of assembling user interfaces for databases by allowing the *developer* to drag and drop table fields onto a canvas. This approach works well for the problems these tools were designed to solve—quickly putting together front ends for databases, which constitute a large fraction of the applications that are written today. Unfortunately, it becomes completely inadequate when the schema is subject to dynamic and unpredictable customization at runtime, as is the case with our data model. It is unreasonable to assume that a user will call the IT staff to redesign the database client every time he or she makes a change to the data model. Using

a semistructured data model allows programs to internally express custom relationships at runtime, but exposing a user interface with which the user can capture the power of the underlying data model is a nontrivial task.

The common approach to dealing with a customizable data model is to have an interface that is generic enough to work regardless of the level of customization. However, users will not benefit from an editor that exposes the raw generality of the data model, as does a directed graph editor. Graph editors often present interfaces that do not meet users' expectations: unlike organization and flow charts, some forms of data, such as calendars and text documents, do not lend themselves well to being displayed as directed graphs. Trying to schedule an appointment with a colleague by finding his or her node, finding a node corresponding to the required date and time, and drawing the proper arc between is patently overkill in comparison to the conventional method of creating an entry in a timetable. Many rudimentary solutions to the problem of exposing a user interface for semistructured data have relied on graph editors [26]. Similarly, knowledge representation editors such as Protégé, while relying on a frame-based representation instead of a graphical one, force users to think about their information and how it is organized in terms of first order logic [11].

To address these problems, one could mandate that programs be reformulated to include more advanced forms of end user customizability. However, there are common generalities that pervade the problem of presenting information in a customizable data model that warrant systemwide support. Indeed, without such base level support for customizability, each developer will be back to square one, developing their own interface paradigms and possibly creating programs with radically incongruent notions of customizing ontologies. For example, both Microsoft Word and Microsoft Outlook support custom key/value pairs on their documents, but in order to use a Word custom key to find a document, one must use Windows Explorer to search; to use a custom field to search for a contact or an appointment, one must use Outlook's search functionality.

6.2 Need for dynamic interfaces

We have observed that many user interface elements, such as hierarchies and forms, had specific properties that made them intuitive and useful under certain circumstances but not others. The problems exist not with the interface elements themselves but the under-

lying assumption that the user will always want to use a generic interface under too broad a set of circumstances. By adding support for allowing the user to see a particular interface style under appropriate circumstances, we can begin to correct some of these problems.

Accordingly, user interfaces should be constructed at runtime based on the user's data model. For example, when displaying the properties of a resource or of a set of resources, one approach is to dynamically introspect the objects' schemata and to present user interface elements that are capable of displaying specific properties of that schema in an appropriate fashion [24]. Modern desktop database applications are able to display automatically-generated form views for simple tables, assembling pre-built string editors, table widgets, etc. to present the properties of a resource. However, a problem arises when the interaction model does not match the user's mental model of how the information is characterized, such as using a form view to manage a calendar. Even assuming a flexible data model, it is the responsibility of the user interface to bridge the gap and expose the data model in an intuitive fashion.

Any solution to the semistructured information visualization problem must incorporate a flexible visual component model for allowing data of various types to be displayed to the user. The manner in which data should be presented depends on context and on what has been deemed important by the user at the time. For example, a graph representation of a workflow is useful when the user wishes to see an overview, but a tabular representation may be more helpful when the user wants to investigate the roles of the people involved in the workflow, and different components will be required for each case. Furthermore, a solution must take into account the differences between how schema designers model the structure of information for the purposes of knowledge representation and how users perceive the same. Either these differences can be specified declaratively, or specific user interface components can be created that are aware of these differences.

A paradigm in which the system can call upon a vast array of visual components for dynamically constructing a user interface represents a change in thinking in a number of different ways. First, instead of assuming that programs will present information to users in certain specific, predetermined ways, the assumption is that programs will be ex-

tensible to allow them to handle types of information of which their designers were unaware. In other words, programmers should design components that handle the rendition of certain kinds of information under specific circumstances and let the system choose the right component based on what circumstances are present at runtime. This is consistent with the notions of end user customizability and flexible ontologies.

Second, one reason overly general user interfaces are developed is that it is one of the few ways developers can build enough smarts into an application to handle a variety of different types. However, because of the wide availability of the Internet, it is now acceptable to assume that programs will be able to extend their functionality dynamically, eliminating the need for a one-size-fits-all solution. As new forms of information, schemata, and ontologies are developed, so can components for displaying and manipulating them be developed and shared with others. (The issue of collaboration will be discussed in Chapter 11.)

Finally, because adopting a semistructured data model dramatically increases the complexity of the data users can describe, programs must now be expected to be capable of presenting data in different ways in order to make it easier for users to comprehend and analyze their information. A crude analogy can be made to multidimensional manifolds: when viewing a two-dimensional curve, a person is able to understand the representation with one screen; when viewing an n-dimensional manifold, numerous cross sections are required since people are incapable of perceiving more than a fixed number of dimensions.

6.3 Views

Our mechanism for allowing information to be presented on screen in multiple, flexible ways is centered about a key concept called a **view**. Views are user interface elements that serve as on-screen proxies for resources. Views are also represented in the data model and have properties that characterize the style of presentation for a resource. Several views may be associated with any one resource; for example, a person may be displayed as an icon (an icon view) or as a large key-value pair listing as is the case in an address book (a property listing view). New views may be introduced into the system, allowing freedom and flexibility in terms of how an object may be presented in a manner most suitable to the current context. The work presented here on using views for

representation was done in conjunction with David Huynh; related discussion can be found in his thesis [94].

There are several elements of a view; the element most affecting the user is the **image**—the pixels on screen that are produced. Views are the only mechanism by which resources are displayed on screen. Because of this principle, we can specify that views' images will serve as first class *representatives* of their underlying resources on screen for the purposes of interaction. One benefit of this correspondence is that users are made able to work with their information by *direct manipulation*, an HCI concept that has been found to be successful in past research [51]. For example, users can drag and drop an image of a person into an image of an address book list as a means of indicating to the system that the person should be added to the address book list.

Views also have a presence in the RDF representation. A v**iew description** is a resource that describes, in some domain-specific fashion, a way in which a resource can be transformed into an image on screen. A view description, like most other resources, has a type, called a **view description class**. A view description class can be characterized according to the RDF classes of the resources that derivative view descriptions can characterize.

6.3.1 View parts

The view system is implemented by a special class of parts called **view parts**. The developer of a view part specifies in RDF what view description classes the view part is capable of rendering and the resulting image size class (e.g., full screen, in a one line summary, as an applet, etc.). Like other parts, view parts are initialized with a context property bag, which contains the resource whose image is to be created and the view description that characterizes how to create the image, and can embed other parts. However, the lifecycle of a view part differs somewhat from that of other parts. The first difference regards the way view parts are initialized. Views are usually presented on screen through an indirect request by the programmer in the form of a **view container prescription**. In such a prescription, the programmer specifies the resource to be displayed, a view description, and optionally the size class desired. Alternatively, the programmer can leave out the view description and the system will choose or create one automatically. For any given view description, several view parts may be available to render it,

differing usually by size class. Rarely does the programmer specify a specific view part to use; instead, he or she allows the view container to choose one based on certain conditions. The manner in which a view part is identified as being suitable for use by the view container part is a specific example of the graph matching technique introduced in Section 4.1.3.

Any part can take advantage of the view part "marketplace" when displaying information by identifying portions to be displayed by sub-parts. For example, the definition of a meeting resource might have a specification of the meeting place, the date and time of the meeting, and a list of attendees. However, a meeting view part needs not be responsible for actually rendering the date and time of the meeting itself; it can delegate that portion of the information by embedding a view container part to find a view part whose specification matches the date and time portion of the meeting resource.

6.3.2 Supporting direct manipulation

Also, because the system is responsible for instantiating view parts and keeping track of where child images are to be embedded within parent images, the system can provide default implementations of certain direct manipulation features for free, such as drag and drop and context menus. For example, when the user starts to drag on an image, the system knows what resource is being represented by that image, such that when the image is dropped elsewhere in the user interface, the drop target can be informed of what resource was involved instead of simply the textual or graphical content of the particular image that was dragged.

As another example, consider the act of filling in a list of meeting attendees on a form. Instead of retyping or copying and pasting names of people from an address book, a user can drag and drop contact images from an address book into an area on the form. Because the images representing contacts in the address book are associated with the resources they represent and not just the names of the contacts, the identities of the contacts' resources can be preserved. The alternative opens the possibility for ambiguity because information is lost. For example, what if there are two people named "John Doe" known to the system? Specifying the text string alone is not sufficient to disambiguate which John Doe is intended, even though it is clear that the John Doe desired is the one that the user selected in the address book.

Context menus, a popular interface mechanism, can be similarly supported by the base system. When a user right clicks on an image, the system can determine from context what the underlying resource is and present a list of possible actions. In addition, because of view nesting, sometimes the specific resource being referred to by the user is not clear. For example, when a user right clicks on an entry in an address book, he or she may be referring to that entry or the address book as a whole. By using the context property bag chain, the system can determine the possible resources of interest and allow the user to choose an action against any of these resources. The specific means by which context menus and drag and drop are implemented is described in Sections 7.2.1 and 7.4, respectively.

6.3.3 Layout managers

Often, view composition takes the form of displaying a list or set of resources. Ozone supports the notion of a **layout manager** for this frequently used paradigm. A prescription for a layout manager specifies a data provider prescription that tells the layout manager what resources need to be rendered, and the layout manager then lays them out in a certain fashion. Basic layout managers include the row stacker, which displays a set of views one on top of another, the column stacker (analogous to the row stacker), and the list view, which like the row stacker displays a list of resources but includes more powerful functionality such as scroll bar support and multiple selection.

6.4 Navigation model

Ozone's navigation model utilizes views in several ways to make it easy for users to work with the system. The web browser paradigm serves as the basis for our navigation model. The concept of the current page being browsed is extended to include all resources, including those not named by URLs. Back, forward, and refresh buttons work as they do in the web browser. Furthermore, a home button is provided to take the user to his or her home page, which is displayed on startup.

The current resource displayed in Ozone's main **content pane** is rendered by means of a special kind of view part called an interactive view part. These parts are designed to be displayed full screen. In addition to the content pane, the **start pane** provides a convenient area for applet-sized views of resources to be docked for frequent use. Furthermore, several kinds of smaller summary view parts in the system are designed to be used like

hyperlinks: when such a summary view is clicked upon by the user, the main content pane navigates to the underlying resource represented by that view.

6.5 Aspects

We have characterized views as extensible and context-sensitive means of displaying visual representations of resources. We broke down the monolithic approach used by many programs that binds data to a specific mode of presentation to one in which methods of visualization are first class and interchangeable. Here we make one further analysis into the ways in which information about resources can be presented to users.

One useful way to think about what information to show concerning any given resource is to decompose a resource into its different **aspects**. For example, considering a vacation itinerary, one might come up with the following (incomplete) list of aspects, i.e., the kinds of things someone might want to know about an itinerary:

1. The dates of the vacation
2. The destination
3. Price
4. Who is going
5. The weather at the destination
6. Lodging
7. How it fits into the budget

From this standpoint, one could consider the process of creating views to be a matter of ensuring that some combination of relevant aspects appears on the screen. While views answer the question of how to show something, aspects answer the question of what to show of something. In this regard, aspects deserve to be considered as first class concepts just as views are. In this section we formalize an ontology for describing the process of using aspects for presentation and show how aspects are implemented within Haystack.

6.5.1 Aspects versus properties

From the perspective of knowledge representation, the process of deciding what aspects exist for a given class coincides significantly with the process of deciding what properties exist for that class. In fact, it would not be uncommon for a schema designer to designate the first four aspects given in the example above as being properties of the itinerary class. What is less certain is whether the remaining three deserve to be called properties in their current incarnation. In terms of making a succinct ontology for trips, the weather at the destination is not so much a property of the itinerary as it is a property of the destination itself. Likewise, if an itinerary is modeled in terms of a list of events (e.g., flights and hotel stays), then lodging is not so much a property *per se* as it is a restriction of the list of events to just the hotel stays. Finally, how the itinerary fits into the budget is more of a summary of a variety of different objects associated with the itinerary, including the financial transactions from the funding bank accounts and the ratio between the total cost of the trip (which is itself a property derived from the cost properties of each individual event in the itinerary) and the amount allocated for travel in a given budget. In other words, the final three aspects would be more efficiently expressed as queries or formulas over existing properties.

For any given schematization of a class, there are likely to be multiple, overlapping sets of property specifications one could come up with. Nevertheless, while treating the final three aspects as first class properties from an ontological perspective may seem redundant to some, there is no denying the human-centric utility of these aspects being first class. Given a first class name, we can make assertions such as "when displaying a travel itinerary, it is important to show how it fits into the budget." To adjust for this inadequacy of properties for describing the salience of arbitrary aspects of a resource, we define an aspect to be a function that, when applied to a resource, yields some relevant piece of information—a generalization of RDF properties. Note that concerns about redundancy are part of our justification for the existence of aspects that are not actualized as RDF properties. Nothing fundamentally stops either a user or a developer from creating a property called `travel:howItFitsIntoTheBudget`. We contend that developers are likely to understand the difficulties of keeping such derivative properties in sync; later we describe how aspects can mitigate some of these synchronization problems.

6.5.2 Characterizing aspects

RDF Schema and DAML+OIL are ontologies for characterizing properties—a special case of an aspect in which the values produced by the aspect (if you consider the aspect to be a function) with respect to any given resource are materialized in the RDF store as statements of the form:

```
[given resource] [RDF property] [value]
```

The key characteristics given by RDF Schema are domain and range. Turning towards aspects in general, we seek to define an ontology for aspects that allow us to describe user-level concepts such as the aspects given in the earlier example. To do this we define a set of aspect classes; this is in analogy to DAML+OIL, which divides properties into classes including literal properties, resource properties, unique properties, etc. Below we define two basic aspect classes, data set aspects and group aspects.

A data set aspect (`aspect:DataSetAspect`) is an aspect that is defined by a data source (see Section 5.4.5). Typically, data set aspects are used to describe aspects where the relationship between the argument and the values can be expressed in terms of a data source. The lodging and weather at destination aspects could be defined in this fashion.

A group aspect (`metadata:MetadataAspect`) allows other aspects to be aggregated together. For example, an address aspect may be described as a group aspect in which the street name, street number, city, state, and zip code properties (i.e., aspects) are combined. Group aspects are described in more detail in Section 9.1.1.

Other aspect classes can be defined by the developer to characterize aspects in greater detail. Custom aspect classes are created when the semantics of an aspect, such as the "how it fits into the budget" aspect, cannot be adequately described with existing vocabulary because there is embedded heuristic or presentational knowledge not being conveyed; the new class can be defined to incorporate such heuristic or presentational knowledge into its semantics. Most of the specialized aspects described in this thesis, such as the Organize aspect (Section 8.3.2) and the Annotation aspect (Section 9.2), are examples of instances of custom aspect classes. We discuss below how support for such aspects' semantics can be implemented in Haystack.

6.5.3 Aspect parts

The semantics of aspects are implemented by **aspect parts**, which come in two flavors. Visual aspect parts are similar to view parts in that they take an argument and produce a presentation on screen. Like view parts, visual aspect parts can also be members of size classes. Data provider aspect parts similarly take an argument but expose a data provider interface; they allow programmers to evaluate the value of an aspect given a resource. Like all parts, aspect parts use a context property bag to initialize themselves, can embed other parts, and can be embedded in other parts.

6.5.4 Embedding aspects in the user interface

One natural use of visual aspect parts is for use in views. However, it is worth noting that by using data providers, Slide elements, view container parts, and layout managers, we can already achieve a large range of different presentation possibilities. What visual aspect parts add to this picture is twofold.

First, views can be constructed completely automatically from a listing of aspects. These views are called **aspect views** (i.e., views composed of aspects). An aspect view of a weather agent can be found in Figure 19.

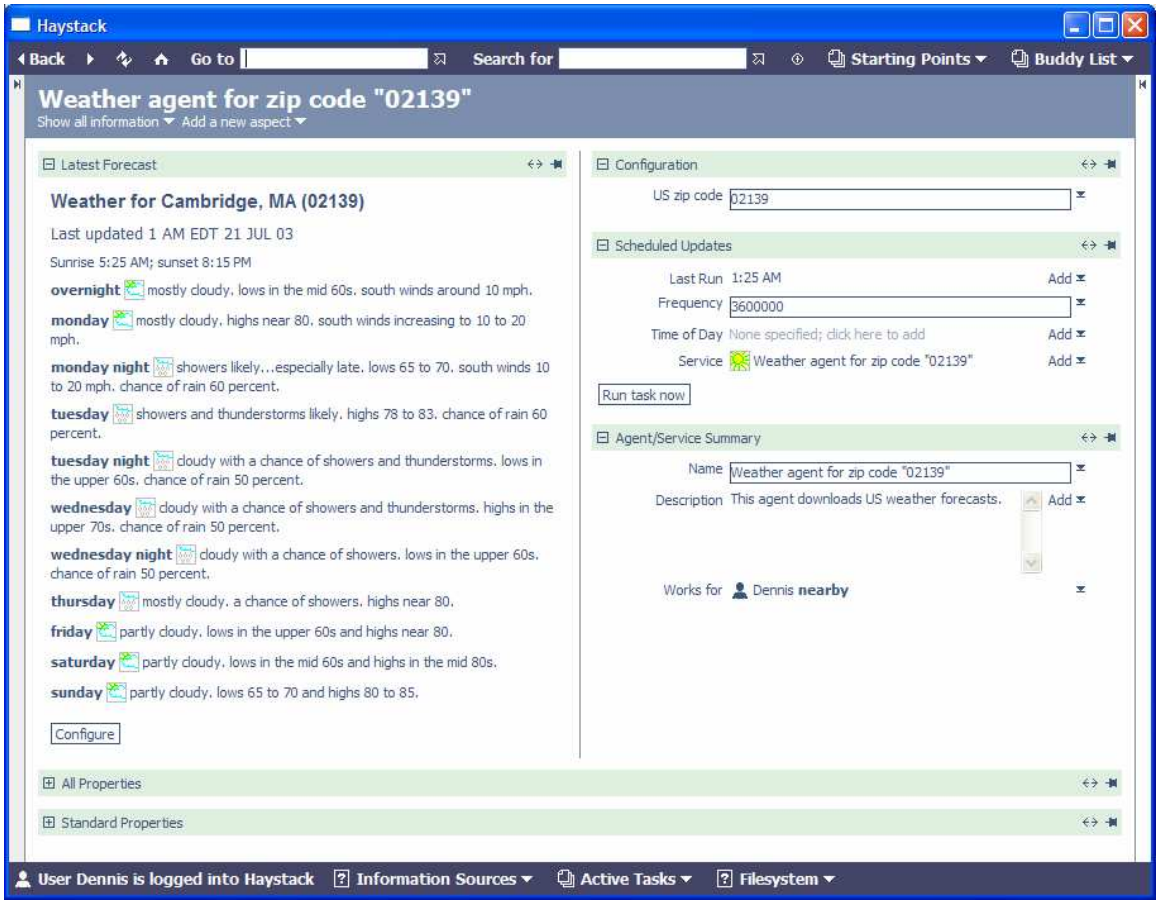

Figure 19: Weather agent aspect view

While it will still be possible to write view parts that are highly tailored to any given resource being presented, aspect views require no programming to assemble. Generally, aspect views display all aspects associated with the types of the resource being rendered. Classes can specify an associated aspect with the `aspect:aspect` property. This approach is also useful for when a resource has more than one type; the aspect view will fold the different aspects from the different classes together into one display.

The second way visual aspect parts are useful during view construction is that they act as reusable components for displaying information. The process of embedding aspects is analogous to that of embedding views. Because there can be potentially several different visual aspect parts capable of handling a given aspect, an **aspect container prescription** is usually employed so that an appropriate visual aspect part can be selected based on the size class needed.

View part implementations are the primary place one will find aspects being used, but another example is the **tool pane**, found on the right hand side of the Haystack user interface. The tool pane acts as a permanent extension of the view presented in the content pane and a docking area for aspects; frequently-used aspects may be dragged and dropped onto the tool pane. Aspects located there display information regarding the resource being viewed in the main content pane. Used outside of an aspect view, aspects act as magnifying glasses, exposing some "aspect" of whatever resource is selected in the content pane. Users can select the resource being "inspected" by use of the "Apply tools to" command on the context menu.

Chapter 7 Manipulating information

The previous chapter introduced the fundamental abstraction by which Haystack presents information to users. In this chapter we discuss operations, the corresponding abstraction for supporting the manipulation of information. Following a common theme found in this thesis, we show that by adding declarative metadata to ordinary Adenine methods, operations can be exposed to the user without the developer having to manually specify much of the normal user interface scaffolding employed in user interface frameworks today, such as custom dialog boxes and tool palettes. Operations can be organized and customized by users in ways not previously possible because of the way the framework takes advantage of the metadata describing operations. We also discuss the use of a specific kind of operation called a constructor, which abstracts the notion of resource creation.

7.1 Exposing functionality in the user interface

The role of many user interface components is to allow users to work with their information. Toolbars, menus, and dialog boxes figure among the more popular means exposed by programs today for allowing users to manipulate information. At their core, toolbar buttons, menu items, and modal dialog boxes encapsulate specific pieces of functionality, which we will term **operations**. In an abstract sense, an operation represents some function exposed to the user.

For an environment such as Haystack, which seeks to expose a highly expressive data model as faithfully as possible, one might ask what can be achieved with operations that cannot be accomplished by more direct means (e.g., addition of RDF statements). The answer lies not only in the fact that encapsulation is a fundamental mechanism for maintaining the integrity of the data model, but also in that users are unlikely to be familiar with the way in which any particular ontology should be mutated at the raw RDF level. In Haystack, operations therefore serve a necessary albeit redundant purpose, unlike the case with other systems, in which they are not redundant because direct mutation is impossible. In Chapter 9 we will explore means for allowing users to manipulate data at a rawer level and explore when this might be appropriate. The choice of to encapsulate or not to encapsulate therefore is analogous to the argument that can be made between ex-

posing methods and fields[4] in a class of an object-oriented program: fields are for primitive properties, and methods are for more complex manipulations or mutations.

Operations as they are exposed in today's applications carry with them a number of user interface problems. As is the case with modern data models, dialog boxes usually embody fixed, immutable specifications. While menus and toolbars are sometimes configurable, the extent to which they can be configured is usually limited to superficial rearrangement, as there are few possibilities for customizing functionality (e.g., adding new operations) given a fixed data model. Developers often take a fixed data model for granted and as exhaustively as possible envision the most useful manipulation elements for the application. These elements form the basis for a program's menus, toolbars, and dialog boxes.

On the other hand, when programs expose information interfaces partially composed of pieces defined by other programs, toolbar, menu, and dialog box specifications must be reformulated to accommodate foreign data types. Several attempts have been made in the past to create schemes for merging manipulation elements from multiple applications together, including Microsoft's Object Linking and Embedding (OLE) in-place activation, but these schemes only overlay otherwise separate application interfaces.

Instead, we contend that like the well-encapsulated data objects discussed earlier, toolbars, menus, and dialog boxes must be broken down into their component operations, in order to effectively create interfaces for manipulating resources that effectively address whatever user interface context is at hand. Some operations require several parameters, which are often requested from the user by means of a dialog box, while some operations require only one parameter, usually the selected object. The properties of an operation (e.g., its name, parameters, etc.) can then be described in our semistructured data model much as the properties of any given resource can. Also, different means of presentation (e.g., toolbar button, menu item, dialog box) can be effected as different views for an operation. Our discussion of operations as resources builds upon previous work done on the use of declarative specifications for commands; Myers et al. applied the technique of treating commands as objects for the purposes of supporting undo [64].

---

[4] By field, we also include the notion of getter/setter methods; in languages such as C# and Python, getter/setter method pairs can be exposed as fields.

Because operations are encapsulated as first class entities, their availability is not limited to certain "applications" that remember to expose them. Rather, the user interface framework can make operations available whenever a user is working with objects to which that operation applies, based on objects' type signatures and the user's current context. As a result, users can manipulate any particular resource in the same fashion regardless of the containing application because the operations defined for that resource will have been defined centrally in a standard manner. Similarly, specifying operations declaratively in the data model also enables many hard-coded user interface elements to be generated dynamically and consistently according to the user's data model. Dialog boxes and menus can be constructed based on the definitions of the operations they represent while taking into account contextual information.

7.2 Operation ontology

Most systems provide some mechanism for exposing prepackaged functionality that can be applied under specific circumstances. For example, in Java one can expose methods in a class definition that perform specific tasks when invoked. In C one can define functions that accept arguments of particular types. Under Windows, one can define verbs, which are bound to specific file types and perform actions such as opening or printing a document when activated through a context menu in the Windows Explorer shell. In general, these mechanisms all permit parameterized operations to be defined and exposed to clients.

Operations are simply Adenine methods marked up with additional metadata for use by the user interface framework. This additional metadata is best explained in the context of an example. The definition of the "Browse To" operation is given in the following code snippet.

```
@prefix op: <http://haystack.lcs.mit.edu/schemata/operation#>


add { :target
    rdf:type              op:Parameter ;
    rdf:type              daml:ObjectProperty ;
    dc:title              "Target" ;
    op:required           "true" ;
    rdfs:range            daml:Thing
```

```
}

method :browseToOperation :target = x ;
rdf:type            op:Operation ;
dc:title            "Browse to" ;
ozone:icon          <http://haystack.lcs.mit.edu/icons/verbs/browseto.gif> ;
adenine:preload     "true"
    (__context__.getService ozone:navigationMaster).requestViewing x[0]
```

The definition of an operation (e.g. `:browseToOperation`) includes basic information such as its name and icon. Named parameters of the Adenine method (e.g. `:target`) can be exposed to the user and given human-readable names by giving additional information regarding the typing of the parameter. The most basic mechanism for typing is simply specifying an `rdfs:Class` as a parameter's class using the `rdfs:range` predicate. (The reason for reusing the `rdfs:range` predicate will become evident below.) A parameter's type constraint can also be specified by giving an Adenine validator method, which given a value verifies that it can be used for that parameter. Finally, parameters can be specified as either mandatory (`op:required "true"`) or optional.

When an operation is invoked, the values assigned to the operation's parameters are passed to the Adenine method by means of the named parameter facility. In other words, each of the operation's parameters is passed to the Adenine method as a parameter named by the operation parameter's URI. Parameters can have multiple values; for example, a send mail operation may allow multiple recipients to be specified. To allow for this, the Adenine method receives a list of all values for each named parameter.

7.2.1 Exposing operations in the user interface

The operations abstraction allows the functionality of the system to be arbitrarily extended, without special plug-in interfaces or points of extensibility needing to be defined on a per-application basis. Furthermore, developers can declaratively specify new functionality to the system rather than modify monolithic user interface elements such as dialog boxes, menus, and toolbars. This is because such user interface elements are constructed dynamically based on available information.

One of the ways in which Haystack exposes the operations installed in the system is on the tool pane in Haystack in the form of an aspect. The Recommended Operations aspect monitors the currently viewed resource in the content pane and displays a list of operations that apply to that resource specifically (as opposed to operations that can be applied to any resource). This matching is performed by confirming that the selected resource satisfies the type constraints of one or more parameters of an operation. Similarly, when a user right-clicks on a view in the system, a context menu is produced that displays the possible operations that can be performed against the underlying resource.

7.2.2 Operation closures

In the object-oriented paradigm, objects are said to pass messages to each other. The contents of these messages are in effect composed of a message name and a set of parameters. Reifying the notion of an object method call enables certain desirable characteristics. For example, messages can be saved for later reuse. Also, messages can be incrementally constructed and invoked after all parameters have been specified.

The analogous concept in Haystack is the **operation closure**, which is a resource that has, as properties, the named parameters for an operation in progress. Because parameters are treated as properties, the `rdfs:range` predicate can be used to designate the type of a parameter. Furthermore, general techniques for editing metadata, as presented in

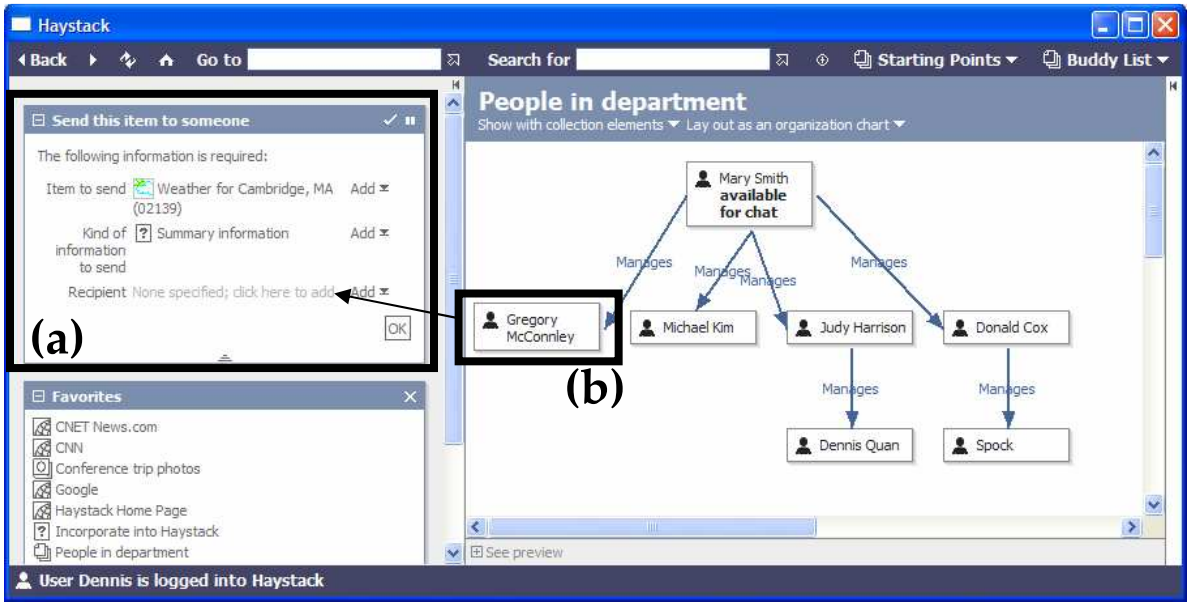

Figure 20: Screenshot of user interface continuation: (a) the user interface continuation; (b) using the context menu to specify parameters to the user interface continuation

Chapter 9, can be applied to the problem of prompting the user for parameters. Using these techniques, Haystack eliminates the need for developers to create specialized user interfaces for user-performable operations in many cases. In other words, an entire class of user interface gadgetry in most applications is reduced to a specific instance of the more general concept of resource editing.

For example, when an operation that requires parameters is activated, Haystack checks to see if the target object (in the case of the command being issued from a context menu or the tool pane) satisfies any of the operation's parameters. If there are unresolved or ambiguously resolved parameters, Haystack exposes the in-progress operation closure as a **UI continuation** (see Figure 20).

The definition of continuation we adopt here arises from the literature on continuation-passing style [114]. Conventional programs use stack frames to keep track of which function is currently being executed. A function completes when it releases its stack frame and returns to the calling function (the parent stack frame). In contrast, continuation passing style does not use a stack; instead, functions are called with an extra parameter known as a continuation. As the name implies, a continuation is a function that represents the remaining flow of execution of a program. Instead of returning a value, a function written in continuation passing style calls the supplied continuation with the return value. By analogy, a dialog box under our scheme calls the supplied continuation with the data gathered from the user.

Like a dialog box, a UI continuation prompts the user for needed information—in this case, the unresolved parameters. However, unlike most dialog boxes, which are modal, UI continuations are modelessly placed on the start pane, allowing the user to use any tool in the system to find the information needed to complete the operation. By default, the system takes the user to a convenient place to find the required information, such as in the case of a send e-mail operation, the user's address book. This interface is similar to a shopping cart on an e-business website: the user can drag and drop relevant items into the "bins" representing the operation's parameters. The user can even decide to perform other tasks and come back to the operation later. When the user has finished obtaining the necessary information and is ready to commence the operation, he or she can click the "OK" button on the UI continuation. The system then performs the operation.

UI continuations also exist outside the context of operations. Consider by analogy the "input" function, which exists in many languages such as Python and Visual Basic, that prompts the user for input, waits for the user to finish typing, and returns the inputted datum captured from the user. Rewriting the input function in continuation passing style, the input function would return immediately and take an additional parameter—a continuation—to which the inputted datum would be passed when the user finishes entering his or her information through the keyboard. This same principle is embodied by UI continuations in Haystack. An Adenine method analogous to the input function is exposed by the system taking two parameters, a validator method and a continuation method. The validator method is used to test whether a given resource can be used for the continuation. When the user has dropped a resource into the UI continuation's view or has selected a resource by means of the context menu for use by the UI continuation, the continuation method is called. (In the case of an operation closure's UI continuation, a helper method that invokes the operation with the specified parameters is used as the continuation method.)

By providing UI continuation functionality, the system frees the developer from needing to design specialized, miniature user interfaces for retrieving information from within modal dialog boxes by reusing the existing browsing environment and at the same time providing the user with a seamless experience. Furthermore, since the UI continuation is displayed using Haystack's view technology, developers are free to customize the display of a UI continuation by defining new view parts.

7.3 Currying

Another use of first class operations is that users are able to save an in-progress operation closure and turn it into a new operation by selecting the option from the UI continuation's context menu that instructs the system to bind the state of the current operation together with the already specified parameters. This binding process can be described as currying followed by partial evaluation, but we will refer to the entire process as currying as a shorthand. Currying is a term used in programming languages such as Haskell and ML that refers to the conversion of a function that takes n parameters into a "curried" function that takes the first original parameter and returns a function that accepts the remaining $n - 1$ parameters (also in a curried fashion). In other words, currying takes a function f of the form:

$f : a_1 \times a_2 \times \ldots \times a_n \rightarrow b$

and turns it into a function of the form:

curry$[f] : a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow b$

Similarly, when a user creates a new operation through currying, he or she is creating a new function from the curried form of the original function in which the parameters that have already been specified have been applied to the curried function. Another way to put this is if a user wishes to curry an operation f with the parameters a_1 through a_m already specified, then the resulting curried operation g has the following form:

$g =$ **uncurry**[**curry** $[f]\ a_1\ a_2\ \ldots\ a_m] : a_m \times a_{m+1} \times \ldots \times a_n \rightarrow b$

There is nothing special about the way in which currying is implemented in Haystack. Instead, currying is exposed to the user as simply another operation that takes an existing operation closure and a name as parameters. Figure 21 illustrates the use of a context menu for saving an operation closure as a new operation.

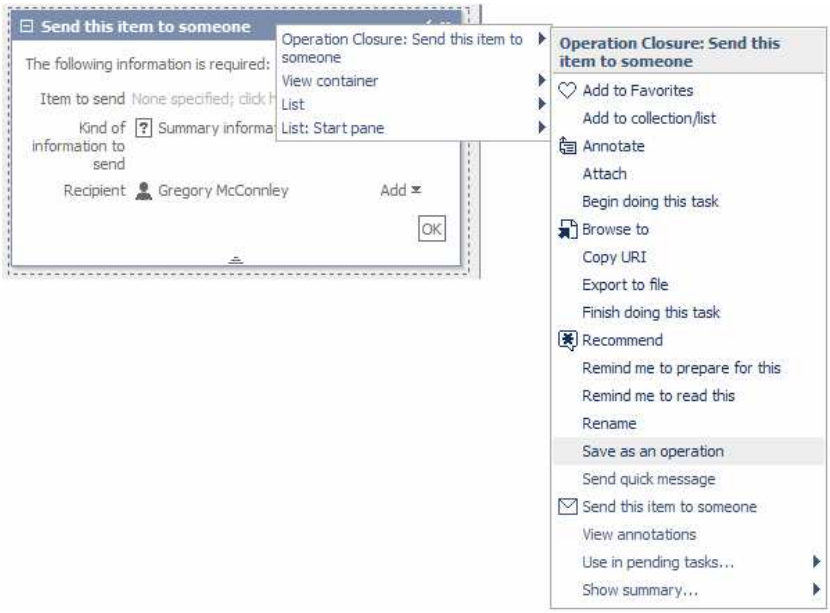

Figure 21: Currying an operation from the context menu

Additionally, the screenshot given in Figure 22 depicts a user filling in a user interface continuation to create a new operation from an existing user interface continuation.

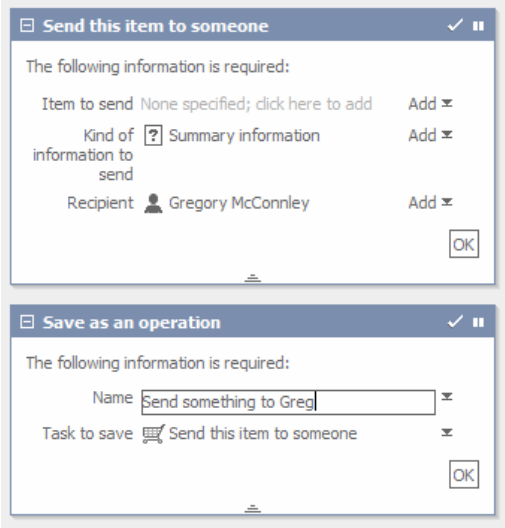

Figure 22: Creating a curried operation

One benefit of currying is its ability to allow users to create specialized commands suited for their own purposes. In the example depicted in Figure 22, the user has likely observed that he or she sends summaries to Gregory McConnley frequently enough to warrant its own command. Most existing environments support this level of customized functionality only through macros or most recently used (MRU) lists. Furthermore, because curried operations and user interface continuations are described in the data model, they can be organized, searched, and shared with others just as documents can (e.g., placed in folders, sent as e-mail attachments, etc.).

Finally, currying can be used to construct first class support for command customizations. For example, the Print dialog box in Windows remembers settings such as copy count, which printer to use, and collation options only for the lifetime of an application. This simple approach obscures two important user interface problems. First, unless the user has observed the Print dialog box's behavior over a long period of time, the circumstances under which a program retains the last used print options may be unclear to the user at first. Second, there is no support for remembering more than one of the user's frequently-used configurations (e.g. double-sided duplicate copies with staples, one-sided single copy to the color printer, etc.). By storing these settings in curried operations, applications can give users first class support for commonly-used groups of settings while removing the ambiguity surrounding an application's policy on maintaining

default options. (Developers would have to expose settings such as double-sidedness as parameters to the print operation rather than as properties of the printer.) Although we have not implemented support for automatic generation of MRU lists, previous work has explored the notion of exposing such MRUs in the user interface [65].

7.4 Drag and drop

Drag and drop is a user interface paradigm that allows the user to indicate that some operation should occur on the object being dragged with respect to the drop target. Accordingly, drag and drop is simply another mechanism for performing an operation that takes at least two parameters. In Haystack, such operations have type `dnd:DragAndDropOperation` and have two distinguished parameters, one having type `dnd:DragParameter` and the other having type `dnd:DropParameter`. Unlike the case with context menus, drag and drop being implemented in this fashion introduces the possibility for irresolvable ambiguity. The system displays a menu to ask the user for clarification under these circumstances.

7.5 Constructors

One final example of manipulation supported by the operation abstraction is object creation. Object creation manifests itself in many different forms, ranging from the addition of a text box to a slide in a presentation graphics program to the opening of a bank account. Applications that support object creation usually expose interfaces for allowing users to choose the appropriate type of object to create or to find a template or wizard that can help guide them through the process of creating the object.

By analogy to the process performed by most programming languages, the process of creation can naïvely be thought of as the coining of a fresh URI followed by an `rdf:type` assertion. The corresponding choice list for creating objects in RDF could be implemented by displaying a list of all `rdfs:Class` resources known by the system. However, there are many issues not addressed by this solution. The user's mental model of object creation may map onto three distinct activities in the programmatic sense: (1) creation of the resource; (2) establishing some default view; (3) population of the resource with default data. For example, the creation of a picture album from the perspective of the data model is straightforward in that a picture album is simply a collection of resources that happen to be pictures. However, if the user begins viewing this blank picture album

with an address book view, he or she may believe that the system has created the wrong object. With respect to the third point, Gamma et al. assert that object creation can come about in various ways, ranging from straightforward instantiation to creating objects according to some fixed pattern [66].

Furthermore, the classical framing of the object creation problem does not address the user interface implications entailed by certain kinds of instantiations. Some objects can be created without further input from the user, such as empty collections, while some objects require configuration data or other information to be properly initialized, such as a POP3 mail agent.

To solve these problems, Haystack makes use of a constructor ontology, which describes resources called constructors that create new resources. Constructors have type `construct:Constructor` and are implemented as Adenine methods that return constructed resources. Constructors that are designed to be exposed in the user interface also have type `op:Operation` assigned to them, allowing them to be invoked in the same ways as other operations.

7.5.1 Basic construction patterns

Constructors can be used to implement traditional object creation functionality, whereby a user is prompted for certain parameters and a new resource is created. A slight twist on this approach is one in which resources are *incrementally* created based on the amount of information that a user is willing to provide at any given time. Instead of requiring a user to fill in all necessary fields in the schema corresponding to the class being instantiated, we allow the user to create small, basic elements and then apply constructors to build more complex resources from these small elements. In particular, the user is not assigning a "type" *per se* to the resource being created but is delaying this decision until later. This notion takes into account the process of refining information that is recorded until it reaches its final form. For example, when one is on the telephone jotting down the details of a meeting from a voicemail, one does not always think of first creating a meeting document and then filling in the details. Instead, one may write down random pieces of text and then look for ways to glue such text together into more cohesive wholes.

One of these elemental constructors is the text fragment constructor, which very simply accepts a fragment of text from the user and creates a resource to identify it. Jotting down bits of text is an extremely frequent activity, and this constructor is designed to support this activity. When the user is ready to promote the text to something more complicated, he or she can apply constructors such as “Make into to-do item” or “Make into message”.

Other simple constructor patterns will be introduced throughout the rest of the thesis. In Section 8.3 we discuss a very simple constructor meant to help the user create a list quickly. In Section 9.3 we introduce the graph editor, which supports an activity corresponding to doodling on paper.

Chapter 8 Managing collections

Perhaps the most important function of an information management environment is allowing users to find relevant information efficiently and effectively. Lansdale states that retrieval begins with the user identifying whatever characteristics are known about what is being sought and ends with the user scanning through the collection of items known to match those characteristics [18]. The more criteria a user can identify up front, the smaller the result set that must be scanned through. Unfortunately, many of these easily-recalled characteristics are not often captured when the information is recorded. Our rich data model provides many opportunities not only to permit such characteristics to be recorded but also to facilitate the retrieval of sought-after items given these characteristics. In order to further the end purpose for organizing information, i.e., enabling information to be retrieved later, it is critical that organizational information be as convenient to record as possible.

In this chapter we explore specifically the concept of a collection, a kind of "unary predicate" that serves as the basic unit of aggregation in Haystack. First we give an analysis of current organizational tools to determine which techniques have worked and which need further improvement. We then recast these techniques in terms of collections and specifically discuss three purposes for collections: *ad hoc* resource specification, categorization, and navigation pathways. We propose several methods for exposing collections in the user interface that improve the convenience with which users can organize their information. Finally, we show with a user study that multiple categorization—allowing resources to be in more than collection at once—is superior in many respects to the hierarchical folder schemes in popular use today. In Chapter 9 we will explore more general methods for allowing users to record information.

8.1 Current approaches to organization

The most popular organizational tool for documents today is the hierarchical folder system, but this system was designed in analogy to that used in filing cabinets for centuries and carries with it problems from a previous era. For example, does a document named "Network Admin Department Budget for 2001" belong in the "Budget" folder, the "2001" folder, or the "Network Admin" folder? Supposing the user finds justification for placing it in just one of these folders at one point in time, it is very possible that the user

may forget this rationale some time later and expect the document to be in a different folder. It may also be the case that some days a user will be interested in a time-based classification and other days a department-based classification.

There is significant psychological evidence that classification of documents into a single folder or category is the wrong approach. In Section 2.2 we discussed Lansdale's observation that the act of classification into a single category is cognitively difficult [18]. He cited research on office space organization done by Malone, who found that people who indiscriminately pile paper documents do so in order to skirt the problem of having to choose between several potentially overlapping categories [19]. (Some have actually suggested that piling may be useful for users who rely on spatial memory to recall documents [21].) Whittaker et al. similarly found in their studies of users and their e-mail corpora that any message of nontrivial length has several axes along which the message may be filed [32].

Unfortunately, current user interfaces do little to encourage simultaneous classification into a folder hierarchy. Few, if any programs can be found whose file save feature prompts the user for all the possible folders into which to place a file. Given a lack of support from the user interface, few users can be expected to save their files in one place, and then separately create shortcuts, aliases or symbolic links in the other folders.

As users' information influxes grow monotonically with time, the importance of supporting convenient multiple categorization increases, both in the data model and in the user interface. Our directed graph-based data model makes it easy to associate any object with one or more categories by means of a "membership" relationship between the object and its categories. To expose such a system to the user, many solutions have been proposed [39]. The most important characteristic of these solutions is that the concept of multiple categorization is emphasized in the design of the user interface.

8.2 Collections

To address the problems given above, we start off with a data model-level generalization called a **collection**. Membership in collections enables us to describe a taxonomy, one of the simplest forms of ontology. Taxonomy is also one of the most highly employed forms of ontology; management of file system directory trees and Web page bookmarks are all examples. Finally, in addition to helping people organize files, classi-

fication into a taxonomy has the end goal of helping users find their information later [18].

Several views exist in Haystack for managing collections of resources. Fundamentally, each serves to embed views of a certain kind to display the resources that are members of the collection. Many different presentation styles result from this simple principle. For example, a collection view that embeds thumbnail-sized views of its members can be used to display photo galleries (see Figure 23), while a view that embeds applet-sized views can be used to display address books. Adding items to a collection can be performed by simply dragging and dropping items into the collection's view.

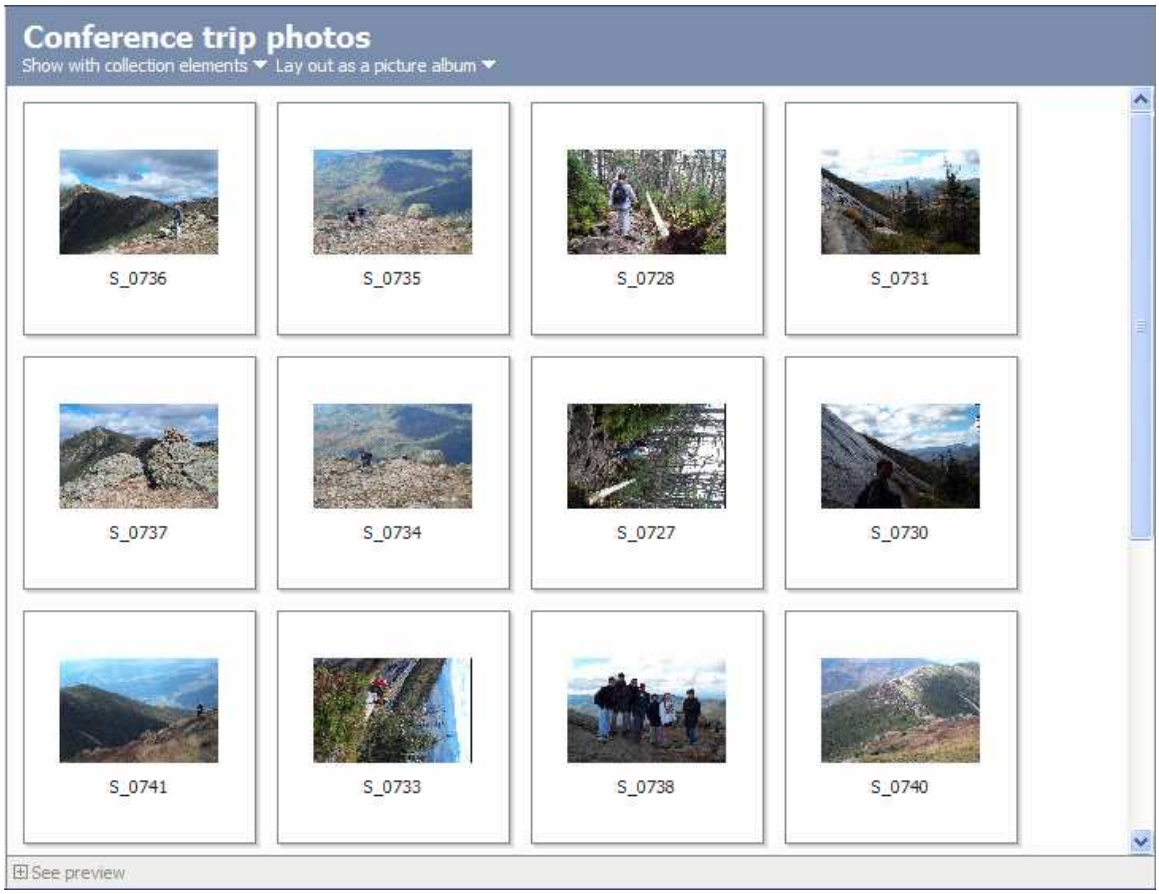

Figure 23: Collection view for photo gallery

Indeed, offering collections as a first class, primitive concept within Haystack enables the user to express a wide range of different types of information. In particular, three distinct purposes for this generic concept arise, independent of the presentation style

used to display a collection. Note that the purposes we enumerate below are inherently user-centric.

8.2.1 Collections as *ad hoc* resources

The collection class has one of the simplest schemas of any concept in Haystack. Only one property is defined, namely membership, which is specified with the `hs:member` predicate. Collections essentially have no semantics beyond being a set of resources. In that sense, membership acts as a sort of super-property of all other properties. In other words, membership is one way to indicate that a relationship exists between several resources (called the "collection") and another (called the "member") without being specific. Similarly, a user may group a set of resources into a collection to indicate that unspecified relationships exist between these resources.

In the physical world this phenomenon occurs often. For example, when applying for college, there are a number of different forms that are required, ranging from essays and recommendations to transcripts and financial statements for applying for aid. There is some amount of structure in such an application: financial statements belong together with the financial aid form, and essays should be attached to the form that asked for them. Nevertheless, it is common for these items to be stacked together and mailed in the same envelope. This "flattening" process is not a result of a desire to be sloppy or of a lack of structure; rather, it is done because the items involved are few and the relationships between the different items are evident upon inspection (i.e., during the scanning phase of retrieval). In Haystack, a user could model a college application as a collection, even though from an ontological standpoint there are "purer" ways to model a college application.

In contrast, many programs with fixed schemas force users into an "all-or-nothing" approach in terms of levels of structure: either provide the information according to the level of structure dictated by the schema or use a text editor, which normally embodies no level of structure, to manage it yourself. We believe a better approach is to allow the user to supply just the amount of structural information that is required. As was discussed in Section 7.5.1, our paradigm allows the developer to create finer-grained abstractions that gives user the flexibility to upgrade the level of structure he or she is comfortable with at any time. In other words, users can start with collections to group re-

lated things together and then call constructors to refine the level of structure at some later point.

8.2.2 Navigational tools expressed as collections

The "menu" style of interface design has existed in computer systems for decades. From the earliest mainframe programs to current graphical user interface-based applications, the menu paradigm has provided users with series of cascading selections from which to choose. Even many touch tone telephone systems have adopted this approach. The Web has generalized this paradigm slightly. Instead of menus having a fixed, rigid-looking structure, people browsing the Web can click through hyperlinks that may be placed anywhere on a page. The basic principle is still the same: following a sequence of steps to arrive at some final destination.

In Haystack we make it possible for users to customize these sequences by constructing nested collections. Because collections describe membership and not containment, items can exist in more than one collection, and hence multiple paths to the same resource are possible, as is on the Web.

Another way to think about this support is to think of menus as "lily pads" that allow the user to jump from a source resource to some destination resource. However, "lily pad" is less of a class of resources than it is a way of using relationships in the system to navigate. However, if a user wishes to introduce a resource into the system simply for the purpose of it acting as a lily pad, using a collection is a very reasonable and generic way of doing so. More discussion on this topic can be found in Section 10.1.

8.2.3 Categories

Categorization is perhaps the most obvious purpose for collections. Users may use collections as categories by placing resources that share some common characteristic into the same collection.

8.3 Creating and managing collections

The Haystack user interface provides several mechanisms for creating collections that draw upon the paradigms that have been discussed in the last few chapters. Collection creation and management is exposed to the user in accordance with the themes presented in the previous section.

8.3.1 Collection views

The system supports a host of different kinds of collection views for use in managing collections. These views can appear full screen, as applets on the start pane, etc. Despite these differences in size, all collection views support drag and drop and context menus for allowing users to add and remove items from a collection. Additionally, the full-screen collection views sport a preview pane, allowing the user to inspect individual members in their full-screen views.

The large diversity of layout managers, which are used by most collection views for displaying their members, permits the user to visually manage collections in a number of different ways. The collection views' direct manipulation features are actually inherited from layout managers. Furthermore, the user can switch layout managers at any time. Because of the generic definition of collection employed by Haystack, resources that may have completely disparate definitions in other systems, such as play lists, bookmark folders, address books, and photo albums, can take advantage of each others' views in Haystack. Address books can be viewed as a series of thumbnails; bookmark folders can be sorted by artist, mood, or genre; photographs can be listed in a tabular fashion. As is the case in Spotfire, it is not necessary for the Haystack user to coerce his or her data into a foreign data type in order to take advantage of a particular presentation style.

The list view is the default and the most frequently used layout manager. In the list view, members are displayed as a grid of aspects, as depicted in Figure 24. In addition, header controls allow the user to control which aspects are shown and to sort the collection according to specific aspects. List views are useful for browsing a set of items according to some characteristic, such as title or date.

Figure 24: List view (Icon, From, Title, and Date are aspects; the values of the aspects are shown in the rows below the header)

The bulletin board view works like a desktop: different items, displayed in applet-sized views, can be arranged in resizable, potentially overlapping rectangular containers. The spatially persistent property of bulletin board views make them ideal for use as *ad hoc* organizational units or previews of sub-collections into which one can navigate. Spatial information is considered independent of the underlying collection and is stored in the view resource. A fresh bulletin board (i.e., a collection in the bulletin board view) can be easily created by means of the bulletin board constructor, another elementary construction pattern.

Icon and thumbnail views provide ways for users to visually scan through a collection of resources using pictorial representations. Like the bulletin board view, these views also retain spatial positioning information between uses. The thumbnail view is especially

useful for photographs, as is shown in Figure 23. The diagram view, as introduced in Section 9.3, is actually a collection view that is similar to the icon and thumbnail views, in which selected predicate arcs can be rendered. Actually, the same implementation is used for the bulletin board view, the icon and thumbnail views, and the diagram view—the only difference is the name given to the view so that the user can jump to well-known configurations quickly.

The portal view allows the user to see applet-sized views of all of the members of a collection while allowing the system to be responsible for laying them out. The portal view flows items from one column to another; the number of columns displayed depends on the width of the view. Members can also be set to display across all columns. The portal view is depicted in Figure 25.

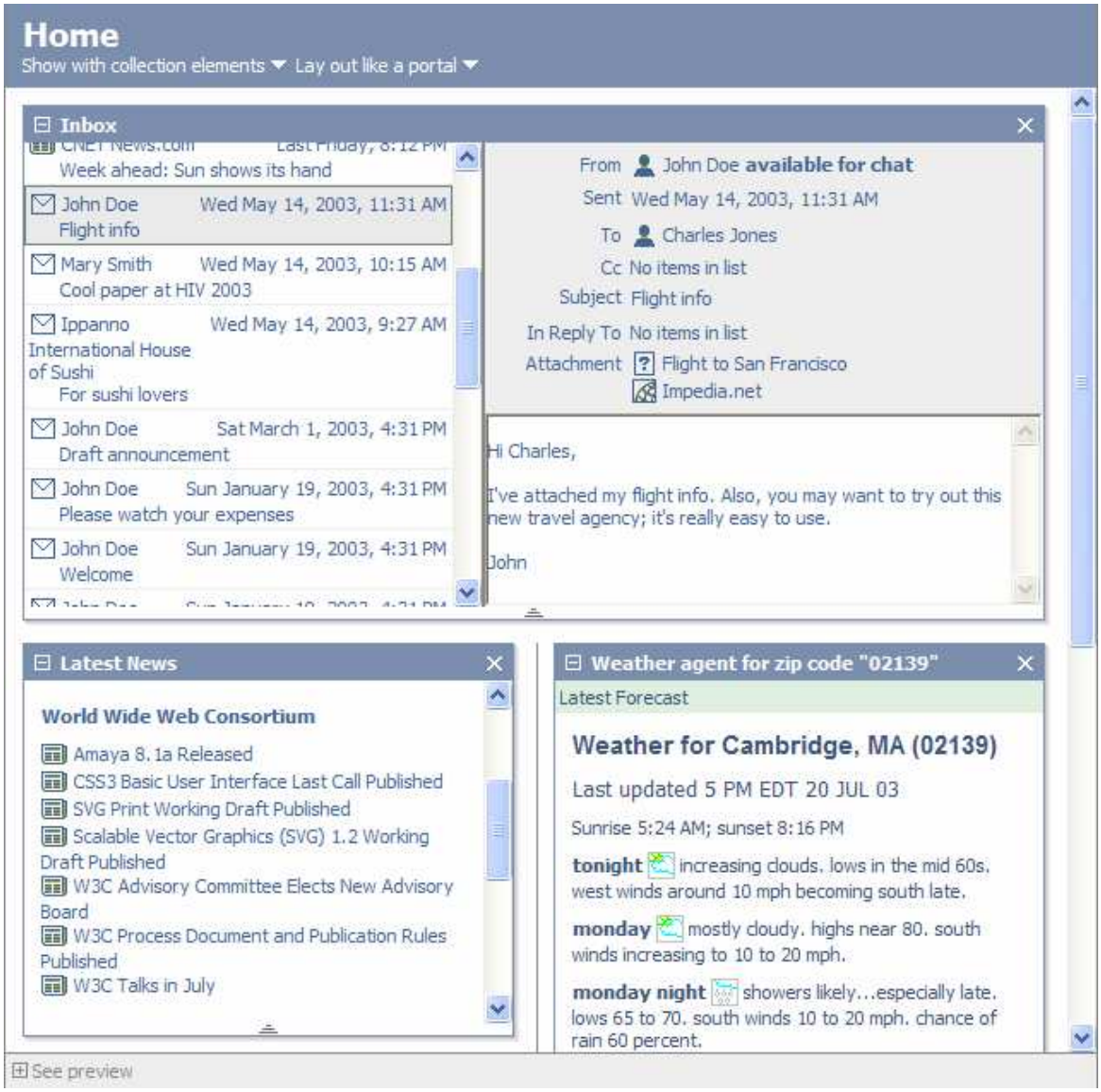

Figure 25: Portal layout

In addition to drag and drop and the context menu, new members can be introduced into a collection with the Add to Collection aspect. This aspect is normally docked to the tool pane and enables the user to conveniently add new resources to the collection being viewed in the content pane using any constructor in the system.

Finally, it is worth noting that the flexibility to change views benefits the user both at collection construction time as well as at retrieval time. Users can switch between the

most useful views for aggregating items into a collection and similarly choose the most appropriate view for finding or browsing items in a collection (see Figure 12 on page 28).

8.3.2 Organize aspect

The Organize aspect provides a convenient means for users to place resources into multiple collections. Abstractly, the Organize aspect displays the presence of a resource in a set of collections. Figure 26 shows the Organize aspect in use. This aspect is also usually docked to the tool pane and can be kept open while documents, Web pages, e-mails, or other objects are being viewed or edited. Placing an object into one or more collections becomes a simple matter of checking the boxes corresponding to the collections desired. The implication of having multiple checkboxes present is that the user is free to place the resource in the content pane into any or all of the categories. Sets of categories to be displayed in the Organize aspect are stored as **categorization schemes**, which are simply collections of collections.

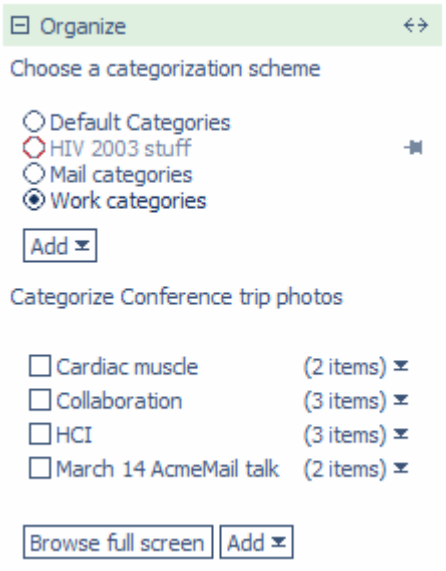

Figure 26: Organize aspect

Making it convenient to file items away and create collections is essential to encouraging users to organize their information properly. Whittaker's study revealed that users did not regard creating categories in current mail clients (most of which are geared towards folder-based organization) as a "lightweight activity" and were hence discouraged from creating them [32]. In a related domain, Abrams et al. found similar problems with the

usability of a folder-based system for organizing bookmarks [1]. They found that users need more scalable tools because even though their bookmark collections grow quickly, the effort required to create and maintain bookmarks remains a constant hindrance. One finding was that a user "cannot place it [a bookmark] in the prescribed folder easily at the mouse click." These problems were important motivations in our attempt to prevalently expose convenient means for creating categories and organizing documents into categories.

In addition to facilitating categorization, the organize aspect can also be used to aid navigation. The collapse/expand buttons beside each category checkbox can be used to expose the members of any collection in a categorization scheme; clicking upon one of these resources browses the content pane to it.

8.3.3 Automatic categorization

The placement of resources into some categories in a categorization scheme and not others helps to illustrate the nature of the condition of membership for such categories to the system. Such information can serve as the basis for automatic classification algorithms. Automatic categorization can help to alleviate the burden of classifying large numbers of documents by hand when it is accurate. However, because not even humans can categorize documents according to one person's tastes precisely, we have taken an approach that allows the user to take advantage of suggestions without forcing these suggestions on the user. We have constructed an agent that monitors membership in specific categories and makes suggestions of other resources that may belong in those categories based on similar characteristics [60]. The agent places a light bulb icon next to the categories in the organize aspect that it believes the currently viewed resource corresponds to. When the light bulb is clicked upon, the user is asserting membership in that category.

Learning algorithms *per se* are beyond the scope of this thesis; for more information on the Haystack learning framework, please refer to related work [60] and further discussion in Section 16.4.

8.4 Retrieving from categorization schemes

We have studied one specific kind of collection browsing in detail—categorization scheme browsing—in particular because categorization schemes can be used to alleviate

some of the fundamental problems associated with items being in one collection at once, as posed in Section 8.1. The problem of facilitating general browsing is often addressed by displaying a "table of contents" that characterizes the corpus. Successive navigation through entries in this table of contents allows the user to select characteristics distinguishing the remaining items of which the user may not have been aware.

However, if the table of contents is not structured properly, the user may have trouble locating the items of interest. For example, tables of contents are usually hand-coded taxonomies, as is the case with popular directories such as Yahoo! and the Open Directory or users' file systems. In such systems the hierarchy presented by the browsing interface corresponds directly to that of the filing interface, presenting a number of problems. The most immediate problem is that the user is not given the flexibility to retrieve information according to some alternate organization scheme.

A more specific problem is that the user must deal with the order in which categories are nested. Refer to Figure 27. If the user is looking for a recipe that involves meat, he or she will have to search all three categories hanging off the root category. The decisions that made sense during the creation of the hierarchy do not always apply during retrieval.

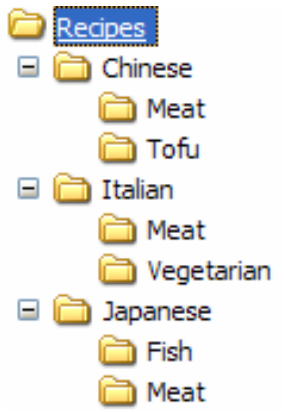

Figure 27: Example folder hierarchy

To address these problems, tables of contents should be presented in a way that facilitates information retrieval. The technique we describe here is an example of metadata-based retrieval [13], which has been explored in the past and used in web sites such as Epicurious.com. We have instantiated this idea of metadata-based retrieval for the case of items classified according to a categorization scheme. This is particularly useful be-

cause collections (and hence categorization schemes) are readily created and utilized by users of Haystack.

Refer to Figure 28 for a screenshot of our categorization scheme view. The categories navigation pane on the left consists of a tree, whose top level contained a list of the categories created by the user and a list of all articles in the corpus. When a category tree node is expanded, the pane displays a list of the articles in that category under that tree node. In addition, a list of categories to which those articles had been assigned is also shown under that node. This list excludes the categories whose tree nodes are ancestors of the articles. When a category tree node is expanded, users can either scan the list of articles or continue to expand child category nodes to refine that list of articles. In other words, the articles under a given category node correspond to the intersection of the categories associated with the given node and its ancestors.

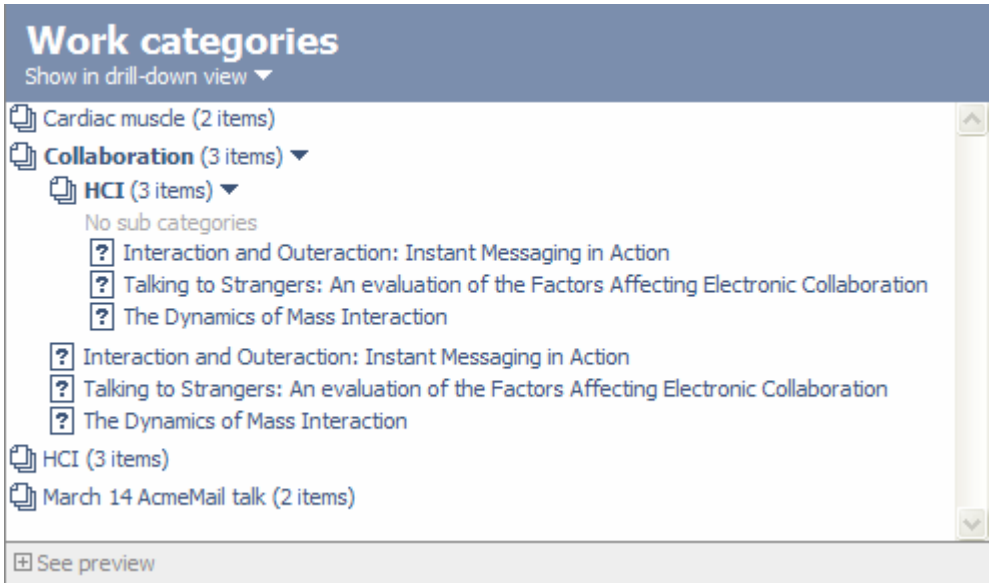

Figure 28: Categorization scheme drill-down view

We conducted a user study to compare users' preferences and performance between multiple categorization and the Microsoft Internet Explorer Favorites manager (representative of the hierarchical folder approach) (complete details of this study can be found in a paper by Quan et al. [39]). Twenty-one MIT computer science students (15 male, 6 female) participated. In the first session of the study, users organized two separate corpora of 60 ZDNet.com news articles in two phases, each phase using a different

approach. We chose this number of articles both to motivate users to organize the articles (smaller corpora may have been manageable with flat lists) and to prevent users from becoming overly bored or frustrated with a larger number of articles. Users then navigated those two organizational schemes in the second session of the study, again in two phases, one phase per approach, and after a one week time lag, in order to answer questions about several topics brought forth in the corpus.

On a one-tailed t-test basis, $t(19) = 1.1$, $p = 0.14$, users took considerably less time (19% reduction) organizing their corpus using multiple categorization (mean 2778.2, σ 833.7 seconds) compared to folders (mean 3441.2, σ 1693.9 seconds). (We considered the first phase only in these results because we observed that users took 30% less time in the second phase overall, evidence of a learning effect; when considering both phases, users took 2754, σ 1434 seconds for multiple categorization versus 2586, σ 1083 seconds for folders, a less statistically significant result.) Furthermore, our study shows that users in general put more organizational information into the system using multiple categorization than folders. On average across both corpora, we found during our study that users created 22 folders and about twice as many categories (45).

Subjective measures of user preference for multiple categorization over folders indicated a similar approval or preference. Of the 21 users who participated in the study, 10 users felt that conceiving and maintaining a folder hierarchy required a greater amount of cognitive effort as opposed to 6 users who felt that multiple categorization required more effort. The remaining 5 users felt that both approaches required an equal amount of effort. This result is encouraging when considering that users have been "practicing" folder-based organization for years.

In keeping with our reasoning from the organization session, we elected to use only data from the first phase of the navigation session. Also, because users were allowed to "give up" on questions they were not able to answer in a reasonable amount of time, we excluded these questions from our averages; we feel this is justified because users had a similar amount of success in answering questions using both techniques (12.4% of questions given up for multiple categorization versus 14% for folders). During this session, users took on average 36.9, σ 8.6 seconds using categories compared to 44.7, σ 12.3 seconds with folders, a 17% improvement, $t(19) = 1.66$, $p = 0.056$, one-tailed test. (For both phases, the results are 38.1, σ 11.0 seconds for categories versus 44.3, σ 13.9 seconds for folders.)

Furthermore, we found that users made selective use of their classifications to their benefit. We found that on average, when looking for a specific article, the number of categories users used to refine the list of articles was only about half of the total number of categories assigned to the desired article. In addition, we inserted 3 pairs of questions into each of our question sets, where both questions in each of the 3 pairs shared the same article as their answer. About 2 times out of 3, users ended up taking different paths to the same answer when navigating using multiple categorization.

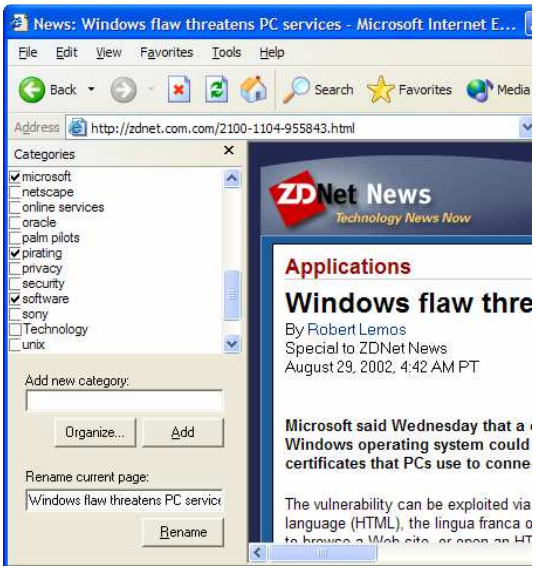

Figure 29: Classification UI

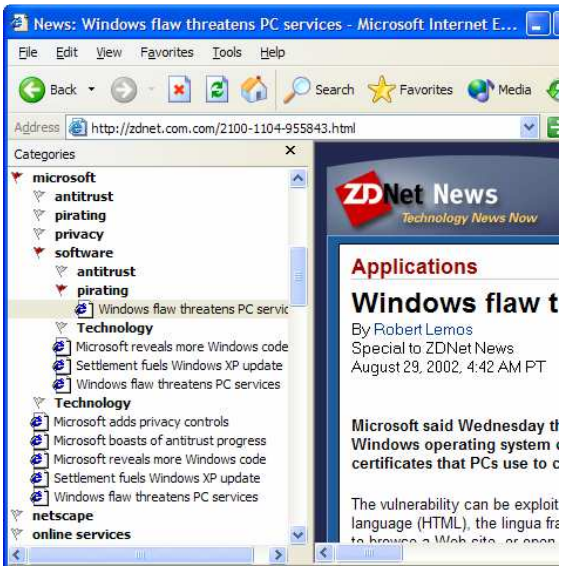

Figure 30: Retrieval UI

Chapter 9 Recording general information

We have proposed that a useful information system can be built upon a flexible data model, views for representing resources in that data model, and operations for mutating the data model. In the last chapter we talked about applying these basic concepts for the management of collections. In this chapter we similarly use these concepts to develop means for recording general information. This activity is inherently user-centric and can take on many forms, involving multiple uses of views, operations, aspects, and other basic user interface concepts. Below we discuss means of inputting information regarding existing resources, including forms, diagrams, and annotation. Despite the fact that these approaches are fairly conventional, fundamentally they are simply means for allowing users to manipulate metadata. In addition to supporting traditional forms of information input, these paradigms attempt to capture connective information that exists between resources and counter the common notion that extracting metadata from users involves typing in XML tags and is unlikely to occur.

Our approaches contrasts with current metadata authoring tools, which generally come in four flavors: (1) ontology/instance editors such as Protégé [10], OilEd [52] and Ont-o-mat [48]; (2) graph-based representation viewers such as IsaViz [49]; (3) schema-specific user interfaces (the type automatically generated by database applications such as Microsoft Access and FileMaker) including Reggie [53] and Ont-o-mat; and (4) taxonomy editors exposed by tools such as Protégé and used by services such as the Open Directory Project [54]. These existing tool implementations have for the most part focused on maintaining ontological constraints and not on addressing the HCI issues of information collection.

9.1 Form-based input

We now turn our attention to mechanisms for enabling users to specify information about resources that exist in their data models. The first paradigm we present is the property editor, which is in essence a form in which users can edit the property-value pairs associated with a resource. A property editor is actually composed of a sequence of views of a specific type that display aspects such as “title”, “creator”, etc. The list of aspects can be manually specified or derived from the RDF Schema definitions of the types of the edited resource. In the common case of the RDF property, in addition to display-

ing the name of the property, an RDF property view detects from context what resource is being edited and displays a list of the values under the property by embedding views for each value, as depicted in Figure 31. (Literal values have a special view associated with them that enable literals to be edited as text strings.)

Because property editors are simply lists of views of aspects, associating new properties with a resource can be accomplished by dragging and dropping properties into the blank area at the bottom of a property editor. Alternatively, by clicking on the Add button, the user can utilize a constructor to enter a new resource or literal value. Certain constructors have been designed especially for use within the property editor.

This use of view embedding is in contrast to what most ontology editing environments available today provide, in which users must work with plaintext representations of properties and their respective values. URIs are meant to be computer-usable names for maintaining the identities of distinct resources in an RDF representation. While some URLs are easy to remember because of marketing (e.g., "http://www.priceline.com/"), the majority are not (in particular randomly generated URNs), and users should not be required to remember them or enter them into forms. In fact, we argue that users should not even need to see them, because users derive no useful benefit by seeing them. By using views to represent resources in the property editor and elsewhere, the system allows users to deal with familiar representations of resources such as icons and human-readable names.

This last point can be further emphasized if one considers that some resources serve as anonymous nodes for gluing together multiple parameters in an n-ary relationship. Take the example of a contact editor that exposes both a home phone number and a work phone number property. Suppose further that the range of these phone number properties is a phone number resource, encapsulating the country code, city code, area code, and local exchange number as properties. A user is not likely to consider a phone number a separate entity, one in which the separate properties of the phone number must be individually entered and displayed every time a contact is shown. In our paradigm, a specialized view can be provided to package together the various properties of a phone number resource and present them in a unified form. In other words, views can be em-

ployed to expose only the RDF property relationships important to the user and not the ones that are present in the data representation for structural and ontological reasons.

Despite our advocacy for a view-based approach, it is worth pointing out that there are times in which a textual specification is the most natural means of input, such as when a portion of the human name of the resource in question can be conveniently recalled. We advocate the use of type-ahead support in these cases, such as that found in most modern integrated development environments, whereby the user would only need to type in as much of the name or description of the resource being described to uniquely match the text input with an existing resource in the system. Type-ahead support is built into a special resource constructor that recognizes the range type of the property being edited and allows the user to match against resources of that type; if the user specifies a title that is not recognized, a new resource is created with that title and type.

9.1.1 Group aspects

The property editor paradigm improves the facility with which a user can input information into a form by taking advantage of our view architecture. In addition, the sets of aspects that constitute forms are also of interest. Aspects can be aggregated into **group aspects** (see Figure 31) for the purposes of abstraction. Like other aspects, group aspects serve as ontological proxies in that they allow the user to describe some aspect of the structure of resources to the system.

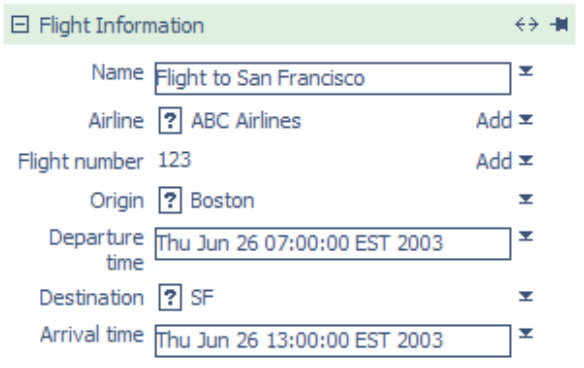

Figure 31: Group aspect

Some of the most basic aspects employed by the system are group aspects. The Standard Properties aspect lists the Dublin Core properties and `rdf:type` for convenience. Fur-

thermore, most types have a summary aspect associated with them that hosts the most basic properties of that type, such as title or author.

9.2 Annotation

Another important way users record information is by making notes to themselves in order to preserve ideas that arise during a variety of activities, e.g., reading documents or attending meetings. The purpose of these notes is often to summarize, criticize, or emphasize specific phrases or events, and notes can also serve as reminders that are used later to improve recollection. Passing annotated documents between colleagues is a highly effective way to exchange ideas and to engage in collaboration. However, as any student who has shared his or her notes with a classmate knows, sharing notations on paper requires considerable effort: either the document must be photocopied, or the author must give up the original for a period of time.

Naturally, keeping documents in electronic form alleviates many of these problems. In fact, most work to date on Internet annotation has concentrated on the types of document- or topic-specific applications mentioned above. For example, Microsoft Research has studied the use of Microsoft's Office 2000 product for posting documents to the Web and engaging in online discussions [98]. We believe that these kinds of software packages go a long way towards allowing users to manage annotations, but there are still many areas of functionality in need of further improvement.

One problem is that sometimes the items that need to be annotated are not nameable in the representation. The makeshift solution for this problem is to give a syntactic description of the position of the items to be annotated, but among other difficulties, such syntactic descriptions can fail to relocate an item if there is any repositioning of neighboring items in the document or space being annotated. Our data model was designed to eliminate this problem from the start by encouraging resources that need to be annotated to be exposed within the data model.

Another frequent issue is the requirement that annotations must be composed purely of text or fit into a limited set of predefined fields. Recent work has attempted to address this limitation. Like Haystack, Annotea uses RDF to describe annotations and allows annotations to be created that follow some RDF schema [99]. However, for reasons discussed earlier, we believe that creating a new resource and always presenting a key-

value pair form is insufficient. Instead, our annotation user interface hosts a constructor, allowing any resource that can be created with a constructor to be used as an annotation. Simple examples such as text strings are supported by means of the elementary constructors given in Section 7.5.1. More complex resources may be created using the full expressive power of our constructor paradigm.

Finally, few annotation environments allow the user to control the kind of connection that he or she is asserting between the annotation and the annotated item. Like the Web, they assume that the connections between resources are unlabeled. Because connections in RDF are labeled, it is not difficult for us to expose this element of expressiveness to the user. Possible relationships that are supported include "is described by", "has comment", and "has reply". Other relationships can be created by the user.

Beyond acting as containers for auxiliary information, annotations serve several other purposes, such as a form of communication. In addition to attaching annotations to interesting resources, sometimes annotations also need to be forwarded to interested parties or CCed to people affected. For this purpose we treat annotations as a special kind of message; this is described in Chapter 11. In addition to messaging, another important use of annotations is to serve as a reminder to aid in later retrieval. This use of annotation is further explored in Section 10.3.2.

Support for annotation is exposed in Haystack as an aspect, which is usually docked to the tool pane for convenient access (see Figure 32). The semantics of the Annotation aspect is to expose a list of annotations attached to the argument through certain selected annotation predicates. In addition to exposing a constructor interface for creating new annotations, this aspect displays annotations that are associated with the resource being viewed in the content pane in a threaded form; because annotations are messages in our model, replies can exist between annotations.

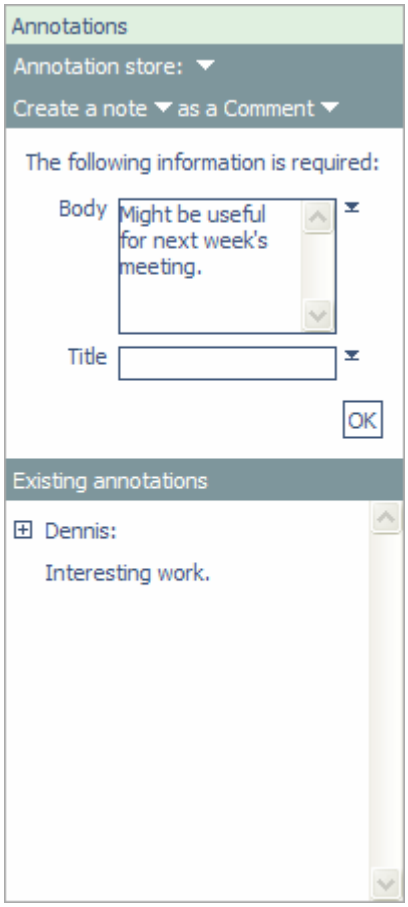

Figure 32: Annotation aspect

9.3 Editing relationships

Finally, we examine the problem of inputting metadata whose primary purpose is to express relationships. We postulate that manipulating directed graphs can be a natural paradigm for dealing with connections between resources because of its similarity to the means employed on paper for displaying connections between concepts—diagrams and charts—which can be especially effective means for recording ideas. Venn diagrams, UML diagrams, family trees, organizational charts, and process, causality or flow charts are used by people in a variety of fields to legibly express relationships between a set of entities or activities. Even looking beyond paper, we note that designers in a number of domains often use sticky notes and a whiteboard to conduct brainstorming sessions. Ideas are first recorded on the square sheets, attached to the drawing surface, and annotated with arrows or other connectors. In a similar fashion, grammar school students

working on school projects often use note cards to capture ideas from various sources, and then lay out the cards in order to solidify the organization of their final reports.

Because diagrams are such a powerful means for displaying and capturing information, we consider graph editing to be a key paradigm for interacting with RDF data when the user wishes to focus on the relationships between resources. A graph editor can present a collection of RDF statements in the obvious manner by representing resources as nodes and properties (or more generally, aspects that produce resources) as arcs connecting nodes. However, taken to its extreme, graphical representations can be extremely misleading to users. Although from an ontological perspective, we can model the relationship expressed in the sentence "Bob is 25 years old" by two nodes named "Bob" and "25" connected by an arrow labeled "age", this notion may not be intuitive to users, who are used to age being a property of people that is entered in a form-like fashion. Even for data that users usually regard as relationship-oriented, one must be careful to only display the relationships and nodes that are important at any given time.

To address these issues, graph editors can embed different types of views to display the resources being manipulated in order to control the level of detail being presented. Figure 33 shows an example of Haystack displaying a conversation as a graph of messages. Although the ontology for messages includes properties such as "From", "To", and "Body", these fields are not visualized as arcs. Instead, the type of view that is specified by the conversation's view for embedding displays a snippet of the message's body as well as an indication of who sent the message. The focus is placed on the "in reply to" connections that exist between messages. To a user looking to gain an idea of the "big picture" of the flow of the conversation, the approach adopted here is arguably more useful than one in which all RDF resources present in the data representation of this conversation, such as the originating and destination e-mail addresses in the "To", "CC", and "BCC" fields, the message bodies, and the attachments (as well as the predicate links connecting them) are visualized. And, as before, using views to render resources to the screen allows the system to display arbitrary collections of resources without requiring graph editors to have any hard-coded notions of how to display graph nodes.

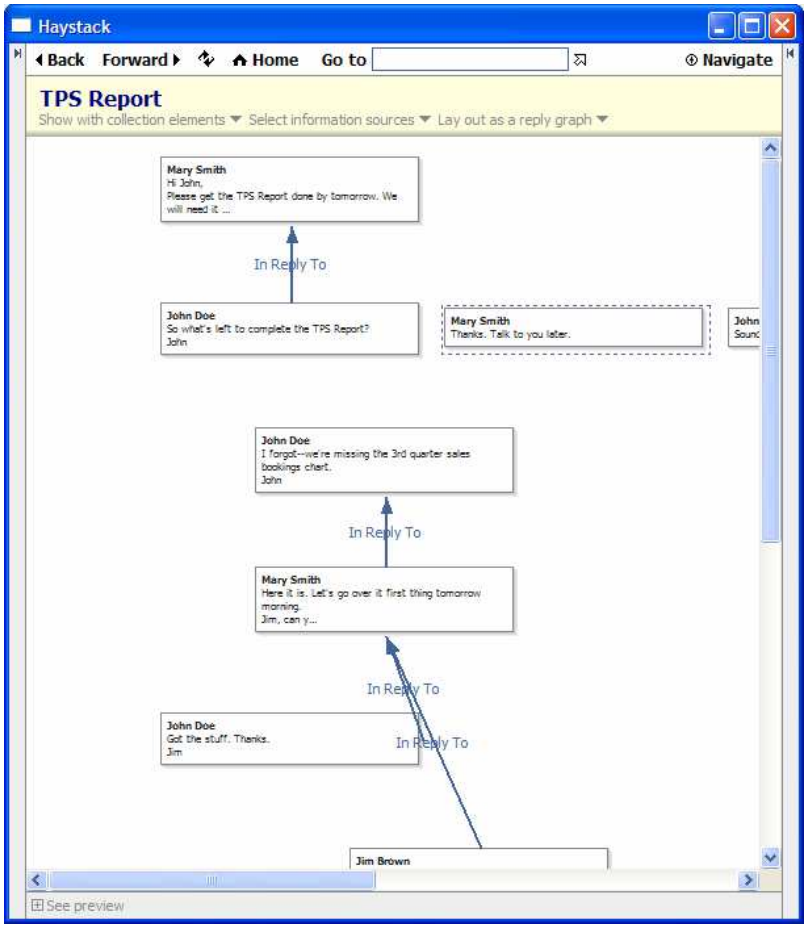

Figure 33: Reply graph

In addition to enabling users to visualize relationships, we allow users to select an aspect from a list and to drag arcs from one node to another. Lists of possible aspects can be derived in many ways; one way is from an ontology by selecting all properties whose domain and range types are those possessed by resources in the graph. Custom lists of aspects are also supported. The palette of possible arcs will be displayed as other collections are displayed—as a sequence of views—meaning that other predicates can be added to the palette by means of drag and drop. Similarly, resources (nodes) can be added to the graph by dragging views of those resources into the graph.

We feel it is important to emphasize that the various paradigms presented here are complementary and can be used together to construct sophisticated user interfaces for working with various kinds of metadata. Figure 34 illustrates a screenshot from Haystack displaying an organization chart in the graph editor that allows a user to work

with the relationships between various people. In addition, the preview pane located on the bottom portion of the figure depicts examples of the property editor, whereby arbitrary properties (as specified by an ontology) can be entered, such as dietary requirements or names. Finally, the pane on the right side of the screen shows the categorization pane and an embedded collection view showing who else is indicated to be a "Senior Vice President."

In theory it would be possible to present all of the displayed information in either the property editor or the graph editor, because both editors are general enough to support the entire RDF data model. However, the figure illustrates that despite the fact that "enjoys cuisines" and "manages" are both RDF properties in the modeling sense, for the purposes of this interface the choice of which paradigm to use to display these properties is key to providing an intuitive user experience.

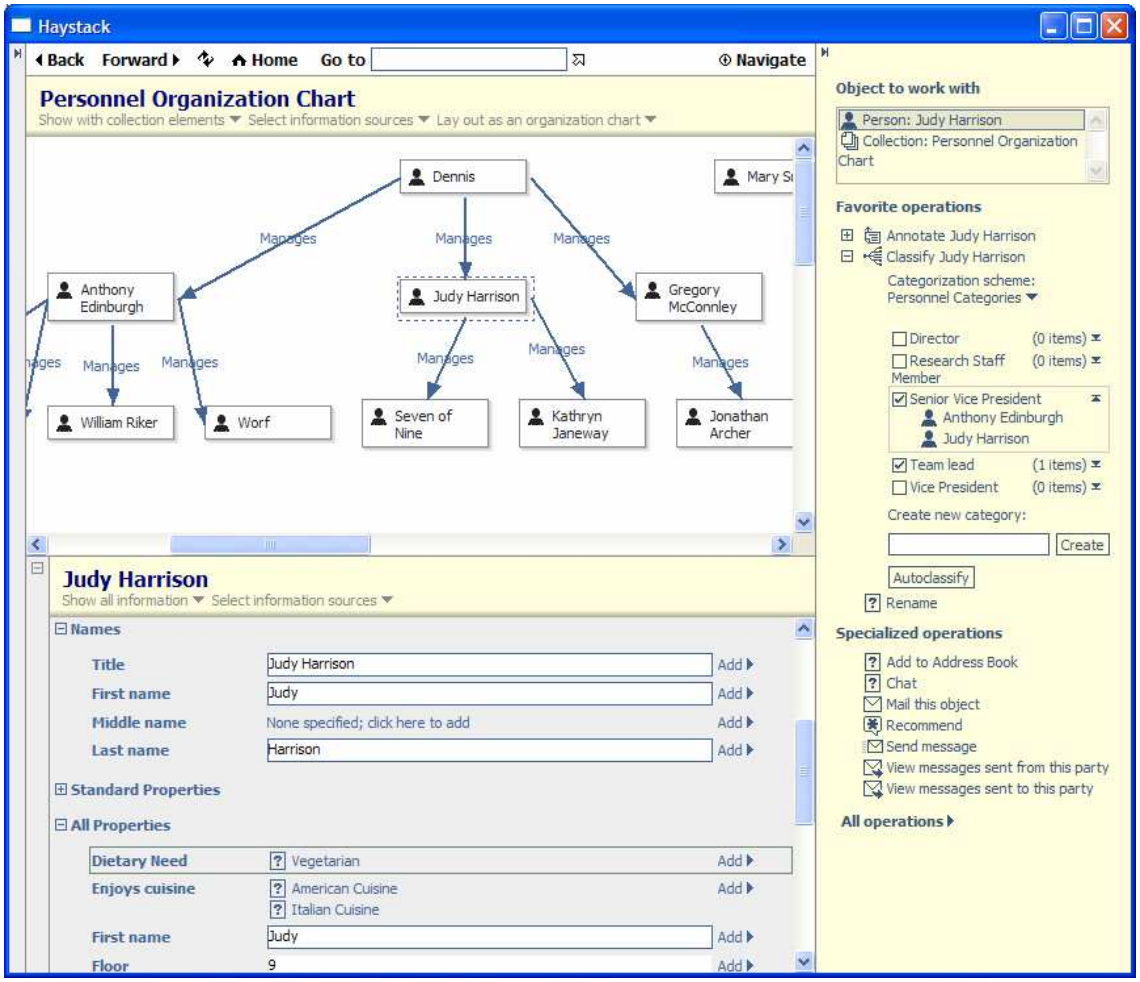

Figure 34: Paradigms working together

Chapter 10 Retrieving general information

In this chapter we discuss generalized mechanisms for retrieval, i.e., locating relevant information satisfying criteria recalled by the user. Traditional information retrieval has focused on assisting users in locating text documents, which has made keyword search perhaps the most frequently-employed means for retrieving documents [80]. However, keyword search is far from perfect; issues such as synonymy, contextualization, and lack of expressiveness rank among the complaints, although much work has been performed in an effort to alleviate these problems. Nevertheless, one of the points of our data model is to support a wide range of different kinds of information, many of which are not primarily composed of text. Our investigation into retrieval will thus focus on how we can leverage the metadata surrounding what is sought after to retrieve information.

We begin by discussing associative recall—the form of retrieval best supported by our semistructured data model. We move on to more traditional approaches to retrieving information, namely various ways for allowing users to issue precise queries to the system. Afterwards, we turn our attention to newer approaches to retrieval, such as natural language question answering.

10.1 Associative recall

When a user is attempting to retrieve some resource, often the characteristics that are recalled are not attributes of the resource itself but more of the path taken to find the resource. This form of recollection, which we will refer to as *associative recall*, relies on the system providing a suitable medium in which users can demarcate and retrace paths. One of the most established applications that takes advantage of associative recall is the World Wide Web. Users have found the hyperlinking model exposed by the Web to be an intuitive and familiar one [106]. Similarly, Haystack enables many of the same forms of information retrieval as the Web.

One reason underlying the success of webpage-based hyperlinking as a form of information access and recall is the large number of visual *features* present on a page that enable a user to determine the next link to pursue. In contrast, most people have probably encountered buildings, such as hotels or office buildings, in which every floor of the build-

ing looked almost identical or that were highly symmetric in some fashion. In these cases, there are very few visual features to associate a particular location with and hence reaching one's final destination is a lot more challenging.

Unlike buildings, personal information spaces are virtual and can be structured and customized to the extent that the organization model of the system supports such customization. In Haystack, we provide collections, discussed in Chapter 8, which can be created, manipulated, and visualized in a variety of different fashions. In particular, collections can act as customizable "breadcrumbs" that permit easy retracement, as discussed in Section 8.2.2. Progressing through a sequence of breadcrumbs is analogous to navigating into a hierarchy. In fact, the Starting Points collection on the start pane provides a means for grounding associative recall trails in a convenient location.

However, sometimes retracement is needed even when breadcrumbs were never explicitly laid out to begin with. The Web browser provides mechanisms such as Back and Forward buttons and history views for this purpose. Haystack also provides Back and Forward buttons in addition to a history mechanism that recalls what resources were visited in the process of doing some task.

10.2 Applying traditional query technology to Haystack

Metadata supplied by the user (e.g., authorship information, titles, importance, categories, etc.) is easily stored in our data model and can be taken advantage of in searches. Additionally, the system also generates a fair amount of metadata passively, such as last visit time, which may also prove helpful. Querying for information given these kinds of attributes is currently achievable with available database technology, as many searches can be expressed in SQL. One major hindrance to the adoption of this approach has been the fact that the use of databases had been reserved for data management "professionals" given the amount of expertise needed to set up such a system. Our data model and the user interface supporting it make possible the use of SQL technology for issuing powerful, expressive searches to the system.

However, another major hindrance has been the difficulty associated with composing SQL queries themselves. Several approaches have been attempted to give users the power of SQL queries without the prerequisite of mastering SQL syntax. Graphical approaches to query building have been explored in popular desktop database packages

such as Microsoft Access and Filemaker Pro as well as in research systems. A related approach is query-by-example, whereby users interact in dialog-like fashion with the system to locate relevant data.

We have designed a generalized query language that is built into Adenine for supporting SQL-like queries, which can be found in the Adenine reference manual [96]. However, this thesis does not investigate the problem of exposing the power of a generalized query language to the user via a graphical interface; instead, we focus more on less traditional approaches, such as natural language and browsing, which despite being key modalities for retrieval have not been highly explored in the past because of lack of support in the data model.

However, a basic requirement of a system such as Haystack is to support search, and as a result we have constructed a highly rudimentary search interface to the system. Refer to Figure 35.

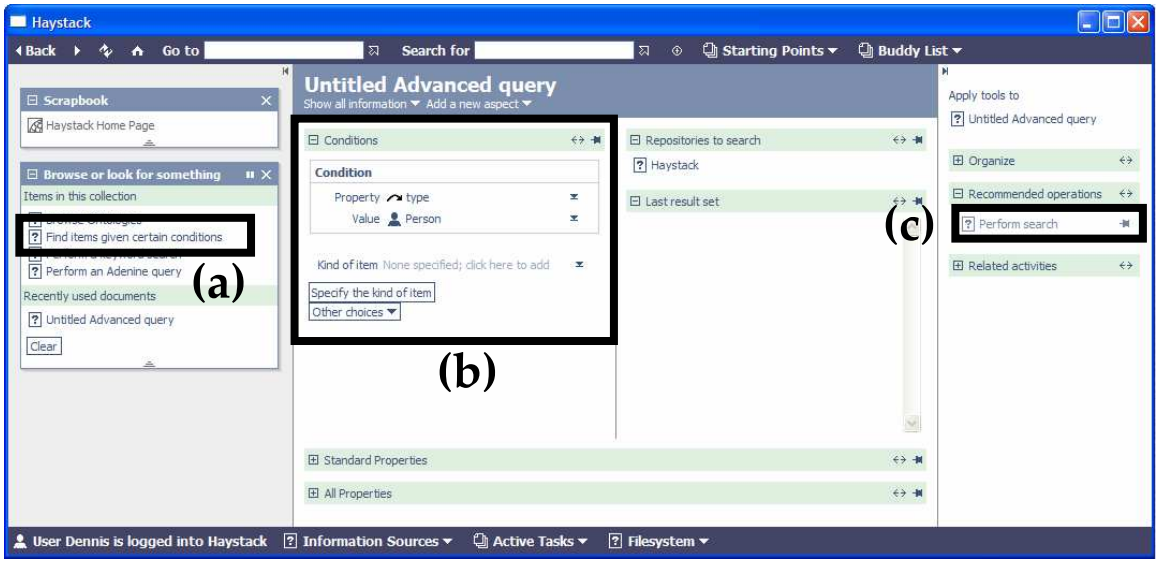

Figure 35: Search interface composed of (a) a constructor, (b) group aspects, and (c) an operation

While not exposing the full generality of our querying capability, the search functionality as implemented currently demonstrates the ease with which one can construct interfaces using constructors (Section 7.5), operations (Section 7.2) and group aspects (Section 9.1.1). Users create queries by clicking on the "Find items given certain conditions" constructor on the left. The query's aspect view contains several aspects, but the most im-

portant is the Conditions aspect. The Conditions aspect is an RDF property that connects the query to condition resources—in essence, reified statement patterns such as `?x rdf:type hs:Person`. The constructor shown at the bottom of the Conditions aspect permits creation of these statement patterns. Query evaluation is implemented by an operation that collects the conditions together, performs the query, places the results into a collection, and displays the collection to the user.

10.3 Natural language retrieval

Perhaps the most intriguing approach to retrieval we present in this thesis, natural language query technology is being developed to enable a user to pose questions to the system using a technique he or she is already skilled at: his or her native language [16]. To enable natural language retrieval, a mapping between elements in our data model and corresponding natural language phrases is necessary. Alternatively, natural language can be stored directly in the form of pieces of text, as discussed in Section 7.5.1.

We have explored the connections between natural language retrieval and our data representation using the START system, a natural language question answering system [16]. START stores information about resources as fragments of natural language text called natural language annotations (we will refer to START's natural language annotations as NLAs because they are distinct from our notion of annotation). NLAs can be used to describe not only other textual resources but also multimedia content, database queries, and even parameterized code fragments. Haystack itself has no natural language processing functionality; this functionality is completely contained within START. We feel this is a typical setup, and as a result the approaches we describe below maintain the separation between the natural language engine and the information repository (Haystack).

Question answering systems accept questions phrased in natural language for *specific* pieces of information such as "What is the capital of Massachusetts?" and return the answer. This is in contrast to approaches that force the user to wade through large numbers of documents and other items in search of the answer. The answer itself can be displayed in a number of different ways depending on the nature of the answer [56]. In the case of requests for information from Haystack's RDF store, result sets can be presented as collections of resources or as literal values.

10.3.1 Natural language schemas

The first approach we discuss for integrating natural language search focuses on means for mapping RDF-encoded data into NLAs for use by START's natural language question answering facility [17]. Despite the fact that START's annotation format and our data model are both based on a triples representation, the mechanism we describe below maintains the abstraction barrier between Haystack and START. We mentioned earlier that NLAs need not be text fragments but can also be parameterized. As a result, we can construct **natural language schemas** that map between RDF schemas and NLAs. For example, consider the following RDF schema for describing basic properties of a state of the United States, such as state bird:

```
add { :State
       rdf:type     rdfs:Class ;
       rdfs:label   "State"
}

add { :bird
       rdf:type     rdf:Property ;
       rdfs:label   "State bird" ;
       rdfs:domain  :State
}
# ... more property declarations

# The following is hard coded but could be entered by the user (cf. Chapter 8)
add { :alabama
       rdf:type     :State ;
       dc:title     "Alabama" ;
       :bird        "Yellowhammer" ;
       :flower      "Camellia" ;
       :population  "4447100"
       # ... more information about Alabama and other states
}
```

We can then map questions that may be asked by the user into equivalent RDF queries by means of the following natural language schema:

```
@prefix nl: <http://www.ai.mit.edu/projects/infolab/start#>

add { :stateAttribute
       rdf:type     nl:NaturalLanguageSchema ;

       # This annotation handles cases like "[state bird] of [Alabama]"
       # and "[population] of [Maine]".
       nl:annotation@( :attribute "of" :state ) ;

       # Code to run to resolve state attribute
       nl:code      :stateAttributeCode
}

add { :attribute
       rdf:type                  nl:Parameter ;
       nl:domain                 rdf:Property ;
       nl:descriptionProperty    rdfs:label
}

add { :state
       rdf:type                  nl:Parameter ;
       nl:domain                 :State ;
       nl:descriptionProperty    dc:title
}

# The identifier [state] will be bound to the value of the named
# parameter :state. The identifier [attribute] will be bound to the
# value of the named parameter :attribute.
method :stateAttributeCode :state = state :attribute = attribute
       # Ask the system what the [attribute] property of [state] is
       return (query { state attribute ?x })
```

The `:stateAttribute` definition is parameterized by `:state` and `:attribute`, which allows one definition to be used for an entire class of possible expressions, such as "state bird of

Alabama" or "population of Maine". The `nl:annotation` attribute of `:stateAttribute` gives the natural language annotation that can be used by START. The `:stateAttributeCode` method is called to evaluate the result of state attribute expressions.

The question answering retrieval process then occurs as follows:

1. The user poses a question such as "What is the state bird of California?" to Haystack.

2. Haystack passes the question to START.

3. START is capable of handling a variety of different natural language variations, such as passive constructions and different question forms. START determines that the question is of the form given by the `:stateAttribute` natural language schema.

4. START binds "state bird" and "California" to `:state` and `:attribute`, respectively.

5. START passes these bindings and the name of the relevant natural language schema (`:stateAttribute`) to Haystack.

6. Haystack executes the associated Adenine method with the given bindings as named parameters.

7. The resultant information is then presented to the user.

10.3.2 Annotation-based retrieval

In this section we explore one other use of natural language question answering: annotation-based retrieval [15]. This model relies on the premise that certain forms of information, such as subjective feedback and recommendations, are not easily cast into RDF without convoluted ontologies. Such information is most easily captured in natural language and as annotations, as discussed in Section 9.2.

Annotations are useful both in an individual setting and a collaborative environment for locating items that have been subjectively found (perhaps by others) to match certain criteria. For example, when buying a product on an e-commerce website, one can often find customer commendations or complaints on the webpage featuring the product.

What if the specific product being sought has not yet been identified? One major advantage of going to a bookstore over buying online is the ability to speak with a sales associate and ask in person, "Can you suggest a mystery novel for my long flight to Tokyo?" Most websites are unable to offer such advice, not because they do not expose customer feedback, but because their interfaces are geared towards allowing users to *browse* annotations for their own sake, rather than using them to *find* objects given a specification.

By supplying START with the text of natural language annotations, we can use START to retrieve items of interest based on these natural language descriptions. In fact, to START, the natural language bodies of our annotations are supported as a special case of START's NLAs.

Chapter 11 Contextualizing information

One of the focuses of Haystack is presenting relevant information and tools to the user. Relevance can be established not only by the nature of the resource being viewed (e.g., type of the resource), as has been discussed extensively earlier in this thesis, but also by the current task being performed by the user. We group ontological concepts such as operations and aspects that benefit some end purpose of the user into resources called contexts. Contexts manifest themselves in the tasks performed by users, and by giving users the ability to manage their tasks within Haystack, we are able to derive the current context to some extent.

11.1 Contexts

We define **context** to be a resource that acts as an embodiment of some state of the user. The context class in the Haystack ontology does not have many predefined properties; instead, a context resource is meant to be extended with custom properties, such as recently used operations or keywords in documents viewed by the user while investigating some topic. A large space of potential properties has been explored by the related work, which is discussed later. Our purpose is to provide a generalized framework in which other components of the system can benefit from the contextual knowledge gathered by specific tools.

Properties that are supported by the base system include most recently used lists, related operations, associated categorization schemes (see Section 8.3.2) and associated aspects. When a context is active (activating a context is explained below), the resources visited by the user are tracked and associated with the context. Also, developers can specify that certain aspects and operations should only appear when certain contexts are active. Aspect views respect these semantics in order to reduce the clutter of aspects that are not salient at the moment. Grouping operations by context can be used to reduce clutter the same way "applications" are used today for this purpose. For example, operations can be grouped according to contexts ranging from "writing a research paper" to "sending out wedding invitations to friends." There is no need for a word processor to expose a "generate bibliography" command when the user is writing invitations; similarly, "mail merge" can be hidden away when the user is writing a journal article.

11.2 Tasks

In order to take advantage of contexts, we must have a means for tracking the current contexts of the user. Haystack provides an ontology for describing **tasks** such as appointments, projects, activities, and to-do items. Tasks represent instantiations of context, and users can inform Haystack of when they are performing tasks as a means of specifying the current context. The `task:Task` class is the base class of all tasks and has only one property defined, `task:context`, which associates a task with a context. Below we outline several predefined subclasses of `task:Task`.

A **to-do item** is perhaps the simplest example, but it illustrates many of the common characteristics of tasks in general. To-do items are represented by the `task:Todo` class, which derives from `task:Task` and `note:Note`. A to-do item is pictured in Figure 36.

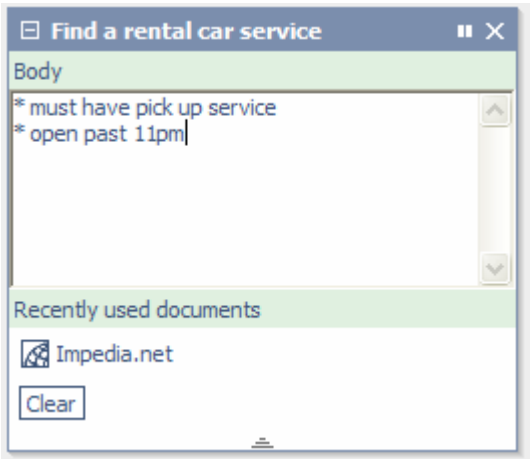

Figure 36: To-do item

As is the case in most personal information managers, to-do items have a space for jotting down notes. In addition, in Haystack to-do items, being tasks, can be started and stopped; Haystack then records the recently used documents browsed during the session. This record can be useful for retrieval later, especially when the only characteristic of the browsed resource remembered is that it was done in the context of some task.

Tasks are considered active when they are open on the Start Pane and inactive when they are removed. Tasks can also be marked as temporarily inactive by clicking on the pause button on their title bars.

A **project** (`task:Project`) is a task derived from `task:Task` and `hs:Collection`. Projects act like other collections except that they can have contextual features associated with them, such as project-specific categorization schemes and most recently used resource tracking.

UI continuations and operation closures are also examples of tasks. See Section 7.2.2 for more information.

Finally, an **activity** has type `activity:Activity` and is a resource that represents some long term process, such as reading e-mail or jotting down information. Most contexts have a single activity associated with them, making activities a generic way to indicate to the system that a context is active. Activities have a list of associated resources that act as starting points. Furthermore, as singleton instantiations of contexts, activities play a role similar to applications in today's environments. As an example, the "Jot down information" activity, which allows the user to take notes and make lists, is shown in Figure 37.

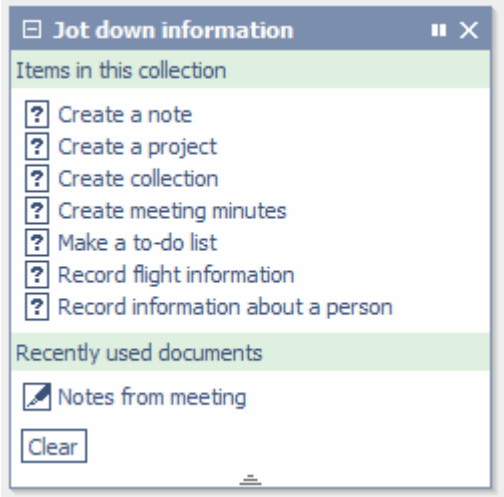

Figure 37: Jot down information activity

Activities can be associated with certain types of resources. For example, when a user is modifying a travel itinerary, the Related Activities aspect displays a list of activities that are commonly associated with travel itineraries, such as expense reporting or travel planning. The user can place a check in the box next to the listed contexts, and the user interface will update to display the then relevant aspects, operations, etc. In the future,

the system may endeavor to guess what the current context is by observing the user's behavior. See Figure 38.

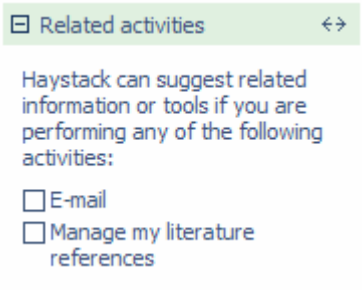

Figure 38: Related activities aspect

There are several predefined classes of tasks, but any resource can be dynamically turned into a task. When a user drags and drops a resource onto a to-do list (a collection in the to-do list view), Haystack asserts the resource to have type `task:Task` and assigns a new context to it. This reflects the reality that users associate the notion of a task with arbitrary resources, such as e-mail messages, people, or web pages. Also, users do not always realize up front when a resource, such as a collection or a note, needs to be considered a task (a project or a to-do item, respectively).

11.3 Previous contextual information systems

Previous work has investigated the encoding, determination, and utilization of contextual information. Many tools have been developed to learn about the context of the user and to recommend related information based on this context. SUITOR tracks the user's interactions with desktop software and, using a keyword-based approach, displays recommended actions [45]. Cheese tracks mouse movements in a web browser to guess the user's context [42]. Our work focuses less on what can be learned from observing the actual "physical" interaction between the user and the computer (e.g., mouse movements, etc.) and concentrates more on ways for allowing users to tell the system what they are doing (e.g., to-do list items, navigating to resources of certain types, etc.). This approach gives us a more concrete specification of users' notions of context, which is then suitable for distribution to others. It is also similar to the approach taken by the remembrance agent, which enhances both the desktop and wearable computer experiences using a combination of textual and physical features (e.g., who was physically

next to you when a piece of information was referenced) to locate relevant information [44].

Other systems have encouraged input from the user in order to build better models of their information interaction experience. Powerscout uses semantic profiles built from keywords appearing on pages browsed to or explicitly added by the user to "scout" web search engines in order to determine recommendations for related web pages [40]. Haystack's notion of context is similar to semantic profiles, but because Haystack supports a richer set of document classes (e.g., e-mails, travel itineraries, address book contacts, etc.), we are able to rely on more semantic attributes such as type rather than on textual attributes such as keywords in the characterization of contexts.

Finally, the approach taken by de Rosis et al. promotes a development model in which program functionality is placed in the context of some user task [41]. Their approach uses Petri Nets and other declarative programming techniques. While our approach may not be as expressive, we have focused on providing interfaces that capture contextual knowledge from users, not programmers, through the use of ontological proxies.

Chapter 12 Messaging

In this chapter we take a slight detour from our discussion of techniques for managing information to talk about communication, a necessary element of our system that enables the collaborative features described in the introduction. The manner in which we cast the communication problem is similar to how the other abstractions we have described so far have been built, such as operations and aspects. We argue for a communication abstraction that treats the notion of a message as a basic unit and describe the metadata framework required to support many of the notions of messaging superstructure in use today, such as dialog management and reply chains. Our messaging abstraction is also designed to encompass many of the existing messaging systems today, and we describe how e-mail, instant messaging, and even annotations are incorporated into our framework. As a result, communication can be initiated, browsed, and organized using the techniques illustrated in previous chapters. Furthermore, a semistructured collaboration environment should support the sending and receiving of "metadata" describing the information in the repository. Categorization taxonomies, ontologies, schemata, and customized ways of looking at information represent useful insight into how to organize information—information that is often present in an organization but "trapped" on personal computers because it is not highly transportable. By including support for other types of messages other than textual ones, we can provide users with a solid basis for collaboration.

12.1 Current messaging paradigms

One of the greatest accomplishments of the Internet has been enabling communication in various forms, and as a result many of our modern notions of how electronic communication occurs have been shaped and characterized by the Internet. Perhaps the most notable technology in this space is e-mail, but instant messaging (IM), Internet Relay Chat (IRC), and Usenet newsgroups have also found widespread use for related applications. The success of these communications protocols can be attributed not only to the degree to which they address shortcomings of their non-digital counterparts but also to their ubiquity and prominence as standards.

Communities have developed around each of these systems, and significant CSCW research has revealed the principles underlying their success. E-mail has been found to be

especially effective for long term, asynchronous communication and task management [32], whereas instant messaging is well suited for lightweight, short term coordination [58]. Newsgroups have been characterized as a public medium in which establishment of common ground is easily discernable from previously recorded conversations [59].

These systems have been developed somewhat independently over the past half century and continue to be extended with new functionality that addresses the broadening needs of their users. While the systems share some common notions, such as e-mail addresses and MIME headers, the amount of perhaps unnecessarily duplicated infrastructure is becoming increasingly evident. For example, all of these protocols have different authentication and identification systems. The most obvious indicator of this phenomenon is the overlap in functionality in the clients for these systems. E-mail clients, instant messengers, and IRC clients all have widgets for displaying lists of people and means for notifying senders of a recipient's absence. Newsgroup readers and e-mail clients both have threaded message views and different mechanisms for filtering out messages from specific people.

This line of reasoning may seem like an impetus for a shared component architecture, but it extends further as the uses for the different communications channels have started to overlap. For example, the same tasks can be and are accomplished via e-mail and IM, e.g. sending quick messages for short term coordination, reminding others of pending tasks, and asking questions [58]. Furthermore, multiple modalities may be involved in the completion of a single task: to notify a friend that you are going to be out for the day, you may start off by attempting to send an instant message, but if he or she is not online, you will likely switch to an e-mail client because most IM systems will refuse to take messages for offline users. If a user has a technical problem with a program, he or she may have to send a helpdesk request through several different channels separately.

The current separation that exists between different messaging systems also creates usability problems. For example, the shortcomings of the simple (often hierarchical) organizational tools that pervade communications clients today become aggravated when used to view a discussion thread flooded with thousands of messages from dozens of contributors. Often people will not have the time to sift through a discussion one message at a time or even one thread at a time. Instead, users may need to examine an ex-

tended conversation along different axes, such as topic or author. (The visualization tools discussed in Chapter 9 address many of these issues.)

Despite their differences, the extent to which it is actually necessary for people to conscientiously distinguish these forms of communication is surprising. Currently it is essential for one to know which program is needed to send or receive a message over a certain channel, just as it is necessary with productivity software to know which program is needed to handle a given document type. However, when a person needs to look up information previously received from a colleague, the fact that it was sent over e-mail or was received on his or her mobile phone becomes less relevant. Just as people do not file paper correspondence by what delivery service it was sent by, users should not be required to categorize their incoming messages by the channel it arrived on. Accordingly, our discussion of messaging echoes the theme of shared object space common to the rest of the thesis.

12.2 Applying our data model

The overlapping problems described above are to be expected, as these systems are all addressing different aspects of the same fundamental problems of interpersonal and group communication. While in the beginning, the different communication channels crystallized functionalities specific to key activities, users have grown more reliant on these systems and are now bumping against the limitations of the abstractions. As a result, the opportunity exists to take a "bigger picture" look at the situation and to recast the problem in terms of a broader messaging abstraction. Once the existing systems are unified under a common model, we can also enhance all forms of messaging by incorporating features that are currently present only for specific messaging paradigms. We can achieve this unification not by disposing of the current approaches but by learning which key concepts have emerged over decades of use, capturing these concepts (e.g., message, address, conversation) in our data model, and recasting current messaging paradigms in terms of these basic concepts.

By storing electronic messages in a semistructured data repository, users are able to experience a number of benefits. One important feature of this data model is the ability to annotate objects with arbitrary properties, as was discussed in Chapter 9. Often these annotations can be useful to those with whom the annotator is working. For example, if

a person marks something as junk mail, it would be convenient for that person's friends to also benefit from this notation. Similarly, if someone takes the trouble to classify a document into a taxonomy, perhaps this metadata would be useful to potential readers of this document. These scenarios are made possible by using a semistructured representation as a *lingua franca* for electronic messaging.

12.3 Message as a unit of communication

In order to incorporate the various forms of messaging available, we define the class `msg:Message` in a very general manner. In our system a message is a unit of expressive communication transported from one or more senders to one or more recipients. This definition allows us to unify the concepts of instant messages, e-mails, newsgroup postings, annotations, chat, and even articles delivered via news feeds.

However, care is needed to distinguish messaging from the general recording of information. We argue it is possible but not useful to define messaging to include all forms of information passing from one component to another. For example, does saving a document to disk constitute the sending of a "message"? Most users are unaware of the inner workings of a computer, and the transmission of a document from memory to disk occurs within the same "black box" as far as the user is concerned. Like all models, our ontology implicitly places a practical boundary around the types of objects we wish to describe. To achieve this, we wish to restrict our modeling of messages to include only communication where it is useful to acknowledge the transmission process from the sender's perspective. Messaging in the sense presented in this paper must therefore involve communication between either human parties or autonomous agents that are working as substitutes for human parties, such as forum moderators.

In addition, it is helpful to distinguish the idea of "audience" from "recipient". When a user posts a webpage to an Internet HTTP server, he or she is declaring the audience of that webpage to be the general public. However, the user is not "sending" this page to anyone; in other words, in the beginning the page has no "recipient". Contrast this situation with that of information being leaked to the press: a reporter may be the recipient of information, although the originally intended audience of that information certainly does not include him or her. For our concept of messaging, we care only about message

recipients, emphasizing the notion that messaging concerns only the aspect of transmission, not intended audience.

Messages come in various forms. The bulk of all messages are textual, but in the context of the Semantic Web it is also useful to provide for messages that conform to some established schema, such as meeting requests, money orders, or even bank statements. However, it is important to note that objects such as text documents, financial statements and currency can exist outside of messaging environments. The notion of message is independent and to some extent orthogonal to the notions of text documents and financial statements. Therefore, we define a message as an object for which a sender and a set of recipients are specified. A message also contains a body, a resource that can be of any type known by the system.

Messages' bodies can be of any type, and most often messages have textual bodies. Serialization of a textual body is not only simple but is also often supported by underlying messaging protocols, such as SMTP. Preparing other types of bodies for transport is a more difficult problem; in these cases we use the graph extraction techniques described in Section 4.4 and transmit the relevant RDF subgraph as RDF/XML.

12.4 Messaging transports

Our ontology is designed specifically to work with existing messaging systems. As a result, the base of our messaging infrastructure consists of a series of mail drivers capable of sending and receiving messages over protocols such as POP3, SMTP, and Jabber. When a message is to be sent, the system must be able to determine which of the available messaging drivers is best suited to delivering the message given the circumstances. In our system, messaging drivers are described as having type `msg:MessageSendService`.

Messaging drivers are responsible for emulating functionality that is not normally available in the underlying protocol. For example, e-mail uses MIME headers to describe metadata concerning the messages, whereas IRC messages typically have no metadata. Techniques such as encoding messages in SOAP envelopes can be employed in these cases .

Messaging drivers are also responsible for incorporating messages into Haystack's RDF repository, which makes messages accessible via the user interface. This results in all messages being persistent.

As noted earlier, each messaging protocol currently maintains its own address scheme. For example, SMTP servers are programmed to route e-mail messages according to recipients' e-mail addresses. These addresses are represented directly in our ontology as URIs. In the case of e-mail, a well-defined scheme for forming a URI from an e-mail address is available, which simply directs the system to prepend the `mailto:` protocol scheme to the e-mail address. There have been similar schemes devised for the other protocols.

When a message is specifically directed to be routed by means of a particular address, the system needs to be able to resolve the address to a driver capable of interpreting it. We specify a base class called `msg:Endpoint` that represents addresses handled by mail drivers. This allows the system to recognize resources as being addresses without relying on the syntactic form of a resource's URI (e.g., whether the URI starts with "mailto:"). We can then derive classes such as `msg:EmailAddress` and stipulate that e-mail address resources be asserted to have this type.

Whereas most current messaging systems have a single identifier which is used for both identification and addressing, our unified messaging ontology necessarily distinguishes between the two concepts since the same person may have a different address for message delivery depending on the message type. However, it is not necessary for those sending messages to concern themselves with the specific transport by which a message will be sent. Instead, people can be represented directly by means of the `hs:Person` class. Recipients and senders are specified by instances of the `msg:AddressSpecification` class. Address specifications can specify either a specific address or a person resource, or both. Addresses can be associated with people with the `msg:hasAddress` predicate.

Furthermore, some protocols support the notion of presence, allowing users to tell when their contacts are online and available for communication. We model this notion by associating `msg:Endpoint`'s with the `msg:onlineStatus` property. Drivers for protocols that support presence are responsible for keeping these properties up to date.

12.5 Conversations

Built up from messages are higher level aggregations that model patterns of communication. We highlight two specific forms that pervade electronic communication, threads and conversations, while noting that these two forms can exist independently of each other and that other forms can be defined.

A thread is a connected subset of a collection of messages. Threads typically indicate a stream of messages exchanged between different parties on a very specific topic. Keeping track of threads is important because responses can sometimes be interpreted only in the context of other messages in the thread. Our concept of threading is indicated by the presence of the `msg:thread` property, linking a message to a `msg:Thread` object.

Threading is not directly supported by most e-mail protocols, but one method for constructing `msg:Thread` objects is to reconstruct threads based on `msg:inReplyTo` connections corresponding to the "in reply to" relationships present in e-mail and newsgroups. However, users often misuse the "Reply" feature of e-mail (e.g., sending a message to someone by locating the last message received from the person and clicking Reply) or fail to use it at all. Still, we feel that using `msg:inReplyTo` is a useful indicator of a thread, despite occasional human error. Perhaps providing a user interface that makes more prominent use of reply chains and gives users more control over how messages are sent will help to fix this problem.

Also important is the concept of a conversation. Like threads, conversations consist of a collection of messages, but the connection between messages in a conversation tends to be more loosely defined by a more generalized topic than those in a thread. Conversations in our ontology have type `msg:Discussion` and correspond to newsgroups, instant messaging conversations, and IRC chats. `msg:Discussion` derives from `hs:Collection` and instances thereof can be manipulated in the ways described in Section 8.3.

In addition to being a collection of messages, conversations also maintain state in order to help facilitate changes in the interaction. A conversation resource keeps a record of the current participants in the conversation, as this list may change during the course of the conversation. Furthermore, whether the conversation is currently considered public or private or being held synchronously or asynchronously is recorded.

Depending on the protocol, conversations can sometimes serve as endpoints of messages. For example, messages in IRC chats and newsgroup postings are often not directed at any individual in particular but instead to the group, represented by the conversation object.

There are many ways users may choose to initiate communication with their colleagues. A user may know *a priori* that the message he or she is about to send will bring about an extended conversation. In this case the user can create the conversation explicitly by selecting "Start discussion" from the "Communicate/work with others" menu, akin to creating a private newsgroup or IRC channel. The discussion that is created is peer-to-peer in that it is maintained over instant messaging and e-mail protocols. Newsgroups and IRC channels can be supported with the same conversation user interface, highlighting the uniformity our system provides to communication.

However, in other situations, a user will send a quick informative message without expecting the numerous replies that ensue. At some point the resulting dialog may become important enough to warrant separate treatment as a conversation as modeled by our ontology. It is crucial that the user interface support *a posteriori* changes to the way a user organizes his or her messages and conversations.

When users are ready to start managing a stream of messages as a conversation in the user interface, he or she can select View Discussion from the context menu of a message to instruct the system to assemble all of the relevant messages into a conversation. Currently, the system uses the `msg:inReplyTo` relationships that exist between messages to locate the connected subgraph of the entire message corpus to identify the conversation.

In other cases, users may wish to collect messages that are loosely related into a conversation but that do not form a subgraph as defined above. Haystack allows users to drag and drop messages into the view of a conversation to indicate that the messages are a part of the conversation. Users can also add messages to conversations by treating the conversation as a category.

12.6 Shared annotations

As discussed in Section 9.2, annotations are simply messages with a well-defined topic resource in our model. Like messages, the concept of being an annotation is completely

detached from a resource having other types, so any object can be made to be the body of an annotation. All of the previously described characteristics of messages apply equally to annotations; in particular, annotations can be replied to. Annotations have type `ann:Annotation`, derived from `msg:Message`, and the object being annotated connects to an annotation with a predicate derived from `ann:annotation`.

The ontology presented here mainly addresses the notion of an annotation in a collaborative context. In such cases annotations are most often persisted into an annotation store. Annotation stores are similar to newsgroup servers in that they both contain groups of threaded messages loosely related by common topic spaces, and as such, annotation stores can be modeled as mail drivers in our model. However, our implementation of shared annotations makes no assumptions about the underlying store; a conversation held over an IRC channel makes for an equally suitable medium for holding annotations.

Chapter 13 Capturing information expertise

Up to this point we have discussed techniques for capturing connective information from users, for allowing users to take advantage of such connective information for retrieval, and for distributing connective information to others. In this chapter we examine the meta-problem of characterizing the forms of connective information that exist within a domain. We will refer to such characterizations as *information expertise*.

Conventional interfaces for capturing information expertise have focused on maximizing expressiveness and precision at the expense of usability. Despite the fact that those unfamiliar with frame-based logics, programming, or knowledge representation techniques have been excluded from taking advantage of these tools, it is actually the non-computer scientists who have the most to offer in terms of describing their domains.

In this chapter we describe techniques for allowing unsophisticated users to describe selected forms of information expertise to the computer and to share this knowledge with others using the approaches given in Chapter 11. Users must be given easy mechanisms for assembling the building blocks provided by developers, including operations, view parts, aspects, and contexts (a concept introduced in this chapter). We show how the user interface approaches discussed throughout this thesis can be applied to enable users to assemble these building blocks and to take advantage of these assemblies automatically to filter for relevant information with respect to the domain.

13.1 What is information expertise?

Information expertise can be characterized in a number of ways. One of the most common means is ontology—a body of knowledge that specifies a means for representing objects, concepts, and other entities in a domain of interest and the relationships between these entities [36]. Ontologies can be used to bring order to collections of data for the purposes of visualization, exploration, and analysis. One widely explored form of ontology is schema. For example, database schemas enable relational databases to efficiently store and retrieve information. On a smaller scale, object-oriented class hierarchies facilitate code reuse and maintainability. Schematic knowledge can be exposed to the user in a number of different ways, from automatically-generated database forms to desktop productivity applications such as address books and e-mail clients. Another

important form of ontology is taxonomy, which is used by sites such as Yahoo! to give users a logical view of the resources available on the World Wide Web. Tools for creating taxonomies exist and are employed by IT consultants to produce solutions that help users of a domain navigate and manage their information more efficiently.

Two less prevalently exposed forms of information expertise are contextual and presentation knowledge. The knowledge of when and in what form certain pieces of information are useful and relevant plays a key role in helping users to work with their data. For example, address books are embedded with contextual knowledge that indicates that important attributes such as a contact's name and address should be presented more prominently than less important attributes such as anniversary date and digital certificates. Similarly, context menus are often designed to present the commands that are most useful given the current selection (e.g., copy and paste for a word processor) and leave out less frequently-used commands (e.g., view at 50%).

Information expertise is often considered to be too difficult for the average user to encode, which perhaps explains why tools such as schema designers, form designers, and taxonomy editors have been designed for use by "experts" (no pun intended). Indeed, the most common packaging of information expertise is the commercial software application—an item whose construction is out of the reach of all but sophisticated software engineers. For example, Microsoft Money, a personal financial management tool, embodies knowledge of how to visualize account statements, track stock portfolios, and calculate capital gains. Microsoft Money also knows that maintaining a list of frequently-used accounts is useful while maintaining a list of frequently-used dollar amounts is not.

In reality however, people frequently actualize information expertise in their daily lives. Students keep homework and notes from different classes separated by using a highly personalized ontology, whose physical manifestation is implemented by means of binders, dividers, etc. People maintain card files with the information that matters most to them personally, such as favorite color, birthdays and perhaps even the last time a telephone conversation or a letter was exchanged. Also, some forms of information expertise are experiential; for example, travel agents know that when their customers travel to foreign countries, they may need visas, and that it is important to check the weather before leaving in order to pack properly.

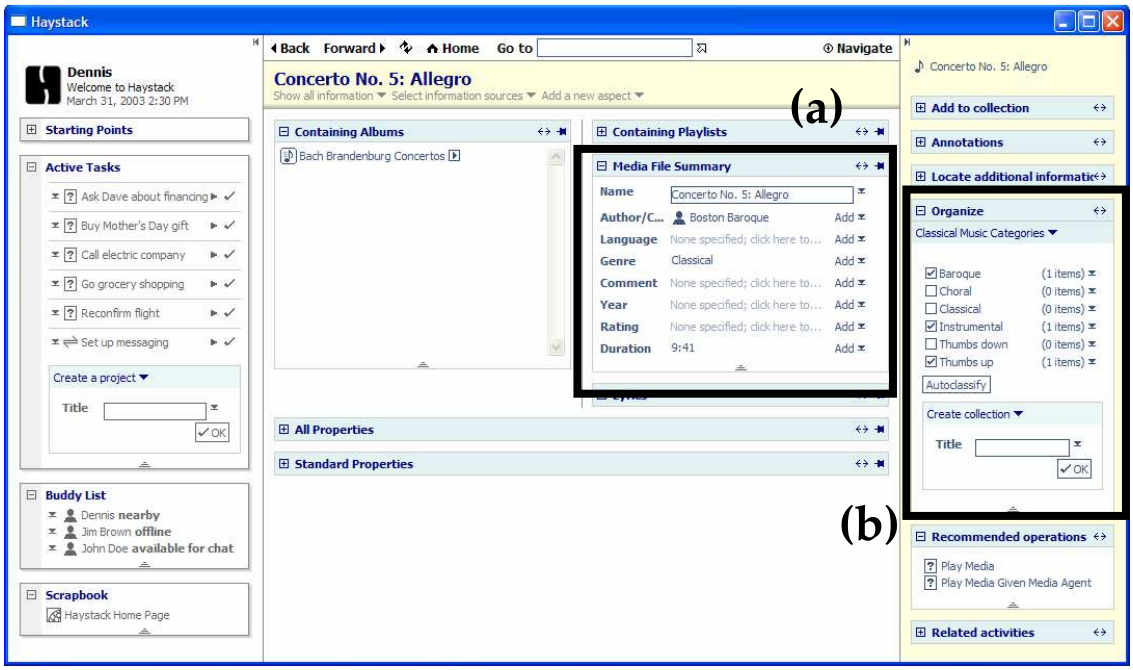

Figure 39: Display of an MP3 file: (a) one example of an aspect (other aspects include Containing Albums, Organize, All Properties, Annotations, etc.); (b) the Organize aspect

In a world in which more and more information is becoming accessible electronically, it is imperative that software enable users to specify information expertise in order to better manage their information. Tools such as Yahoo! are critical to bringing order to domains with so much information that any one person could not possibly manage it. However, these tools must be accessible to the average user. It is not practical to assume that such a user, who is now becoming familiar with digital photography, MP3 files, and e-mail, will hire IT consultants to customize software to allow him or her to group pictures by the trip on which they were taken, music by moods named by the Seven Dwarfs [47], or messages by customer engagement account.

Making information expertise explicit within our data model is also essential because Haystack does not divide functionality into applications as is commonly done today. On one hand, users may benefit from seeing integrated views of their information regardless of application, but on the other hand, applications served as artificial filters on the amount of information that would ever be presented at once. For example, one does not see the list of music sung by a person or his or her family tree when looking at that person in the mail merge feature of a word processor. The mechanisms we give here for describing information expertise play the role of "application" as a grouping mechanism without the detrimental limitations that result from true information segregation.

Furthermore, organizational insights encountered by specific users or domain experts may prove useful to those less familiar with a domain. In a customer relationship management group, the information kept about customer accounts may differ slightly from account manager to account manager. When new account managers join, instead of requiring them to start from scratch, existing managers should be able to transfer their expertise, in the form of an ontology, to others. To facilitate sharing, it is important that information expertise be modeled in a consistent, unified fashion so that it can be serialized into a transportable representation and incorporated into others' systems, as can be done with our data model.

13.2 Putting the pieces together

To allow users to specify information expertise, we cannot expose raw "meta-concepts" such as classes, properties, and constraints. Instead, Haystack monitors the users' interaction with certain visual elements called *ontological proxies* as users work with actual instance data. Ontological proxies represent pieces of information expertise on the screen, such as the presence of a birthday field or a "cheery mood MP3 files" category, and when users drag and drop proxies from one area of the screen to another, the system can interpret this activity to imply something about the ontologies being worked with.

We examine several kinds of ontological proxies in this paper. In particular, the forms of information expertise we are interested in exposing to the user include not only the traditional notions of schema and taxonomy but also presentation information encapsulated by views and aspects. A lot of presentation information, such as the order in which address book fields should be presented or the fact that a corporate directory can be displayed as an organization chart, is important information expertise not encoded by schemas and taxonomies alone. We are also interested in encoding contextual knowledge (cf. Chapter 11), such as the fact that when one looks at a travel itinerary before a trip, the weather report for the destination city is of interest, but not when one has completed the trip and is creating an expense report.

13.3 Ontological proxies

Ontological proxies are elements of information expertise that represent meta-information about a domain, such as properties, operations, and categories. Like all re-

sources in Haystack, ontological proxies are exposed on screen as views. These views not only provide a handle by which users can manipulate the underlying ontological proxies, but often these views are also responsible for actualizing the information expertise they embody. Examples of this embodiment will be given in our discussion below.

Ontological proxies act like other resources in our system. For example, views of ontological proxies can be dragged and dropped around into appropriate containers. Just as contact resources can be dropped into address books, properties can be dropped into group aspects (see Section 9.1.1). The main observation here is that Haystack makes no special distinction about the "meta"-ness of ontological proxies.

Categorization schemes are the basis for taxonomical information expertise. The same direct manipulation approaches that were used for other ontological proxies apply equally in this case. Just as items can be placed into specific categories by means of drag and drop, categories themselves can be added to categorization schemes by drag and drop. Categorization schemes can be browsed and manipulated by means of specialized views (discussed in Section 8.4) and the Organize aspect (Section 8.3.2).

Operations are created to specify what can be done with resources of a certain type. While most operations must be written by developers, they can be manipulated to some extent by users. For example, currying is one mechanism by which end users can extend the set of possible operations, and often used curried forms, such as "submit form to second line manager" as a specialization of "submit form to someone", represent encodings of ontological knowledge.

Aspects are intended to be user-level, reusable ontological proxies for characterizing presentation knowledge. Some examples include:

- A list of e-mail address associated with the selected resource
- A list of the people known to live in the selected location resource (e.g., when viewing a city or country resource)
- Full name (i.e., a first, middle, and last name)

When aspects are displayed in certain containers, such as group aspects (Section 9.1.1) or aspect views (Section 6.5.4), a label or a title bar is provided to name the aspect. The label or title bar represents the aspect itself in the user interface (i.e., the label or title bar is a view of the aspect) and can be used as a handle to the aspect itself for the purposes of drag and drop. Aspect containers act as collections of aspects that relate to some specific class of resources. Manipulations to this collection are recorded directly as information expertise. For example, if a user drags the e-mail address aspect from a person's view into a hotel's view, he or she is likely indicating that hotels have e-mail addresses, in his or her way of thinking. By making aspects first class, we are imparting a higher level of expressiveness to the user interface. Unlike current systems, in which the exact constitution of a presentation is hard coded, this constitution meta-information is first class, exposed directly, and can thus be directly manipulated.

13.4 Example scenarios

To motivate the kinds of problems we wish to address by allowing users to manipulate ontologies, several example scenarios are described in the next section. Afterwards we discuss a series of different ontological components and the ways in which they may be manipulated by means of ontological proxies placed in the user interface.

To demonstrate how users are able to impart information expertise to the system, we illustrate the use of ontological proxies with some examples.

13.4.1 MP3 files

Haystack comes with support not only for playing MP3 files but also for organizing them. Figure 39 shows a screenshot of the system displaying the default view of an MP3 file. However, as noted earlier, people vary in their approaches to organization, and these differences are accentuated in the particular case of organizing music. Suppose a user enjoys singing karaoke and only cares about seeing the lyrics, the language of the song, the length of the song, and the authors. In this case the user can create a new view specifically suited for karaoke and remove the excess fields and aspects.

Suppose another user is taking a class on classical music appreciation and wants to get started organizing his or her newly growing classical music collection. Instead of waiting until the end of his or her class, when he or she will have learned all of the ways by which classical music may be categorized, the user may directly import a taxonomy

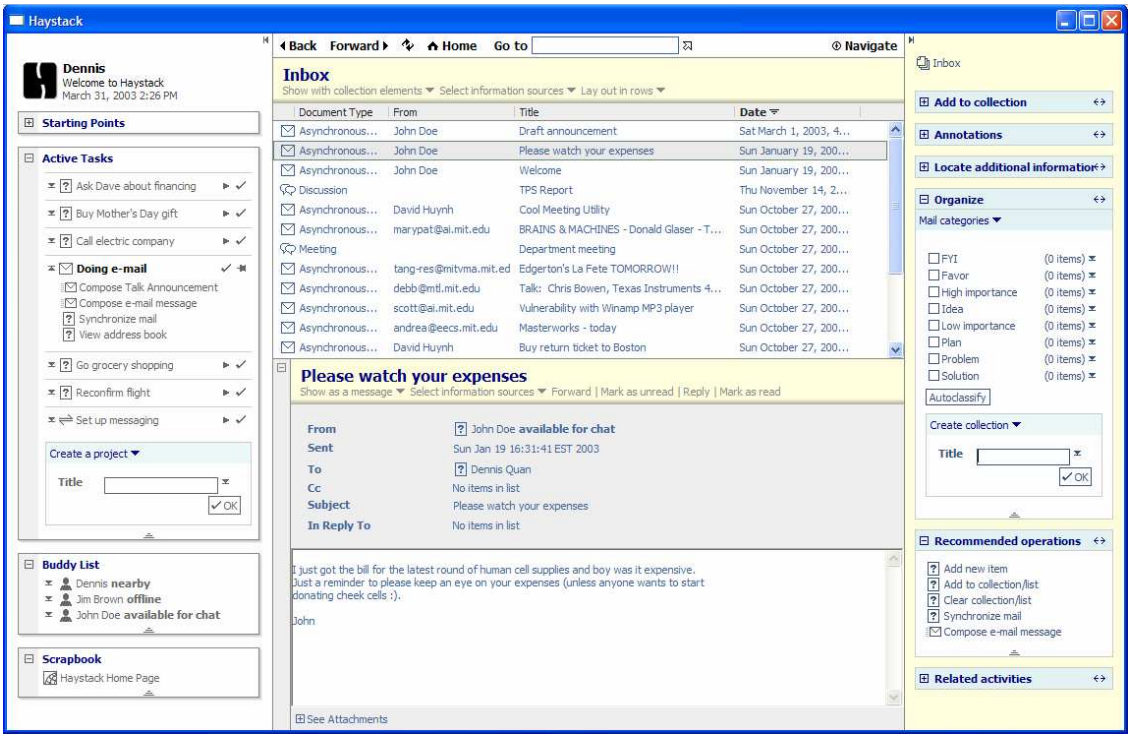

Figure 40: E-mail context in Haystack

from a friend, the teacher of the class, or off of an enthusiast's website, which he or she can then use to file away his or her music. Furthermore, the user is free to customize this taxonomy with two extra categories, "thumbs up" and "thumbs down", for recording the user's own.

13.4.2 E-mail

Users benefit from seeing relevant information and tools rather than being overloaded with all possible information. Haystack uses contextual knowledge to do this filtering. For example, when users click on the E-mail context on the left side of the screen, Haystack presents a list of relevant operations, including composing a new e-mail and synchronizing e-mail accounts (refer to Figure 40). Similarly, the taxonomy shown on the Organize aspect can be set to show categories that are often used for filing away e-mails, rather than music.

13.4.3 Contact manager

Showing relevant information is also important when working with contacts. The view for contacts is shown in Figure 42 and is composed of aspects for names, addresses, electronic addresses, etc. Because of the people-centric way we conduct our lives, a virtually

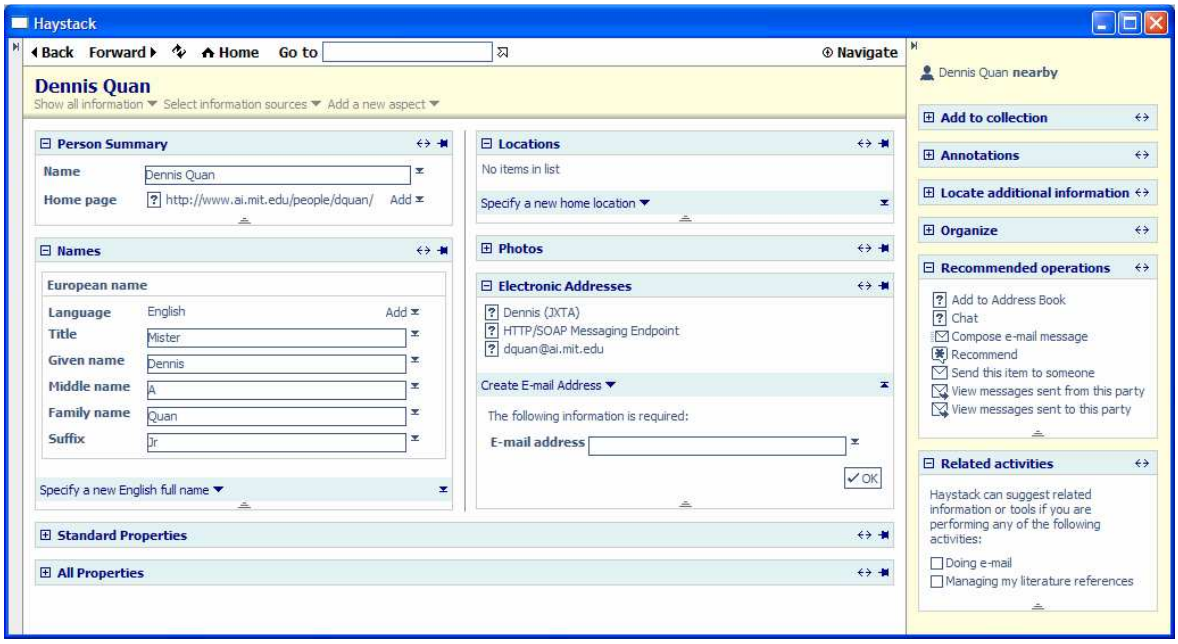

Figure 42: Contact editor

limitless amount of information could be shown on such a view: a list of the person's relatives, how/when the person was met, birthdays and anniversaries, etc. Instead, we use contexts to help show only the information that is important. For example, when the user is working on managing his or her literature references, we can show a list of all papers written by the contact. On the other hand, when the user is doing his or her e-mail, we can show the correspondence written to and by the contact.

13.4.4 Customer records

Similarly, when maintaining customer records, there are numerous different possible aspects that could be displayed. In Figure 41 we have depicted aspects for recent orders, the current shipping address, and contact information. Suppose the user's company is upgrading to Internet-based customer relationship management. The user could go into his or her address book and drag and drop aspects for specifying e-mail addresses and tracking correspondence into the customer record view. These actions would indicate to the system that e-mail addresses and correspondence are also important for the domain of customer record management.

Chapter 14 Application: personal information management

In the first eleven chapters of this thesis we have characterized the semistructured data model and shown how it can be employed to give users a more expressive environment in which to not only hold their information but also capture the connections between different parts of that information. In this chapter we apply these principles to the area of personal information management. We draw upon a rich body of previous literature that has identified the problems plaguing existing personal information management software and has pointed to the need for better integration. First we examine how personal information has been managed in the past and identify the common problems that have been well characterized by the literature. Then we describe how an integrated information space and organization tools are necessary to address the problems uncovered in previous disparate environments. Finally, we present the results of a qualitative user study that determined that users are not only willing to supply additional cross-task organizational information given the right tools but also do indeed benefit from the availability of such information during retrieval.

14.1 What is personal information management?

Personal information management is the problem of using computers to deal with day-to-day information affecting our personal lives. At present, this activity is performed through a wide range of different desktop productivity applications. Here, we characterize personal information management along three axes: task coordination and management (what someone is working on and how), archiving items of interest (storing items not being worked on), and communication (sharing work with others). The first of these two axes are discussed below; the third was explored in detail in Chapter 11.

14.1.1 Task coordination and management

People keep track of their work using a variety of methods, including to-do lists and day planners. Many of these approaches have direct electronic analogs; products such as Microsoft Outlook and Lotus Notes allow users to create task lists and manage appointments on calendars. Documents themselves are typically stored on the file system, where folder hierarchies can be constructed to mimic their paper equivalents—bins and piles—to some extent.

These software products have also evolved to support other forms of task management. E-mail—perhaps the key functionality of Outlook and Notes—can serve as a task management system both for tasks that involve significant amounts of collaboration [32] [67] and those that are self-directed. In particular, a document's presence in certain containers, such as an inbox or the desktop, can serve to remind people of its significance and timeliness [32] [68].

14.1.2 Storing items of interest

In the paper-based world, documents are filed into folders, binders, piles, and filing cabinets when they are no longer useful in the short term. Similarly, the file system provides an analogous mechanism for the long term filing of electronic documents [68]. However, there are many aspects of filing electronic documents that have no analogs in the paper world. First, letters and other forms of correspondence have been and are still routinely filed away in the normal course of business. However, with the advent of e-mail, the scale of the filing problem has changed dramatically. Similarly, for many forms of information such as that found on the Web, people are now filing away *links* to such information instead of actual copies of the information. Because linking is such a lightweight process, the Web has greatly increased the amount of information that can be filed away [1]. Finally, with multimedia materials such as digital photographs, the need arises to be able to conveniently browse through these items in a variety of different ways [70].

14.2 Making the case for integration

The same human computer interaction literature that has characterized the use of personal information management software is also overflowing with analyses of the problems of such software, most of which was designed to solve some arbitrary cross section of the problems in our daily lives. For example, budgeting for a vacation and booking airline tickets are related activities, as are paying bills and submitting feedback on the power company's website about why electricity rates have recently increased, yet the applications for performing these tasks—financial management software and web browsers—divide this space counterintuitively. It should come as little surprise to learn that many of these problems arise from artificial distinctions created by applications.

14.2.1 Task-based organization

To be fair, many of the problems observed by researchers are problems that exist in the physical world too. A common complaint is the lack of expressiveness and manageability of folder hierarchies [1] [18] [67]. Problems include the unnecessarily high cognitive load needed to file something into a specific folder and the clutter associated with managing dense folder hierarchies [1] [18] [32], as discussed in Section 8.1. Despite the computer's ability to host more powerful abstractions for organization than those possible with paper, the folder system remains the dominant paradigm.

To address the shortcomings of the folder paradigm, researchers have looked to other, more natural grouping mechanisms. Because e-mail has become an environment for managing tasks, Bellotti et al. have examined how to allow users to manage their e-mail in terms of tasks [71]. Their prototype, Taskmaster, automatically grouped e-mail threads by task and also allowed users to add e-mails to and remove e-mails from tasks. Support for items other than e-mails was accomplished by creating stub messages with attachments and making attachments first class within the environment. A related example is Lifestreams, an application that uses timelines of semantically-grouped document links to help users visualize and organize their document spaces [73].

However, task management and information organization exist beyond e-mail alone, and there are limits to what can be accomplished with a single, focused application. Along these lines, Abrams et al. noted that all good bookmark managers require the user to leave the browser, and that users would be more willing to provide organizational information if the tools were embedded into the application they happened to be working with [1].

14.2.2 Visualizing information together

Another oft mentioned problem is the inability to visualize different pieces of related information together. Many have noted that current personal information managers function as several, glued-together mini-applications; in particular, despite the connection between task management and e-mail, the task list manager and e-mail functionality of Outlook are kept separate [71]. Several projects have experimented with placing information from multiple domains together in the context of some particular activity. The aforementioned Taskmaster prototype centered information management around

tasks. Contactmap presented information in terms of how it relates to the people belonging to the social networks surrounding the user [72].

Such highly specialized approaches to information visualization are likely useful for certain situations but not others. Other tools have given users more generality for working with arbitrary data sets, giving users more flexibility in terms of the kinds of information that can be visualized and managed. Semantic Regions is a standalone visualization tool that lets people create regions on screen that correspond to the presence of certain attributes [74]. These regions can be composed with Boolean operators to produce useful visualizations of related information. Presto allows users to manage general personal information in an attribute-oriented fashion, as discussed in Section 2.3.1.

14.2.3 Repurposing applications

In an effort to give users a richer environment for managing their information, many previous approaches have attempted "bundling" useful features into popular applications. One possible reason for such approaches is that finding a usability paradigm that reaches wide acceptance is rare; examples include e-mail and the World Wide Web. Furthermore, research into how any specific application is actually employed by users often unveils ways of using the application not intended by its designer, e.g., e-mail used for task management and archiving [32]. As a result, we see applications being "repurposed" to bridge users over to these extra features; for example, some have designed Web portals that host word processors and spreadsheets in response to the ubiquity of Web browsers.

14.3 Personal information management with Haystack

We have now seen that several key problems with personal information managers have been examined by previous work. Two observations that have been noted on numerous occasions are that users do not enjoy having to switch contexts to benefit from functionality, but that users want to see related forms of information presented together regardless of which application it happens to be supported by. The answer to personal information management must integrate all of these different discoveries. We describe below how Haystack provides the flexibility to address the concerns raised in the previous section.

14.3.1 Context model

Task-based management proved to be an especially effective methodology for e-mail because people often think of their work in terms of the tasks they need to do. The study of Bellotti et al. indicated that their change to make attachments and other non-e-mail documents first class in their environment was in response to the role other documents play in task management [71]. In order to make all information truly manageable in a task-based fashion while keeping the organization mechanism integrated into the environment, we have built the context mechanism in Haystack into the user interface paradigm. As discussed in Chapter 11, contexts can be used not only to group related items together but also to track recently used documents and to inform the system of contextually-related information. Contexts are managed by the user with a to-do list, so that information that is already being supplied to the computer is reused for the purposes of tracking information in a task-based fashion.

14.3.2 Collections

The ability to group related information from separate applications together, as was seen in Contactmap and Placeless Documents, requires not only that information be integrated into a common abstraction, as is the case in Haystack, but also that generic mechanisms for aggregation be provided. In Haystack, collections provide this capability. Many of the different visualization and grouping examples shown in previous research can be accomplished in Haystack with collections being displayed in different views, as was discussed in Section 8.3.1. We also addressed the problems of managing deep hierarchies and providing convenient mechanisms for filing documents in Sections 8.2.3 and 8.3.2, respectively.

14.3.3 Relationship-based browsing

To bring different forms of information together, the user interface must emphasize the connections between information. This is achieved in Haystack in a number of ways. First, aspects are used to present context-dependent pieces of related information, such as the list of recent e-mails from a person (a feature present in Contactmap). Second, our view-based interface permits direct manipulation of resources. As a result, users can point and click to navigate from resource to resource, regardless of application. Finally, graph browsing shows users the bigger picture and is a generalization of the relationship browsing process, as was discussed in Section 9.3.

14.4 User study

We conducted a qualitative user study on four participants, all experienced with computers, over a four week period. The overall goal of the study was to determine whether users would benefit from a personal information management system that integrated task-based organization and flexible forms of visualization.

14.4.1 Hypotheses

The initial hypothesis was that users would prefer working with common tools for managing all of their information in one place. We felt that the task-based model, which was successful for e-mail, should be equally successful for all aspects of document management, provided users do not have to leave their work environment to use it. Users should prefer flexible tools such as collections and tasks to keep track of their information. Furthermore, being able to record relationships between information from different "applications" should encourage use of the system by reducing unnecessary duplication, increase the amount of metadata being recorded, and give users a bigger feature space over which to search.

14.4.2 Procedure

Participants were given a copy of Haystack to use for a four week period. We encouraged them to use the system for as much of their personal information as they felt comfortable; in particular we encouraged the use of Haystack for managing e-mails, photographs, flight itineraries, bookmark collections, and text notes. In addition, we requested that users utilize the to-do list system so that we could study the effects of merging task list management (i.e., jotting down to-do items) with organizing information with respect to these tasks (similar to what was done in the study performed by Bellotti et al.).

Users were given a brief survey at the beginning to determine their initial preconceptions of task-based information management. We then introduced them to Haystack's basic usability model, including context menus, drag and drop, the task list, collections, the organize pane, and our browser-like navigation model. During the course of the study, we were available to answer questions and fix technical problems with the system. Participants were given another survey half way through the study eliciting freeform responses to questions regarding the features of the system. Finally, a combination multiple choice and comments survey was administered at the end after we encouraged us-

ers to try features they had not yet had a chance to experiment with. Most multiple choice questions asked users to rate the usefulness of features on a 5 point scale from "not useful at all" (1) to "invaluable" (5).

14.4.3 Initial conceptions

Our initial survey, which was given to participants before they began to use Haystack, revealed a number of patterns. Most indicated that they had an extremely systematic hierarchy at work in their file systems, but that this hierarchy was mirrored in other places, such as on IMAP servers. A lot of freeform information was also being kept on paper as a result of lack of support from their information environments. Finally, some participants pointed to a desire for an organization scheme based more heavily on associations.

14.4.4 Overview of results

Although users were often bound by time and other constraints (e.g. security restrictions of data in their possession) that prevented them from switching completely over to Haystack, all of our participants did use Haystack to manage information for one or more personal projects, ranging from hunting for an apartment to keeping design notes for web site design. This real life use of Haystack bolsters the results we detail below in terms of organizational flexibility, but another study that examines the use of Haystack over a more extended period of time is warranted to determine if retrieval performance has been enhanced.

One of the biggest complaints about the system was its performance. Admittedly, the developers of the system have had access to state-of-the-art machines with a gigabyte of RAM and at least two gigahertz processors, while our test subjects generally had 512 megabytes of RAM or less and at best two gigahertz processors. Further optimization of the system, along with the inevitable improvement of hardware, should ease these problems over the next year.

The mid-survey revealed that although instructions were given to participants detailing the features of the system, the method for utilizing some of these features was not as evident as was anticipated. A related comment was that the system encompasses a lot of information, which produced a corresponding initial learning curve. As is the case with any early prototype of the system, usability issues were present, and many areas of im-

provement were isolated. However, some of the bigger themes of our study were indeed confirmed, in spite of usability problems.

14.4.5 Results from the use of collections

Overall, users responded overwhelmingly positively to our support for collections. Some aspect of collections represented the favorite feature of every one of our participants. Support for items being in multiple collections at once was one feature whose usefulness was surveyed. Three users rated this support with at least a 4, and the fourth user rated it a 3 because he felt he did not have enough data to gauge the usefulness of this support.

We received similar feedback for the Organize aspect, which was introduced in Section 8.3.2. Three users rated the usefulness of the Organize aspect a 4 or higher, with the fourth user saying that he did not use the feature. Evidently, the fourth user created and maintained collections outside of the Organize aspect; this may point to more usability work being needed. Overall, the Organize aspect seemed to have effectively implemented the idea studied during the user study discussed in Section 8.4.

14.4.6 Results from the provision of heterogeneity

Users especially appreciated the ability to place items of different types into the same collection. Two users rated this feature a 4, and the other two rated it a 5. Users also reported in the freeform text responses that this feature was used in the course of managing actual projects, confirming the hypothesis that multiple types of documents are used in the course of a project.

We asked users whether the support for quickly jotting down notes was useful. Three rated the feature a 4, while the fourth did not try the feature. One user noted that he was able to use the support for heterogeneity to quickly jot down notes instead of sending himself an e-mail, reinforcing the theme that repurposing does not address the root problem. E-mail is fundamentally an interpersonal communication tool, so keeping personal notes is an example of a use of e-mail for which it was not originally designed (repurposing).

On the issue of switching views, our users reported that this feature was not used much. Two users did not try the feature at all, one of which reported that he preferred viewing

collections as lists. One user found the performance of the system inhibiting his use of this feature.

14.4.7 Results from the task-centric paradigm

Finally, we received mixed results on the task-centric paradigm exposed by Haystack. On one extreme, two users reported that our support for jotting down to-do lists was minimally useful (2 on the scale). Both noted that the support in Haystack was cumbersome and difficult to get used to. One of these users wanted support for embedding tasks within tasks in order to manage more fine-grained notions of task management. On the other extreme, one user rated the support a 4 in light of the flexibility Haystack gave him in managing to-do lists.

Our users also had a mixed response to the play and pause buttons for telling Haystack when a task was being performed. One user was annoyed about needing to tell the system about his activities, while one user enjoyed having this ability. One of the remaining users felt worried that because he was not meticulously telling the system about when his tasks started and stopped that perhaps the system was deriving invalid conclusions about the way in which he worked. Consequently, when asked about keeping most recently used documents lists on a per-task basis, most users reported that they had not used the system enough or that it was just minimally useful.

14.5 Discussion

Our study pointed towards validation of our hypotheses on a number of fronts. The biggest success was found our users' positive reaction to the flexibility of collections in Haystack. This flexibility arose both in terms of being able to file multiple kinds of items into a collection as well as in terms of being able to file a single item in multiple collections as once. One user described this model as "infinitely useful…maps directly to how I think about organizing". Another user enjoyed the uniformity of being able to interact with all collections in the same fashion.

Furthermore, the support for multiple first class types of resources in the system was received well. In addition to e-mail messages, text notes, to-do items, and of course collections ranked among the most frequently used resource types. We anticipate that other types would have been more frequently used if we had provided more support for interfacing with existing systems.

Nevertheless, these two findings—usefulness of collections and multiple type support—point to the notion that there is nothing special about e-mail that makes it deserving of special treatment by the system. In fact, one could say the system was built around collections, which is by nature an aggregation mechanism for other types. We believe this is strong evidence towards repurposing being less of a proper usability paradigm and more of a symptom of how information environments have been implemented.

Some of our hypotheses found only weak support from our study results. For example, very few users found need to switch views. Also, the task integration into the environment was natural to some but not others. In general we found that these features are satisfying the needs of some of our users, although more extensive tests are needed to ferret out the specific problems at play here.

In particular, on the issue of task management, we found that the interface for telling the system when a task was being actively pursued was not intuitive enough. We believe that users are already familiar with telling their systems about what task is active because they are doing so in a limited sense when they switch between applications—a weak form of task-specific information centralization. Usability paradigms adopted by previous work, such as the Rooms project [101] and Plaisant et al.'s notion of role management [102], may be of help here.

Chapter 15 Application: bioinformatics

In this chapter we give one final application of the techniques described in this thesis to the burgeoning area of bioinformatics. This field is typified not only by the huge growth in instance data that has resulted from the mapping of the genomes of several organisms, but also by the expansion of the ways in which such data can be characterized, since new scientific discoveries will undoubtedly uncover new ways to understand various biochemical processes. We propose ways in which Haystack can allow bioinformaticians to overcome many of the problems plaguing current bioinformatics systems while showing how our support for managing information expertise and collaboration can help enable biologists to focus more on their research and not on integrating their back end systems. We also discuss how biological information hosted by relational databases or web servers can be brought into Haystack in ways that satisfy the needs of information technology personnel responsible for these databases, further demonstrating the feasibility of our methods.

15.1 The problem space

Life scientists face many challenges in their task of managing biological information. Looking at the bioinformatics workflow at a low level, we observe that data from experiments is entered into databases, where scripts written in languages such as Perl are employed to filter and analyze the data and to compare them with other known samples [75]. Results are then prepared as a combination of numerical data and prose for publication.

At a more conceptual level, the information that is used and generated by life scientists, including chemical pathways, annotated gene sequences, and protein structures, is highly connected in nature. For example, an enzyme that catalyzes some specific pathway has a specific, definite structure and genetic sequence that encodes how to construct it. Connections also exist between any given protein and other proteins that are similar to it, either in terms of functionality or composition.

Historically, these different forms of information (e.g., annotations, pathways, structures, sequences, etc.) have been stored in a series of incompatible databases using disparate identifier schemes and distinct data formats. As a result, life scientists have been pre-

vented from working with their information at the desired high level because these databases must be bridged, manipulated and normalized (usually by hand-written Perl scripts) to accomplish all but the simplest of tasks. While recent advances in parallel processing and database technology have sped up data analyses, techniques for organizing the data themselves remain primitive and are perhaps a hindrance to accomplishing the actual objectives of biologists and others who work with such information.

15.1.1 Scattered databases and identifier spaces

Perhaps the most superficially obvious problem in today's systems is the abundance of different naming schemes for biological concepts and data. Currently, identifiers are database-specific and must be qualified with the name of the database from which the identifier was assigned. The ability to use an identifier from a foreign database is dependent on there being a provision for using that database's identifier in the particular format or schema being used, and the manner for encoding such a foreign identifier differs from format to format. In addition to requiring special case code to be written for each database, the lack of a unified identifier scheme hinders the development of tools that can refer to multiple databases or that can adapt to use the naming conventions of new databases when they come online or are included in research.

A related problem is the challenge of retrieving biological information given an identifier. Each database exposes its data in a different way, using a collection of programmatic interfaces, Web forms or pages, and FTP directory structures. Because the access instructions for each database are written in human-readable form, writing resolution code that can deal with identifiers from multiple, potentially unknown databases is a challenge. Furthermore, when data is finally retrieved from a database, users must be cognizant of the format in which the data are recorded. Formats range from custom text structures to XML Schema-based XML formats, and entire books have been written on how to parse this Tower of Babel [62].

The point of having bioinformatics databases available in the first place is to enable sophisticated analyses of data spanning multiple databases, which results in new information in the form of annotations and collected result sets. The proliferation of infrastructure such as annotation systems, Grid/Web Services-based data processing services and

visualization tools on top of a series of incompatible standards is stymied by the repeated chore of special-casing for each database.

Finally, the format problem is an impediment to users' own management and organization needs. When users wish to inspect data sets on their local machines, they must remember where the information was derived from, e.g., a database table, a file on the file system, on a web server, or perhaps on an FTP server. Also, describing the ontological connections that exist between data, such as catalytic pathways or regulatory dependencies, or referring to such concepts in literature or correspondence requires that the concepts being connected together be referable in the data representation.

15.1.2 Organizational tools

Even without a unified approach to biological information management, one may still note that a huge wealth of information is available publicly. Despite data being universally accessible for any given research area, what distinguishes a novice from an expert is the knowledge of what information is *relevant*. At the moment, such relevance information is for the most part trapped within research papers—one reason why graduate students spend so much time reading them. Knowledge representation technology has enabled some of that expertise to be represented in a machine-readable form, such as with the Gene Ontology [63]. However, these representations suffer from the "hindrance" of having to map identifiers from system to system.

On the other hand, if creating conceptual maps of research areas were as easy as making homepages of bookmark collections, simply from the result of the barriers to entry being lowered we would be likely to see more expertise modeled in this way. When concepts are mapped in terms of ontologies, they are then searchable by the machine, potentially expanding the power of query engines. One can imagine a user aggregating several kinds of objects together—gene sequences, literature references, web pages, even the names of noted experts in the field—into custom, user-defined collections. Such collections represent valuable interpretations of relevance and could then be shared, sent between colleagues, and searched.

15.1.3 Visualization

The space of information available to life scientists today is huge, and locating relevant information can be tricky if the scientist cannot express his or her information need in

terms of a query phrased with respect to standard vocabularies. A good example is literature research: At one point or another all scientists engage in literature research to learn about an unfamiliar field, discover related work for a paper being put together, etc. However, most tools for searching the literature are keyword-based. Retrieval is fundamentally a two-phase process: first the user identifies the characteristics that are known about the items being sought (searching), then he or she scans through the result set to locate the particular items desired (browsing) [18]. If a scientist is looking for papers in a specific topic space or that reference a peculiar molecule with nonstandard nomenclature, i.e., the characteristics that are known are not easily keyword-expressible or no characteristics are known (pure browsing), then retrieval may be severely hampered and time consuming.

Instead one may argue that scientists should be able to browse the literature space much like customers can browse books in a library or bookstore. The set of all research papers forms a significantly connected graph, with connections ranging from inter-paper references to ontological connections that exist between the biological concepts discussed within the papers. Such browsing is made possible by allowing users to take advantage of connective metadata, and in related domains browsing has been shown to facilitate discovery of items that can be hard to characterize in words [77].

Furthermore, viewing literature or other data on the same topic but from different databases is inhibited by the separation of this data onto separate web sites. Users must understand the navigation structure of each site and in particular how to search for the topic of interest in each site. One result is that biologists must frequently open dozens of web browsers manually in order to learn about the multiple facets of any species. A worse problem though is that the different steps taken to get to information about any particular species for each site cannot be automated at the moment. Both information space visualization and the aggregation of information for the purposes of visualization will need to be improved in order to improve accessibility to the wealth of information locked in databases.

15.2 Applying our data model

Many of the data modeling themes that arose in our discussion of the bioinformatics problem space are common to the problems of personal information management and

other domains discussed in this thesis. We tackle below the practical issues that accompany the integration of biological information into Haystack and in particular discuss specific issues regarding managing information in this area.

15.2.1 Naming

In Sections 3.2 and 3.3 we emphasized the importance of universal names and the role they play in facilitating an integrated information space. Naturally, we have adopted the use of URIs for naming resources in this space. However, in the case of bioinformatics, there are a few issues that must be taken into consideration because of the nature of the life sciences industry and the work styles of bioinformaticians. As discussed in Section 4.3, a special class of URNs called Life Science Identifiers (LSIDs) have been developed by the I3C [91] to address the needs of the life sciences community. As URNs, LSIDs are location-independent identifiers for information. Despite the name, the technology behind LSIDs is not specific to bioinformatics and may possibly be used for other domains in the future.

LSIDs have the following syntactic form:

```
urn:lsid:authority name:namespace id:object id[:revision id]
```

Specific details for the standard and its motivation are beyond the scope of this thesis but can be found in [91]. The main idea is that LSIDs use the authority name both as a persistent identifier of the originating party (important to the industry) and as a hint for later retrieval of associated data and metadata (discussed in the next section).

15.2.2 Data retrieval

The standard was designed to permit retrieval of both data and metadata associated with an LSID. Herein, the distinction between data and metadata is noticeably artificial but important because of the way in which LSIDs will be phased into use. By data, we are referring to files containing information such as genetic sequence data or protein structure data encoded in some standard (usually textual) format. Metadata, by the way the term is used in this thesis, encompasses the definition of data just uttered, but at the moment is used more to connect LSIDs together by means of RDF predicates. In the future, we envision that all data will be modeled as RDF, and the distinction will no longer be useful.

In terms of data (as opposed to metadata) retrieval, the LSID resolution mechanism supports a number of practical features, including the choice of transfer protocols such as FTP, HTTP, and SOAP and caching. The base level LSID ontology also supports the notion of resources referring to data. The predicate `<urn:lsid:i3c.org:predicates:hasContent>` is used to connect conceptual resources, such as "human hemoglobin", to individual renditions of the data associated with a concept into text formats, such as "human hemoglobin protein structure data in PDB format".

15.2.3 Schema management

Schemas and other forms of information expertise are vital to the proper interpretation of biological metadata and can be similarly retrieved via the LSID resolution mechanism. As mentioned in Section 3.4.5, the `rdfs:isDefinedBy` predicate connects classes and properties to the ontology resource that defines them. When an LSID-named resource is retrieved in Haystack and is of a type that is named by an LSID but is not recognized by the system, Haystack can follow the `rdfs:isDefinedBy` predicate to determine the ontology resource name and download the ontology definition and other related presentation knowledge and information expertise.

15.3 User interface

Haystack begins to address the user interface needs associated with the presentation of biological originating from disparate sources using the techniques outlined in Chapter 6, Section 9.3, and Chapter 8.

15.3.1 Aggregation with aspects

Different facets of the information pertaining to any particular resource, such as related literature, sequence data, or homology, must be viewed in separate web browsers because the data is contained in different databases and hosted by separate web sites. Once the data corresponding to different facets is incorporated into the user's repository, they can be viewed together. In particular, each facet can have a corresponding aspect for use in visualizing that facet. The process of turning information displays originally encompassed by single web pages into visualizations produced by aspects results in a display that looks not unlike a portal (see Figure 43).

Figure 43: Unified view of RNA sequence

However, there are several advantages to our approach over a portal, which merely integrates several web pages together into the same page. First, aspects can be defined that cut across the different databases and display information resulting from the overlap of multiple databases. Second, aspects allow manipulation of the data, so users can both inspect, modify, and even create notations on the data. Finally, aspects can be created by the user to incorporate new elements of information expertise into the display—a level of customizability not often exposed by portals.

15.3.2 Graph browsing

In a connected space such as bioinformatics, it is often important to be able to visualize the connections that exist between different resources. Many tools exist today for enabling this process, but they often work only over specific databases and on a predefined

set of types [93]. In contrast, the Haystack graph browser, described in Section 9.3, can visualize arbitrary relationships between an unbounded set of different types of resources. Refer to Figure 44.

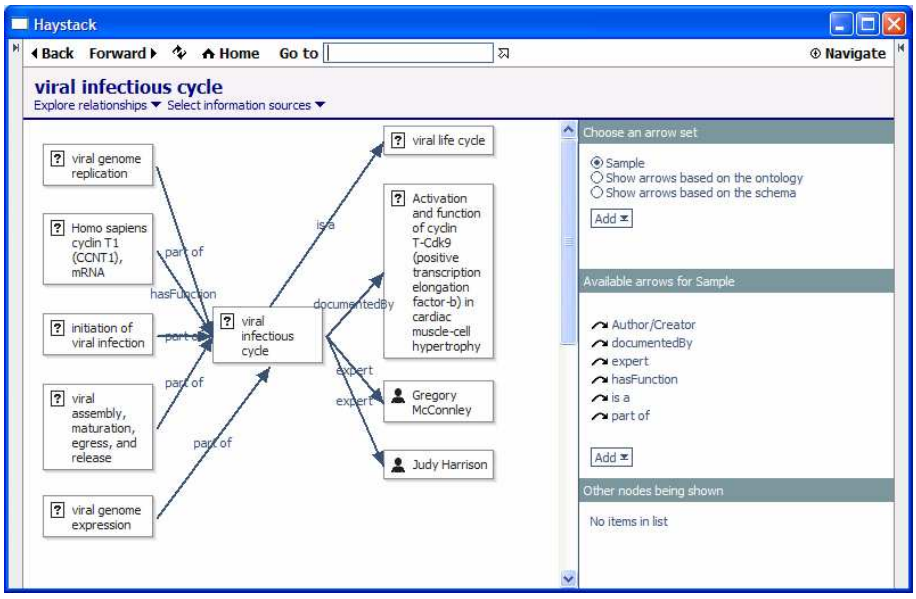

Figure 44: Example of browsing a graph of biological information

15.3.3 Making collections

We mentioned earlier that client-side aggregation of information has been inhibited by differences in formats and protocols exposed by the various databases used by biologists. With biological information expressed as RDF, users can not only benefit in terms of unified visualization but also in terms of being able to create new organizational structures on top of this information. We believe that Haystack's support for collection management will allow users to aggregate relevant pieces of biological information. This support was discussed in detail in Chapter 8.

15.4 Discussion

Our application of Haystack's tools to the bioinformatics space has begun to demonstrate several ways in which our approach to information management can be practically applied to many domains. First, despite Haystack's somewhat unique reliance on RDF as a core data model, information that exists today in more or less structured forms than RDF, as is the case in the life sciences community, can be incorporated incrementally into the Haystack/Semantic Web world. LSIDs are one technique for naming enti-

ties and providing uniform data and metadata retrieval support in a world dominated by a myriad of different kinds of data encoded in various text formats. The migration to RDF-like formats can be achieved in steps, either on the server side, where data is gradually converted into metadata, or on the client side, which uses transformations such as those encoded in XSLTs to produce RDF.

Similarly, information does not need to be federated on servers in order for users to benefit from unified visualizations of the data relevant to them. Because RDF graphs can be automatically merged by RDF stores, information about resources of interested, once encoded in RDF (via one of the mechanisms described above), can be brought into the user's repository and treated uniformly in the user interface.

Furthermore, client-side manipulation of information, in addition to permitting various forms of visualization, also enables users to add their own personalized notations to the data. By using tools such as collections, categorization schemes, group aspects, and the graph browser, we are enabling biologists to create metadata that would have previously required special purpose workgroup or public databases to be established to hold such metadata. Biologists are able to retain more control over this information because it resides on their own machines, and options for sharing this information with others are expanded.

The existing work in this area—such as that in high performance computational services such as homology search engines and simulators—is easily incorporated into the Haystack picture. Such services, when packaged as Web Services with WSDL descriptions [90], can be treated like other services and agents in the system (see Section 4.2), and specific pieces of functionality exposed by these services can be abstracted into operations. Furthermore, Web Services that either require structured input or produced structured output (or both) benefit from the user interface support provided by Haystack for inputting the needed parameter data to the service or for rendering the produced output data to the user, respectively. In fact, services are starting to support more sophisticated forms of interaction, such as parameters specified in RDF graphs, leading to the name Semantic Web Services. Similarly, services that run over computation clusters, called Semantic Grid Services [78], can be interfaced with Haystack in the same fashion. In general, functionality exposed by services that process data in some specific fashion, e.g.,

parallel processing, higher order inference, etc., can be incorporated into Haystack; conversely, high intensity data processing requirements can be offloaded onto services without changing the user interface exposed to the user.

Finally, LSID authorities are simple to construct, and support for publishing data to LSID servers can be built into Haystack and can be achieved either through Web Services interfaces to LSID authorities or even by hosting an LSID authority from within a user's own Haystack.

Chapter 16 Concluding remarks and future work

Throughout the thesis we have described how to apply the RDF data model to the problem of giving users a flexible enough information environment for effectively managing their information. We have also shown two specific applications of our approach in order to characterize the breadth of our methods. In this chapter we summarize the contributions of Haystack. We then turn to some of the fundamental limitations of our data model, including the inability for the user to extend our data model in certain specific ways. We talk about important issues that need to be addressed in any practical implementation of our data model but were not implemented in Haystack, such as security and privacy. We also point out several future directions for building on Haystack, including automation and collaboration, that should benefit significantly from the expressiveness and availability of user information encoded in Haystack's knowledge representation. Finally, we describe the potential consequences of widespread adoption of the ideas described in this thesis.

16.1 Summary

In this thesis we posited that a semistructured, semantic network-based data model provides many benefits to users in managing their personal information. The three characteristics of extensible schemas, fine level of granularity, and shared object space give users an abstraction in which to store metadata about resources from multiple domains at varying levels of precision—ranging from "sloppy", *ad hoc* connections to formal, schematic descriptions. The data layer of Haystack makes this abstraction practical, placing a user's information into a centralized RDF store and assigning agents the job of shuttling information in and out of the RDF store and the RDF representation. The Adenine and Ozone languages are embedded in the RDF model and allow developers to naturally express both imperative and declarative programs that use RDF data.

Exposing Haystack to the user is an interface built around four core ideas. First, resources need to be exposed to the user in a first class fashion in order to support direct manipulation and metadata input. Second, when manipulating a resource, multiple presentation styles are potentially important, and the system must not glue data models to specific styles. Third, an important method for characterizing resources is in terms of aspects, an extension of RDF properties intended to include human-centric notions of

presentational and heuristic knowledge. Finally, commands can be encapsulated into operations, which are simply Adenine methods with additional metadata. A common theme in the user interface is that because of the expressiveness of RDF, data model-level concepts, such as classes, properties, and methods, can be directly transformed into user-level concepts with the addition of metadata.

With these basic user interface principles in mind, we adapt modern information collection, organization, and navigation paradigms, including folders, annotations, forms, diagrams, messaging, and search, to our data model. Collections are like folders in that they are used to aggregate resources together, but the limitations that prevent resources from being in multiple folders at once and that prevent resources of different types from coexisting in the same folder have been removed. We demonstrated that this flexibility addresses a huge set of usability problems and enables a large class of metadata to be expressed easily. Annotations, forms, and diagrams are familiar paradigms and enable more general forms of metadata to be specified by users, demonstrating that metadata input is not simply limited to the notion of surrounding pieces of text with XML tags, a process unlikely to be welcomed by users.

Abstractly, today's desktop applications are aggregations of the very elements discussed above—operations, views, aspects, etc. One further notion remained to be untangled: context. We described a context model that treats to-do items, projects, and other kinds of tasks as indicators of context. Applications artificially restrict the information and tools available for use on any particular resource within windows with predefined menus and toolbars, while in Haystack, all tools and information are accessible regardless of the "window" that is open. However, this approach can lead to information overload; context is used throughout the system to reduce clutter on the screen and to present users with relevant information and tools. The notion of application was then broken down into activities and contexts that group together the functionality possessed by an application.

Finally, we showed that Haystack's framework can support several higher-level purposes. The elements of an "application" can be manipulated directly by users with ontological proxies, allowing new levels of customizability and the ability to create new "applications". Also, we discussed how Haystack addresses many of the fundamental prob-

lems that exist in personal information managers and the field bioinformatics; common themes that arose are the artificial separation of information into different formats, networks, databases, and applications and the inability to create arbitrary levels of structure on top of existing information.

16.2 Limitations of the RDF data model

One of the great properties of semantic networks is the ability of separate fragments to be merged into a single unified semantic network. This feature relies on nodes referring to the same concept in different fragments to share names. However, names in RDF are opaque and are not supposed to contain semantics, bringing about another class of problems: Whenever a name must be coined to refer to some derived property of a set of resources, a need arises to make sure that this name will be the same name coined by others to refer to the same derived property. The common solution—using a formula or a query to refer to the derived property, perhaps itself written in RDF—requires inferential support in the store. For systems such as ours without such inferential capability, intension needs to be embedded within a name in order to maintain consistency in all cases; we discuss this below.

Furthermore, the RDF data model was designed with two key extensibility characteristics in mind. First, resources are given opaque names that bear contractual meaning across distributed systems, and the set of names is unbounded. Second, the set of properties with which one describes a resource is unbounded. While these two characteristics permit a large class of metadata to be expressed, one particular form of extensibility that is not well supported—the ability to extend binary predicates into ones of higher arity—which is discussed in this section.

16.2.1 Intensional naming

To concretize the somewhat abstract nature of the problem of intensional naming in a system without inference, we pose the following example. Suppose an ontology is designed to model families in terms of person resources and sibling and spouse predicates. Now consider how one would go about making an assertion about the list of siblings of a particular person, e.g., Bob. To make an assertion in RDF about something, that thing must have a name and be considered a resource in the graph. One could create a re-

source whose job it was to represent the list of Bob's siblings and assign some random name. Then, this resource could be used as the subject of an assertion.

There are several problems to this approach. What happens if someone else makes assertions about the list of Bob's siblings? Every asserter would come up with a different name for the sibling list. In the spirit of maintaining a shared object space and consistent names, this explosion of possible names is not ideal; anyone looking for assertions concerning the list of Bob's siblings would have to search through a potentially unbounded set of names.

Ideally, every would be asserter would come up with the same name every time an assertion needed to be made about the list of Bob's siblings. This approach is the basis behind naming schemes that incorporate some identifying string or primary key into the URI of the resource. For example, e-mail address endpoints, discussed in Section 12.4, are uniquely named by an e-mail address, so a URI can be created by prepending "mailto:" to the address. Similarly, our approach to naming statements by their MD5 hash, as brought up in Section 4.1.1, uses this same underlying idea. However, two problems arise from applying this approach in general. First, it requires that for every kind of reification, such as "sibling list", a convention be adopted that allows every possible asserter to construct the correct URI, which for infrequent cases is unrealistic. The other problem is that the identifying property of a reification may not be a single property but rather some general logical expression. Not only would it be difficult to embed, in effect, a canonical RDF expression into a URI for the purposes of uniqueness, but it would also be difficult to check for equality in the case when canonicalization is not possible. It is hardly desirable for name determination to be potentially undecidable!

Another way to look at the difficulty posed above is to say that we are embedding equivalence class detection into the naming convention. Instead, perhaps we must adopt a looser notion of equivalence that permits resources to have multiple names. In this case, we can allow the semantics that would otherwise have to be embedded within the name to be expressed as RDF assertions. In other words, the name itself does not identify the extension of the resource but is rather an intensional placeholder for an unnamed resource (by unnamed we mean that there is no standard name), which is the

approach taken by systems such as DAML+OIL [100]. The problem with this approach is that graph queries—indicated in Section 4.1.3 to be a prevalent and efficient mechanism for extracting information from an RDF graph—can no longer depend on name equality and must invoke inference in order to ensure equality between intensions. It is our experience that introducing inference into such a low level of the system would be detrimental to overall system performance in an actual implementation.

Perhaps a useful compromise would be to embed support for resource name equivalence into the data layer. It was mentioned in Section 4.1.3 that Cholesterol maintains a virtual node for each resource referenced by the database. If multiple names were permitted for every node and a primitive operation on the database were supported whereby clients could register URIs as referring to the same resource, there would be little loss in efficiency to the system. The details of what resources would be considered equivalent would have to be managed by some sort of per-user equivalence class table, much like a belief network as discussed in Section 4.6. In addition to addressing the problem of intensional naming, this approach also covers any case when a single resource may have been given multiple names by multiple parties.

16.2.2 Extending binary predicates into *n*-ary predicates

Because of its triple structure, RDF is fundamentally built upon the notion that relationships can be broken down into a series of binary predicates. Support for predicates of higher arity is accomplished by introducing an intermediate node—a reification of the relationship—and attaching the different parameters of the predicate to this node. Unlike the process of asserting a new property to a resource, the process of increasing the arity of a relationship from binary to ternary or higher requires an intermediate reification step [29]. (Once reified, a ternary relationship can easily be turned into a quaternary relationship or higher.) This is a fundamental limitation of the RDF data model if we assume that we cannot upgrade the representation in the database every time such a change is made.

We do not claim to know the solution to this problem, but we discuss several possible solutions here. One possibility would be to mandate the creation of a new predicate and a new type for the intermediate node, together technically bearing the same semantics albeit with different representational "syntax", every time a binary predicate needed to

be extended. For example, if we wish to qualify the fact that John likes Mary with a certainty probability:

```
<john> <likes> <mary>
```

would become

```
<john> <likes-version2> <intermediate-node>

<intermediate-node> rdf:type <Liking>

<intermediate-node> <personBeingLiked> <mary>

<intermediate-node> <withCertainty> “60%”
```

This solution carries with it the burden of requiring systems to either cope with the possibility of predicates being expressed in one of these two fashions or to force an upgrade of the entire data representation. An alternative would be to create a reification of the binary predicate assertion and then annotate that:

```
<relationship-reification> rdf:type rdf:Statement

<relationship-reification> rdf:subject <john>

<relationship-reification> rdf:predicate <likes>

<relationship-reification> rdf:object <mary>

<relationship-reification> <withCertainty> “60%”
```

This solution has the benefit of not requiring systems that do not understand the extra parameter to change, but extension of the arity of a relationship comes in a somewhat unnatural way.

Theoretical investigations into the nature of representation and conversion between ontologies, including those that differ only in superficial syntactic form, can be found in previous work [103].

16.3 Security and privacy

Two important features of an information management environment—security and privacy—were not discussed in this thesis. Clearly, any implementation to be deployed on a large scale will need proper security and privacy controls to prevent unauthorized users from accessing personal information and to filter private information from being sent

out to others. In this section we survey some thoughts on how these problems may be addressed and point out related solutions in previous work. A related issue, belief, is important for ensuring that information that enters the system is not able to distort the accuracy of the user's representation and was discussed in Section 4.6.

16.3.1 Access control lists on RDF

One natural mechanism for controlling security privileges is the access control list (ACL), which has been used in a variety of different systems ranging from file systems to web servers. The approach typically taken to applying the ACL idea is to control access at the resource level. Systems such as Lotus Notes are based on this idea. However, this methodology is problematic in our environment because information at different levels of security access concerning the same resource exists within the same repository, relatively undistinguished. Another approach is to control access at the database level of granularity, which has been adopted by systems such as T Spaces [87]. The problem with administering permissions at the database level is that it forces users to separate their data out into different databases based on access control, which is antithetical to our principle of maintaining a shared object space.

An ideal level of security granularity would be to allow ACLs at the statement level. This approach is explored in the work of Baeza-Yates et al. [86], where access control is administered at the attribute level. They observed that it is attributes and not objects whose access should be controlled. In order to expose this power to users, we propose that views be supported for visualizing statements. In this way, users get a tangible representation of statement-level security privilege administration. For example, consider a person resource with name, age, and social security number attributes. Haystack could display the person resource in a special view in which all of the metadata concerning that person was spelled out at a statement by statement level. A user could right-click on the user interface element labeled "The social security number of John Doe is 123-45-678" and select access control on the social security attribute. This approach requires presentation information to available for most predicates in the system, lest the user see overly general presentations such as "The property named 'hasDaughter' has value 'Jane Doe'". Other challenges exist in the problem of exposing access controls for large sets of attributes to the user; having the user set the level of security for each property manually and individually is likely not to be practical.

16.3.2 Controlling access to computation

One final concern regards the fact that any query request made against a Haystack embodies a certain amount of inherent computation. This observation leads us to explore the possibility that the complexity of a query may be used to either gather information about a Haystack without the user's knowledge or to launch a denial of service attack on the system. The former can result from the addition of graph fragments to a query whose only purpose is to test for the existence of certain information. The latter can result from queries being posed simply to burn computation cycles on a target machine. Further study will be needed to determine the ideal balance between allowing relatively open access to Haystacks and safeguarding users' computing resources.

16.4 Automation

In addition to helping users keep track of information, metadata recorded in computer systems has a further added benefit—computers can use metadata to automate mundane tasks [85] [4]. The agent infrastructure discussed in Section 4.2 can form the basis for families of agents that use metadata aggregated within a user's repository to coordinate tasks with others, optimize the user's organizational scheme, and find related information. Below we touch upon some of these possibilities and discuss how they may be applied to Haystack.

We began to discuss the topic of machine learning in Section 8.3.3 by discussing Mark Rosen's work on automated text classification support in Haystack [60]. Areas ripe for the application of machine learning include any situation in which users are making classifications, such as context switching. In Section 15.4 we mentioned that some users were annoyed with having to tell the machine when a task was started and stopped. Given user behavior data collected over time regarding tasks, machine learning could be applied to allow the system to more intelligently determine when tasks are active. Similarly, the system may be able to learn about the connection between frequently-used operations and views and the kinds of resources that are used in conjunction with these operations and views and provide intelligent recommendations that help to shorten the list from which the user has to choose in these situations.

Using semantics to improve search is an area at the intersection between information retrieval and knowledge representation [7] that has begun to gain traction lately. Col-

laborative filtering, discussed below in Section 16.5.1, is another example of this same principle. Tools such as TAP enable a concept known as semantic search, whereby the system seeks out additional information that may be related to the resource being viewed based on semantic connections [107]. TAP begins to address the problem of locating information scattered across multiple Internet servers when no common ontologies have been established. Similarly, the Edutella project has investigated the problem of locating related metadata across a peer-to-peer network [105].

Finally, one of the basic motivations for systems such as the Semantic Web is the possibility of agent-based negotiation. In an article on the Semantic Web by Berners-Lee et al., an example scenario of two siblings trying to coordinate the ideal time and place for taking their mother to the doctor is depicted, demonstrating the usefulness of being able to express semantically-rich constraints to agents that take advantage of these constraints to perform mundane tasks [4]. While Haystack does not incorporate an inference engine—the mechanism by which coordination was achieved in the above scenario—the user interface components in our system can be used to help capture and store user preferences in order to enable agent-based negotiation.

16.5 Collaboration

We explore one last area of potential future work: collaboration. In Chapter 12 we began our discussion of support for collaboration with the topic of messaging—a fundamental building block. However, there is significant research into computer-supported cooperative work that has elucidated higher level usability paradigms for taking advantage of underlying communication infrastructure. In this section we give an overview of these ideas and how they might be applied to Haystack.

16.5.1 Collaborative filtering

Collaborative filtering is a specific example of the idea posed in Chapter 1 that a user's colleagues may benefit from the expertise gained by a user. In particular, collaborative filtering looks at the problem of recommending resources that were found to be interesting by people with similar interests to the user. Applications of this technology include GroupLens, a system that locates relevant Usenet articles to readers based on their shared interests with other readers [108]. Similarly, commercial systems such as TiVo and amazon.com have incorporated collaborative filtering techniques.

Haystack's contribution to this space may come in the form of user interface support. Opinions can be collected from users in a variety of ways in Haystack, but perhaps the most obvious is to use the categorization mechanism discussed in Section 8.3.2. Ratings are simply a specific example of a categorization scheme, whose categories include, say, "thumbs up", "thumbs down", and "neutral". There are also interesting research problems in the area of presenting recommendations. The developers of the GroupLens project [109] and our support for automated classification (cf. lightbulb indicator discussed in Section 8.3.3) provide insights into this problem.

16.5.2 Shared information repositories

Another classical area of computer-based collaboration research involves the notion of repositories of shared information. Current work in this area comes from domains ranging from the groupware area (e.g., Lotus Notes, Groove, and [110]) to the database community [111]. Some solutions, such as Lotus Notes, have exposed a flexible data model for collaboration, while others, such as Answer Garden [112], gave users a higher level interface for managing shared expertise. We feel it is possible to provide both levels of user interface support in Haystack through different views.

Interesting challenges exist at the plumbing layer in terms of distributing shared information. Lotus Notes, Groove, and Microsoft Exchange employ a replication model in which users cache and work with copies of remote databases on their local machines. Offline updates are then sent back to the server during synchronization. However, the corresponding problem in the RDF space is much more difficult due to the granularity and flexibility of the underlying representation (cf. work on DAML diff at http://www.daml.org/2001/04/iow/mit/). The solution we have outlined in this thesis is to encapsulate shared information in terms of messages (see Chapter 12), but other possibilities deserve future exploration.

16.5.3 Ontological conversion

We pointed out in Chapter 1 that a fundamental motivation behind the construction of Haystack was to allow users to customize their data model for their own benefit. When considering this fact in the context of collaboration, a problem that arises is trying to make sense of one user's data encoded with respect to that user's ontology in another user's system—ontological conversion. We have proposed simple mechanisms for al-

lowing developers to specify conversions from one ontology to another by means of Serine patterns, as discussed in Section 4.5. The support for views in Haystack, as discussed in Section 6.3, can also help to alleviate the problems of ontological conversion by letting developers create view parts for corresponding types in different ontologies that expose similar user interfaces.

However, there are other approaches to this problem that have been explored by previous work. In the knowledge representation domain, work has been done to examine the problem of mapping superficially different representations onto each other [103]. In the database community, this problem is known as federation [113]. Approaches providing frameworks for supporting point-to-point translators between different ontologies have also been explored in past research [104]. These solutions could be introduced into Haystack either at the data abstraction layer by creating virtualized RDF stores or at the agent layer by creating translating agents like Serine.

One might also consider the problem of trying to minimize the need for ontological conversion in the first place. Tools such as Guha et al.'s TAP [107] can be used to resolve human-readable names of concepts into their URIs, helping to stem the growth of multiple names for identical concepts.

16.6 The Haystack revolution

Widespread adoption of the core tenets of Haystack could have several consequences. First, the notions of application and desktop operation system would fade away. To imagine what would replace it, consider the advent of the World Wide Web. The notion of custom document networks began to disappear as the totality of documents belonging to a greater network dominated over the idea of closed environments. Similarly, personal computers could become personalized access points to a greater Semantic Web. Second, more information could be stored electronically. A fundamental reason why so much information remains in paper form is that the structure imposed by current software is too limiting or requires too much cognitive effort to force freeform, looseleaf ideas into. Finally, information retrieval would become more "natural". Many of the problems encountered by information retrieval today are caused by a lack of metadata, and the field is characterized by an abundance of techniques for overcoming this dearth. These problems could be overcome if users' software not only produced metadata but

were also capable of utilizing it. The tools for searching and browsing information corpora would morph into a continuous spectrum, and metadata from providers such as Google, Yahoo!, and the Open Directory could be combined and consumed uniformly.

References

[1] Abrams, D., Baecker, R., and Chignell, M. Information Archiving with Bookmarks: Personal Web Space Construction and Organization. Proceedings of CHI 1998.

[2] Adar, E., Karger, D., and Stein, L. Haystack: Per-User Information Environments. Proceedings of CIKM 1999.

[3] Berners-Lee, T. Primer: Getting into RDF & Semantic Web using N3. http://www.w3.org/2000/10/swap/Primer.html.

[4] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. Scientific American, May 2001.

[5] Box, D., Ehnebuske, D., Kavivaya, G., et al. (ed.). SOAP: Simple Object Access Protocol. http://msdn.microsoft.com/library/en-us/dnsoapsp/html/soapspec.asp.

[6] Brachman, R. J. What's in a Concept: Structural Foundations for Semantic Networks. International Journal of Man-Machine Studies 9, 1977, pp. 127-152.

[7] Brachman, R. and McGuinness, D. Knowledge Representation, Connectionism, and Conceptual Retrieval. Proceedings of SIGIR 1988.

[8] Cutting, D., Karger, D., Pedersen, J., and Tukey, J. Scatter/gather: A cluster-based approach to browsing large document collections. Proceedings of SIGIR 1992.

[9] Dourish, P., Edwards, W.K., et al. Extending Document Management Systems with User-Specific Active Properties. ACM Transactions on Information Systems 18 (2), April 2000, pp. 140–170.

[10] Dublin Core Metadata Initiative. http://dublincore.org/.

[11] Eriksson, H., Fergerson, R., Shahar, Y., and Musen, M. Automatic Generation of Ontology Editors. In Proceedings of the 12th Banff Knowledge Acquisition Workshop 1999.

[12] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. (ed.). HTTP Extensions for Distributed Authoring – WEBDAV. http://asg.web.cmu.edu/rfc/rfc2518.html.

[13] Hearst, M. Next Generation Web Search: Setting Our Sites. IEEE Data Engineering Bulletin, Special issue on Next Generation Web Search, Luis Gravano (ed.), September 2000.

[14] Huynh, D., Karger, D., and Quan, D. Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. Proceedings of Semantic Web Workshop, WWW2002. http://haystack.lcs.mit.edu/papers/sww02.pdf.

[15] Karger, D., Katz, B., Lin, J., and Quan, D. Sticky Notes for the Semantic Web. Proceedings of IUI 2003.

[16] Katz, B. Using English for indexing and retrieving. Proceedings of RIAO 1988.

[17] Katz, B., Lin, J, and Quan, D. Natural Language Annotations for the Semantic Web. ODBASE 2002.

[18] Lansdale, M. The Psychology of Personal Information Management. Applied Ergonomics 19 (1), 1988, pp. 55–66.

[19] Malone, T. How Do People Organize Their Desks? Implications for the Design of Office Information Systems. ACM Transactions on Office Information Systems 1(1), 1983, pp. 99–112.

[20] Malone, T., Lai, K., and Fry, C. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. ACM Transactions on Information Systems 13 (2), April 1995, pp. 177-205.

[21] Mander, R., Salomon, G., and Wong, Y. A 'Pile' Metaphor for Supporting Casual Organization of Information. Proceedings of CHI 1992.

[22] Marshall, C. Annotation: From Paper Books to the Digital Library. Proceedings of DL 1997.

[23] Marshall, C. Toward an Ecology of Hypertext Annotation. Proceedings of Hypertext 1998.

[24] McKay, S., York, W., and McMahon, M. A Presentation Manager Based on Application Semantics. Proceedings of UIST 1989.

[25] O'Hara, K., and Sellen, A. A Comparison of Reading Paper and On-Line Documents. Proceedings of CHI 1997.

[26] Pietriga, E. IsaViz. http://www.w3.org/2001/11/IsaViz/.

[27] Quillian, M. Semantic memory. Semantic Information Processing, M. Minsky (ed.), pp. 27-70. Cambridge, MA: MIT Press, 1968.

[28] RDF Model Theory. http://www.w3.org/TR/rdf-mt/.

[29] Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[30] Resource Description Framework (RDF) Schema Specification. http://www.w3.org/TR/1998/WD-rdf-schema/.

[31] Szolovits, P., L. Hawkinson, et al. An Overview of OWL, a Language for Knowledge Representation. MIT Laboratory for Computer Science TM86, 1977. http://medg.lcs.mit.edu/people/psz/OWL_overview1.html.

[32] Whittaker, S. and Sidner, C. Email Overload: Exploring Personal Information Management of Email. Proceedings of CHI 1996.

[33] RDF Site Summary (RSS) 1.0. http://www.purl.org/rss/1.0/.

[34] Huynh, D., Karger, D., Quan, D., and Sinha, V. Haystack: A Platform for Creating, Organizing, and Visualizing Semistructured Information. Proceedings of IUI 2003.

[35] Quan, D., Karger, D., and Huynh, D. RDF Authoring Environments for End Users. Proceedings of Semantic Web Foundations and Application Technologies 2003.

[36] Howe, D. The Free On-line Dictionary of Computing.

[37] Quan, D., Huynh, D., Karger, D., and Miller, R. User Interface Continuations. To appear in Proceedings of UIST 2003.

[38] Quan, D., Bakshi, K., and Karger, D. A Unified Messaging Abstraction for the Semantic Web. Proceedings of WWW2003.

[39] Quan, D., Bakshi, K, Huynh, D., and Karger, D. User Interfaces for Supporting Multiple Categorization. To appear in INTERACT 2003.

[40] Lieberman, H., Fry, C., and Weitzman, L. Exploring the Web with Reconnaissance Agents. Communications of the ACM 44 (8), August 2001.

[41] de Rosis, F., Pizzutilo, S., and De Carolis, B. A Tool to Support Specification and Evaluation of Context—ustomized Interfaces. SIGCHI Bulletin 28 (3), July 1996.

[42] Mueller, F. and Lockerd, A. Cheese: Tracking Mouse Movement Activity on Websites, a Tool for User Modeling. Proceedings of CHI2001.

[43] Kang, H., and Shneiderman, B. MediaFinder: An Interface for Dynamic Personal Media Management with Semantic Regions. Proceedings of CHI2003.

[44] Rhodes, B. The Wearable Remembrance Agent: A System for Augmented Memory. Personal Technologies Journal Special Issue on Wearable Computing 1, 1997, pp. 218-224.

[45] Maglio, P., Barrett, R., Campbell, C., and Selker, T. SUITOR: An Attentive Information System. Proceedings of IUI 2000.

[46] Gifford, D., Jouvelot, P., Sheldon, M., and O'Toole, J. Semantic File Systems. Proceedings of SOSP 1991.

[47] Kinder- und Hausmaerchen Gesammelt Durch Die Bruder Grimm. Mit Erinnerungen an die Bruder als Einleitung Herausgegeben von Herman Grimm. Mit 4 Uquarellen von V.P. Mohn. Berlin, Germany: Verlag von Wilhelm Hertz, 1899, pp. 149-155.

[48] Handschuh, S., Staab, S., and Maedche, A. CREAM—creating relational metadata with a component-based ontology-driven annotation framework. Proceedings of K-CAP 2001.

[49] Pietriga, E. IsaViz. http://www.w3.org/2001/11/IsaViz/.

[50] Eriksson, H., Fergerson, R., Shahar, Y., and Musen, M. Automatic Generation of Ontology Editors. Proceedings of the 12th Banff Knowledge Acquisition Workshop, 1999.

[51] Shneiderman, B. Direct manipulation for comprehensible, predictable and controllable user interfaces. Proceedings of IUI 1997.

[52] Bechhofer, S., Horrocks, I., Goble, C., Stevens, R. OilEd: a Reason-able Ontology Editor for the Semantic Web. Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence, Springer-Verlag LNAI 2174, pp. 396–408.

[53] Distributed Systems Technology Centre Resource Discovery Unit. Reggie—The Metadata Editor. http://metadata.net/dstc/.

[54] Open Directory Project. http://dmoz.org/.

[55] Davis, R., Shrobe, H., and Szolovits, P. What is a Knowledge Representation? AI Magazine 14 (1), 1993, pp. 17-33.

[56] Lin, J., Quan, D., Sinha, V., Bakshi, K., Huynh, D., Katz, B., and Karger, D. What makes a good answer? The role of context in question answering systems. To appear in INTERACT 2003.

[57] Smith, M., Cadiz, J., and Burkhalter, B. Conversation Trees and Threaded Chats. Proceedings of CSCW 2000.

[58] Nardi, B., Whittaker, S., and Bradner, E. Interaction and Outeraction: Instant Messaging in Action. Proceedings of CSCW 2000.

[59] Whittaker, S., Terveen, L., Hill, W., and Cherny, L. The Dynamics of Mass Interaction. Proceedings of CSCW 1998.

[60] Rosen, M. E-mail Classification in the Haystack Framework, MIT Master's Thesis, February 2003.

[61] Zhurakhinskaya, M. Belief Layer for Haystack, MIT Master's Thesis, May 2002.

[62] Tisdall, J. Beginning Perl for Bioinformatics. O'Reilly, 2001.

[63] The Gene Ontology Consortium. Gene Ontology: Tool for the Unification of Biology. Nature Genet 25, 2000, pp. 25-29.

[64] Myers, B., and Kosbie, D. Reusable Hierarchical Command Objects. Proceedings of CHI 1996.

[65] Terry, M. and Mynatt, E. Slide Views: Persistent, On-Demand Previews for Open-Ended Tasks. Proceedings of UIST 2002.

[66] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns. Boston: Addison Wesley, 1995.

[67] Ducheneaut, N. and Bellotti, V. E-mail as Habitat: An Exploration of Embedded Personal Information Management. Interactions, September-October 2001.

[68] Barreau, D. and Nardi, B. Finding and Reminding: File Organization from the Desktop. SIGCHI Bulletin 27 (3), July 1995.

[69] Fertig, S., Freeman, E., and Gelernter, D. "Finding and Reminding" Reconsidered. SIGCHI Bulletin 28 (1), January 1996.

[70] Rodden, K. and Wood, K. How Do People Manage Their Digital Photographs? Proceedings of CHI 2003.

[71] 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 CHI 2003.

[72] Nardi, B., Whittaker, S., Isaacs, E., Creech, M., Johnson, J., and Hainsworth, J. Integrating Communication and Information Through ContactMap. Communications of the ACM 45 (4).

[73] Freeman, E. and Gelernter, D. Lifestreams: A Storage Model for Personal Data. SIGMOD Record 25 (1).

[74] Kang, H. and Shneiderman, B. MediaFinder: An Interface for Dynamic Personal Media Management with Semantic Regions. Proceedings of CHI 2003.

[75] Buttler, D., Coleman, M., Critchlow, T., Fileto, R., Han, W., Pu, C., Rocco, D., and Xiong, L. Querying multiple bioinformatics information sources: can semantic web research help? ACM SIGMOD Record 31 (4), December 2002.

[76] Goble, C., Stevens, R., Ng, G., Bechhofer, S., Paton, N., Baker, P., Peim, M., and Brass, A. Transparent access to multiple bioinformatics information sources. IBM Systems Journal 40 (2), 2001.

[77] Yee, K., Swearingen, K., Li, K., and Hearst, M. Faceted Metadata for Image Search and Browsing. Proceedings of CHI 2003.

[78] Goble, C. and De Roure, D. The Grid: An Application of the Semantic Web. SIGMOD 31 (4), December 2002.

[79] Losee, R. Text Retrieval and Filtering: Analytic Models of Performance. Kluwer international series on information retrieval. Kluwer Academic Publishers, Hingham, MA, 1998.

[80] Kobayashi, M. and Takeda, K. Information Retrieval on the Web. ACM Computing Surveys 32 (2), June 2000.

[81] Bush, V. As We May Think. The Atlantic, July 1945.

[82] Shneiderman, B. Creating Creativity: User Interfaces for Supporting Innovation. ACM Transactions on Computer-Human Interaction 7 (1), March 2000, pp. 114-138.

[83] Kossmann, D. The State of the Art in Distributed Query Processing. ACM Computing Surveys 32 (4), December 2000, pp. 422-469.

[84] Kaplan, S., Kapor, M., Belove, E., Landsman, R., and Drake, T. Agenda: A Personal Information Manager. Communications of the ACM 33 (7), July 1990.

[85] Dertouzos, M. The Unfinished Revolution. New York, NY: HarperCollins, 2001.

[86] Baeza-Yates, R., Jones, T., and Rawlins, G. A New Data Model: Persistent Attribute-Centric Objects. http://dcc.uchile.cl/~rbaeza/ftp/paco.ps.gz.

[87] Wyckoff, P., McLaughry, S., Lehman, T., and Ford, D. T Spaces. IBM Systems Journal 37 (3).

[88] Freeman, E. and Gelernter, D. Lifestreams: A Storage Model for Personal Data. SIGMOD Record 25 (1), March 1996.

[89] Ullman, J. and Widom, J. A First Course in Database Systems. Upper Saddle River, New Jersey: Prentice-Hall, 1997.

[90] Web Services Description Language (WSDL) Version 1.2. http://www.w3.org/TR/2003/WD-wsdl12-20030303/.

[91] Werner, P., Liefield, T., Gilman, B., Bacon, S., and Apgar, J. URN Namespace for Life Science Identifiers. http://www.i3c.org/workgroups/technical_architecture/resources/lsid/docs/LSIDSyntax9-20-02.htm.

[92] Buttler, D., Coleman, M., Critchlow, T., Fileto, R., Han, W., Pu, C., Rocco, D., and Xiong, L. Querying multiple bioinformatics information sources: can semantic web research help? ACM SIGMOD Record 31 (4), December 2002.

[93] Goble, C., Stevens, R., Ng, G., Bechhofer, S., Paton, N., Baker, P., Peim, M., and Brass, A. Transparent access to multiple bioinformatics information sources. IBM Systems Journal 40 (2), 2001.

[94] Huynh, D. A User Interface Framework for Supporting Information Management Tasks in Haystack, MIT Master's Thesis, May 2003.

[95] JavaBeans Activation Framework. http://java.sun.com/products/javabeans/glasgow/jaf.html.

[96] Quan, D. Adenine Reference Manual. http://haystack.lcs.mit.edu/documentation/adenine-reference.pdf.

[97] HyperText Markup Language (HTML) Home Page. http://www.w3.org/MarkUp/.

[98] Cadiz, J., Gupta, A., and Grudin, J. Using Web annotations for asynchronous collaboration around documents. Proceedings of CSCW 2000.

[99] Kahan, J. and Koivunen, M. Annotea: an open RDF infrastructure for shared web annotations. Proceedings of WWW10.

[100] DARPA Agent Markup Language. http://www.daml.org/2001/03/daml+oil-index.

[101] Henderson Jr., D. and Card, S. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. ACM Transactions on Graphics 5 (3), July 1986.

[102] Plaisant, C. and Shneiderman, B. Organization overviews and role management: inspiration for future desktop environments. Proceedings of CHI 1995, Conference companion.

[103] Maluf, D. and Wiederhold, G. Abstraction of Representation for Interoperation. Proceedings of International Symposium on Methodologies for Intelligent Systems, 1997.

[104] Halevy, A., Ives, Z., Mork, P., and Tatarinov, I. Piazza: data management infrastructure for semantic web applications. Proceedings of WWW 2003.

[105] Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmer, M., and Risch, T. EDUTELLA: A P2P Networking Infrastructure Based on RDF. Proceedings of WWW 2002.

[106] Carr, L., Hall, W., Bechhofer, S., and Goble, C. Conceptual Linking: Ontology-based Open Hypermedia. Proceedings of WWW10.

[107] Guha, R., McCool, R., and Miller, E. Semantic Search. Proceedings of WWW 2003.

[108] Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., and Riedl, J. GroupLens: Applying Collaborative Filtering to Usenet News. Communications of the ACM 40 (3), March 1997.

[109] Herlocker, J., Konstan, J., and Riedl, J. Explaining Collaborative Filtering Recommendations. Proceedings of CSCW 2000.

[110] Natvig, M. and Ohren, O. Modeling Shared Information Spaces (SIS). Proceedings of GROUP 1999.

[111] Kim, W., Ballou, N., Garza, J., and Woelk, D. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. ACM Transactions on Information Systems 9 (1), January 1991, pp. 31-51.

[112] Ackerman, M. Augmenting organizational memory: a field study of answer garden. ACM Transactions on Information Systems 16 (3), July 1998.

[113] Sheth, A. and Larson, J. Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys 22 (3), September 1990.

[114] Steele, G. and Sussman, G. LAMBDA: The Ultimate Imperative, MIT Artificial Intelligence Laboratory Memo 353.