THE BUZZ: SUPPORTING EXTENSIVELY CUSTOMIZABLE INFORMATION AWARENESS APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

James R. Eagan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2008

THE BUZZ: SUPPORTING EXTENSIVELY CUSTOMIZABLE INFORMATION AWARENESS APPLICATIONS

Approved by:

John T. Stasko, Advisor
College of Computing
Georgia Institute of Technology

Beki Grinter
College of Computing
Georgia Institute of Technology

Mark Guzdial
College of Computing
Georgia Institute of Technology

Saul Greenberg
Department of Computer Science
University of Calgary

Keith Edwards
College of Computing
Georgia Institute of Technology

Date Approved: August 2008

For my father.

ACKNOWLEDGEMENTS

This work is the product of input and help from many sources. Many thanks to my committee for their time, effort, and insight in shaping this work, and especially to my advisor, John Stasko. To my peers who helped me shape and refine my ideas, James Hudson, Jochen Rick, Chris Plaue, Zach Pousman, Erika Shehan, Dugald Hutchings, Brian Dorn, Allison Tew, and the members of the Information Interfaces lab at Georgia Tech. Thanks to the U.S. National Science Foundation (IIS-0414667) for supporting much of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation

Two technology trends have emerged that are creating an exciting opportunity to create new tools to help people to manage the information they encounter on a routine basis. The first of these trends is increasingly ubiquitous access to information. Every day, people send more than 60 billion email messages [65], make over 1.6 million blog postings[1], and post over 32 thousand videos to YouTube [42]. Every year, more than four times the data in the U.S. Library of Congress is added to the world wide web, not including unindexed web pages and information stored in dynamic databases [65]. Not only is more information becoming available on the Internet, but also access to the Internet itself is becoming increasingly ubiquitous. Miniaturization is allowing network access to be embedded throughout the environment, with wireless Wi-Fi and 3G internet access becoming available in many municipalities.

At the same time, the costs of display technologies is falling. It is becoming increasingly feasible to embed displays throughout the environment [96, 85, 24, 73]. Visions of the future varying from those of Hollywood's Minority Report to Weiser's dream of ubiquitous computing [105] are coming closer to reality. Figure 1 shows former Vice President Al Gore's office, with three Apple Cinema displays on the desktop and a flat-panel display on the wall.

The combination of abundant and ubiquitous access to data with falling costs of display technologies is forming an exciting opportunity to create new and interesting information awareness applications. These applications can gather data from a variety of sources, compound them together in interesting ways, and create new interfaces on the data. For example, mashups focus on taking data from one or multiple sources and integrating them together with a different interface [62], as in combining housing data from Craig's List with a mapping interface from Google [88]. These

[1] As of November, 2006, Technorati.com tracks and average of 1.6 million posts to 60 million weblogs [5]

Figure 1: Al Gore working in his office, with three displays on the desktop and another on the wall.

mashups, however, are difficult to create. They usually involve reverse-engineering the underlying data format, the interface to a particular unpublished user interface, or both. For example, one of the first mashups was the aforementioned Craig's List and Google Maps mashup. At the time, Google did not make their maps API available. Creating that mashup involved scraping data from the Craig's List website and converting them into a format suitable to be injected into Google's private mapping interface.

Other applications focus on presenting the data via calm, ambient or peripheral interfaces [87]. For example, the InfoCanvas [96] conveys information through dynamic artwork on the wall. Various elements on the canvas update to encode some particular piece of information. To the uninitiated observer, the canvas looks like an electronic picture hanging on the wall, but to the owner, the various elements of the picture depict the forecasted temperature, the stock market, or the number of emails in one's inbox.

These interfaces offer the exciting potential to help people to better manage interruption [15, 50] and reduce feelings of information overload [30, 21]. They work by allowing the user to glean information opportunistically through lightweight interactions. Rather that explicitly searching for a particular piece of information, the user might happen to glance at the interface during a natural break and happen to see some relevant information. This approach may not help for demand-driven

information foraging sessions, but can transfer monitoring activities [56] from an active pull process to a passive push process.

These information awareness tools, however, are inherently personal in nature. The needs of one individual, however, are frequently different from the needs or interests of another. Information that is relevant to a manager, for example, is often different from the information relevant to front-line worker. Such systems, therefore, need to support some degree of customization in order to allow the user to tailor the content to his or her own personal interests.

Different software systems have taken various approaches to supporting users at customizing their software to varying degrees. One way to consider the degree of customization is in terms of the expressiveness of a particular customization with regard to the amount of effort that the user must expend. In these terms, a relatively minor customization might not require much of effort. As such, this customization might entail low effort for low expressiveness. A complex customization, in contrast, might involve a high degree of expressiveness with significant effort on the part of the user.

Although it may seem that there is a tradeoff between effort and expressiveness, such a tradeoff is not necessarily inherent. Consider, for example, defining a search query for housing listings. An interface that requires the user to compose a basic SQL statement and an interface that allows the user to compose that query via sliders before executing it might afford the same degree of expressiveness, but the dynamic query interface requires significantly less effort from the user. Furthermore, modifying the sliders to continuously execute the queries as the user adjusts them might increase the expressiveness of the interface by allowing the user to dynamic changes and thresholds in the data as the query parameters change [13, 12]. At the same time, this immediate feedback might also reduce the effort required by the user. Thus, depending on the task and the interface, effort and expressiveness are not necessarily a continuous, monotonically increasing function, although they may typically appear as such.

One goal of customizable software is often to support customization across this space: to support users of different skills at being able to perform different kinds of customization with varying

degrees of expressiveness in accordance with their capabilities and motivations. To support customization across this space, applications typically partition the interface to focus on different capabilities. These systems recognize that different users have different motivations and skills. Users are typically ascribed to three roles: end users, tinkerers, and developers [68, 66, 40]. End users are competent computer users but have little intrinsic interest in the underlying functionality of the software. Tinkerers tend to examine the inner workings of the software are are more experimental in their usage, whereas developers possess some degree of programming skills and can develop software or plugins.

Using these terms, Apple's Dashboard [11], for example, supports end users at subscribing to widgets, and developers, through a separate IDE, to create such widgets. In this way, the two interfaces afford vastly different kinds of customization, and users of one are not required to use the other. Similarly, HyperCard uses an interface through which the skill level of the user determines the interface options available. Thus, a beginning user is not exposed to a hypercard stack's scripting capabilities, while an intermediate user might be able to perform limited scripting capabilities, and and advanced user might have full reign over the interface.

These approaches adapt the interface to the capabilities of the user, either by making the user explicitly switch between interaction roles, as in Dashboard, or by adjusting the presentation of the interface itself, as in HyperCard. Like Dashboard, Yahoo Pipes [7], a system for creating mashups, also partitions the interface between a user and developer role, but further provides support within the interface for the user to transition between these interfaces. Nonetheless, these interfaces all take the approach of providing separate interaction interfaces for users in different roles. Through these different interfaces and different roles, these interfaces support the user at making customizations at different points along the expressiveness spectrum.

I have created a research prototype information awareness application, The Buzz [34], which aims to support users, from competent computer users through programmers, at performing customizations across the full expressiveness spectrum via a unified interface. Rather than partitioning the interface through distinct roles, as in Dashboard, or adapting the interface to the user's skill level, as in HyperCard, The Buzz aims to provide a fluid interface through which the user can adjust

his[2] effort to correspond to the complexity of the underlying task.

The system attempts to provide a nested interface where more basic customization operations are available alongside cues to expand the interface for more complex operations. Through disclosure dialogs and nested dialogs, the user can "dive down" into deeper interfaces to perform more complex interactions. If these cues align appropriately with the user's mental model of the task, it is reasonable to expect that the user should be able to identify the relevant interfaces [86]. Through a modular component architecture, these interfaces support the combination of reusable and configurable components to create powerful, flexible customizations that can handle a wide variety of situations. Furthermore, by providing this depth of interfaces, The Buzz aims to support users at performing basic customizations, such as subscribing to content and adjusting basic properties of their presentation; more advanced, rich customizations, whereby the user is controlling abstract behaviors of the system but without programming; to defining new behaviors such as by writing plugin code. It is through this software that we can examine the thesis statement of this work:

Thesis Statement

> It is possible for an information awareness application to enable end-users, tinkerers, and developers to use, modify, create, and share powerful, flexible customizations over the content and presentation that the system provides. Such an application can provide more extensive customization capabilities than existing systems without requiring significant programming effort.

This thesis statement focuses on several components. The first of these reflects the notion that not all users are alike. Instead, they each possess different skills and motivations, which will influence their use of an information awareness application. The second of these components recognizes that not only do different users possess distinct skills, but also they do not always customize their software from scratch. Instead, they may use others' customizations as a starting point. The third component focuses on the complexity of the actions the user can take. By *powerful*, we mean that a user can create customizations that go beyond controlling simple parameters of the awareness system to actually controlling abstract behaviors. By *flexible*, we mean that the complexity

[2]Except where otherwise stated, gendered pronouns should be read in the neuter.

of the customization can vary with the power needed to express it. The user should be able to make a choice about how complex a customization to make depending on the challenges involved in creating or modifying a particular informative presentation. The final component argues that, by supporting these powerful and flexible customizations, such a system can enable more complex customizations than are currently possible in other systems without significant programming.

This thesis statement raises the following questions:

RQ1 Is it possible for an information awareness application to give users increased power in their customizations without requiring significant programming effort?

RQ2 Is it possible for an information awareness application to give users increased flexibility in how they create their customizations?

RQ3 What dimensions characterize the customization space for awareness applications?

RQ4 What kinds of customizations can users create with a powerful, flexible customizable information awareness application?

The first of these questions, RQ1, focuses on the power, or expressiveness, of the customizations that a user can create without significant programming. There is a small amount of ambiguity in just what constitutes *programming*. While there are certainly clear examples of programming and non-programming interfaces, the boundary between the two is fuzzy. The word *significant* in this question and in the thesis statement reflects this ambiguity. As such, this question demands not a yes or a no answer, but rather more of a characterization of the interaction. The significance of programming will depend on a combination of the extent and complexity of the programming or programming-like interactions that a user must perform.

While RQ1 focuses on the capabilities of the customization, RQ2 looks beyond the power of expression that the customization interface affords. Rather, it considers how the user can scale that expression. What kinds of choice does the user have in the complexity of a particular customization he might perform? Does he have to create a new channel from scratch? Can he modify an existing channel instead to yield an artifact that is "close enough" to his goal?

RQ3 takes a step back and looks at the broader customization space for information awareness

applications. What approaches have been taken by existing systems to support users at customizing their experience? What dimensions help to define this space? Does these dimensions reveal unexplored areas?

Finally, RQ4 focuses on what kinds of customizations users can make. What kinds of content can the user present? How can she represent those data? Are there examples of customizations that real users have made using such a software system?

The Buzz supports this kind of powerful and flexible customization. Throughout this work, we will examine this system and how it answers these questions. In Chapter 2, we examine two supporting models that will help frame our discussion of these questions. The first of these models focuses on the user and describes a user ecology in which users assume different roles depending on their skills, motivations, and goals at a particular moment. The second of these models describes the underlying technical model to represent information awareness processes.

In Chapter 3, we use these models to frame our examination of various related work in both the information awareness and customization domains. What kinds of information awareness systems have been developed? How do they convey information to the user? What kinds of configuration do they support? This exploration describes the information awareness landscape that The Buzz inhabits. Additionally, it examines various technical approaches that have beent taken in creating these tools. What kinds of data do they support? How do they gather those data? What techniques do they use to present them? How can the user control these processes?

Chapter 4 describes The Buzz prototype software that exemplifies our customization approach. In this chapter, we present The Buzz software system and explore The Buzz user experience. What does the software do? How does the user run it? What kinds of customizations can the user perform, and how might she perform them? Through this discussion, we demonstrate both the power (RQ1) and flexibility (RQ2) of The Buzz, both from the perspective of how the system design supports them and from the perspective of how the user can make use of them. Chapter 5 then explores the underlying technical design of The Buzz. It describes the architectural underpinnings of the system and relates them to the interfaces exposed to the user.

In order to understand the customization capabilities that The Buzz supports within the awareness context, Chapter 6 compares six related systems in the awareness customization domain.

Through this analysis, we identify various dimensions that help to characterize this space (RQ3). We further explore how The Buzz (and each of the other systems within the space) behaves with respect to each of these dimensions. This explorations shows not only how The Buzz supports a different class of customization from other systems, but also gives a sense of what the information awareness customization domain looks like.

We conducted two small deployment studies to get a sense of whether users will be able to take advantage of the customization capabilities available in The Buzz. Chapter 7 describes these deployments, and shows how users created their own channels (RQ4). Although no users within the study group performed especially complex customizations, some users did share their customizations with others and made use of complex customizations that had been created by the system designer, further reinforcing the importance to design for the broader ecology of users. Finally, Chapter 8 presents the research contributions of this work and potential future directions for research.

CHAPTER II

SUPPORTING MODELS

We used two models throughout the design of The Buzz: a user model and a technical model. These models help to frame the analysis of information awareness customization domain, and are used extensively throughout the design of The Buzz software, and in this document. The first of these models draws from the notion that users are not all the same. They bring different skills and different motivations, which will alter how the interact with an awareness system. Section 2.1 describes this user ecology.

The second of these models is the data pipeline. This data pipeline model helps to describe the technical issues that surround the information awareness domain. It describes the process of collecting data from publishers scattered across the Internet and on the local machine and transforming those data in a representation for the user. Section 2.2 describes this pipeline in more detail.

2.1 User Ecology

Customizable software recognizes that not all users are alike. It is because of those differences that the software allows the user to tailor its behavior to better suit his own individual needs. Just as different users may have different needs from the software, different users may also have their own peculiar needs from the customization system.

Different people have different skills and different motivations that affect their use of the software system. A merely competent computer user is likely to customize his software very differently from an expert programmer, who might immediately look for hidden "dotfiles" to tinker with.

Not only will these particular skills be different, but so might their individual needs. An executive, for example, for whom a large part of the job is to maintain networking contacts, might need to keep a keen sense of the various people with whom he interacts on a semi-routine basis. A customer service representative, in contrast, might not need to maintain a breadth of contacts, but might instead need a detailed overview of outstanding problem tickets.

Finally, people often do not customize their software in a vacuum. People rarely create complex

customizations from scratch. More commonly, they share their customizations with each, using someone else's configuration as a starting point. Within an office setting, one coworker might be meeting with another and observe a non-standard operation, prompting him to ask something like, "How'd you get it to do that?" This practice is so common that many web sites have been set up to facilitate sharing of customizations, from desktop wallpapers to email rules, to music "smart playlists." Through this sharing, many kinds of customization belong to a larger ecology, in which one user might create an initial customization, share it, and find it to evolve throughout a larger community.

This notion of an ecology of users has been studied within various customization domains. For example, Maclean *et al.* describe users as being workers, tinkerers, or programmers [68]. Gantt and Nardi identify three similar roles in the context of CAD users: end users, local developers, and professional programmers [40]. Furthermore, they identify a special kind of local developer, the gardener (also called the guru), who receives explicit organizational support to provide customized tools for the group or groups with which he or she interacts. Finally, Mackay identifies various patterns of sharing of Unix dotfiles, with different roles of end users, translators, and programmers [66].

Although all of these studies focus on customization roles within different contexts, and although all of these studies use different terminology for the various roles the users take, they all identify the same three basic classes of users: the end user, who may be competent with the software, but uses it as-is; the tinkerer, who has no intrinsic interest in the inner workings of the computer but will, nonetheless, work "under the hood" to support domain-specific tasks; and the programmer, who typically has formal training to develop new software applications, plugins, and other related support tools.

In *A Small Matter of Programming*, Nardi further observes that these roles are not static [81]. For example, many spreadsheet users, who tend to have no formal programming training or any intrinsic interest in programming, write complex spreadsheet formulae to solve domain-task-specific problems that arise. Even though a user might not have a high degree of skill at programming, her or she may still put forth the effort necessary to learn if there is a perceived work value to doing so. On the other end of the spectrum, consider the Unix guru who uses a Macintosh because it "Just Works." He or she may be highly capable, but chooses not to customize his or her software simply

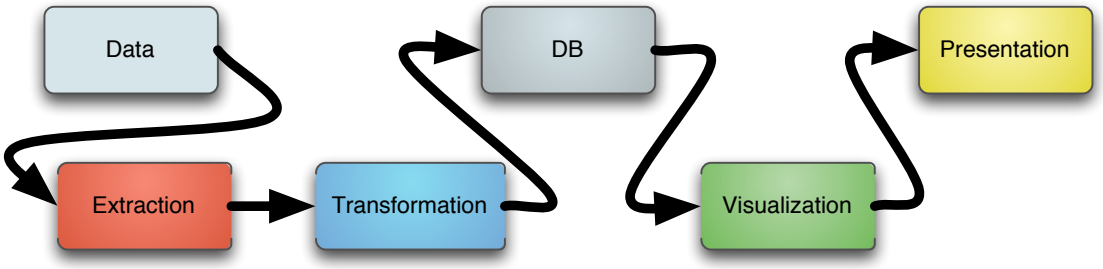

Figure 2: The awareness data pipeline.

because it is not perceived to be worth the time. In this way, the role that a particular user fulfills may be fluid, based on a combination of base skills and motivations.

This variety of users and their mixtures of skills and motivations drives the design of The Buzz. The system should support a user at performing customizations that scale not only with the individual skills of the user but also with her motivation. For example, a merely competent user or one with relatively low motivation should be able to control the content of the awareness system, whereas an advanced or highly motivated user should be able to exert finer-grained control over the system.

2.2 Information Awareness Data Pipeline

In order to present information to the user, the awareness system must gather data from various data sources and transform them into an appropriate representation for the user. The data pipeline model helps describe this data gathering and transformation process (see Figure 2). This pipeline reflects the notion that information awareness tools focus on taking data on one end and transforming those data into information in the user's head. Thus, on the front end of the awareness pipeline, lies data as it is published by the various content producers. On the back end of the pipeline, we have an informative representation of those data. The various operations necessary to convert those data into such an informative presentation lie in the middle of the pipeline.

The front-end of this pipeline is concerned with content. This content is the input to the awareness application. Content may be in the form of a web page intended for human eyes, or it might be encoded in a machine-readable format such as RSS. It could be data from a database, such as provided by the U.S. National Weather Service [99] or even from raw sensor output. Regardless of the source, the data is rarely in a format suitable for directly presenting to the user. Even web pages

often present a specific representation of the data that reflects the publisher's intended use rather than the user's. For example, an information aggregation tool might attempt to unify the presentation style for common data from a variety of web sites. Publishers may embed distracting animated ads in their web pages that would be unsuitable for a peripheral awareness display. As such, even if the data are presented in a human-readable form, that presentation may be incompatible with the goals of the awareness application.

Thus, the next stage of the pipeline involves extracting these data from the publishers. If the data are in a well-structured database or if the publisher provides an appropriate API, then it may be relatively straightforward to extract such data. Nonetheless, such data extraction still requires knowledge of the encoding format or API. If the data are not in such a machine-readable format, then it may be even more complex to extract the relevant data from the irrelevant and to infer any sort of semantics.

Once the data are extracted, they may still be in a different format than desired. Thus, is may be necessary to apply various transformations. For example, temperatures might be in degrees Fahrenheit, degrees Centigrade, or Kelvin. Locations could be a street address, city and state, postal code, or latitude and longitude. For example, the U.S. National Weather Service provides radar data keyed by radar site name, whose location is provided in latitude and longitude. Thus, to get the radar data for a city, the city must first be geocoded into a latitude and longitude from which the nearest radar site can be computed.

The data extraction process is frequently a lengthy process. If data from multiple sources are to be integrated, then they will need to be stored in some intermediate format suitable for aggregation. Thus, the next stage of the pipeline is the database that stores these data as extracted and transformed from the server.

At some point—which may be immediately upon extraction—these data will need to be presented to the user. Somehow the data stored in the database must be represented to the user in the desired fashion. That is, the data must be visualized[1] to yield an information presentation.

For a simple visualization, breaking down the process into this information awareness data pipeline may seem painstaking. For example, a Google Gadget might simply embed the image of

[1] or represented through some other modality such as audio

the day from some particular website. In this case, the data the gadget uses might be a static URL whose content changes daily. Thus, the gadget is merely an HTML template that gets embedded on the user's personalized portal page. The data is implicitly loaded by the user's web browser whenever she loads the portal page. Breaking this process down into the constituent parts of the pipeline may seem overly complex in this case.

However, this pipeline models a wide variety of awareness applications. Thus, while some stages of the pipeline may be short-circuited for some applications, they may be necessary in others. For example, many Google Gadgets actually cache the data they load onto the Google servers so as to avoid overloading the web servers that provide the source content. As such, a gadget may use more stages of the pipeline than may at first seem apparent.

Furthermore, the structure of the pipeline allows for flexibility in the design of information awareness applications. Data transformations could take place at the time data are extracted, at the time the data are visualized, or a combination of the two. Whichever approach is taken can depend on the particular needs of the application.

CHAPTER III

RELATED WORK

Supporting extensive customization in information awareness applications involves the intersection of two domains: end user customization and information awareness. This chapter comprises three sections related to these two topics. We first examine information awareness applications in order to better understand what kinds of uses these systems support (Section 3.1). What kinds of data can they convey? How do they go about conveying those data? How does a user interact with the system, if at all?

With an understanding of this awareness domain, we can then focus on the underlying technical tasks necessary to support both the gathering and conveying data and the customization of those processes (Section 3.2). Section 3.3 considers work from the end-user customization domain. In addition to this Related Work chapter, Chapter 6 provides a detailed comparison and evaluation of six systems in the awareness and customization space.

3.1 Information Awareness Applications

Numerous informations awareness systems have been built. Some of these systems focus on aggregating information from numerous sources and providing a unified interface to that information, or on combining data from one or more sources and "mashing them up" with another interface. Regardless of how they aggregate their data or what interface they use, these tools all share in common the ability to *repurpose* data.

In their repurposing of these data, awareness applications strive to help people to better manage their attention and to handle the onslaught of information that inundates their lives. Some awareness tools provide informative dashboards, such as Sideshow [18] and Irwin [74]. These tools run in a small region along the side of the screen or in a corner. These tools comprise small applets or tickets, each of which conveys a specific piece of information, such as the state of an email inbox, main memory usage, or the weather.

Sideshow even provides a C++-based software developer's kit (SDK) through which programmers can create new tickets to add to the ticker. These tickets run in the periphery in the sense that they reside on the side of the screen. Some of these tickets, however, are fairly dynamic in nature and may draw the attention of the user. Furthermore, the focus of these tickets is to provide ready access to information through their display and interaction. As such, the data representations are usually appealing and visually active, and provide extensive interactive support to dive down into the information conveyed. For example, the email ticket allows the user to see an overview of the inbox, read a full email, and trigger the email client to send a reply.

Konfabulator [6], Apple's Dashboard [11], Vista Sidebar [75], Google Gadgets [4], and Windows Live Gadgets use a similar applet-based approach through which the user can create a full-screen collection of applets. Developers can create new applets (often called widgets or gadgets) using a combination XML, HTML, and JavaScript. Because these tools use web-based frameworks, they can run in various configurations. For example, both Live Gadgets and Google Gadgets can run on the desktop in a full-screen mode as described, in a side bar as Sideshow, or embedded in a web portal page. Thus, they support a flexible style of information presentation that can vary from peripheral (as in the sidebar) to something that integrates with the user's active information foraging habits.

So far, all of the systems we have considered focus on conveying personally-relevant information to the user. That is, the information is tailored to the individual. Other awareness systems have focused on promoting group awareness and interaction. The Notification Collage [45] and MessyDesk [36] attempt to promote group awareness and interaction by providing a full-screen canvas, or bulletin-board, to which group members can post various snippets of information—images, weblinks, live webcams, text notes, *etc*. These systems are tailored to run on the individual user's desktop (usually on a secondary monitor), but they can also run on a large, shared, projected display.

Whereas the Notification Collage and MessyDesk provide an interactive tool through which group members can exchange information, the What's Happening screensaver [111] passively conveys information through automatically generated collages from the community web server. Accenture Technology Labs' UniCast [73] uses a similar approach, but running on a secondary display to

convey information in the periphery by displaying collages from sources that the user has selected. Huang's Semi-Public Displays [49] further attempt to promote group awareness by encoding presence and activity information in the display, while attempting to balance awareness and privacy issues.

Churchill's PlasmaPosters project [23] combines approaches from the previous few systems to convey user-submitted group content via large-screen plasma displays embedded throughout the environment in places such as heavily-trafficked corridors or break rooms. The Café Life project [24] deploys a similar system in a coffee shop to further promote community interaction and awareness.

Because many of these awareness systems are intended to run on a peripheral display on the desktop or even on a wall-based display (such as a flat-panel display or projected image), several approaches emphasize the aesthetics of the information they convey. The InfoCanvas [79, 96] depicts a scene in which the elements change to abstractly represent dynamic information. For example, in a beach scene, the color of a woman's bathing suit might change depending on the traffic on the freeway, the number of empty drink cups near her might depend on how full an email inbox is, and a bird might soar higher in the sky depending on the current temperature. In this way, what might look like an image of a beach scene to a casual observer is actually an informative display to the owner. The Kandinsky [38] and Informative Art [91] similarly convey information via aesthetic presentations, but emphasise artistic representations over more direct information representations.

Many of these tools operate in the periphery, so they must balance the needs of effectively conveying information with the need to avoid unnecessary interruption [14, 28]. Judicious use of animation [85] and change-blind transitions [54] can help to avoid unintended interruptions. Nonetheless, sometimes it may be necessary to alert the user under exceptional circumstances or with urgent information. Matthews *et al.* created the Peripheral Display Toolkit to help support designers of peripheral displays at managing notifications and interruptions [72, 71].

There are many other information awareness systems and approaches, focusing on conveying information via methods from ambient interfaces [87, 31, 20, 48] to tangible interfaces [44, 55, 20]. As it becomes increasingly feasible to embed networked display technology throughout the environment in both larger and smaller form factors, designers will continue to create new and exciting ways to convey information.

3.2 Data Gathering

Creating these information awareness tools, however, involves solving technical challenges at each stage of the data pipeline. Challenges arise in gathering data from multiple data sources in a multiplicity of formats and encodings, integrating them into a unified and coherent format if they are to be aggregated, and effectively conveying them to the user in whatever manner is used by the particular awareness system. The remainder of this section focuses on work related to solving these technical challenges.

3.2.1 Data Scraping

There are many data formats in current use, all tailored to various properties of the data they encode. Understanding all of these data formats and representations is an enormous task which few, if any, systems could support. It would be too time-consuming to be able to decode and make sense of so many data formats.

Furthermore, not all data are stored in semantically-rich formats. The vast majority of data on the World Wide Web, for example, is stored in HTML or XHTML. This markup format provides for rendering markup to describe how data should be displayed to the user, but it rarely contains semantic markup to describe the data themselves (although there are some efforts to graft semantic markup into HTML [58, 37, 109]).

Although HTML is primarily a presentation markup language, there is often an implicit semantic structure to HTML documents. For example, most newspapers use a common template for all of their articles. Two different articles in the same section of a newspaper are likely to have a similar high-level structure from which the various individual data components can be inferred. The headline of the article might be in an `<h1>` tag while the text of the article might be in a separate class. Thus, given this implicit information, it may be possible to extract the underlying data from an HTML document (see Figure 3).

There are two primary forms of data available on the web: those that are in presentation-oriented markup languages and those that are in semantic data formats. The first kind of format might encode data to be shown to the user, with, for example, a title in a bolder, larger typeface. In this way, the document format does not specify that the bolder, larger text is the article title. A semantic format,

```
<span class="tel">
   Our <span class="type">work</span> number:
   <span class="value">
   +1.415.555.WORK</span>
</span>
```

Figure 3: A microformat to provide semantic information for a phone number in an HTML document [58].

in contrast, would specifically designate this underlying meaning of the data.

This second approach may seem relatively straightforward at first. If a publisher exposes an API for their data, then there is a clear method by which to gain access to and to extract the content. Difficulty arises, however, when we consider many publishers. If each publisher exposes a different API for their data, then each potential data source may need a separate translation mechanism to convert and extract the data from each publisher.

There are, however, some existing approaches to attempt to resolve the complexity of abundant data access frameworks. The Web Service Description Language (WSDL) is an attempt to create a standard interface by which web services can publish their interfaces in a machine-readable format [22]. A web service provider can publish their interface as a WSDL document. This document provides a combination of human-readable descriptions of the interface along with a machine readable specification. The U.S. National Weather Service, for example, publishes the interface to the National Digital Forecast Database (NDFD) in WSDL format [100].

Many of these web services are built using protocols such as SOAP or REST [112, 46], which attempt to create a lightweight RPC-over-HTTP mechanism. Many so-called Web 2.0 [84] services publish interfaces that use these simple protocols. For example, Yahoo provides such access to its search services [8]. The Flickr photo sharing site provides rich access to its data via REST [3], as does the social bookmarking service, Digg [2].

These interfaces may help in getting access to the various data provided by content producers, but a human still needs to make sense of the interface. These APIs specify a sort of a "contract" for how to access the data, but a human still needs to make sense of what it means to gather, for example, the interestingness of a photo on Flickr.

As a middle ground to providing such rich APIs, many publishers make extensive use of more rigid standardized formats that provide semantic information about common attributes of the content, but allow for a flexible document structure. Really Simple Syndication (RSS) [106], the Resource Description Framework (RDF) [9], and Atom [53] describe article-style documents in a machine-readable format, delineating titles, summaries, full stories, authors, *etc*. RDF is also used extensively in the Semantic Web community [109].

These formats have been adopted by a variety of publishers in a variety of industries, including the BBC World News, Flickr, and Apple's iTunes Store. These feeds provide semantic markup to distinguish various properties of each entry, such as title and description, but the specific format of some of these values may be only loosely defined. For example, the description of an entry might be a one or two line summary in plain text, or it might be the full text of the article in HTML format. Flickr updates, for example, use a specific HTML template where Flickr-specific data, such as number of comments on a photo, are only encoded in the HTML. In order to extract the number of comments on a photo from this feed, an application would have to scrape it out of the HTML. The web service could, however, (and many frequently do) change what content appears within the entry description field or how it is represented. In this way, general purpose semantic formats can help make data more available to information awareness applications, but there is still a certain amount of wizardry involved in finessing the data into the appropriate format for many tasks.

3.2.2 Web Data Scraping

Even as more services make their data available through web APIs or feeds, much of that content is likely to remain in HTML format. As such, scraping data from HTML and other non-semantic formats can be a valuable tool to gathering data to be repurposed. A lot of work has been performed in the area of web data extraction (see [60] for a brief survey).

One approach is to use a pattern expression to extract content from a text-based document, regardless of whether it is in plain text, HTML, RSS, or some other XML-based format. Because regular expressions are difficult even for programmers, let alone end users, many data extraction tools either hide the regular expressions from the user or provide simplified "wildcard" expressions instead [76, 47]. One advantage of this pattern matching approach is that it can find data that is

embedded within data of another format. For example, a pattern that looks for HTML `<image>` tags in textual data will find such tags even if they are embedded in the document with inline JavaScript, or if they are a part of an HTML block embedded within an RSS document. However, because the pattern pays no attention to context, it is likely to match erroneous data. For example, an HTML `<image>` tag matcher will probably match against images that have been commented out in the underlying HTML document. Furthermore, such patterns are difficult to write and often incorrectly handle more subtle features of the underlying markup format.

To address the limitation of blindly looking for matching text in a marked-up document, encoding-aware patterns can be useful. For example, the W3C's XPath query language is designed specifically for matching elements and attributes in XML documents [25]. More broadly, tools such as XWRAP [47], RoadRunner [27], and Dapper [1] operate on parsed data to take advantage of the document object model (DOM) hierarchy. One limitation of this approach is that they are relatively fragile when dealing with malformed documents. Most modern web browsers support a "quirks mode" to handle malformed markup. As a result, many of the pages on the web render in most web browsers, but might not parse correctly to a standards-compliant parser.

As an alternative to hand-crafted data scraping methods, various machine learning-based methods have been used. One approach is to use various data detectors [82, 35], capable of identifying data items in an arbitrary textual document like a web page or email message. These data detectors can recognize things like URLs, phone numbers, email addresses, *etc*., in a document. This approach can be useful when the desired data is relatively easily and unambiguously detected, but it may have difficulty distinguishing between a ten-digit phone number and a ten-digit product code.

To help limit data detection errors, some systems combine a library of data types with explicit information from the user to identify the entities in the document [52, 78]. Thus, a user can click on the various items in a document and the system can infer what kind of the data they are, as well as identify similar items in the document. This approach can be particularly useful, but care must be taken to ensure that the user's model is aligned with the inference system and to detect possible outliers [77].

Rather than relying on the user to identify the data itself, tools like RoadRunner [27] and Dapper[1] allow the user to specify multiple instances of the same web page. The system then

compares these examples to attempt to identify the relevant parts of the document, possibly with the aid of the user. For example, a user could specify three days' worth of the front page of an online newspaper as examples, from which the system could try to identify how to extract the headlines.

So far, all of these examples have operated on raw data such as an HTML or XML document. Greenberg and Boyle's notification engine instead operates on the rendered output of webpages [43] (see Section 6.2.6 for a more detailed description). The user identifies regions of interest on a rendered web page, relative to a fixed position on the page or some "visual anchor." This approach takes advantage of the web browser's ability to handle the vast majority of data published on the web. If a particular web page presents its information using Flash or Java, the browser is already capable of displaying it. However, because this approach operates on the rendered output, it does not provide semantic access to the data. It scrapes out a collection of pixels rather than the data they represent. Nonetheless, for many peripheral awareness applications, it may be sufficient merely to extract the visual information to present to the user as-is.

3.2.3 Data Storage

The data storage referred to in this section does not have to do with how to represent data on disk in the more traditional databases and data representation senses. Rather, it refers to how to store data from a collection of different data sources in a variety of different formats and for a variety of purposes in a common format. Ideally, similar data in different formats could relatively easily be combined to be used in compatible tasks. For example, data from an RSS feed of news articles and data on auction listings might share similar attributes: both might have a title, a description, and possibly some pictures. Although the data sources and formats are different, a common data format might allow them to be represented in the same fashion.

However, using a rigid structure can lead to unnecessary complexity when defining new data types to store in the database. A complex new definition may need to be defined to extend the existing database schemas. Alternatively, a designer might attempt to shoehorn the new data format into an existing schema in a way that is not strictly compatible with the schema or the data being stored. These difficulties illustrate a tradeoff between complexity and mutual intelligibility [33].

An alternative approach is to use a folksonomic approach using tagging [102, 70]. Under this

approach, new data types are added using the appropriate existing tags or adding new ones as necessary. The lightweight nature of this approach is also its biggest detractor: without any central management of tags, the meanings of tags might inappropriately evolve. These approaches are prone to tag-creep, whereby the different tags might be used for equivalent purposes or the same tag might be used with a variety of meanings.

3.2.4 Data Presentation

The last stage in the data pipeline process is to transform the data into an information representation. Creating such visualizations in general is a difficult challenge, whether on the screen [95] or off the screen [98], and regardless of whether the purpose of those visualizations is to support active analysis or more peripheral awareness. Most general-purpose awareness presentation interfaces delegate the task of depicting the data to a programmer, as in the dashboards and SideShow, where the widget developer has full programmatic control over individual pixels.

Others still provide a more restrictive model of the kinds of information that they can convey and provide a framework for visualizing those kinds of data. The InfoCanvas, for example, lets the user drag and drop visual operators to the screen, which represent a particular encoding for a piece of data. For example, a user could drag a slider operator to screen and associate it with an image and a numeric data value. The particular value of the number would then control where along the specified path the image would be drawn.

Restricting the class of presentations that a system can support, however, can allow for significant flexibility. North and Shneiderman's Snap-Together Visualizations [83], for example, help to enable users to create custom coordinated-views visualizations without programming. Using this system, users were able to create complex visualizations out a collection of pluggable components. North and Shneiderman further describe three levels of customization available for visualization systems (paraphrased from [83]):

1. Data. The user can specify what data to use with an existing visualization.

2. Visualizations. The user can specify what data to use and choose an appropriate pre-existing visualization for them.

3. Coordination. The user can specify how multiple visualizations can be combined to coordinate amongst their data.

Beyond their categorization, a fourth level might involve the creation of new visualizations, as supported through tools like the dashboards, SideShow, and The Buzz (see Chapter 4).

Various frameworks can help guide users at creating visualizations of their data. Operating at level 2, Bell Labs' InfoStill [26] enables users to embed dynamic visualizations into web-based reports using an easy to learn markup syntax. For creating more complex visualizations, Apple's DashCode allows widget developers to create their widgets using a drag-and-drop GUI creator to construct a view using reusable interface components. Thus, the widget developer need only write a reduced set of widget-specific code to connect the components and use any specialized rendering methods necessary.

3.3 End User Customization

The End User Programming community [57] is primarily concerned with supporting end users at writing better programs. These approaches often involve trying to make programming easier, such as by visual programming techniques. Such programming approaches, however, are frequently concerned with supporting general purpose computation in a broad collection of problems. Because of their general purpose nature, these approaches necessary entail significant abstractions necessary in computing.

More specialized programming languages such as Processing [90] can help to align programming abstractions to the particular domain in which they are applied. In the case of Processing, the language has enabled artists and designers to create a variety of creative information visualizations. Nonetheless, even an applied programming language tailored to a particular domain can still require significant technical effort beyond the needs or desires of many users.

Many non-programmers have, however, used programming techniques to extensively customize their software. Spreadsheet users frequently create artifacts that demonstrate complex conditional logic and computation [81], but such computation is typically highly scaffolded and aligns closely with the particular task domain of the user. In this way, even though the user may be writing a complex spreadsheet program, there is enough structure to the process and domain orientation that

many non-programmers are able to create such programs.

Much of the work in the end user programming community can help to solve end user customization problems through a trickle-down effect. Nonetheless, many programming solutions, such as those that involve training users to use guard conditions [17] or to aid in debugging [59] may have varying utility within more focused customization domains. In these contexts, those performing customization may not desire to use software engineering practices; primary concern is of getting something that just works, or works well enough.

Because of this domain-centric nature of customization problems, much of the end user customization work has been applied in nature. Such customization frequently takes place within the context of performing another task, such as recording scripts of web actions [63] (see Section 6.2.5), creating mashups [108, 7] (see Sections 6.2.2 and 6.2.1), extracting data from web pages [52, 32] (see Section 6.2.4), customizing web-based interactions [16], writing web crawlers [76], creating interactive "scrapbooks" out of pieces of webpages [97], or otherwise piecing together elements from web pages [39]. All of these approaches focus specifically on tasks within a particular domain. For example, Chickenfoot might help a user to extract a set of product listings from a web page and sort based on price, even if the publisher does not provide that ability [16].

Each of these customization approaches involves different degrees of complexity or even programming activity, depending on the complexity of the underlying task. For example, the aforementioned Chickenfoot tool makes extensive use of JavaScript to support many customizations that might not otherwise be feasible.

CHAPTER IV

THE BUZZ

This chapter describes The Buzz, the software system designed to exemplify our flexible customization approach over a broad range of customizations. The goal of the research driving the application is to be able to examine how to support end-users, tinkerers, and developers at being able to create their own personal information awareness streams, with tailored content reflecting the combination of the users' own skills and motivations.

In contrast, the user's goal is to maintain awareness of routine and personally relevant information. Customizing the system itself is related to this goal, but is not the goal itself. As such, the user's motivation to perform such customization will likely relate to the perceived benefit of performing such customization. Although supporting such awareness is an important function of the software under study, it is beyond the focus of this research. While the software is designed with the intent of promoting awareness, we make no claims about the effectiveness of the software toward that purpose. Rather, we argue that users have expressed a desire to customize content to their interests. Furthermore, it makes intuitive sense that content tailored to the user's own interests should be more useful than arbitrary content. As such, this research focuses on supporting such customization within this awareness context.

4.1 Formative Interviews

In order to understand what sorts of information users might wish to see and what sorts of customizations they might wish to make to the data, we interviewed twelve potential users of the software. These interviews consisted primarily of lightweight information gathering sessions rather then more stringent requirements analyses. Our goal was to understand both the kinds of information that users expressed a desire toward seeing in such an information awareness tool as well as their existing information monitoring and awareness practices.

We conducted interviews with twelve participants from the Georgia Tech community. All but two of these interviews were conducted at the participants' normal workplace. In the interviews,

we asked users to describe their typical day and the sorts of information they routinely access. This included email, web sites, databases, *etc*., as well as offline media such as television and radio. Our goal in this line of questioning was to identify what information sources might be important to support in a personal information awareness application.

We then asked users to walk us through their bookmarks and web browser histories. For privacy reasons, participants were given a chance to self-censor their bookmarks and web browser histories before our interviews. For each web site, participants described how they used that resource. What sorts of monitoring habits did they use? Did they tend to monitor a particular portion of a web page or web site, or did they monitor the web site as a whole?

Additionally, we asked what kinds of customizations they had made to their software. Did they change any of their preferences in any of their software applications? Did they use any mail handling rules in their email client? Had they customized a web portal page? Had they ever downloaded or used any software plugins or other types of hacks? In particular, we were interested in what experience our participants had in customizing their software and to what degree they were willing to do so.

Finally, we gave our participants a stack of index cards with various potential data sources on them. We asked our users to rank each data source into three piles: those that they would particularly like to see in the awareness application; those that they might like to see; and those that they did not care about. For each source, they were then asked to describe why they cared about that particular source and what sorts of customizations they might wish to make to them (*e. g.* location for the weather).

These sources on the index cards included those used by the then-existing system along with sources that we as the designers of the system thought might be useful to include and sources that users had suggested in our pilot interviews. Finally, we also gave the participants a stack of blank index cards on which they could add their own.

4.1.1 Interview Observations

Within the College of Computing, we interviewed three Professors, seven graduate students, an administrative assistant, and an administrator, all working in Computer Science or related areas. The

College of Computing being a technical organization, we suspect that our participants demonstrated a higher degree of capability and willingness to customize their software. We believe, however, that this is also similar to the demographic that is most likely to use a customizable information awareness application.

Interestingly, in our interviews, a quarter of our participants had used customizable portal pages and all but one of them had customized their portal pages. The participant who had not performed any customizations indicated that he had examined the options available but had chosen not to use them because they were too rudimentary. Furthermore, of these users who had customized their portal pages, none of them frequently visited those pages. These users had taken the time to customize the portal pages, but did not actually use the resulting artifact. This lack of use suggests that users may find portal pages too lacking in customization and/or presentation capabilities to be useful. For example, one user remarked that he had customized his page to figure out what he could do with the portal, but once he had done so, did not feel compelled to use it. Rather than condemning portal pages or information awareness applications in general, we see this behavior as potentially calling out the need for richer customization capabilities in information awareness applications. If users are unable to adequately customize their awareness tools, they may be unlikely to use them. Further study is necessary to tease out reasons for this behavior.

Our primary goal in probing our users for their information monitoring habits, however, was to gain insight into the sorts of triggers that existed to their accessing particular pieces of information. In Kellar *et al.*'s terminology for monitoring activities [56], we were looking at when participants tended to monitor data sources for browsing, fact finding, or information gathering purposes. Each of these different purposes suggests a different kind of interaction. For example, a user participating in a browsing activity may be content merely to sit back and let the system determine what content appears and when. For a fact finding activity, however, the user is motivated to see a particular piece of information at a particular time. As such, a direct path to that data would need to be available. Therefore, an application such as The Buzz, which provides a slideshow style of presentation, should provide a shortcut to an index of all of the available content if it is to support fact finding activities.

Finally, we used the index cards to provide an indication of what kinds of data sources our users

News headlines (10), pretty things (8), Atlanta events (8), weather (8), weblogs (8), traffic (8), stocks (4), comics (4), calendar events (4), class wikis (3), CoC calendar (3), todo items (2), MARTA announcements (2), campus trolley times (2), class websites (2), radio listings (1), sports box scores (1), jazz cds (1), Atlanta Journal-Constitution articles (1), public school alerts (1), Slickdeals.com (1), airfare alerts (1), movie listings (1), grocery store sales (1), travel planner (1), instant messaging buddy lists (1), local bulletin board (1), dating info (1), computer system status (1), conference deadlines (1), TiVo listings (1), jokes (1), celebrity gossip (1), bible verse of the day (1), new mail messages (1)

Figure 4: Potential channel information sources suggested by users in our interviews. Numbers in parentheses indicate how many participants expressed a desire for that particular topic.

might want to monitor. While the standard sources such as news headlines, traffic, weather, and stock quotes quickly rose to the top of the list, there were also some surprising sources identified. One user described a "pretty things" collection of pictures she had taken on her travels: "It sparks conversations and makes me happy." Figure 4 summarizes the various items suggested by users in interviews.

Other sources tended to be relatively one-off in nature. For example, a professor wanted to see "what the snake says" on an introduction to programming class wiki. On the web page for the class in question, the professor puts quick class announcements in a speech bubble next to a cartoon Python snake. Seeing what the snake says would help him to keep in sync with the announcements he or any of the other teaching assistants might be making to the class. This particular example demonstrates an inherent limitation in awareness applications that focus on handling data in particular formats. Although data in semantic markup formats such as RSS is increasingly common, most content is more *ad hoc* in nature. For example, the speech bubble of the snake is a clever presentation trick made by the designer of the class web page. As a result, a general purpose heuristic to extract the meaningful content from a web page is unlikely to find "what the snake says" without some more specialized guidance. Instead, the "What the snake says" channel was implemented using a custom extraction pattern.

4.2 Prototype Design Goals

Using these information sources as inspiration, we were able to identify a combination of both popular information sources that the system should probably support, and sources that might pose an interesting challenge to support. For example, eight of our twelve participants indicated that they

desired to see interesting pictures from their photo collections, others' collections, or both. As such, one of the concrete goals was to make sure that it was possible to create such a photo gallery using the customization capabilities of the system.

Moreover, our primary goal in designing The Buzz was to provide extensive support for customization. Such support should provide a flexible degree of control over the content, such that a relatively novice user could perform comparatively basic customizations, while an advanced user desiring to perform a complex task might be able to use an extensive customization interface. In this way, the complexity of the interface should scale with the complexity of the task the user wishes to perform.

One of our approaches to addressing this goal is, wherever possible and reasonable, to use general purpose methods that could support configurability. By using these general purpose methods, the system could more readily support users at defining different kinds of data or presentation methods without resorting to writing specialized code. Nonetheless, there is an ease-of-use tradeoff between general purpose and specialized methods. We have attempted to balance these tradeoffs in The Buzz by providing a combination of general purpose and specialized tools.

Further, in recognition of users' practices of sharing customizations, we felt that it was important to explicitly support this operation within the interface. If possible, the interface should help to foster a community of sharing by promoting such behavior within the system. Through this mechanism, users should be able to find, modify, and share customizations via a lightweight interface.

In addition to these customization goals, the awareness system should operate as a peripheral interface. Although the system runs in both a shared, public context and as a secondary display on an individual's desktop, the latter environment is of primary focus. Both environments entail different design decisions and use models. As such, where the two are at odds, we chose to optimize for the individual. In this environment, the awareness display operates in close proximity to the user's normal work place. As such, it is especially important for the interface to remain in the periphery unless explicitly called to the forefront of the user's attention. As such, the interface should update in a change-blind fashion [54] so as to minimize undesired interruption [14, 50].

Nonetheless, at some point the information presented by the system will transition from the periphery of attention to the focus. Thus, while the system should avoid unintentionally drawing

the user's attention, it should also support the transition to a focal task. Thus, if the user sees information of interest and desires to follow up on it, the software should provide an appropriate mechanism to transition to a more directed information-seeking task. As a result, The Buzz allows channel designers to specify actions to be taken when the user clicks on a region in the presentation.

4.3 Iterative Prototypes

Using these interviews as a foundation, we identified technical capabilities the system would need to possess. What kinds of data will the software need to be able to support? What kinds of behaviors will it need to support, and how will the user need to be able to control them? What kinds of customizations will it be necessary to rule out, given the relative complexity of supporting such data sources?

By combining these technical requirements with user-oriented goals of the software, we began designing the system. To help ensure that users would be able to make use of the customization capabilities of The Buzz, we followed a user-centered design-based approach with a series of increasing-fidelity prototypes. During this iterative prototyping process, we asked a small set of 3-4 potential users of the software to help critique the designs.

Initially, we created a collection of sketches of potential interface approaches. At this phase of the design, it was not clear even what metaphor to use to represent a data source and presentation. Through examining this collection of sketches and our lengthy discussions with users, we eventually settled on a channel metaphor and a channel-selection interface reminiscent of a screensaver control panel.

After selecting a metaphor and interface style from these sketches, we used paper prototypes to examine the interaction of this interface. It was not until we had implemented a high-fidelity prototype, however, that it became clear that the screensaver-style interface would not scale appropriately to support a large number of channels (see Figure 5). With a large number of channels, the channel titles blended together and did not provide sufficient cues for the users to easily distinguish them. As a result, we revised this interface to the one shown in Figure 8. The new channel browser interface supports channel icons, which allow the user to more quickly recognize the channels based

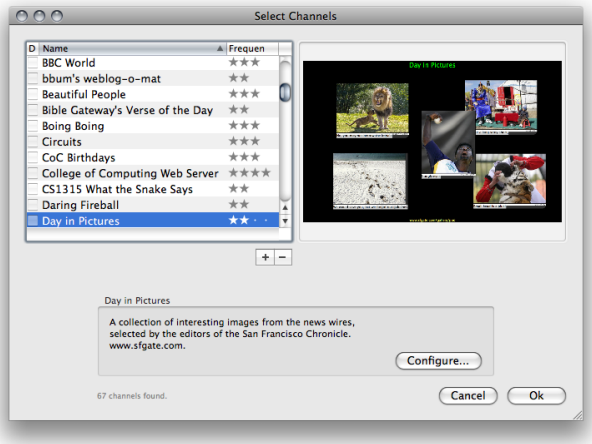

Figure 5: The original channel selector interface, which was replaced by the channel browser shown in Figure 8.

on their represented icon rather than requiring them to read and decipher the channel names[1].

Once the system had evolved to the point of high-fidelity prototypes, we continued to ask our user testers to perform various tasks in the interface while we observed. These users were encouraged to try to perform their tasks without intervention from the designer, but help was available if they became stuck. By watching the users perform or fumble these tasks in the interface, we were able to identify tasks and interface elements that were potentially too difficult to use.

4.4 Running The Buzz

The Buzz runs on an extra monitor on the desktop (see Figure 6) or on a large, public display (see Figure 7). Approximately every minute, it cycles through a different slide, automatically generated from a particular data source, or channel. These slides depict primarily web based content, such as news headlines, weather, traffic, weblogs, photo galleries, organizational directories, *etc*. Although content primarily comes from the web, content can also come from the local system, such as a local Excel spreadsheet or the contents of an Apple Mail inbox.

When running on the desktop, The Buzz displays its slides on a dedicated secondary display. While the user is focused on his primary task, this slide is intended to remain in the periphery. Whatever content is displayed should remain outside of the user's attention. At some point, when the

[1]The channel browser interface also supports additional capabilities that were added later in the design process. Consequently, the channel selector does not support as many capabilities, such as sharing or copying channels.

Figure 6: The Buzz running on a extra monitor on the desktop (at right).

Figure 7: The Buzz running in a public lounge.

user takes a natural microbreak, such as to adjust his glasses or take a sip of tea, he may glance at the display, opportunistically gleaning whatever information happens to be displayed at that moment. If the information shown is irrelevant, the user can ignore the content and redirect his focus back to his primary task. Because the encounter occurred opportunistically during a natural break, there should be little additional cognitive cost associated with the user's attentional shift [14].

If, however, the information displayed is relevant, the user may quickly digest it and return to his primary task. For example, he may glance at the display, see that the weather forecast calls for rain, and make a mental note to bring an umbrella tomorrow. Or, he may interrupt his original task to dive more deeply into the information conveyed. For example, he may see that a student posted a question to a class forum and redirect his attention.

As this example illustrates, it is important to recognize that, at some point, the information shown on the display will transition from the periphery to the focus of attention. As such, it is important for the system to accomodate this shift in attention. Too many interruptions can be detrimental to productivity, but relevant interruptions can be of benefit [50]. Thus, the system should attempt to avoid unnecessary interruptions, but support the transition from peripheral task to focal task. To support this transition, the user can click on an item in the slide. The system will open the URL, application, or document related to the item shown on the screen.

To help avoid unnecessary interruptions, The Buzz displays only static content[2]—typically images and text. Furthermore, when the display updates from one slide to another, it uses a slow-in, slow-out [61, 51], smoothly animated transition to promote change-blindness [54] and avoid inadvertently grabbing the user's attention [85]. By using such smooth transitions, the user will ideally only look at the display when she chooses to. There is a concern, however, that even if the information display updates in a sufficiently change-blind fashion, it may ingrain itself into the user's habits, causing her to habitually glance at the display in much the same way that one instinctively continues to reach for an open bag of potato chips, even when not hungry [50]. Nonetheless, as much as is feasible, the display attempts to remain in the periphery of the user's attention, unless she explicitly glances at it.

[2] Although the system only display static content, it is possible for a plugin author to add support for dynamic, animated content. Nonetheless, the system provides no explicit support for such visualizations and the designer discourages them.

In this way, The Buzz attempts to promote opportunistic information encounters. These opportunistic encounters are lightweight interactions intended to convey information briefly and with minimal intrusion into the user's primary task [111]. The Buzz uses a presentation style based on the What's Happening screensaver, which has been successful at promoting opportunistic information encounters within the context of the community awareness domain [110]. Rather than running as a screensaver, however, The Buzz runs on its own dedicated display. Additionally, The Buzz adds extensive user customizability, whereas What's Happening provides a one-size-fits-all solution.

4.4.1 Users of The Buzz

Many information awareness systems attempt to support end users at customizing their software. But end users are not homogenous. Different people bring with them their own different sets of skills, capabilities, and motivations. In fact, it is this very heterogeneity that boasts the need for supporting customization in software.

As such, it is important to understand what is meant by "the end user." Indeed, the term "the end user" itself is potentially problematic because it conflates many different sorts of end users. In reality, individual end users possess their own unique skills and motivations. As such, the various triggers and barriers that might encourage or discourage one user from performing a particular task may be very different from the triggers or barriers of another user.

This heterogeneity of skills, motivations, and information needs calls out the need for flexible customizable software. In addition to possessing distinct skills and motivations, different users have different interests. Information that is interesting to one user may be dull or irrelevant to another. For example, someone who commutes via rail is unlikely to care about how heavy the traffic is on the freeways. A field technician will likely need very different information about pending problem tickets than a manager or executive. As such, the user should be able to customize the awareness system to suit these different interests.

Not only do these users need to be able to express different information interests, but also they need to be able to do so via interfaces that align with their particular skill sets and motivations. As such, the interface should provide a flexible interface that allows the complexity of the interaction to scale with the complexity of the task. Thus, we must consider the degrees of customization and

the complexity of the interactions in understanding the various information awareness systems.

4.5 Channels

Users can choose what information appears by subscribing to *channels*, which correspond to the different slides that the system displays. Each channel encapsulates the process of gathering data from various content publishers and transforming them into a representation on the screen. In this way, channels are conceptually similar to widgets in the Dashboard systems.

In terms of the data pipeline (Section 2.2), the channel describes the process of *extracting* content from the data publishers and *transforming* those data into a suitable intermediate format to *visualize* on the screen. It also provides various metadata suitable for the management of a collection of channels, called a "channel lineup" in The Buzz. (Section 4.9.4 describes the components that control this process.)

As such, the channel is the primary artifact with which users interact in The Buzz. It controls what content the system displays, when that content will be shown, and how. The channel is effectively the document around which the user interacts. Users can interact with these channels both within The Buzz user interface, or they can treat them as documents and interact with them through the file system, just as they would with any other application document.

4.6 Customizing The Buzz

The channel, being the primary artifact of The Buzz, is what users see within the system and what they customize to control their awareness content. The Buzz offers a flexible customization system to support a range of different sorts of customization, from selecting amongst existing channels so as to create a personal channel lineup to creating plugins that define new data extraction algorithms or visualizations. These different customization capabilities focus on the different user roles within the system in an attempt to make simple customizations simple and complex customizations feasible. As such, these interfaces attempt to connect together fluidly so that the interface can support the user as she transitions from one role to another.

Just as it is difficult to classify individual users into rigid roles—users instead transition from role to role as their individual skills and motivations fluctuate—so, too, is it difficult to partition the interface into distinct modes. Instead, one of the design goals of The Buzz is to provide cues

as to how to accomplish more complex operations, thus providing a link between more complex user goals and the actions necessary to accomplish them [86]. In this way, the different interfaces for accomplishing different tasks support the user at diving deeper into the configuration to control increasingly complex and advanced capabilities. To support this diving to deeper levels within the interface, each level provides cues to suggest that there are more advanced controls available.

The rest of this section focuses on some of the different tasks that users can perform with The Buzz and the interfaces and interactions that support those tasks. These customization tasks are broken into four high-level activities: creating a channel lineup from existing channels (Section 4.7), modifying and creating channels (Section 4.8), extending the capabilities of the system with plugins (Section 5.7), and sharing customizations (Section 4.10).

4.7 Creating a Channel Lineup

The first time the user launches The Buzz, the system displays a welcome screen that describes the software and displays an overview of the channels in the user's current lineup (see Figure 8). This view presents the channels in the system's default lineup, which is catered to the broader community of Buzz users. As such, one of the user's first goals will be to customize what information the system conveys.

The channel browser consists of three primary regions: the middle region depicts the current lineup of all loaded channels. On the right side of the screen are details about the currently selected channel and controls to display and customize that channel. The left side of the screen allows the user to switch between viewing the current channel lineup and browsing available channels from a shared repository.

The channel list (in the middle of the screen) represents channels as icons, much in the same way that they might be shown as icons on the file system Desktop. Each channel has its own icon, to help the user to recognize and distinguish the various channels. The dots on the left side of the channel convey the frequency with which the channel is to be displayed, much in the same way that relevance bars convey relevance to query terms in search results. Thus, a channel with all four dots will be displayed very frequently while a channel with no dots will not be shown.

On the right side of the screen is additional information about the currently selected channel.

Figure 8: Viewing the current channel lineup in The Buzz. Channels are shown in the middle region. Information about the currently selected channel is shown on the right.

This detailed information is intended to help the user to more easily find a desired channel amongst the list of many channels. While the icons may often help at such disambiguation, the detailed summaries can provide additional aid when necessary.

At the top of this details panel is a preview of the most recent slide generated by the channel. Below this preview is the title and detailed summary of the contents of the channel. Under this summary, the control panel allows the user to show the currently selected channel on the main display, to configure the selected channel, to adjust how often it is shown, or to share it.

The design of this interface is inspired by the design of the icon-based desktop. By mimicking the desktop interface, the design of the system attempts to leverage the skills and capabilities that a competent computer user will already possess. As such, the design attempts to make it easy for a user to easily understand how to perform basic channel management operations: clicking a channel to select it, double-clicking to open and modify a channel, *etc*. (Section 4.8 describes configuring and modifying channels in more detail.)

When the user clicks the "Available Channels" button on the left side of the screen, the view switches from browsing the user's own channels to viewing channels in a shared repository of channels (see Figure 9). This toggle again mimics the desktop interface, through which a user can browse remote files in the network neighborhood using the same interface he would use to browse his own.

Channels shown in the repository are broken into different categories, represented as folders. Within these folders are the various channels available for the user to download. Just as the user can click on one of her own channels to view a detailed summary and preview slide, so, too, can she click on one of these shared channels to view the same summary and preview. The primary difference between the two interfaces is that some of the available operations change their behaviors to accomodate the difference between local and remote channels. For example, the controls to configure the frequency with which a channel is shown disappears, since it does not make sense for a channel not in the current lineup. Additionally, the "Configure" button changes to a "Download" button, and double-clicking downloads the channel rather than configuring it.

In this way, a user can view additional available content directly within the interface, using the same interactions to browse shared channels that she would use to browse her own channel.

Figure 9: Browsing available shared channels in The Buzz. Shared channels are broken into categories, which are represented by folders (top). When a user browses into a category, the channels shared within the category are shown (bottom). Channels with a checkmark indicate that the user already has downloaded that channel.

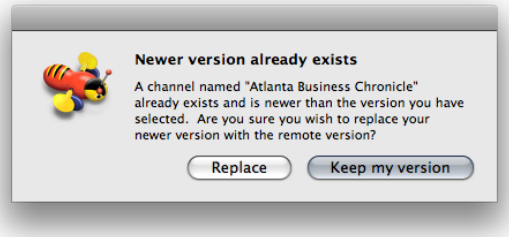

Figure 10: Downloading a shared channel that already exists in the user's current channel lineup.

To help distinguish between new channels and those the user already has, badges on the icons of shared channels indicate whether the user has already subscribed to a particular channel. A user can download a channel simply by double-clicking on it. The system will automatically fetch the channel and install it into the current channel lineup. No additional interaction is necessary. Attempting to download a channel again will prompt the user to confirm whether she wishes to overwrite the channel currently in the lineup with the shared channel (see Figure 10).

Once the user has downloaded the channel through the interface, the channel is a part of her channel lineup, just as any other channel. This browsing interface allows the user to adjust his or her own channel lineup at a relatively coarse level. The user can add and remove channels and can adjust the relative frequencies with which the different channels are shown. In this way, the user can exercise control over the content selection with a relatively straight-forward interface.

4.8 Modifying and Creating Channels

In a study of Unix users, Mackay identified various triggers that encouraged users to customize their software [67]. In that setting, users typically customized their software when the software changed (*e. g.* after software upgrades) and when faced with a software breakdown. Furthermore, in our own interviews with users of customizable software, we found that those participants who customized their software frequently explored software preferences the first time they ran the software and subsequently when faced with a problem to "scratch an itch" (see Section 4.1).

Applying these observations to channels in The Buzz, users are most likely to customize a channel the first time they run the system or when they load a new channel. For example, when a user subscribes to a weather channel, one of the first actions is typically to set the location whose

weather to depict. Additionally, however, users are likely to customize a channel when faced with a particular "itch" when the channel breaks down. Such a breakdown might involve displaying different content than the user desires or displaying it in a different way. For example, a user might grow tired of seeing the same content from an infrequently updated weblog and wish to modify the channel to only show postings made within the past few days.

To address these two different common scenarios, The Buzz provides two primary means of customizing a channel. First, when browsing the channel lineup, as is the case the first time the software launches and when the user subscribes to new channels, the user can double-click to bring up the channel customization interface. Additionally, when viewing the slideshow on the peripheral display, a customize button appears when the user moves the mouse to interact with the system. This button acts as a shortcut past the channel browser and directly into the customization interface for the currently-displayed channel.

When the user begins customizing a channel, the system automatically and transparently creates a copy of the channel for the user to work on. From the user's perspective, he is directly customizing the channel. This copy, however, enables two important capabilities: it is easy for the user to abort the customization and revert to the channel's previous state; and any changes made to the channel can be saved as a copy without altering the original channel. Thus, if a user makes a change he is uncertain about, he can save the channel as a copy instead. By always operating on a copy of the channel and allowing the user to undo or cancel those changes at any point, we aim to foster experimentation on the part of the user.

4.8.1 The Channel Editor

In order to customize the content and presentation of a channel, the user can open the channel editor dialog (see Figure 11). Through this interface, the user can control what content to show, what presentation style to use, and various channel metadata such as its name, description, and icon. There are many attributes of each channel that the user could potentially control. If all of these attributes were to be put into a single dialog, the interface would quickly become overwhelming. Given the complexity of the task, the dialog is already fairly complex. To help reduce this complexity, the dialog makes use of three tabs to partition the tasks relating to the data, presentation, and channel

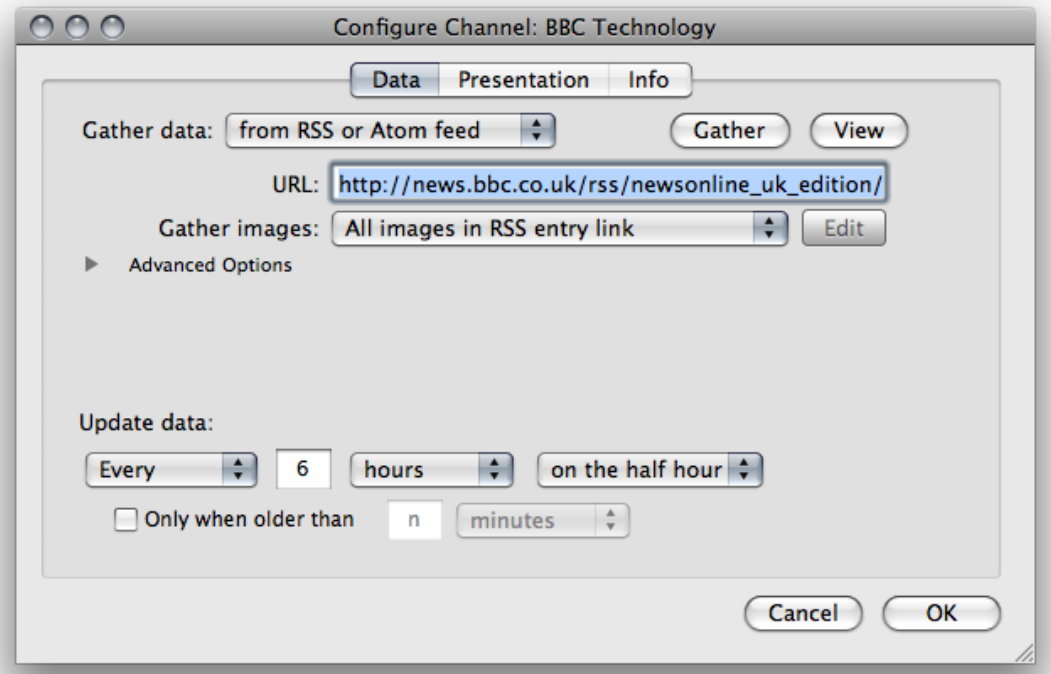

Figure 11: The channel editor in The Buzz.

info. Initially, the data tab is shown.

Most commonly, a user will edit an existing channel—whether to modify its behavior (*e. g.* to show the weather for San Francisco instead of Atlanta) or to create a new derivative channel (*e. g.* to apply the current presentation style to a different data source). Users rarely create a new channel entirely from scratch [66]. As such, when the user sees the dialog shown in Figure 11, the channel configuration is typically in a state that reflects the goals of the user. For example, when the user modifies a Flickr channel to show a different collection of photos, the channel is already configured to extract a particular set of photos from the Flickr photo sharing website and to apply a particular slide layout to those photos. As a result, when the user desires to change the collection of photos, she needs only to change those properties of the channel related to the photo selection criteria.

Channels provide different data gathering methods. Each of these data gathering methods, in turn, defines various properties that are relevant to the method. Thus, a channel that gathers data from Flickr will provide properties for tags, users, and groups, using Flickr terminology. In contrast, a weather channel will provide a property for the zip code of the location whose weather to show.

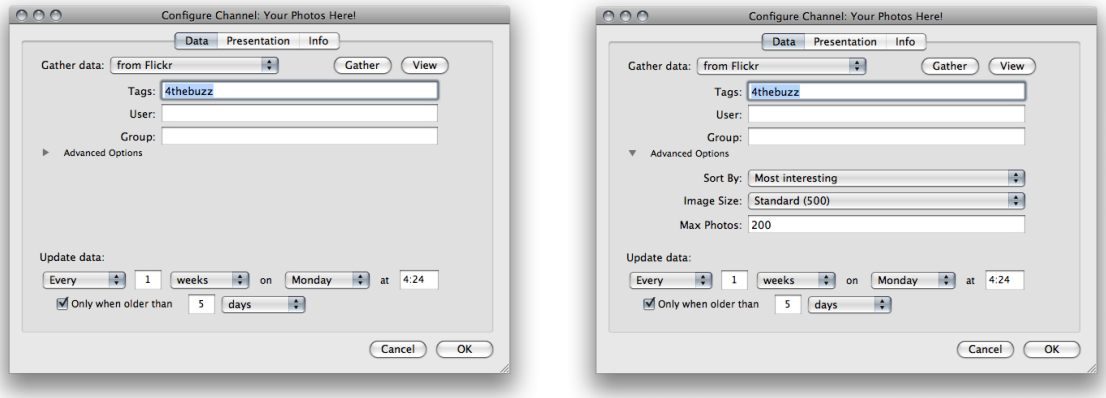

Figure 12: Standard (left) and Advanced (right) options for configuring a Flickr channel.

The underlying data model for this process is described in more detail in Section 4.9.4.

In the previous examples, the channels have gathered data from specific sources using data gathering methods specialized to the particular sources they use. As such, the Flickr channels can use Flickr terminology and the weather channel can use terms relevant to the weather. The user can choose which of these methods to use by selecting from the "Gather data" pulldown menu at the top of the dialog. The particular data gathering method chosen determines the properties that the user may choose. In this way, the number of configuration options for the user to fill in depend on the particular data source being used. Thus, the weather can provide a single property for the user to fill in: the zipcode. Meanwhile, Flickr can provide the user with more extensive control, choosing amongst any combination of tags, users, and groups.

Nonetheless, there may still be additional properties over which the user may desire some control, but which are not strictly necessary for behavior of the channel. The designer must forge a balance between the flexibility of the options over which the user can control and the simplicity of the interface. In many customization domains, fewer options often lead to a more effective user experience and, counterintuitively, encourage more customization.

To address this balance, The Buzz allows some of these properties to be defined as "advanced" properties. As shown in Figure 12, these advanced properties are collapsed under the "Advanced options" disclosure triangle. Thus, if some options may enable finer control over the data for some users, but are not necessary for typical users, they can be hidden under the advanced options section. This technique allows the designer to strike a balance between the simplicity of the customization

and its flexibility.

In addition to these specialized data gathering methods, which can tailor their terminology to the particular data they gather, The Buzz also provides more general-purpose data gathering methods. These methods, such as the web crawler and the RSS gatherer, operate at a lower, protocol level. Because these methods are not tied directly to the content on which they operate, the terminology that they use for configuration is not presented in task-related terms.

Consider, for example, a user who wishes to create a channel of the most recent travel photos from the Flickr user "eaganj." She could use the Flickr gathering method and supply the username "eaganj" and tag "travel." Or she could use the RSS gathering method, supply the URL to Flickr's RSS feed for all of eaganj's photos, and apply a filter to omit any entries in the RSS feed that do not contain "travel." In the first case, the user is able to specify these customizations entirely in terms familiar to a Flickr user: tags and users. In the second case, however, the user must first specify a "magic" URL for the desired Flickr photostream and specify a filter using terminology relevant to the underlying RSS and HTML protocols that encode each photo entry.

Although specialized data gathering methods offer the potential to provide easier interfaces that are better tailored to a particular domain, they do not offer the same degree of flexibility of general purpose methods. For example, to alter a Flickr channel to gather photos from a different photo sharing service may require significant modifications to the underlying channel. Changing a generalized data gatherer, in contrast, could potentially be as simple as adjusting the starting URL and filtering criteria. Thus, while specialized methods can provide the user with a simpler interface, a generalized method can be simpler to repurpose.

Although specialized methods may be able to provide the user with a more intuitive interface, they do also present a more complex management problem. If the system provides a few general purpose data gathering methods that handle most of the cases that the system might encounter, the user only needs to choose from that small number of gatherers. If, however, the system provides specialized methods, it will need to provide such a specialized gatherer for every potential data source. The user would need to choose amongst a large number of data sources, necessitating a higher level management scheme for gathering methods.

As such, there are tradeoffs to be made between general purpose and specialized data gathering

methods. Specialized methods offer the benefit that they can provide a user interface that more closely aligns with the user's task, but they take more effort to develop and manage, and they are difficult to repurpose. General purpose methods offer greater flexibility and can be more easily repurposed. Because of their generality, a relatively small number of such methods can resolve many tasks. But they are also more difficult to customize and may require fairly technical expertise to customize.

4.9 A Tale of Three Channels

To help illustrate how the user can create and configure a channel, let us consider three example channels. In this section, we will create three channels from scratch to show the various options the user can configure and how they interact. The subsequent sections will then describe some of the underlying models on which these interfaces are built. There is a chicken-and-egg problem in this discussion, in that the underlying models do not necessarily make sense without understanding the interfaces they support. Similarly, the interfaces depend on the underlying models. Nonetheless, this interface-first approach should help illustrate the underlying models, which will later help to reinforce their design.

4.9.1 Webcams

One of The Buzz's popular channels is the "My Webcams" channel, which displays a collage of various webcams from around the world (see Figure 13). The channel maintains a list of webcams that the user can modify and allows the user to change the layout of the various webcams, such as to change the display from a collage of multiple webcams to a single one at a time. In this example, we will examine how the user can customize this channel by adding a new webcam and alter the presentation to show a particular layout.

A version of the "My Webcams" channel is included in the default channel lineup for The Buzz. To configure what webcams the channel displays and the presentation it uses, the user can double-click on the channel in the channel browser or click on the "Configure..." button, bringing up the channel editor window, as shown in Figure 14.

In this figure, the user has clicked the + button to add a new webcam to the list. This task is complicated by the fact that many webcam publishers make the camera image available in different

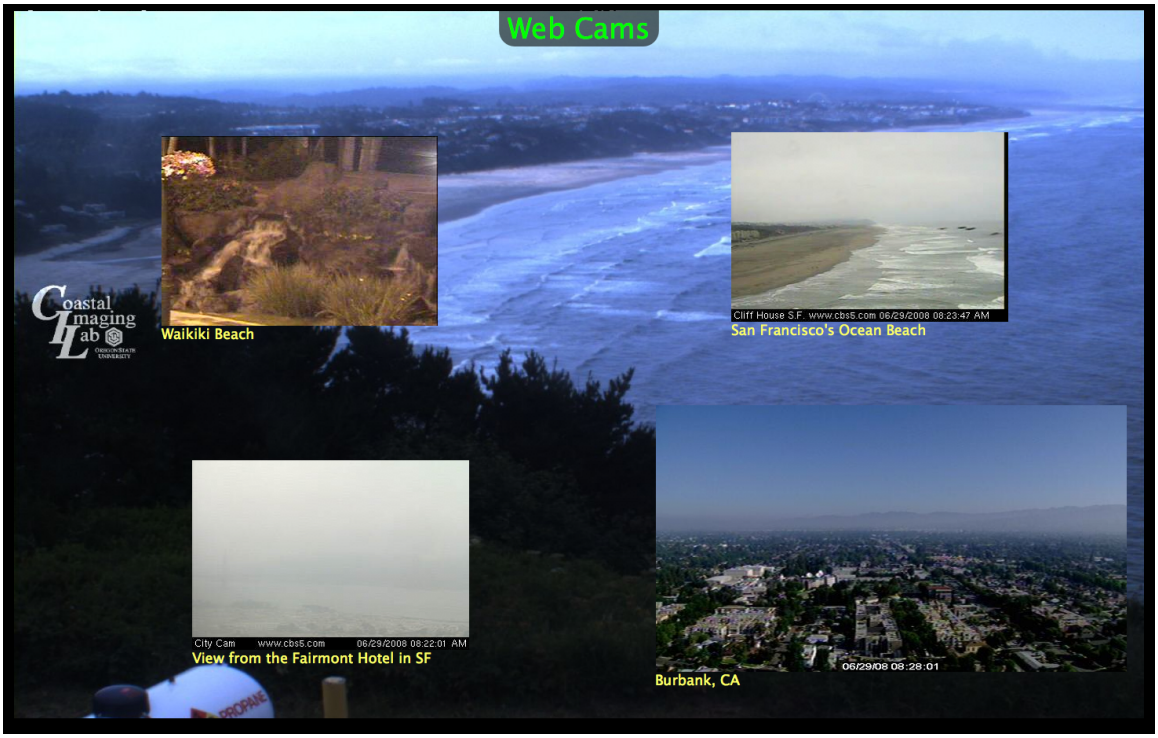

Figure 13: The "My Webcams" channel in The Buzz.

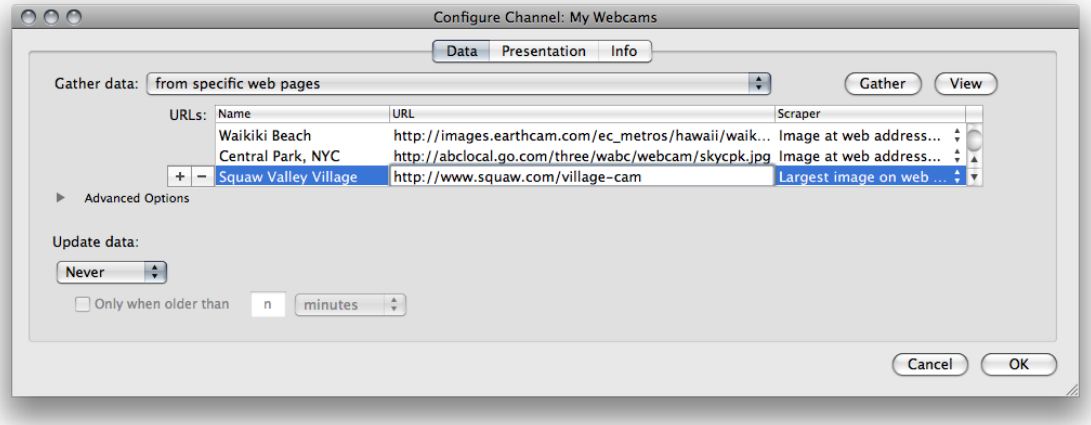

Figure 14: Editing the list of webcams used by the "My Webcams" channel.

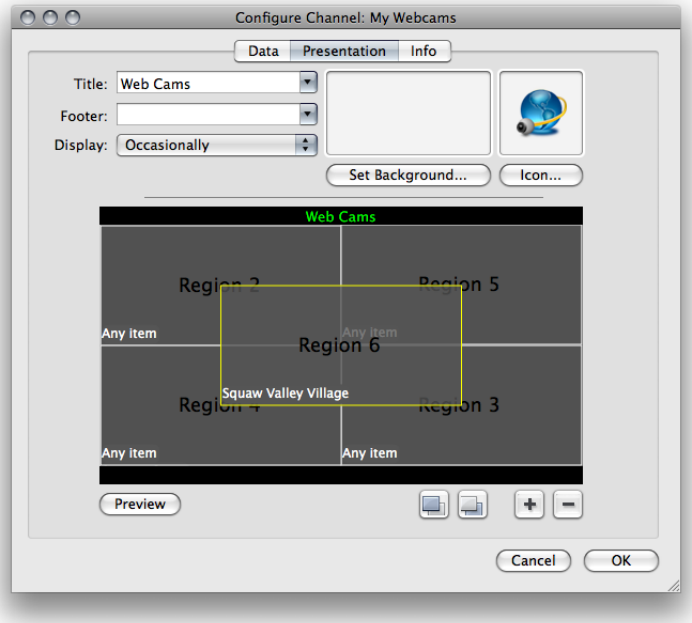

Figure 15: Modifying the presentation of the "My Webcams" channel. The interface shows a thumbnail mockup of the screen, comprising several regions the user can position and resize.

ways. Sometimes they may publish to a static URL, the contents of which update to show the most recent image. The webcams at the top of the list simply refer to the image at the particular address for the desired webcam.

Other webcams, however, use a different URL for each image and simply change the URL that the embedding web page uses to reference the most recent image. This technique is often combined with JavaScript or a CGI backend. In this case, there is usually a webpage with a static address, but which embeds the image from the appropriate URL. In this case, a heuristic of grabbing the largest image on that web page is usually sufficient. As such, when we add this new webcam, we specify the URL of the web page that embeds the camera image and specifies that it should gather the "Largest image on web page."

After adding the new webcam to this list, we will modify the presentation. Instead of showing a collage of five random webcams, we wish to see our new webcam displayed on every collage along with four others, chosen at random. To modify the presentation, we select the "Presentation" tab at the top of the screen. When we change tabs, the system will prompt us to reload the data, because we have updated the data configuration.

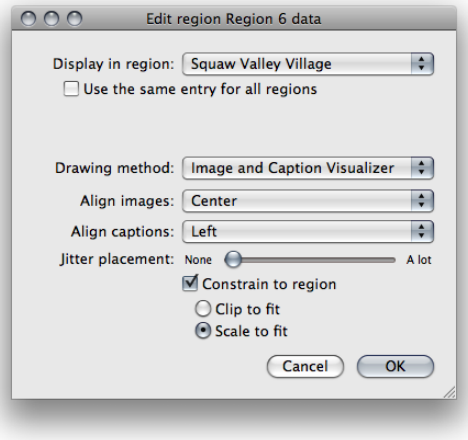

Figure 16: Viewing the configuration for a region.

Figure 15 shows the presentation editor with the existing collage-like layout, which defines six regions: a large, full-screen region bound to the Oregon Coast webcam (to act as a background); and five smaller regions, each bound to a randomly chosen webcam. We will modify the middle region to display the new webcam.

To modify the region's properties, we can double-click on it, revealing the region editor, as shown in Figure 16. At the top of this dialog, we can select which data entry to display in this region. The default configuration binds this region to "Any item," which selects an entry at random. This pulldown menu is populated by each of the webcams added under the "Data" tab. By changing this to our new webcam, as shown in the figure, we can change what is shown in the region. The other options in this dialog allow us to change how the data are drawn. We will leave them as-is.

4.9.2 Digg

Digg is a social bookmarking community in which users submit links. The submitter includes a link, a title, and a brief description of the link. People can vote in a thumbs-up/thumbs-down fashion to promote or demote these submitted links. Additionally, people can discuss the links through comment-areas associated with each link. In this example, we will create a channel to display the articles on the front page of Digg.com, which contains the most highly dugg articles. Furthermore, we would like to see any related images that might be embedded in the dugg page. Figure 17 shows our goal.

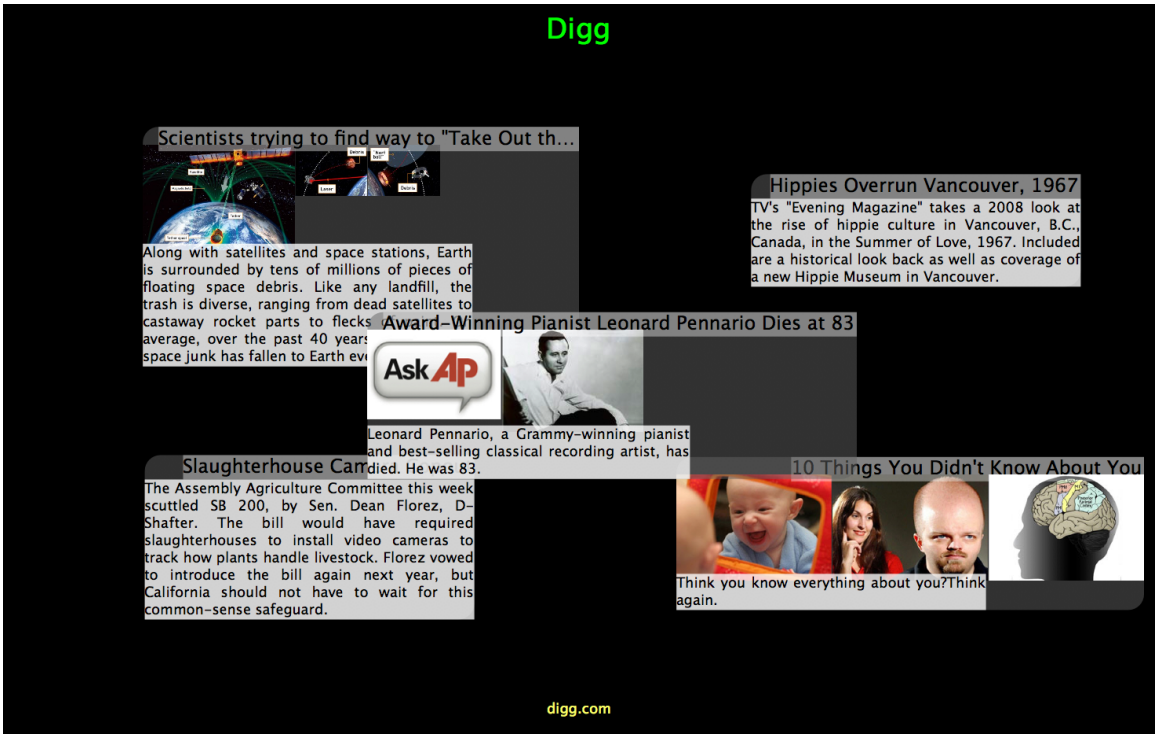

Figure 17: The Digg channel in The Buzz.

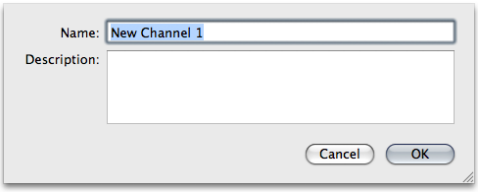

Figure 18: Setting the name and description of a new channel.

We will begin by creating a new channel from scratch. To create a new channel, we can click on the + button in the channel browser window (see Figure 8). When we click this button, The Buzz will first prompt us to give a name and optional description of the channel (see Figure 18). Either of these can be changed at any time.

The default gathering method for new channels is the RSS harvester. The Buzz is capable of gathering data using a variety of methods, of which RSS is just one. Because RSS is the easiest to configure gathering method that supports a wide variety of data sources, it is the default. These gathering methods are discussed in more detail in Sections 4.9.4 and 5.3.

To configure the data, the user need only specify the URL for the RSS feed provided by Digg. Most website operators make this RSS link readily available on their web pages, designated by

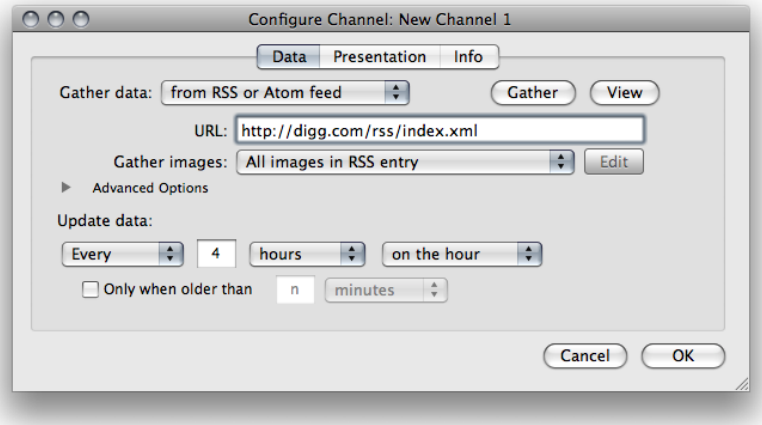

Figure 19: Configuring the Digg data gatherer.

either the label "RSS" or with a bright orange icon. The user can simply copy and paste this URL into the configuration dialog.

This configuration will gather the titles, descriptions, and links to all of the Digg articles on the front page, but because the feed does not include any images, it will not include any of the images from the dugg articles. It is fairly common for publishers not to include any images in their RSS feeds, even if there are images associated with the linked article.

Many publishers view their RSS feeds as "teasers" to lure readers to their web sites, which may contain the full content of the article (and where they can generate ad impressions). As a result, many RSS entries contain only a brief snippet of a linked article. These summaries are usually ideally suited to the peripheral nature of The Buzz, where the user can follow up to the full article if she is interested. But the presentation style of The Buzz is optimized toward image-oriented content. To address this situation, The Buzz can gather all of the images embedded in the web page linked by an RSS entry instead of just the images embedded in the RSS entry itself. Thus, the user can change the "Gather images" pulldown from "All images in RSS entry" to "All images in RSS entry link."

For most cases, this change would be sufficient. However, Digg is different from most websites. Rather than producing its own original content, Digg aggregates this content and augments it with its users' ratings and commentary. As such, the link in the RSS entry is not to the original article, but to the Digg page that describes the original article. In order to gather the images associated with

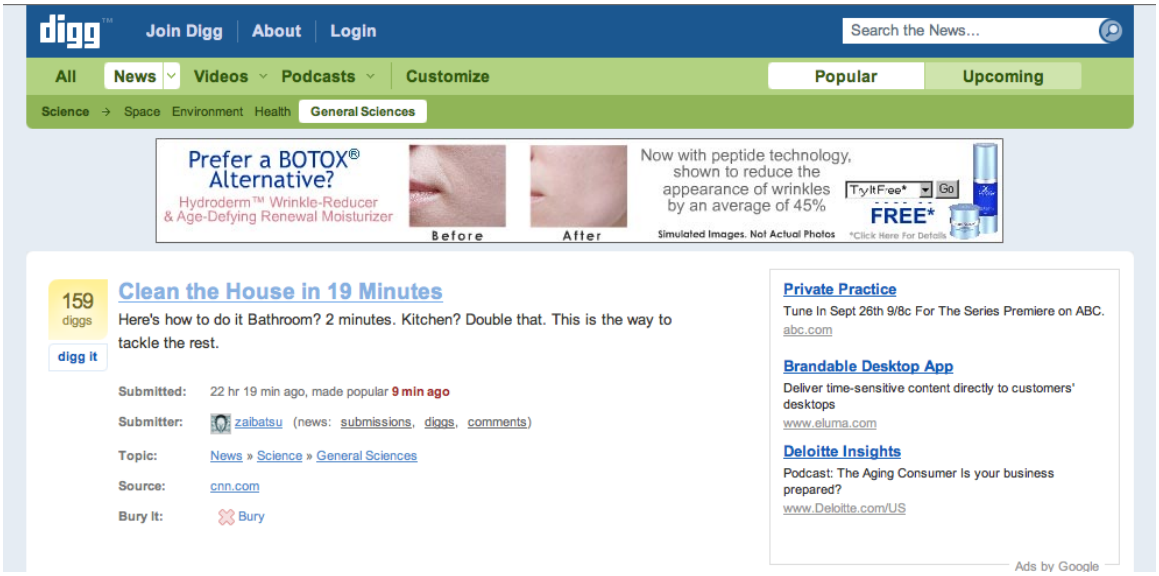

Figure 20: A Digg article summary page. The link to the actual article itself is located in the lower left, next to the "Source:" label.

the original article, we will need to add an extra layer of indirection, by following the link from the feed entry to the Digg page, finding the link within this page, and gathering the images from that page.

Unfortunately, there is no obvious generic way to find that link on the Digg page (see Figure 20). To extract the link to the source article, the user must define an extraction pattern to find this link on an arbitrary Digg article page. To handle this case, we must change the image gathering method to gather "Items from a pattern-matched link on entry web page ...," which allows the user to specify an extraction pattern and an image gathering method (by default, "All images (with captions) on web page"). The extraction pattern will be applied to the RSS entry's linked web page (the Digg article page) and the image gathering method will be applied to the web page that matches the extraction pattern.

Writing extraction patterns is a challenging task. Pattern matching languages involve nuance and subtlety. Even though these pattern languages do not contain branching or looping, they are effectively a form of (non-Turing-complete) programming language. To simplify the task of writing these patterns, The Buzz provides a pattern editor, as shown in Figure 21.

As the user enters a pattern, the pattern editor displays the matches found or any syntax errors in the pattern. This immediate feedback provides for a tightly-coupled feedback loop in which the

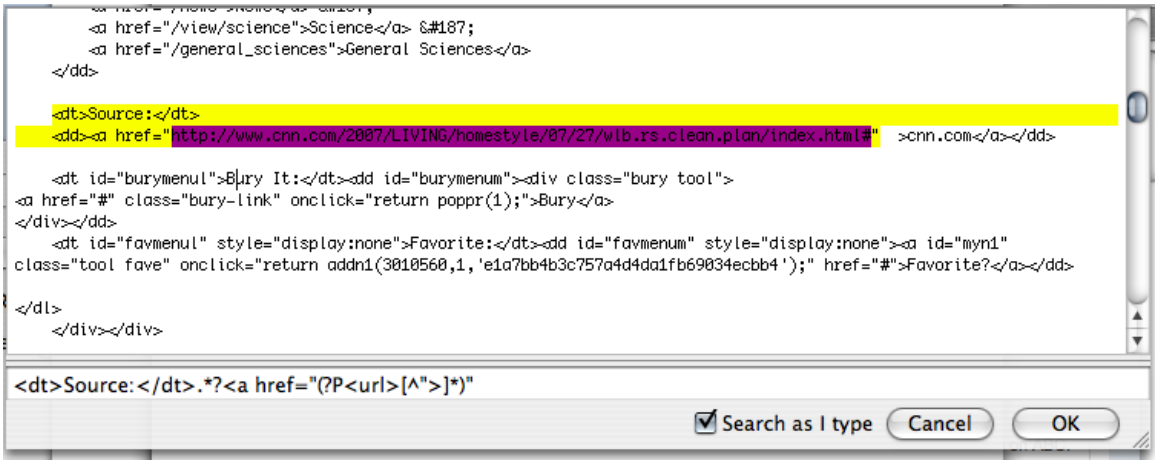

Figure 21: The pattern editor in The Buzz. The top portion of the window shows the HTML source to which the pattern will be applied. As the user types a pattern in the lower portion of the window, the top portion updates to show matching text, or any syntax errors that might exist in the expression.

effects of the pattern are dynamically displayed. Thus, the user can more easily and quickly identify any syntax or semantic errors in the expression as it is being written. While this feedback will not instruct the user as to what pattern to enter, it will help him to refine it and more easily recognize bugs in the pattern.

By providing this extensive scaffolding around the pattern writing task, The Buzz aims to narrow the context in which the user must program. Without incorporating this pattern matching capability, we would need to give up on gathering images from the source article, or we would need to write a plugin to handle this special case. Instead of writing a complete plugin to implement this data gathering method, we can write a comparatively simple extraction pattern.

With this pattern, The Buzz will now follow the link embedded in the RSS entry, apply our matching pattern to the Digg article page, and extract all of the images on the web page loaded from that match. We now need only to configure our presentation. In our case, the default presentation template for the RSS data gatherer will be sufficient, yielding the slide in Figure 17.

4.9.3 NSF News

In the previous example, we explored some of the power that The Buzz provides in handling the various complexities that publishers may cause in how they make their data available. In this last example, we consider the flexibility that The Buzz affords in allowing the user to selectively determine

how much work she might expend in creating a channel.

The U.S. National Science Foundation (NSF) publishes a collection of news articles on its web site. Because the NSF is a broad organization with many different scientific areas under its umbrella, the web site allows one to filter these articles by NSF priority area. In this example, we will attempt to create a channel that displays news articles (and their associated images) in the Computing & Information Science & Engineering (CISE) priority area.

There are a several ways to create this channel. The easiest is to use their RSS feed. Unfortunately, while the web site breaks the articles down by priority area (including CISE), the RSS feed does not. As such, it would be difficult to use RSS to create a channel just for Computing news. This particular limitation may not be a significant problem in that general scientific news is likely to be of interest to computer scientists. Nontheless, the goal is to create a CISE news channel. Creating a broader NSF news channel would be a (merely acceptable) compromise.

The RSS gatherer does, however, provide support for filtering entries. Using this filtering capability, we might able to configure the gatherer to ignore any entries in the feed that are not in the CISE topic area. Filtering by topic, however, is complicated by the fact that neither RSS entry nor the linked web page provides any indication as to which priority area(s) the article is in. The only indication of where in this hierarchy the article lies is through the web site itself, which does allow for a human-readable filtering of the articles by topic area.

In order to filter the RSS entries by priority area would then involve comparing the articles in the feed with the articles on the web page. If an article in the feed is also linked by the CISE news web page, then it should be accepted; otherwise it should be filtered out. This filter, however, is further complicated by the fact that the NSF uses different link formats for the articles from the web site versus from the RSS feed, even though either format will refer to the same article. Thus, such a filter would need to dissect the relevant components from the URL and perform its comparison against those components. The only way to define such a complex filter in The Buzz is to write it in Python code and load it as a channel plugin. If a user were especially motivated and capable, she might be able to write such a filter. For all intents and purposes, however, this approach is not well supported by The Buzz.

The filtered list of articles may not be available through the RSS feed, but it is available on

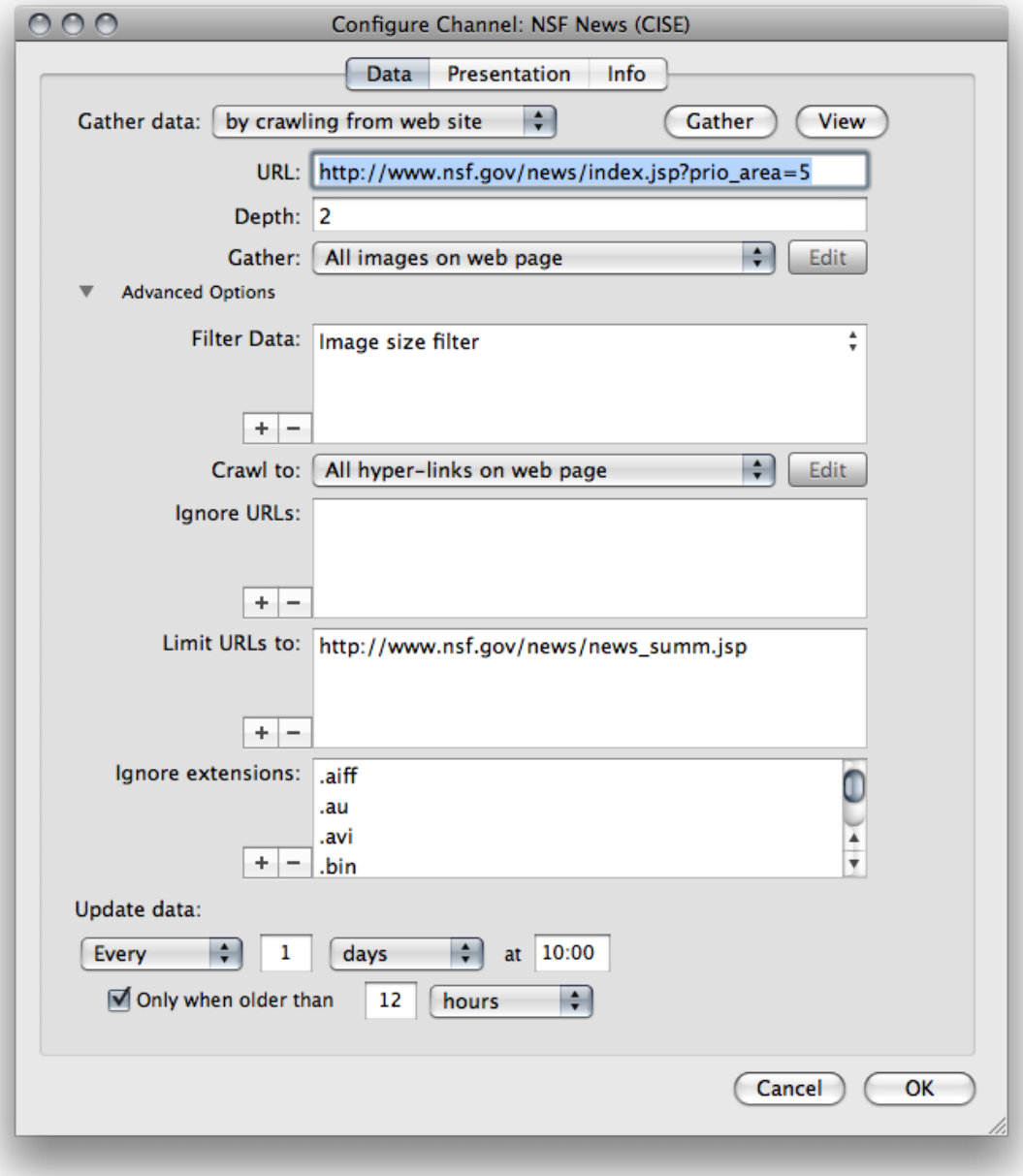

Figure 22: Configuring the CISE News channel to gather CISE-related news articles from the NSF. The advanced properties at the bottom of the screen are normally collapsed until needed.

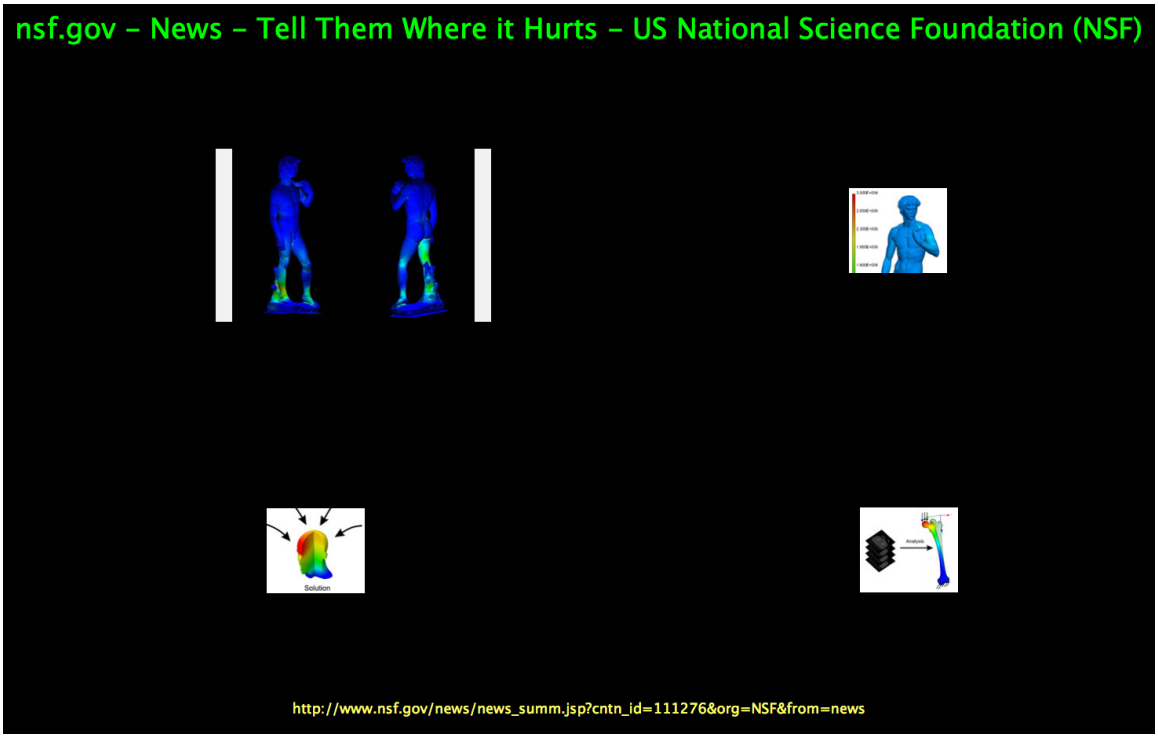

Figure 23: Images extracted from the CISE section of the NSF News website.

the web site. Thus, we may still be able to create a CISE news-only channel using a web crawler. Figure 22 shows the configuration for this web crawler. By setting the start URL to be the CISE news index page and a depth of 2 (meaning that the crawler will traverse the start URL and any pages linked from it), we can crawl all of the news articles linked from that page.

This crawl, however, will still collect too much data. It will include any page linked from the index page, not just links to articles. As such, we will need to limit the articles to those accessed through `http://www.nsf.gov/news/news_summ.jsp`. Furthermore, we do not wish to include header graphics or button, so we can set a filter to ignore images smaller than a certain size, say 100 by 100 pixels. We can control these behaviors through the advanced properties of the web crawler, as shown in Figure 22

This web crawling method is well suited to creating indexes of images on web sites because it is relatively straightforward to identify images in HTML documents. It does not, however, extract any of the text from the articles. As such, this approach allows us filter the news articles down to CISE news only, but it also eliminates the text of the articles, as can be seen in the slide in Figure 23.

In, instead, we wanted to gather not only the images but also titles and descriptions, similarly

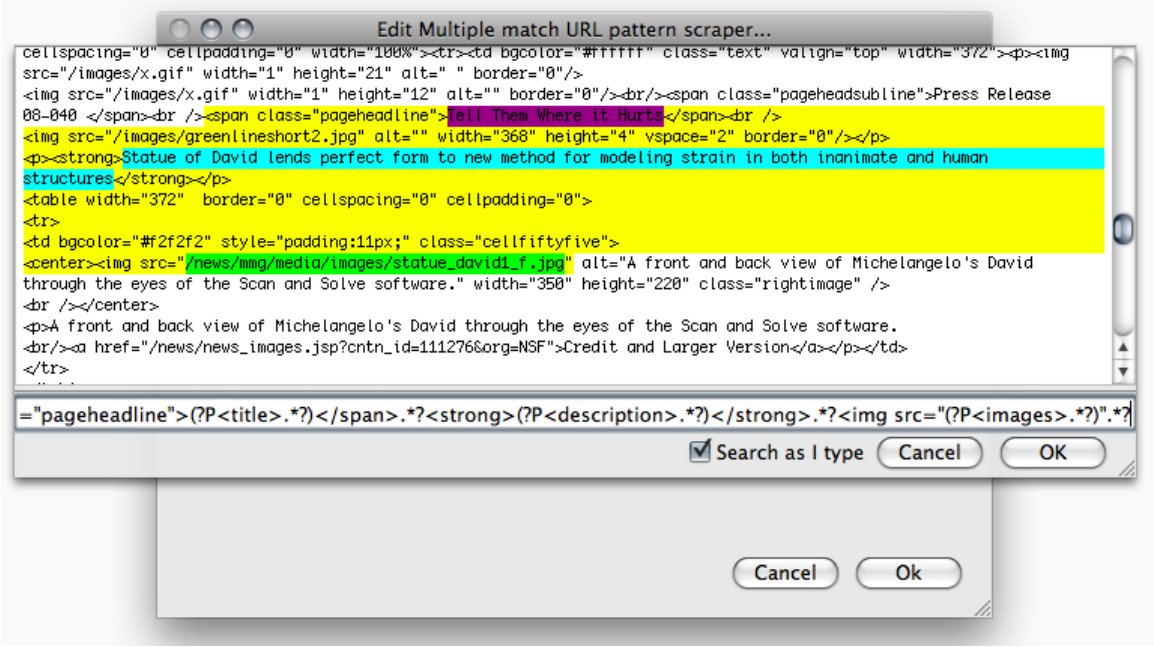

Figure 24: Creating an extraction pattern for NSF News articles.

to what the RSS gatherer would extract, we will need to write our own extraction pattern to gather these data from the article page. This technique requires significant effort, so if we are content with extracting either all of the news articles, regardless of whether they pertain to the CISE focus area, or just extracting the images without the descriptions, then we can limit ourselves one of the earlier methods. If we truly wish to create a CISE News with summaries channel, however, we will have to create an extraction pattern[3], as shown in Figure 24.

As this example illustrates, content publishers often make their data available in ways that are more or less amenable to data scraping techniques. Even when publishers are thoughtful enough to provide RSS feeds for their content, these feeds may not necessarily align with the user's goals. Adapting to all of these special cases can be cumbersome.

Through this example, we have seen three different approaches that a user could take to create a CISE News channel (see the final result in Figure 25). The first is to simply settle for the content that is already available in an RSS feed, even though it includes extraneous data. If our user truly desired only to see CISE news, she could expend a little more effort to configure a web crawler to extract

[3]Inference-based extraction pattern-generation schemes, such as used in [52, 108, 32], could obviate the need for manually writing such patterns in many cases. The Buzz, however, does not currently support these techniques.

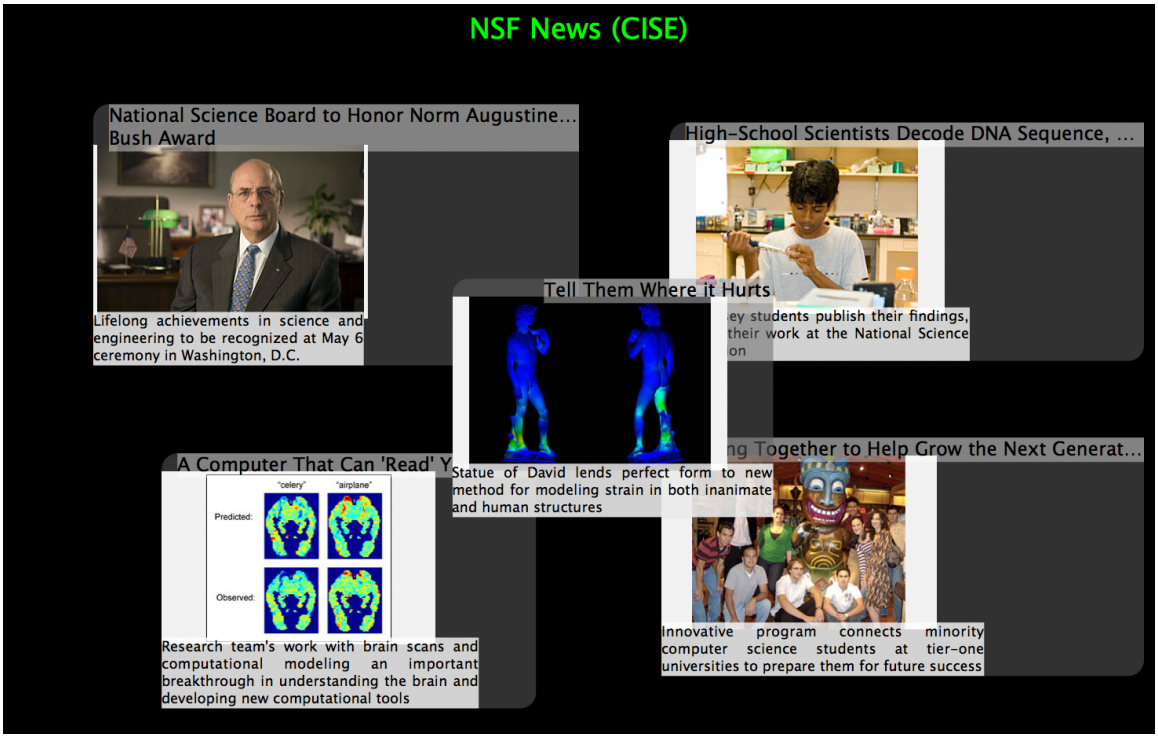

Figure 25: The CISE News articles channel, including article summaries.

only the images from the CISE news articles. Or with a bit more effort, she could write a pattern extraction rule or even a custom filter plugin. Through this variety of methods, The Buzz supports flexibility in the degree of customization that a user can perform, depending on the combination of the particular data desired and the effort the user is willing to expend.

4.9.4 Channel Data Model

In order to understand how the data and presentation customization interfaces integrate, it is useful to consider the underlying data model used by The Buzz. This data model reflects the data pipeline (see Section 2.2), with various components to perform each of the stages of the pipeline. Thus, where the data pipeline defines stages for *extraction*, *transformation*, a *cache*, and *visualization*, the channel model defines *harvesters*, *scrapers*, *filters*, a *database*, and *visualizers*. As shown in Figure 26, the harvester is responsible for gathering data from the content publishers and transforming them into a format suitable for the intermediate database. The extraction itself is performed by the scrapers, while the filters apply any transformations (including to the empty set, $\emptyset$).

One of the goals of this data model is to provide flexible support for a variety of different types

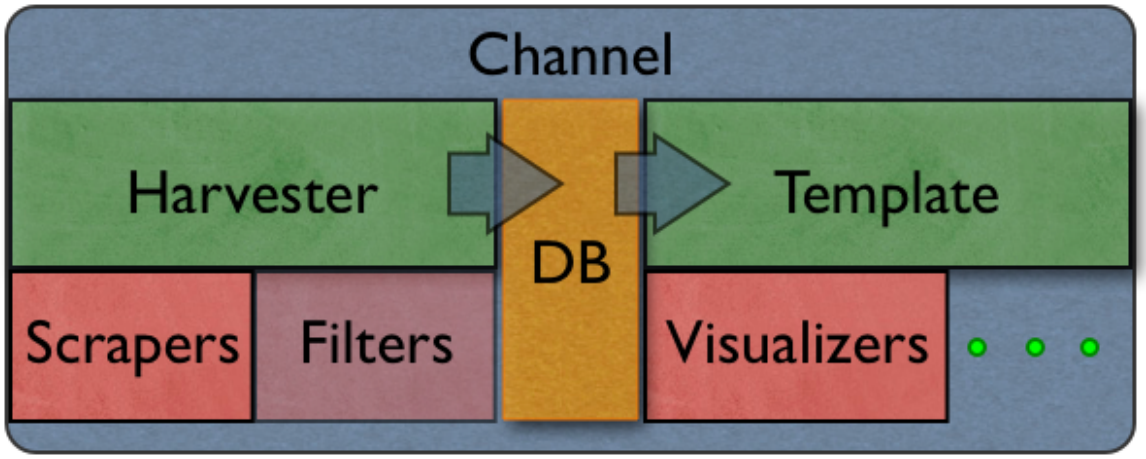

Figure 26: The Channel model in The Buzz.

of data. Content publishers use a variety of different formats to encode their data, depending on the kind of data they make available and their intended use model. For example, most publishers on the web use HTML to encode their data, but are increasingly making their content available in more interactive, richer presentation formats such as Flash, Java, and JavaScript. Furthermore, how those data are encoded on the web page varies. Some publishers may take advantage of the spatial layout of HTML to encode their data, where the positioning of content on the screen is important. Others may simply use HTML as a branding mechanism to give their content a particular look and feel. Finally, not all data are on web pages. For example, much useful data can be found in email accounts, spreadsheets, and local databases (*e. g.* iPhoto).

All of these different approaches to content publishing influence the task of gathering the data and extracting sufficient semantic structure to repurpose them for presentation within an awareness system. The goal of The Buzz's data model is to provide sufficient flexibility to handle a wide variety of content, but simultaneously to scale the complexity of the customization interface with the complexity of the user's task.

In order to handle a wide variety of these data formats, The Buzz provides various data gathering methods, called harvesters. Some of these harvesters are specialized toward specific data sources. For example, the Flickr harvester accesses the Flickr APIs directly to specifically support extracting content from Flickr's database, and the Weather harvester accesses the U.S. National Weather Service's SOAP interface to extract local weather forecast data. Others of these harvesters are more

Figure 27: Configuring the presentation of a channel with potentially stale data.

general purpose in nature, crawling web pages or parsing syndication feeds (*e. g.* RSS, Atom).

The general purpose harvesters often perform the same high-level task but with different specific behaviors. For example, crawling a website to extract all of the images on a photo gallery and crawling a website to extract people's profiles from an online directory both involve crawling a website, but performing different actions with the individual web pages. To accomplish these tasks, the harvesters allow the user to specify what content to extract. Thus, the user can configure the web crawler to "Gather all images on web page," implicitly specifying a *scraper* that will perform the actual content extraction. (Section 5.4 describes scrapers in more detail.) Additionally, the user can apply *filters* to modify the web pages or content that are extracted during the gathering process.

4.9.5 Channel Presentation Model

At the point that the user customizes the presentation, it is assumed that the data have already been configured. This assumption is subtly implied by the order of the tabs at the top of the screen, with the data tab first, followed by the presentation tab. This particular order is important for several reasons. First, the order reflects the mental model implied by the data pipeline, where data is first gathered from the various content publishers before eventually being presented to the user. Second, by loading the data first, the interface can inspect the data to tailor the presentation options available to those that are relevant to the underlying data format. Thus, for example, the interface can omit configuration options that require images if a particular data source is purely textual.

By requiring the data to be pre-loaded, the interface can present the user with data-aware configuration options, but this requirement does come with some cost. Most significantly, the data must be loaded in order for this inspection to behave properly. The first time that a user customizes a

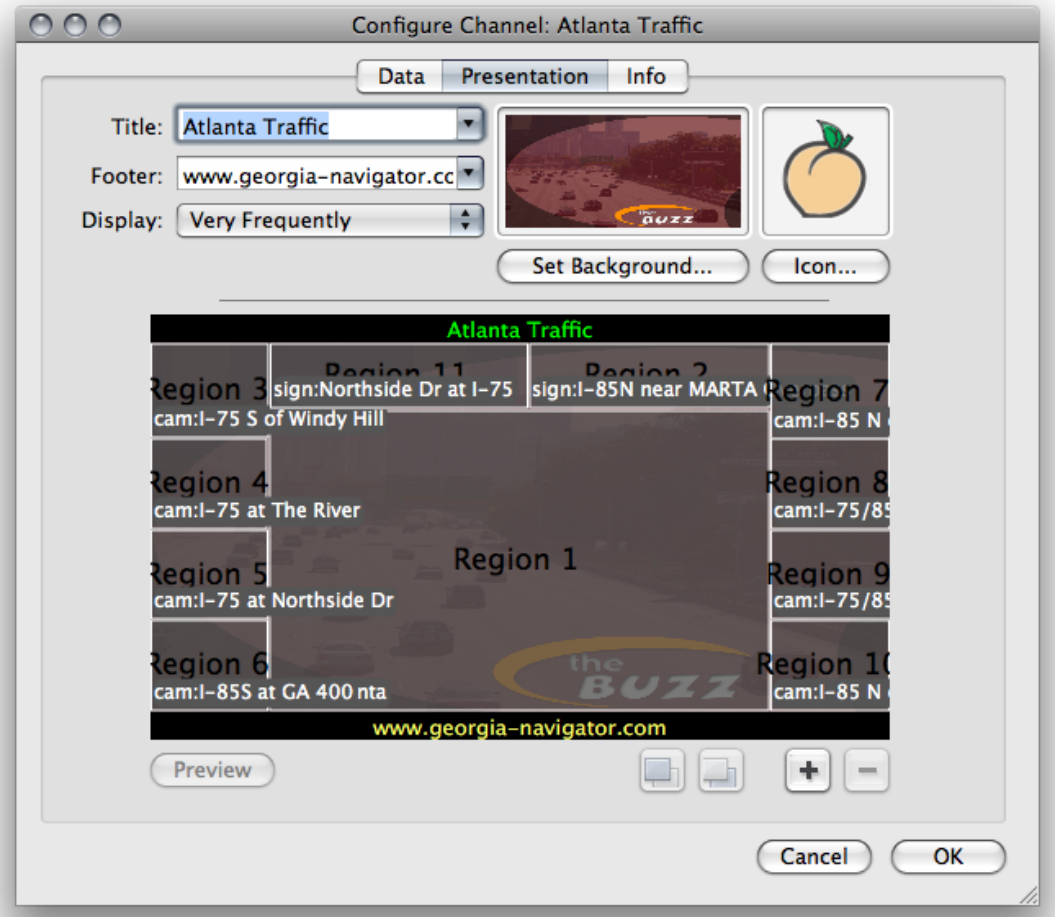

Figure 28: Customizing the presentation of a channel in The Buzz.

channel, however, no valid data will have been loaded. As such, the system must make the user wait to customize the presentation until it has had a chance to load the data. For some data sources, this data gathering process may take several minutes. As a result, if data are loaded, but may be potentially stale (as the result of a change to the harvesting process), the system will allow the user to continue without pre-loading the data (see Figure 27). If the changes the user has made do not alter the overall structure of the data, then continuing with stale data is safe. Otherwise, the data-driven customization may cause inappropriate configuration options to be shown.

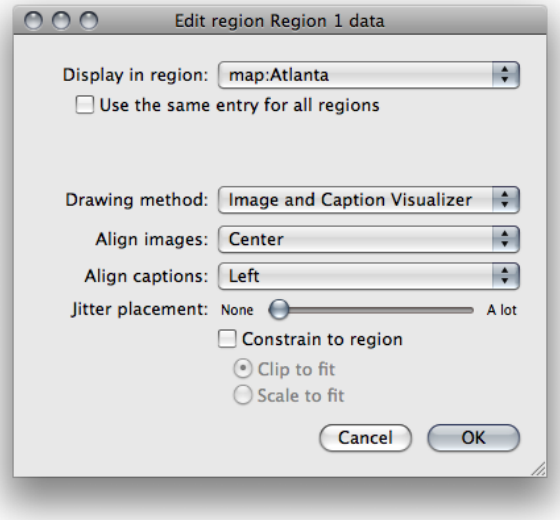

Figure 29: Editing the bindings for a particular region in a channel's presentation template.

4.9.6 Layout Templates and Regions

The Buzz attempts to provide a simple but flexible presentation model for the user to control how the data for a particular channel are shown on the screen. To accomplish this goal, The Buzz uses the notion of a layout template, through which the user sketches out the structure of the presentation. Through this mechanism, the user can specify which data to draw on which portion of the screen and how to represent them (see Figure 28).

In order to define the presentation of a slide, the user must specify one of these layout templates. Each template represents the screen space used for a single slide, in device-independent coordinates. The template itself is comprised of various rectangular regions, where each region designates the space for a single data entry. The user can create new regions by clicking on the + button (see Figure 28). She can then drag the region around on the screen using a direct-manipulation [93] style interface, repositioning and resizing the region as desired.

By adding and positioning regions on the screen, the user specifies placeholders into which different data elements will be drawn. In this way, he can "sketch out" the structure of the presentation. To specify which information should be drawn in which region, the user can double-click on a region to edit its *bindings* (see Figure 29). These bindings allow the user to define various relationships between the various regions and the data. Regions can be bound to a particular data

entry (*e. g.* the five-day weather forecast or the freeway traffic sign at 10th St) or to a random data entry (*e. g.* some photograph in a photostream or some news article). Additionally, regions can be bound in such a way as to share a single data entry. Thus, if an RSS entry, for example, includes many photographs, different regions can depict the different images from the same entry instead of depicting different entries in each region.

This binding process can be tricky for users, since it involves such a low-level detail and a kind of thinking that people typically do not exhibit in normal practice. As such, the different harvesters define default templates and default bindings that attempt to provide what most people might expect given the data. The RSS harvester, for example, provides a layout template the conveys four news articles at a time in a tiled fashion (plus some jitter for aesthetic reasons). The web crawler, by default, uses a collage-like layout to generate collages of all of the images from the same page (data entry). Through this defaults mechanism, users do not *need* to edit the data bindings for their templates, but they may do so if they wish to.

In addition to binding which data are shown in which region, the bindings also let the user control the visualizers used to depict the data. By default, regions use an automatic binding scheme whereby the visualizers inspect the data to determine the best visualizer, if any, to represent them. Thus, a user need not specify a specific visualizer unless she wishes to override the automatic mechanism (*e. g.* if multiple visualizers are capable of depicting the data and she wishes to use a specific one). Finally, the visualizers can define configuration options to control their behavior. Through this region editor, the user may adjust these properties to control scaling behaviors, jitter, *etc*.

4.10 Sharing Customizations

Section 2.1 described an ecology of users in which individuals bring with them different sets of skills and motivations. In this ecology, not all users are alike, and not all users perform the same kinds of customizations. Despite these differences in skills and motivations, or perhaps because of them, various patterns of sharing have been identified amongst users of customizable software [66, 40].

Various systems have taken it as an expressed goal to foster this kind of sharing. The Buttons project [68], for example, explicitly aimed to foster a tailoring community. The Buzz strives to

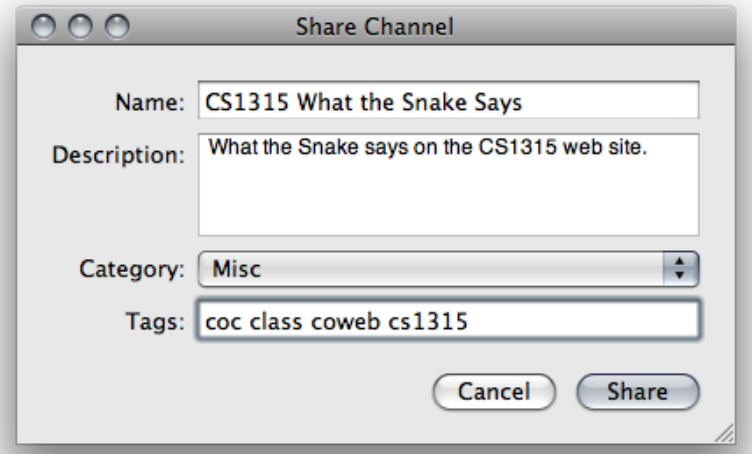

Figure 30: Sharing a channel in The Buzz.

promote such a community of customization and sharing by including integrated sharing support directly within the interface. In this way, sharing and finding customizations should be as easy a task as is feasible.

As such, sharing a channel in The Buzz is a fundamental operation on a channel. In the channel browser interface (see Figure 8 in Section 4.7), the user can share any channel in her channel lineup. Next to the "Configure" button is another button, labeled "Share...." The first time the user selects this button, she is prompted for her login credentials or to create them if she has not already done so.

She is then prompted to update the channel's name, description, and to supply any tags to categorize the channel, as shown in Figure 30. To make this process as lightweight as possible, these fields (with the exception of the tags) are filled in with the channels current name and description. When the user clicks the "Share" button, The Buzz automatically packages up the channel, compresses it, and uploads it to the shared repository, making it immediately available to all other users of The Buzz.

While The Buzz attempts to make it easy to share channels directly within the interface, this mechanism is not the only way for a user to share his channels. All channels are stored in a folder in the user's Documents folder. (See Figure 31.) The user can share any channel by finding it in this

Figure 31: The Channels folder containing all of the user's subscribed channels.

Figure 32: Browsing shared channels through the web-based interface.

folder and sending it to another user via email, ethernet, sneakernet, or any other desired method.

Only channels shared from within The Buzz, however, will be available to all other Buzz users through the channel browser interface. This interface integrates with a central repository of shared channels. Through this interface, the user can, directly from within The Buzz, browse through a listing of categories and channels made available by other users.

In addition to this built-in browsing mechanism, however, users may also browse this shared repository using a web browser. (See Figure 32.) The repository exposes its interface in two forms: a JSON encoding used by The Buzz client to query the repository, and an HTML representation of the same JSON data. Thus, a user can browse channels and even send standard web links to any channel in the repository.

By providing this flexible approach to sharing channels, we aim to reduce the barriers to sharing. Users should not feel that it is a difficult task to share their content. By making it easy to browse shared channels both within the interface and via a web browser, we hope to reinforce the sense

of sharing, that the existing channel collection is a living, breathing entity to which any user can contribute.

CHAPTER V

THE BUZZ ARCHITECTURE

In this chapter, we examine the low-level system implementation of The Buzz. The challenges and solutions that arise may be valuable for designers of future systems, but can be safely skipped by others.

In order to provide a flexible and customizable approach to gathering, manipulating, and presenting data to the user, The Buzz uses a modular design. We have already seen most of the components of this architecture in our examination of the system. This section describes how how these components fit together and discusses the primary components of the system in more detail.

Figure 33 shows an overview of The Buzz architecture. The central unit of this model is the channel. The channel embodies the implementation of the data pipeline for a particular informative presentation, or slide, shown by the system. These channels use harvesters, scrapers, filters, visualizers, and a database to implement the various stages of the data pipeline. To coordinate the operation of multiple channels, the dispatch operates as a supervisor to control when each of the various channels should update their data and/or presentation. These components are described in more detail in the remainder of this section.

5.1 Database

The channel database serves as a cache, allowing the channel to refresh its display without requerying the content publisher. This cache allows the screen to redraw quickly and efficiently. Furthermore, it also allows the channel to display different subsets of the data without reloading the entire data set. For many data sources, the harvesting process is time consuming. For example, generating the BBC World News channel involves processing the web page and images for each article. Repeating that process for every data source used, every time a slide is shown is wasteful. The database allows these data to be stored until explicitly refreshed.

Not only may the harvesting process be time consuming, but it may also be expensive. For example, the Flickr harvesters use Flickr's API, which imposes a limit on the number of API queries

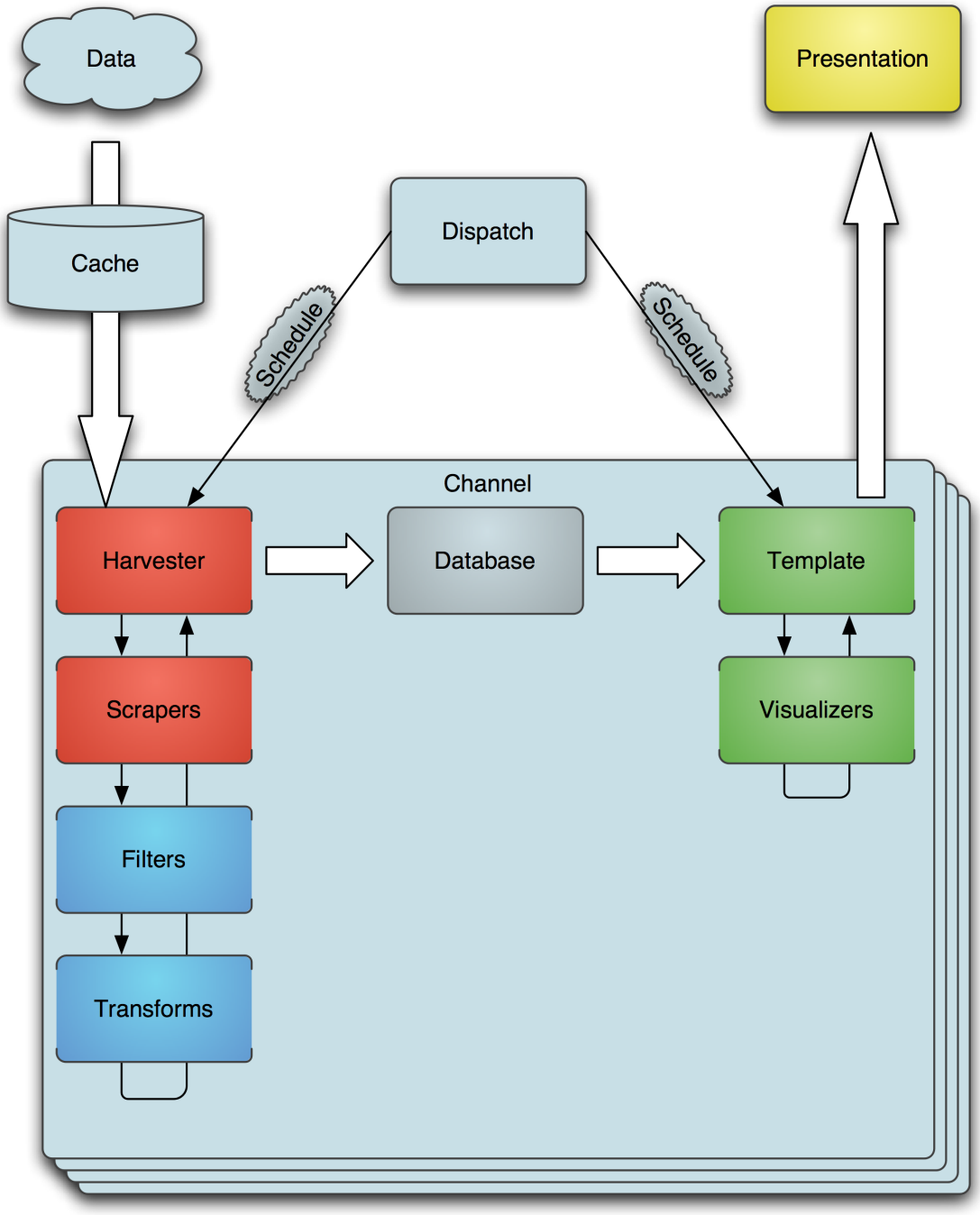

Figure 33: The Buzz Architecture

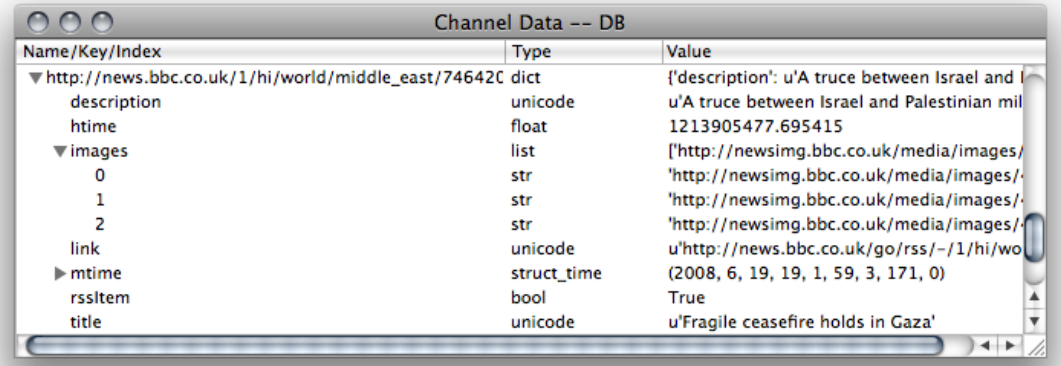

Figure 34: A BBC World News article as stored in the channel database.

that may be performed without charge per day. Thus, this database can help to reduce the cost of querying the underlying data sources.

Each channel in The Buzz has its own database, storing only the data for that particular channel. This database uses a keyed index of entries. Each entry corresponds to an individual data item in the database. This entry might encode a single entry from an RSS feed, the list of images found on a single web page, or particular element of weather data, such as tomorrow's probability of precipitation. Each of these entries are encoded as dictionaries (hash tables) where each key indicates the type of item in that particular field. Thus, for data from an RSS feed, this dictionary might contain the keys, `title`, `description`, `images`, and `link`, where the `title` and `description` values are HTML strings, the `link` is a URL, and `images` is a list of URLs. Figure 34 shows an example of such an entry.

While the structure of each entry is itself a dictionary, the particular keys used are only loosely defined. Particular entries may define arbitrary keys. Furthermore, the keys that are provided are intended to reflect the function of the data rather than a particular format. In this way, the data entries use a folksonomic tagging-style, wherein there is not a strict taxonomy of tags, but an evolving set [102]. This folksonomic approach leads to more flexibility on the part of developers because the system can grow to handle new tags and thus new data types very easily. This flexibility comes at the cost of a lack of central control over the available tags, leading to potentially conflicting keys. The Buzz design takes that approach that this problem can be ameliorated with discipline rather than

through a strict taxonomy. Nonetheless, it does require developers to exercise diligence. Thus, data that comes from an Atom feed will still be stored in the database using a `description` key, even though this field is called a `summary` in Atom. In this way, the underlying format of the publisher's data and the entry in the database are only loosely coupled.

5.2 Properties

Properties encode attributes over which the user should have control. These properties are used for any configurable object in The Buzz—harvesters, scrapers, filters, and visualizers. In code, properties are a special kind of variable defined with additional metadata to describe the user interface for configuring that variable. Because these properties are a special kind of variable in code, the developer of the module can simply refer to the property just as she would any other variable, assigning values to it and using it as parameters to method invocations.

The associated metadata allows a user interface to be automatically generated to enable the configuration of the properties. Currently, The Buzz defines 10 types of properties:

Number properties define variables that should only contain numeric values. The interface presents a text field into which the user can enter a numeric value.

Fields define attributes of simple scalar strings. The interface presents these values in a standard text field.

Schedules allow users to define a recurring schedule. This kind of property is used to specify, *e. g.*, how often a harvester should run.

Patterns are a special kind of field that provides special support to help the user write regular-expression-style patterns.

Choices provide the user with a popup menu from which to choose from a pre-defined list of possible values.

Booleans are represented as simple checkboxes.

Factories are special properties whose values are arbitrary Python objects. These properties are exposed as a pulldown list from which the user can make a selection. When the user chooses

a new item, the corresponding factory method is invoked to create the object. This kind of property is most commonly used to allow the user to choose from a list of available scrapers, filters, *etc*.

Lists are a compound property that represents an array of a particular other property type. This property is represented as a single-column table whose rows depend on the particular property type stored.

Dictionaries are another compound property type represented as a table of rows.

Custom properties allow the programmer to extend the property mechanism with special interfaces when the standard mechanism is insufficient.

5.3 Harvesters

The harvester is responsible for gathering data from the publishers and storing them in the database. The Buzz defines several kinds of harvesters (see Table 1). To control this process, each harvester defines various *properties* that the user can customize. These properties are described in more detail in Section 5.2. Each property encodes an attribute that the user can configure. These attributes are then used to guide the harvester's data gathering process.

As described earlier, some of these harvesters are general purpose in nature (*e. g.* the web crawler), while others are tailored specifically to a particular data publisher or type of data (*e. g.* the Flickr and Email harvesters). The specialized harvesters are typically implemented in an *ad hoc* fashion. For example, the Flickr harvester communicates directly with the Flickr servers using their web-based APIs and outputs directly into the database. Various properties control what pictures the harvester gathers, but they do not control the overall behavior of the harvester. Because most of this process is peculiar to the specific task of extracting photos from Flickr, there is relatively little reusable code in the harvester.

The web crawler, in contrast, operates in a general purpose fashion. In this general purpose harvester, the properties control the actual behavior of how the crawler traverses web sites to find relevant web pages. To control the actual content extraction, however, the harvester uses various lower-level scrapers. These scrapers are generic modules that operate on particular types of content.

Table 1: Built-in harvesters in The Buzz.

Web Crawler Starts at a specific web page and crawls to the linked web pages. The user can control this process by adjusting the depth of the crawl, restricting the crawl to specific web hierarchies (*e. g.* not follow links to other web servers or outside of the parent's file path), and specifying what content to extract from the various web pages.

Syndication Feed Loads an RSS, Atom, or RDF syndication feed. The user can control what content to extract from feeds (by default, the title, description, and images included in each entry) and specify filters (*e. g.* to ignore entries older than a certain threshold).

Flickr Gathers images from the Flickr.com photo sharing website. This specialized harvester lets the user specify photo tags, users, and groups to control what photos are fetched. Because this harvester uses the Flickr API, the user can control the size of the photos—a capability not available through Flickr's RSS feeds.

Weather Gathers weather data for a specified zip code and extracts local radar data, satellite imagery, a five-day forecast, a detailed daily forecast, and other misc. weather data. The user specifies only a zip code to control the process. The harvester abstracts the process of connecting to multiple U.S. National Weather Service and National Oceanic and Atmospheric Administration databases, each keyed by one of zip code or latitude and longitude.

E-Mail Interfaces with Apple's Mail software and system-wide Address Book to collect unread email messages. Stores the subject of the message along with the name of the sender and associate image, if any, from the address book.

For example, the "Largest Image on Page" scraper inspects all of the `<img>` tags in an HTML document and yields the image with the largest area. In this way, the process of accessing the content and the process of extracting the content are separate.

This separate approach allows scrapers to be reused for different kinds of data providers. Fundamentally, extracting all of the images on a web page and extracting all of the images in an HTML-encoded RSS entry use the same process. Thus, the more general purpose harvesters delegate the actual content extraction to a lower-level component, called a *scraper*.

5.4 Scrapers

While the harvester is responsible for gathering a collection of data from the publishers, the scraper is responsible for extracting the relevant data from a single document or piece of a document. Thus, the aforementioned "Largest Image On Page" scraper can extract the largest image in a single HTML document. So long as its input is HTML, the scraper does not care whether it is operating on

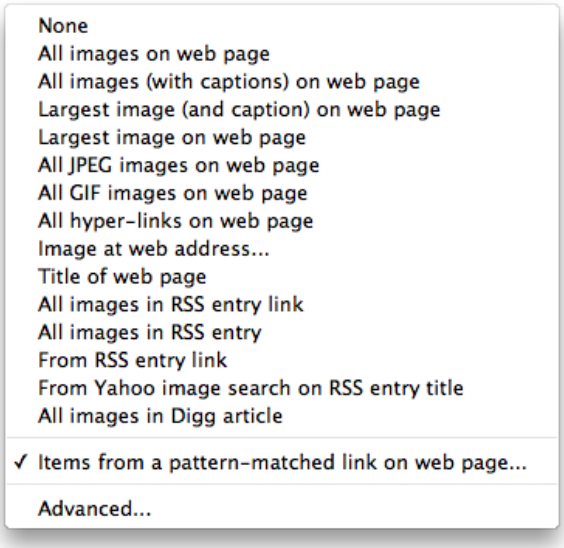

Figure 35: Choosing how to extract data from an RSS feed.

a web page or a snippet of HTML embedded within another document (such as an HTML-encoded entry in an RSS feed).

Thus, while the scraper performs the low-level data extraction, the harvester performs the higher level process of gathering a collection of data. In some cases, particularly with the specialized harvesters, these functions are merged directly into the harvester. Frequently, however, reusable scrapers are applied to extract the data. In this way, a simple RSS harvester can extract a variety of different kinds of data, from gathering all of the images in an RSS entry to gathering all of the images in the web page linked by an RSS entry (both using the same scraper) to gathering images resulting from an image search on the keywords in the article title.

In addition to performing low-level data extraction, some scrapers also define chaining behaviors. That is, some scrapers will use the result of another scraper as its input. For example, the `IndirectURLScraper` lets the user specify a pattern to apply to a document, the result of which is used as input to another scraper that the user can specify. Using this approach, the user can, *e. g.*, specify a pattern to identify a particular link embedded in a web page and extract all of the images from that web page instead of directly from the document itself. (This technique is particularly useful for link aggregation sites, such as `Digg.com`, where the images relating to the content are not on Digg itself, but on the site linked.)

Figure 35 shows a user choosing a scraper in the user interface. Although the underlying code of The Buzz distinguishes between harvesters and scrapers, the term scraper is not exposed to the user. Instead, these objects are referred to as gathering methods, depending on the particular context of their use. Wherever possible, these objects use human terms rather than jargon. Furthermore, all scrapers have functional labels in addition to their programmatic names. Thus, the `IndirectURLScraper` appears in the user interface labelled by the kind of data it gathers: "Items from a pattern-matched link on a web page." While this distinction between harvesters and scrapers is useful in the architecture of The Buzz, it is not necessary to expose it to the user. Users do interact with the different objects, but special terminology for them is not necessary.

5.5 Visualizers

Visualizers are responsible from transforming a single entry in the database into a representation on the screen. Different visualizers operate on different kinds of data. For example, the simple `ImageVisualizer` displays an image and an optional caption on the screen. The `CardVisualizer` displays data, such as from an RSS entry, on an index card-like region on the screen with a title, summary text, and/or images, depending on the data.

Visualizers use a signature-based typing mechanism, or "duck typing [101]" to determine the kinds of data they can depict. Under this duck typing scheme, the visualizer examines the data entry to see if it contains the tags it needs to be able to render the data[1]. For example, the `CardVisualizer` considers the data to have a conforming signature if it defines at least two of the tags `title`, `description`/`summary`/`captions`, or `images`.

Using this duck typing approach, the visualizers and the harvesters can operate in a loosely-coupled fashion. Because the visualizers look only for the tags they need, they are not dependent on a type specification made by the harvester developer. Furthermore, the harvesters can output different tags suitable for different representations of the data. For example, when the weather harvester outputs the day-at-a-glance forecast it can output an `image` tag containing the forecast icon for the day, a `number` tag containing the forecast high temperature, a `description` tag containing a textual description of the day's forecast, and any other custom tags that might be relevant to

[1]The term "duck typing" arises from the phrase, "If it looks like a duck and quacks like a duck, it must be a duck."

the particular data, such as `probability-of-precipitation`. A generic visualizer such as the `CardVisualizer` could then use those tags to generate an appropriate description of the weather forecast, or a more specialized weather harvester could take advantage of the specialized tags output by the harvester.

To control this inspection process, visualizers define two methods. The first, `canHandleData`, simply responds as to whether the visualizer is capable of depicting a particular data entry. This method is used to filter the list of visualizers to only those that are capable of rendering the data. The second method, `shouldHandleData`, provides a confidence ranking of how well-suited the visualizer is to depict the data. For example, the `CardVisualizer` provides a high ranking for data entries output by the RSS harvester, which includes all of the desired tags. A data entry that only provides a subset of those tags, however, will receive a lower ranking. Using this ranking, the system can automatically select a reasonable visualizer to apply to a particular data entry. Alternatively, users can override this decision to specify a specific visualizer. Section 4.9.5 describes how the user combines these visualizers and data to construct a slide.

5.6 Dispatch

The dispatch acts as the supervisor coordinating the various channels. It controls when the various channels' harvesters should run to update their data. It also determines when to show each channel.

The dispatch thread typically runs its loop approximately every minute. Each time through the loop, it first runs a maintenance process to cycle internal memory pools and to identify channels with stale data. Any channels with old data are then added to the harvester queue.

A harvester thread monitors the harvester queue. To avoid overburdening the user's CPU or network connection, the harvester thread limits itself to updating three channels at a time. As long as there are available slots, it will start the next harvester in the queue. Additionally, it imposes a time limit of ten minutes on each harvester. If, after this time, a harvester has not completed, it is assumed to have stalled and is halted. Any harvester that fails to complete, whether because of a timeout or an uncaught exception, is marked as having suffered an error.

The dispatch keeps track of which harvesters have encountered errors and penalizes those harvesters with increasing penalties. If, for example, a publisher's site goes offline and causes a harvester to malfunction, it will receive an increasing penalty each time the harvester fails. Thus, if the publisher experiences a short downtime, the harvester may retry every few minutes and fail a few times before eventually running successfully (when the publisher comes back online). If, however, the problem occurs over a longer duration, the harvester will be deprioritized sufficiently that it might only retry once a day. Through this mechanism, the dispatch thread is able to balance resilience to transient errors with the costs of repeatedly executing flawed code.

After the dispatch has queued up any stale harvesters, it selects the next channel to display. Each channel has a particular display priority specified by the user. This priority controls how frequently a channel is displayed: very frequently, frequently, occasionally, rarely, or never. The specific meaning of these terms depends on how many channels the user has subscribed to. Thus, if a user subscribed to a great many channels all to be shown very frequently, they might still only be shown every few hours. A user with few channels, in contrast, might see a channel scheduled to be shown rarely come up every few minutes. Nonetheless, the dispatch thread uses a priority queue to control the display order of the various channels. Additionally, within each priority, the dispatch uses a random permutation of the channels to control the display order. As such, a channel will not be shown again more than once until all other channels with the same priority have been shown. If an error occurs in displaying a channel (*e. g.* the channel has invalid data or a programming error occurs), the channel is skipped and the next channel with the same or lower priority is shown.

In this way, the dispatch thread is resilient to data gathering and presentation errors. The effect of a runaway data harvester or visualizer (either due to a programming error, transient network anomaly, or invalid data from the publishers) is mitigated through the dispatch thread's throttling and timeout mechanisms. Nonetheless, the user can explicitly override any of these automatic mechanisms and force a channel to run at any time. The user can instruct the dispatch to run a particular harvester right away and can even monitor its progress in the channel browser, or she can override the display priority queue and cause a particular channel to be shown immediately using the playback controls.

5.7 Plugins

In addition to all of the built-in harvesters, scrapers, filters, and visualizers, developers can write their own using The Buzz's plugin mechanism. Thus, if a capable user wishes to augment the behavior of the system in a way that is not readily accomplished with the existing customization framework, she can write her own python plugin.

To support plugins, harvesters, scrapers, filters, and visualizers are `Registerable`s. These objects support dynamically loading new registerables and querying for which registerables are available of a given type. They also define both the programmatic code for the object, but also the user-readable descriptions of the object. Thus, a user could write a custom `DiggHarvester` that integrates with the Digg.com APIs. In registering the new harvester, the developer must provide a descriptive label for the harvester of the form "Gather from ..." and a longer description of the harvester for use in a tool tip. Thus, the software can query the registered harvester to find out it is defined in the class `DiggHarvester` and that it should be presented to the user as "Gather articles from Digg.com" with the hint "Gathers a collection of articles submitted by the Digg.com community."

When The Buzz populates its list of available harvesters, scrapers, filters, or visualizers, it queries for all of the available registerables of that particular type. Thus, a developer of a new plugin needs only to write the appropriate object conforming to the appropriate harvester, scraper, filter, or visualizer interface and register it with the registerable mechanism.

In most cases, the developer need only subclass the appropriate registerable and implement a single method, although the various registerables do provide additional methods that the developer may use to override the configuration behavior of the new object. Figure 36 shows an example plugin scraper to extract the embedded images on a webpage linked from Digg.com.

Channels are stored on the filesystem as a file bundle, a directory with a special structure and which is presented to the user as an opaque file. That is, it looks like a regular file to the user but is actually a folder. A developer can add a new plugin to a channel by putting the python code for the plugin object into the Extensions folder of a channel. See Figure 37. When the dispatch loads a new channel, it will load any plugins that may be included in the channel.

```
import re

class DiggImageScraper(scraper.HTMLImageScraper):
    ''' Scrape images from the web page linked by the Digg article page. '''
    __digg_exp_str = ur'''<dt>Source:</dt>.*?<a href="(?P<url>[^">]*)"'''
    __digg_exp = re.compile(__digg_exp_str, re.I|re.M|re.S)
    def scrape(self, entry, harvester=None):
        ''' scrape(<RSS entry>, <Harvester>) -> [ <data entry>, ... ] '''
        # Load the Digg article and apply regular expression to find the
        # source article URL
        html, base = loadURL(entry.get(u'link', None))
        m = self.__digg_exp.search(html)
        if m is not None:
            # Extract matching URL and let our parent class scrape the HTML
            # for images.
            url = m.group(u'url')
            html, base = loadURL(url)
            return super(RSSDiggImageScraper, self).scrape(html, base)

        # No matches; return empty set
        return []

# Register the new scraper
Scraper.register(DiggImageScraper, u'All images in Digg article',
                 description=u'Extracts all images in the source article '
                             u'for each Digg entry in the RSS or Atom feed.')
```

Figure 36: An example scraper plugin to extract images from articles posted on Digg.com.

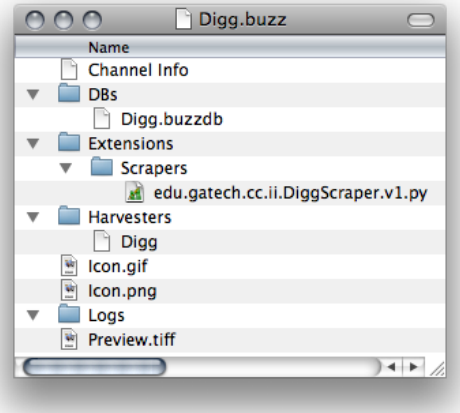

Figure 37: The structure of a channel bundle on the file system.

CHAPTER VI

COMPARATIVE ANALYSIS

The Buzz lies at the intersection of two domains: end user customization and information awareness. The primary goals of the user and of the system are to foster awareness of routine information. Toward that end, the goal of this research focuses on how to support the user at customizing the information the system conveys. These two distinct goals, while complementary, bring different design implications across different design dimensions. In this chapter I describe this customization and awareness space. Through various dimensions of this space, I examine existing approaches to information awareness and customization applications, and I situate The Buzz with respect to these approaches.

6.1 Gentle Slope

Within the end user programming community, a common way to examine programmable software is in terms of both the expressive power of the customization or programming interface and the effort required to make such an expression. These two dimensions define a two-dimensional space, leading to the standard gentle-slope diagram [80]. From this perspective, plotting different kinds of interactions with different tools can present a difficulty curve, in which simple customizations with little expressive power lie in the lower, left corner and difficult customization with rich expressiveness lie in the upper right corner (see Figure 38).

Designers of customizable software then pursue two goals: (a) to keep that curve as low as possible while also (b) avoiding steep increases in that slope, which represent barriers to more expressive customizations. The first of these goals focuses on keeping tasks as simple as is feasible. No matter how complex a task is, the system should support it as easily as is feasible. The second of these goals focuses on the learnability of the software. An incrementally more complex task should ideally require only an incremental increase in effort.

In the context of information awareness applications, the lower left corner of this space contains simple behaviors such as using defaults or performing basic customizations to the content, such as

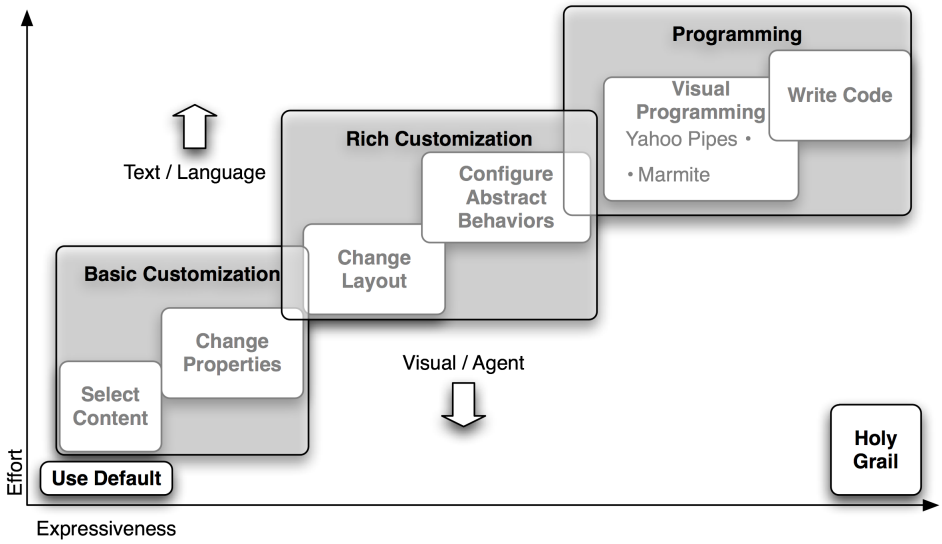

Figure 38: A gentle slope representing the effort and expressiveness of different awareness customization activities.

changing the location for a weather data stream. These customizations are fairly simple to perform, but also offer relatively little expressive capability. The upper right corner, by contrast, contains advanced behaviors such as creating new awareness streams or even new reusable components for defining such awareness streams. These sorts of customizations typically involve complicated interactions, such as writing code, but also offer a high degree of expression.

There is a gap, however, in the middle of this space. Currently, systems typically focus on one or the other of the corners, attempting to make it easier to perform simple customizations or by trying to make programming interfaces more accessible to more users. For example, visual programming interfaces attempt to lower many of the technical barriers to customization. But they still ultimately require the user to possess programming capabilities.

In this middle space, however, users can go above and beyond providing simple data parameters to the information awareness system. Users should be able to control the actual behaviors of the system, controlling how the data are gathered and presented, rather than simply controlling what data are gathered presented. Similarly, the user should be able to control such behaviors without having to resort to programming.

In the spirit of the gentle slope, therefore, awareness systems should support customization across the expressiveness spectrum as easily as is feasible. Thus, a user should be able to easily

choose what information to show, control simple parameters as in choosing to show the weather forecast for multiple locations, create derivative content streams, or even define new ones altogether. Most existing systems, however, focus only on the edges of this space: supporting simple content selection and parameterization or supporting the programming of new content streams.

In this way, users fall under one of two roles: end user or developer. An end user cares only to work within the confines of the system, subscribing to streams, widgets, gadgets, pipes, or whatever metaphor the system uses to encapsulate types of information content. A developer, in contrast, can create new streams, widgets, gadgets, or pipes by writing code. A user who wishes to perform any customization beyond the basic customizations the system supports must transition into a developer's role; there are no intermediate stages along the slope.

Furthermore, most systems treat these two roles as entirely distinct. A user is either an end user or a programmer but not both. In Dashboard, Google Gadgets, Live Gadgets, Konfabulator, and Sideshow, for example, the user can subscribe to widgets[1] and can change the properties exposed by the designer of the widget. If the user wishes to modify the widget, however, she must leave the system and enter a separate developer's environment, instead writing HTML, XML, JavaScript, or even using C/C++-based APIs. Pipes recognizes that these two roles are fluid and provides a transition between end user and programmer entirely within the interface. Nonetheless, these two roles are treated as distinct—a user is behaving *either* as an end user by running pipes, *or* as a developer by modifying their sources.

Nonetheless, while this rich customization space is sparsely populated, it is not entirely barren. Various systems have taken different approaches to supporting the user at performing various degrees of customization. The rest of this chapter focuses on this customization space. Section 6.2 presents different approaches that have been taken to address customization relevant to information awareness systems. These approaches help to illustrate many of the pitfalls and challenges that arise. Section 6.3 uses these different approaches to help describe various dimensions of this customization space. Thus, the rest of this chapter first describes the various awareness systems that populate this customization space. Only after we understand the inhabitants of this space can we

[1] Unless otherwise stated, I use the term "widget" generically to refer to widgets, gadgets, pipes, or similar artifacts created by the system, regardless of which term a particular system prefers.

use them to identify and illustrate the dimensions that make up this customization space.

6.2 Six Approaches

Various approaches have been taken to address the assorted customization tasks related to information awareness systems. If we consider these tasks in terms of the stages of the awareness data pipeline (see Section 2.2), it will help to categorize many of these customization dimensions. The various stages of the awareness data pipeline describe the technical steps an awareness system must perform to gather, aggregate, manipulate, mash up, and convey data as information. Through this lens, Pipes and Marmite (Sections 6.2.1 and 6.2.2) focus entirely on the front end of the data pipeline: extracting, aggregating, and transforming data from the sundry formats used by content publishers. In contrast, the various widget systems (Section 6.2.3), Greenberg and Boyle's Notification Engine (Section 6.2.6), and Dontcheva's Cards framework (Section 6.2.4) address all stages on the pipeline, albeit in very different ways. Finally, Koala/Coscripter uses a programming by demonstration approach to allow the user to create recordings of repetitive tasks, but does not explicitly focus on any particular stage of the pipeline or on awareness interfaces. Nonetheless, its programming-by-demonstration approach makes it a valuable lens to consider.

In this section, I present six systems that demonstrate different approaches to supporting customization. Most of these systems focus specifically on the information awareness domain. Some of these systems, however, focus on problems that are relevant to, but not necessarily within, the information awareness space. An analysis of these systems will help to illustrate many of the concepts and challenges that arise in support end user customization in this context.

6.2.1 Yahoo Pipes

Yahoo Pipes is a visual programming system that enables people to take data from one or more sources, filter them, manipulate them, and create a single, derivative stream. With this approach, a user could create a mashup of Craig's List housing listings and Yahoo Maps, of New York Times headlines through the lens of photos on Flickr, or just remove all the photos of flowers from a Flickr photo gallery.

To define these streams, a user creates a pipe by connecting visual blocks, called operators, together by drawing lines, or pipes, between them (see Figure 40). Each pipe combines simple

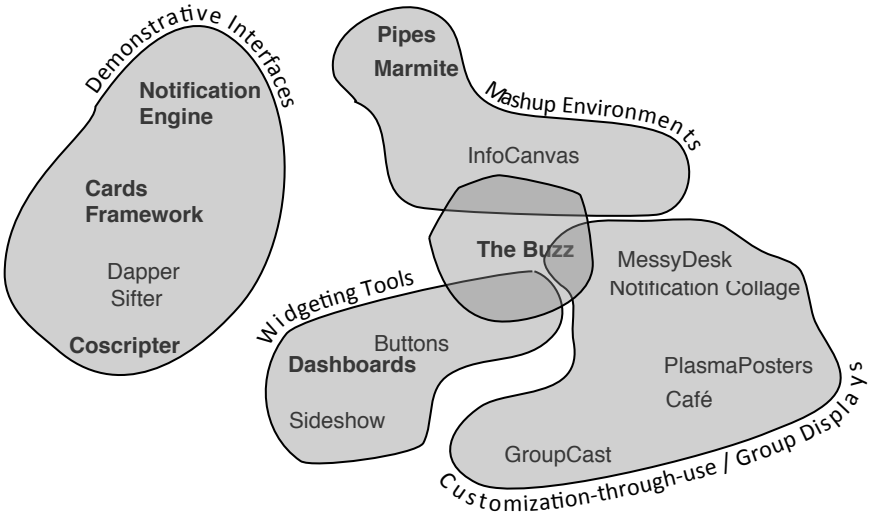

Figure 39: An overview of systems in the awareness customization space. Systems drawn closer to each other within a region are more similar than systems drawn farther apart. Those in bold are spotlighted in this chapter.

operations together, chaining the output of one operation to the input of another. Much in the same way that Unix pipes allow relatively simple commands to be combined to accomplish fairly complex and application-specific tasks, Yahoo Pipes allow the combination of relatively simple operations in order to generate more complex data streams.

Each Pipe gathers input from various sources, primarily RSS[2], JSON, XML, and similar structured input sources. It then applies various operators to the data, such as filters, regular expressions, loops, *etc*. The results are output as RSS, JSON, or KML. By combining different operators, users can create pipes that aggregate data from different sources, perform certain kinds of manipulations or normalizations on those data, and output mashed-up or other derivative data streams.

In this way, Pipes is primarily an aggregation or mashup tool. A user can find existing pipes created by other users or create his own using a visual programming interface. The user typically creates a pipe that aggregates data from multiple data sources, perhaps modifying the data in some way, and outputs it as an RSS feed, photo gallery, or map, depending on the kind of data. In this way, Pipes is especially useful for creating aggregations of data from multiple sources and

[2]and its cousins, such as Atom and RDF.

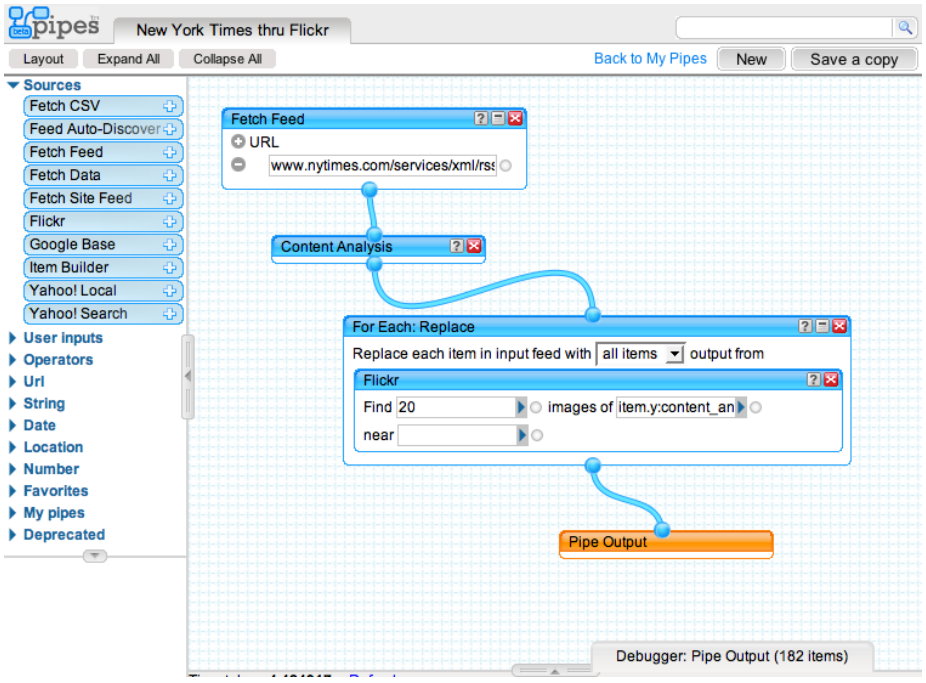

Figure 40: The Yahoo Pipes editor interface to create a mashup of headlines from the New York Times with related photos from Flickr.

combining them with one of these interfaces.

A user can run a pipe in several ways. The Yahoo Pipes web site provides a web interface and an RSS interface to each pipe, similarly to the way that many news publishers provide their content both to the user on the web or via RSS. Thus, a user can view pipe output either in his or her web browser or through any standard RSS reader. Additionally, the web interface provides a link to view and/or modify the source for each pipe. In this way, the interface supports users at transitioning from an end-user to a developer's role.

The pipe editor interface uses a visual programming approach (see Figure 40). In this interface, each operator is represented as a box on the screen. Each operator exposes various configuration parameters, but otherwise functions as a black-box entity. The user can then drag and drop connections from one operator's output to the input of another (see Table 2). Through this visual programming interface, the user can define complex data flows.

For example, a popular pipe gathers news headlines from the New York Times RSS feed, extracts keywords from those headlines, and uses the keywords as query parameters for photographs on Flickr.com. The pipe then outputs news headlines augmented by potentially related photographs

Table 2: A listing of operators available in Yahoo Pipes. Many operators are self-explanatory. Refer to `http://pipes.yahoo.com` for their precise behaviors.

Sources Fetch CSV; Feed Auto-Discovery; Fetch Feed; Fetch Data; Fetch Page; Fetch Site Feed; Flickr; Google Base; Item Builder; Yahoo! Local; Yahoo! Search

User Inputs Date Input; Location Input; Number Input; Private Text Input; Text Input; URL Input

Operators Count; Filter; Location Extractor; Loop; Regex; Rename; Reverse; Sort; Split; Sub-element; Tail; Union; Unique; Web Service

URL URL Builder

String Private String; Yahoo! Shortcuts; String Builder; String Regex; String Replace; Sub String; String Tokenizer; Term Extracter; Translate

Date Date Builder; Date Formatter

Location Location Builder

Number Simple Math

My Pipes Insert New Pipe; *Dynamic list of my pipes*

provided by the Flickr community. Another popular pipe gathers news articles from hundreds of weblogs and merges them into a single RSS feed.

Pipes operates as a web service. All pipes are defined through the web-based interface, through which the user defines the inputs, dataflows, and outputs for each pipe. All data processing is performed by Yahoo's servers. As such, users are limited in the ways in which they can extend the available operators. First, a user can treat another pipe as its own operator, allowing for encapsulation of pipes. Nonetheless, this mechanism does not allow the developer to create a new operator beyond what can already be expressed with all of the existing operators. The only mechanism Pipes provides for developers to define entirely new operators is to create and host their own web service, to which a pipe can then delegate some of the data processing. For most purposes, therefore, Pipes does not support extensible operators.

The focus of Pipes is primarily on data flows. That is, it is a tool to collect data from various sources, aggregate them, manipulate them, and mash them up to define new, derivative data streams. As such, the system provides little support for the presentation of those data. Through the web interface, a user can display a pipe's output as a list, as a slideshow of images, or on a map. Beyond

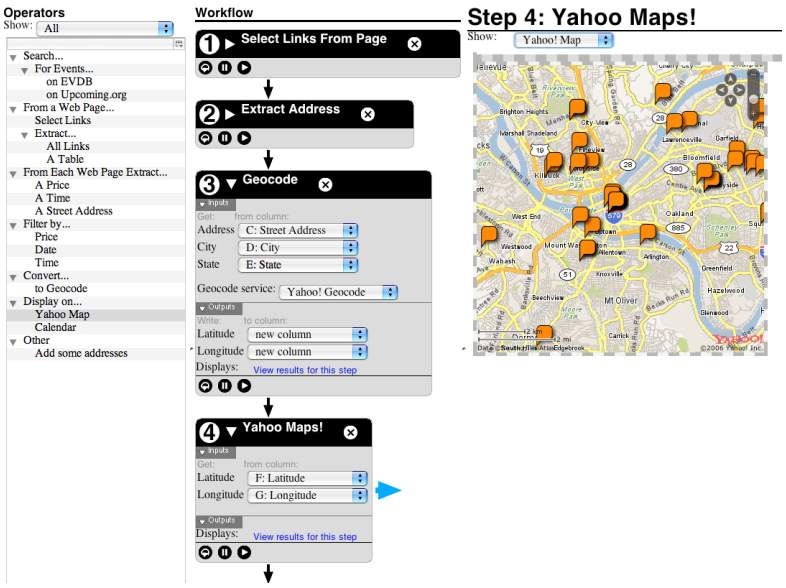

Figure 41: Creating a Marmite mashup of housing listings from Craig's List with a Yahoo Maps interface.

choosing which of those representations to use, the user has little or no control over the presentation.

The primary use model is to subscribe to the RSS feed output of the pipe. In this way, the actual use of the data is delegated to the feed subscriber. Presentation of those data, beyond the rudimentary interface described earlier, is beyond the scope of the system. As such, Pipes focuses primarily on the front end of the awareness data pipeline described in Section 2.2.

6.2.2 Marmite

Like Pipes, Marmite [108, 107] provides a visual programming interface to support users at aggregating, filtering, and mashing up data streams. Using Marmite, a user can define a dataflow to, *e. g.*, extract addresses from housing listings on Craig's List, filter out properties above a certain price, and convert those addresses to latitude and longitude coordinates so that they can be placed on a Yahoo map interface.

Instead of acting as a web service, however, Marmite operates as a Firefox web browser plugin. Through the plugin, users can create a dataflow using an interface similar to that of Apple's Automator tool [10], which allows users to visually define automated workflow processes. For example, in Automator, a user might start with a folder of images, pass the images to a photo manipulation tool to scale them down to a web-scale, create an archive of the resulting images, and attach them to an

email message.

Marmite, like Pipes, allows the user to connect together different operators in order to define the process of gathering, manipulating, and outputting data. Marmite, however, restricts these to linear data flows. The output of one stage is passed to the next stage in the pipeline, or workflow. No support exists for branching or merging dataflows.

The user begins a workflow by specifying an input operator. Marmite offers input operators for extracting data from web pages or from the UpComing.org or Eventful.com event listing services. The web page input operators allow the user to click on a rendered web page to specify the attributes to extract. The system then uses techniques adapted from Sifter [52] to infer an extraction pattern to apply to other instances of the same web page. Thus, by selecting, for example, the names and prices on a web page of housing listings, the system could infer a rule for extracting listings from web pages from the same publisher but for a different location.

The user can then apply new operators to the extracted data, for example to convert a street address to a latitude and longitude. At each step along the dataflow, the system updates a spreadsheet-like representation of the data. In this way, the user can more easily trace the progress of the dataflow in order to verify its correct behavior or to identify bugs.

In terms of the awareness data pipeline, both Pipes and Marmite perform a similar function. Both systems focus on the front end of the pipeline, extracting data from content publishers and transforming it in some way, before outputting a derivative data stream. Marmite provides two mechanisms, however, to try to better support end users at programming these data flows: operator hints and the aforementioned spreadsheet data representation.

When the user begins creating a dataflow, Marmite provides hints as to which operators are appropriate at the current stage of the dataflow. Thus, at the beginning, Marmite suggests using one of the source operators to extract data from some particular data publisher. At the next stage, the system provides hints as to which operators might be appropriate given the output of the previous operator. Thus, if an operator outputs addresses, Marmite might suggest the use of a geocoding operator to transform the addresses to latitudes and longitudes. A geocoding operator might suggest the use of a map sink to visualize the output data points on a map, or a sink to output KML map markup data.

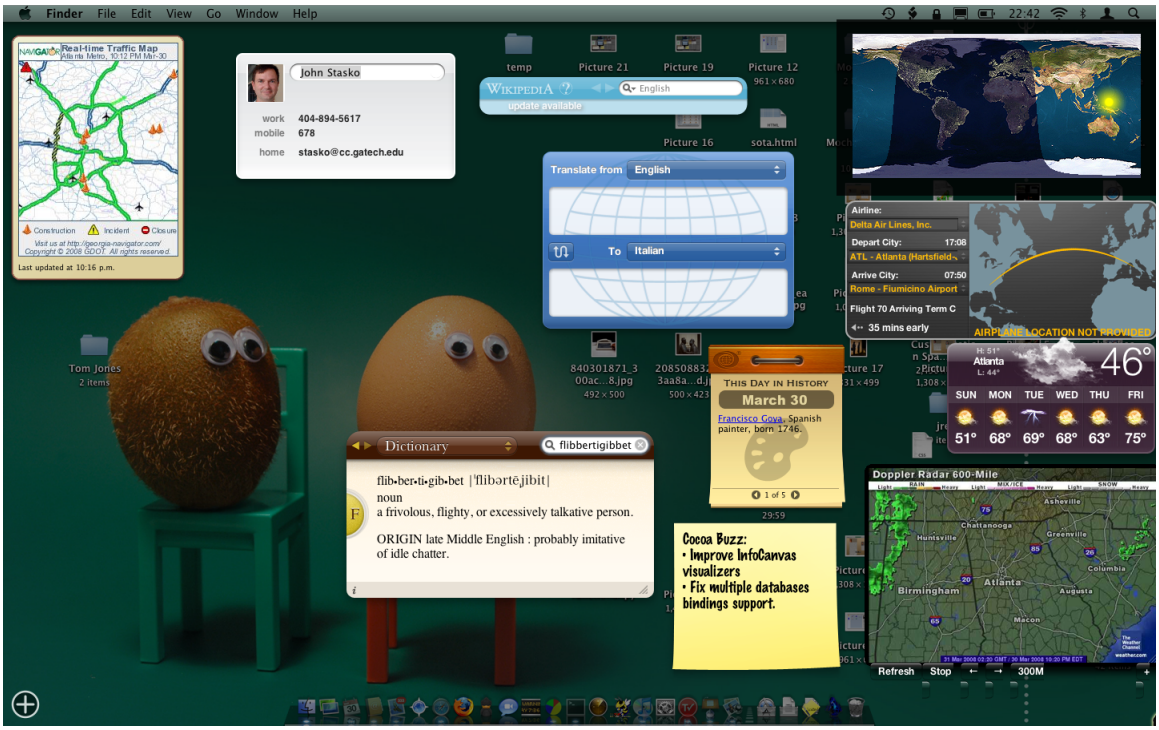

Figure 42: A collection of Dashboard widgets running in Apple's Dashboard.

Through the use of these hints and the spreadsheet data representation, a user study has found that most users with spreadsheet experience are able to create such dataflows in Marmite [108]. This result suggests that the Marmite visual programming approach is able to help lower the threshold required for end users to program within this particular domain. Nonetheless, the study also found that participants who did not already have experience with spreadsheet programming found it difficult to create such dataflows. As such, this visual programming approach is no magic bullet.

Because Marmite runs as a Firefox browser plugin rather than on Yahoo's servers, it does not have the same security limitations on creating new operators. As such, Marmite allows developers to extend the system with new operators written in a combination of JavaScript and Mozilla's XML Binding Language (XBL). Thus, there are two kinds of users the system supports: end user programmers, who create their own data flows, and developers, who can define new operators for themselves or other end user programmers. Nonetheless, the system itself provides no mechanism to support the exchange of data flows or operators amongst fellow users.

6.2.3 Konfabulator/Dashboard/Google/LiveGadgets

Konfabulator [6], Apple's Dashboard [11], Google Gadgets [4], and Microsoft's Live Gadgets [75] all provide a similar widget-based information awareness approach. For most purposes, these systems can be treated as equivalent. Konfabulator was the first of these systems created and was subsequently acquired by Yahoo. Each of these systems enables the user to download information widgets, or applets. These widgets are typically small objects on the screen and which convey a specific piece of information and/or support a specialized task, much in the same way as Buttons [92, 68]. Information-only widgets include the Weather, Traffic, and Day in History widgets shown in Figure 42. The figure also includes interactive task-oriented widgets, such as the Dictionary, Wikipedia, and Translation widgets.

These dashboard systems make it relatively easy for users to find information applets that provide a highly specialized interaction. Information-only widgets might present specific information such as the weather in Poughkeepsie or the number of active bugs in the company bug tracking system. Task-oriented widgets might provide an applet that is optimized for looking up package status based on UPS tracking numbers. Although all of these are tasks that are possible through existing web-based interfaces, these widgets provide a narrow, "just the facts, ma'am," tool for common tasks. These tools let users find and use such widgets and provide support libraries and frameworks for developers to write these widgets.

These widgets are popular among users because they provide a simplified interface for a specialized task, usually without excess user interface cruft or too many options necessary in a general-use interface. They are also popular among developers because they are relatively simple to create, given existing web development skills.

In each of these systems, users can browse a shared repository of available widgets and select the ones they wish to use. Depending on which system is used, these widgets float above the desktop, or in a sidebar, appear overlaid above the desktop when a hotkey is pressed (as in Figure 42), are embedded within a web page, or some combination of the above. Regardless of the framework within which these widgets operate, however, the interaction style is essentially the same. For information-only widgets, the user quickly glances at the widget to gather a status update. For

interactive widgets, the user might monitor the state and possibly follow up with a quick interaction. For example, the Address Book widget shown in Figure 42 serves as a shortcut to the Mac OS X integrated address book database. Rather than requiring the user to launch the Address Book application, the address book widget can be preloaded in the background and available whenever the Dashboard hotkey is pressed.

Similarly, widgets can provide a shortcut for common and well-structured tasks, such as tracking the status of an airline flight, also shown in the figure. Such a widget can provide a specialized interface for a recurring task. Many such widgets have been created by the active communities around each of the respective systems. A user can simply browse the library of available widgets and subscribe the ones of interest. They will then appear on the dashboard, sidebar, or web page. Furthermore, many widgets offer various tailored properties available for customization, such as the zipcode for a weather widget or the language to use for an international widget.

When no pre-existing widget is available for a particular need, however, a user must transition into the role of a developer. Although each of these widget systems may differ in how they present the widgets, they all use the same approach to supporting developers at creating widgets. And unlike Pipes, they do not provide an integrated editing environment. Instead, the developer must load a separate development environment[3]. Widgets are then created using a combination of HTML, XML, and JavaScript. As such, even the widget systems that appear to run as desktop applications or applets still run each widget within a web-based framework.

By using a web-based framework, these systems allow those with already-existing web development skills to create their own applets. The skills from the one transfer to the other. Furthermore, web development tools provide extensive support for controlling the appearance of the artifacts created, making it easier for developers to create specialized visualizations or depictions of certain kinds of data.

Nonetheless, these systems still require the user to demonstrate significant technical skill. Regardless of the development environment, widget developers must write markup or code. As such, these systems treat end users and developers as distinct.

[3] Although some systems, such as Dashboard, do provide an Integrated Development Environment (IDE) specialized for the development of widgets, others rely on the widget developers to use their own preferred environment.

6.2.4 Summaries Framework and Cards

Dontcheva *et al.*'s Cards Framework [32] focuses exclusively on helping end users at creating new information mashups without programming. Users can define extraction patterns by clicking on items on web pages. Items extracted are added a to a card. By dragging from an item on one card to another, the user can specify a relationship between the items, so that when one card is loaded, its data can transfer to the new card so as to merge the two.

Using this technique, a user can create a restaurant listing card stack, where the user can view restaurant pricing and hours gathered from one site, integrated with reviews from another site, and a map with driving directions from yet another. As such, the Cards Framework is primarily an information aggregation tool.

Under this framework, there is no distinction between different classes or roles of users. End users do not transition into a developer role. Instead, the focus is on letting the end user define extraction templates and relations between the data through a direct manipulation interface.

Under the Cards Framework, users can define extraction templates to gather data from various data sources. For example, a user could define a template to extract restaurant review data from a yellow pages service. In the interface, the user clicks on the various elements of the rendered web page to specify the desired data. The system then infers the template to extract the specified entities from the underlying HTML. Using this template, the extracted data are then laid out on an "index card" on the screen.

The user can combine multiple extraction templates by browsing to another website, for example, a restaurant review service. In the same way as before, the user specifies an extraction template to gather the name and overall rating of the restaurant from this new service, creating a new index card for the restaurant reviews.

In order to link these two cards, the user draws a link between the fields for the restaurant name on the two cards. In this way, the user can create a new, derivative card that displays the combined information from the multiple data sources. Now, when the user adds a new card for a new restaurant, the system automatically applies same templates to create the appropriate derivative cards, unifying the data from multiple sources.

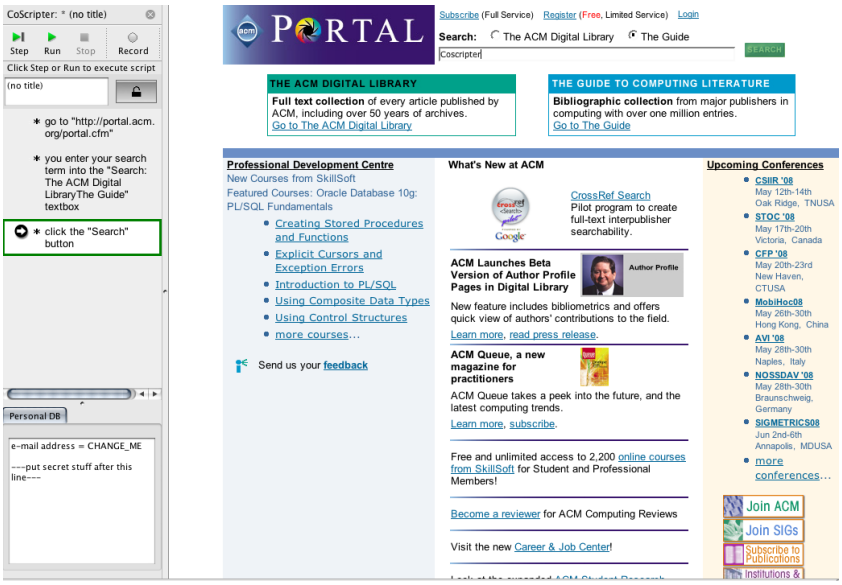

Figure 43: Coscripter (in a sidebar on the left) running a script on the ACM Portal.

Like Pipes and Marmite, this approach relies on the user to interact with a direct manipulation-style interface in order to create a data manipulation program. However, by restricting the kinds of relations the user can define on the data, the Cards Framework is able to provide such an interface without requiring the user to program. The user defines graphical relations from which the system infers the underlying programming. As long as the user is only concerned with specifying these data relations, the Cards Framework can provide a simplified customization interface. If the user need more complex data manipulations, however, the Cards Framework will be insufficient.

In this way, the Cards Framework supports a users at extracting data, aggregating them, and creating new presentations of the data without requiring the user to perform any programming. As such, the system focuses on all stages of the awareness data pipeline: content extraction, transformation, and presentation. Furthermore, it does so by operating squarely in the middle region of the expressiveness versus effort space (Figure 38).

6.2.5 Koala/Coscripter

Coscripter [63], the recording artist formerly known as Koala [64], focuses on the problem of automating repetitive browsing tasks. Using Coscripter, a user can press a "Record" button to store the steps necessary to accomplish a particular task, such as filling out an expense report after a trip. Coscripter monitors the actions taken and stores those steps in a human-readable script. By slightly

modifying the script, the user can replay the steps taken in order to fill out future expense reports.

Whereas the awareness data pipeline describes the process of transforming a publisher's data into knowledge in the head, Coscripter tries to simplify the process of taking knowledge from the user's head and transforming that into the steps necessary to accomplish a cumbersome action. As such, Coscripter does not sit in the information awareness space. It's end user customization techniques, however, are relevant to the task of supporting end users at defining methods for extracting data from web pages and describing processes on them.

Coscripter operates as a Firefox web browser plugin. With Coscripter, a user can automate a recurring tasks by recording the steps necessary to perform it. Using this programming-by-demonstration [29] approach, a user simply performs an action in the web browser while the system watches the steps performed. This recording takes the form of a human-readable list of steps, such as "Go to http://portal.acm.org," "You enter your query into the 'Search the ACM Digital Library' textbox," and "Click the 'Search' button." Figure 43 shows Coscripter in operation.

Steps in a script that start with "you" indicate steps that are to be performed by the user. Thus, in the above script, the system performs the first and last steps, while the user performs the middle step. Furthermore, each step in the script is user-editable. Changing the middle step from "You enter your query..." to "Enter 'End User Programming' into the 'Search the ACM Digital Library' textbox" will appropriately transform the script into a fully automated one.

In this way, a user can program recipe-style scripts simply by performing the action. These scripts are recorded in such language that they serve not only to record the steps necessary for the system to perform, but also in clear enough language that many users exchange these scripts with each other via other means, such as email. For example, an administrative assistant using the system recorded a script for all of the workers she supported, not to automate the task, but to create a set of instructions [63].

6.2.6 Notification Engine

The Notification Engine [43] allows users to create collages out of clippings from web pages. These clippings might consist of a region of a web page that frequently changes, such as the image portion of a webcam, a daily comic strip, an at-a-glance weather forecast, or news headlines, and more. To

Figure 44: The Notification Engine.

create a notification, the user selects these portions of a web page in a special notification composer interface. She can then compose those portions of the region to create a notification. She can then assemble a collection of notifications from different web sites into a collage, as shown in Figure 44.

To generate this collage, the system monitors the rendered output of the tracked web pages for changes. When it detects a visual change in one of the specified regions, it generates a new notification to replace the old one on the collage. Playback controls at the bottom of each notification allow the user to step through recent notifications.

Through its use of rendered output, or *surface representations*, the Notification Engine is able to handle nearly any underlying data format that the web browser is capable of displaying. Even though the Notification Engine has no understanding of HTML, Java, Flash, or any other format that might occur in a web page, the system is still able to generate notifications from these data formats. Because it uses the rendered output of the web browser, however, this approach prevents semantic understanding of the underlying data. From the system's perspective, the extracted data are merely a collection of meaningless pixels. As such, the user can compose notifications based only on visual

changes, not on the actual content themselves.

In many practical examples, however, understanding the data is not necessary. As long as the relevant data can be extracted and presented to the user, the user can make sense of the data himself. Thus, this surface representation approach can be useful to handle many underlying data formats as long as the source presentation itself is sufficient to convey the desired data.

6.3 Customization Space

Each of the systems presented in Section 6.2 illustrates some of the many approaches that have been taken to support users at customizing their information systems. The various presentation interfaces they use and the various interaction styles they employ all influence the various design decisions each of the systems' authors make. As a result, there are many design dimensions on which to analyze these systems. What kind of user does the system support? How does the user specify the content or the presentation of the system? *Can* the user specify the content or the presentation that the system uses? The remainder of this chapter describes these dimensions from the perspective of the six systems described earlier.

Even among these dimensions, there are different approaches to supporting the user at expressing her intent. She might use a direct-manipulation [93] to perform her customizations. Or, perhaps, a programming-by-demonstration [29] interface might infer how to perform the task in the future by monitoring how she performed that task. Agent-based approaches might personalize the system by monitoring her activities beyond the awareness system.

When she explicitly customizes the software, she might do so by writing a program, or by writing dotfiles. Perhaps she uses a structured rule editor, such as in Figure 45. Or she might use a visual programming environment, as in Yahoo Pipes and Marmite. A wizard interface might help to guide her through the customization process.

Each of these approaches will have an influence on the overall ease of use of the particular task the user is performing. Which style of interface is most appropriate will depend on the particular task the user is performing within a particular domain. For example, a programming interface might not be appropriate to support a user at subscribing to specific channels, but if she wishes to define an arbitrary visualization of a complex collection of data, it might be the ideal tool. The style of

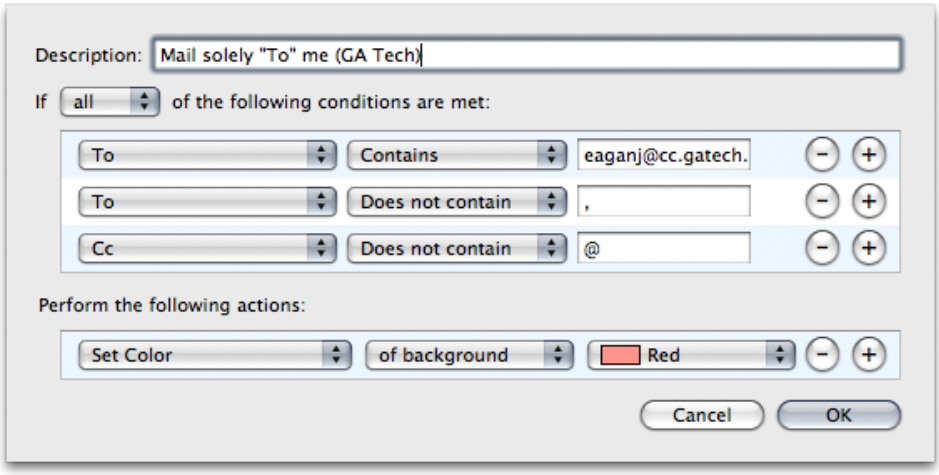

Figure 45: A rule editor interface from Apple's Mail program.

interface used might influence where a system lies on a particular dimension.

To help understand this vast multidimensional space, we consider it in terms of the awareness data pipeline. This data pipeline represents, from a system-perspective, the various steps necessary to transform a data stream into a presentation. Thus, the pipeline can help to structure the various system processes that the system will have to perform.

Because the pipeline focuses only on the data streams relating to a *single* presentation, however, it does not address relationships across representations, or how they are ultimately consumed by the user. As such, while the pipeline model is useful to help structure our analysis of these customization dimensions, it is important to recognize that there is a broader space beyond that which the pipeline encompasses. As such, additional dimensions arise when we consider not only the individual data pipeline, but also the collection of multiple data pipelines.

We consider this space first in terms of the user: who is he or she, what kinds of skills and motivations does she possess, and what is she trying to do (Section 6.3.1)? With an understanding of the makeup of the user, we can then focus on the technical tasks that he or she must accomplish (Sections 6.3.2 and 6.3.3). As we consider each of the stages of the pipeline, we also consider the implications of a multiplicity of such concurrent pipelines. Finally, we can then consider the design implications of the broader awareness context in which these systems are to be used (Section 6.3.4).

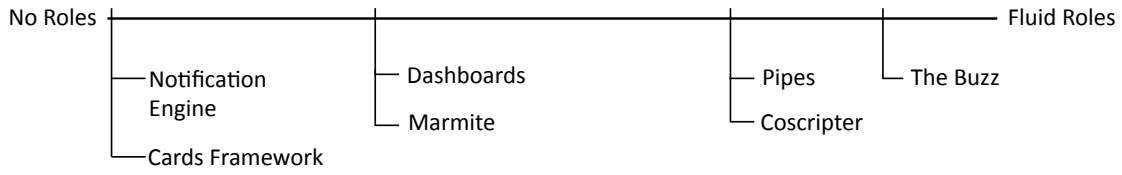

Figure 46: Fluidity of user roles in customizable software systems.

6.3.1 The User

Section 2.1 describes a user ecology, in which users typically fall into different roles: end-users, tinkerers, and developers. These roles, however, are fluid, rather than static. An individual user's skill level and inherent motivation might determine a baseline role, but various triggers and barriers might alter the role a particular user demonstrates at a particular moment of interaction.

A typical administrative assistant, for example, is often a competent computer user with no intrinsic interest in how the computer works. It is merely a tool for a job. However, many such users create spreadsheets with complex conditional logic in them when it helps them in their work. Thus, an end-user might assume a developer role given appropriate motivation and surmountable barriers.

Similarly, many computer scientists, with extensive technical skill, are often content to use their software in its default configuration. Consider, for example, the case of the Unix guru on the Mac. Just because a user possesses a tremendous technical proficiency does not imply that she desires to exercise it. Instead, she may be content to use a system that just works, and only leverage her advanced skills if the need arises.

As such, one important dimension to consider in analyzing such customizable software is the degree to which it supports these different roles of users. For example, systems such as the Notification Engine [43] and the Cards Framework [32] support only a single role of interaction. There is a single style of interaction through which the user creates his or her customizations. The user simply generates his or her collages/cards by specifying the relevant portions of web pages. The system itself is a black box, over which the user cannot take control beyond its typical use.

Alternatively, systems such as Dashboard [11, 4, 75, 6] and Marmite [108] support multiple distinct user roles. In Dashboard, a casual user can select from existing widgets and configure basic properties to customize which content the awareness system presents. Marmite allows the user to perform more advanced customizations using its dataflow model. Furthermore, both systems

also support developers at creating new content through a developers interface, whether through standard tools, a standalone IDE, or through Firefox development libraries (*e. g.* XUL). In this way, Dashboard explicitly supports end users and developers, while Marmite supports the tinkerer and developer roles.

The Buzz, Coscripter [63] and Pipes [7] extend this support for multiple user roles by making the transitions between the roles more fluid. In Coscripter, the user can restrict him- or herself to just using the script playback controls or the automatic script suggestion features of the interface, or he or she can modify or create scripts by editing the text in the sidebar. In this way, the interface provides a natural transition from the end user running scripts to the tinkerer modifying or creating scripts[4].

Similarly, in Pipes, the user can browse and run pipes found in the vast online shared repository populated by the abundant users of the system. Or a user can modify an existing pipe, including ones created by others, simply by clicking the conspicuous "View Source" button. Thus, like Coscripter, Pipes provides a transition from one role to another entirely within the interface.

The Buzz further supports all three user roles: end-users, tinkerers, and developers. End-users can customize the system by merely selecting from existing content and altering basic properties for those channels. Tinkerers can adjust the actual layout and presentation of the channels or even modify some of the data extraction behaviors, while developers can write plugins in code. With the exception of the transition from tinkerer to developer, the interface attempts to provide a gradual transition between each of these roles, where each interface presents cues as to the steps that a user might need to take to perform a more advanced customization.

Within this customization space, the fluidity of these user roles as supported by a particular system will cut across many of the other dimensions. For example, a system that supports only a single role will provide only a particular style of interaction for customizing the data gathering process, if it even allows such customization. Whereas a system that supports multiple fluid roles may provide a variety of such styles of interaction.

[4]One could argue to whether the complexity of the scripts written might boost a user performing a particular interaction into more of a programming role. Nonetheless, the natural-language-like syntax used by Coscripter and its lack of complex programming constructs, such as loops or conditionals, suggests that it focuses more heavily on the tinkerer rather than the end-user programmer.

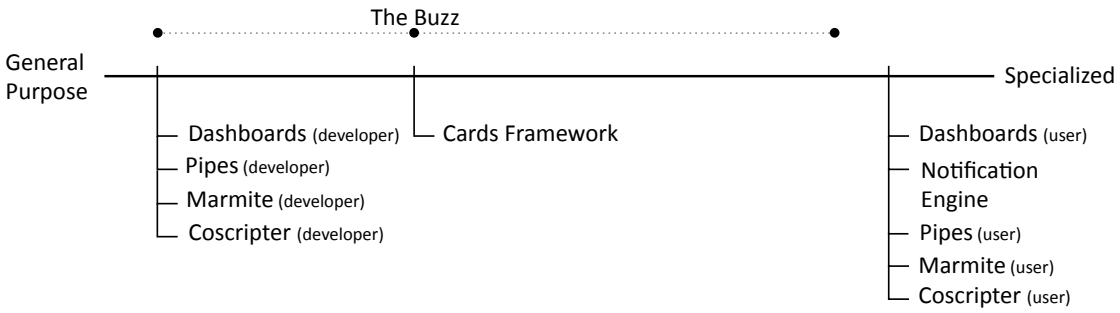

Figure 47: Generality versus Specialization.

6.3.2 Data Extraction and Transformation

The information awareness data pipeline describes the various technical steps necessary to create a single information stream: gathering data from content publishers; transforming the data into a suitable representation; and conveying those data as information to the user. The first half of the pipeline focuses on this data extraction and transformation process.

6.3.2.1 Specialization

Although the data pipeline focuses on the process from a technical perspective, there are many approaches that systems can take to support the user's goal—to tell the system what to show. The Dashboard systems let the end user accomplish this task by selecting from existing content and possibly configuring simple properties. Alternatively, the developer can create any *ad hoc* widget in code.

These two approaches differ in several ways. Not only do they focus on different user roles, as discussed in the previous section, but also, these two approaches differ in generality. When a user selects a particular widget, the options that he or she can configure are highly specialized to the specific widget. For example, a stock widget provides parameters for the list of stocks in a user's portfolio. A flight tracker widget asks for airline, city, and flight number information. These parameters are tightly coupled to the specific task the user is trying to accomplish.

The widget developer interface, in contrast, operates at a very general level of granularity. The interface provides support for loading URLs and painting to the screen. As such, the developer must concern herself with details such as what URL, webservice, or query parameters to supply in order to get the data for a particular airline flight, or how to render a particular piece of data to the screen.

While the widget user operates at a concrete level, the widget developer operates at an abstract level.

Although the Dashboard systems segregate abstract and concrete interactions by user role, these abstractions are not necessarily always so segregated in other tools. For example, Yahoo Pipes provides a combination of general and specialized operators for the Pipes developer to use. For example, a Pipes developer can use an RSS operator to gather photos from Flickr, or he could use the specialized Flickr operator. In the former case, the developer would need to translate photo tags, groups, usernames, *etc*., into the appropriate query parameters to the RSS feed. In the latter case, however, the developer simply provides those properties to the operator, which abstracts away the details of extracting the data from Flickr.

In this way, there is an inherent tradeoff between general-purpose mechanisms and concrete. General purpose approaches tend to be reusable to support a broad variety of tasks, but typically require more effort to tailor the particular approaches. For example, a web crawler is well-suited for gathering a wide variety of data types found on the web. But the user must control many esoteric details of the process. For example, the shallowness or depth of the web hierarchy on a particular server might influence how deeply to crawl. Or, should the crawler restrict itself from following links to other web servers? Many publishers use farms of webservers (*e. g.* www1, www2, ..., or images, ads, content, db, *etc*.), while other publishers provide unrelated content on different subdomains (*e. g.* `news.bbc.co.uk` for news services versus `www.bbc.co.uk` for TV services). Further still, some services, such as Digg.com, provide aggregations to other sites. As such, the desired behavior of the crawler depends heavily upon the particular context in which it is being used and in ways that are difficult to predict *a priori*. In this way, the flexibility offered by such general-purpose tools comes at the detriment of added complexity.

In addition to the technical task itself being complex to express, the user must also control the mechanism using terminology that is unrelated to the user task that he or she is trying to accomplish. Thus, instead of instructing the system to gather all photos on the Flickr photo sharing website that are tagged with "zebra," the user must describe how to crawl `flickr.com` without crawling too deeply or accidentally following a link offsite, but while still getting content from the various servers in the image-hosting farm. As such, there is a cognitive disconnect between the user's goals and the system's expression of that goal.

More specialized approaches can address this cognitive disconnect between the interface and the user's goal. A widget or operator that is tailored for Flickr, can use terminology that matches the user's mental model of the service. But if the user wishes to include photos from, *e. g.*, the Picasa photo sharing service instead, the user must resort to another method. The Flickr operator is too specialized to be useful for such a purpose. In this way, specialized operators lack the flexibility necessary for the user to easily repurpose them to another task.

While specialized approaches may restrict the ability of the user to repurpose them, they do provide greater flexibility through their focus on a specific information stream. For example, not only can a Flickr operator provides its customization interface in terms relevant to a Flickr user, but also it can provide access to data that might not be available through the general approaches. For example, an RSS operator tailored to Flickr through special query parameters may not be able to access the full breadth of features that the underlying data embodies. Many publishers, including Flickr, provide web-based APIs to their data. Through these APIs, a specialized Flickr module can access low-level data features, such as the comments on a particular photograph, that might not be easily accessible through the general-purpose tools.

For each custom API, however, someone must take the time and effort to create an interface to that API. For an awareness system to support the Flickr API, a developer must take the time to implement a widget, gadget, channel, or operator that interfaces with the API. As the publisher updates the API, the developer must update the widget. Multiply this effort by the number of publishers and APIs that the system is to support, and this task can quickly become overwhelming.

Nonetheless, these specialized approaches may be worth the extra effort of writing an *ad hoc* interface to the particular data source. The Dashboard systems, for example, exclusively use these specialized approaches. Each widget is written specifically to present a particular interface to a particular kind of data. Each of the widgets shown in Figure 42 uses separate code to gather data and present the appropriate interface to the user. Most of these widgets were written by distinct authors, yet relatively few users actually write their own widgets. In this way, specialized approaches can take advantage of the user ecology, in which users fall into the different roles of end users, tinkerers, and developers and in which users frequently share their customizations with each other.

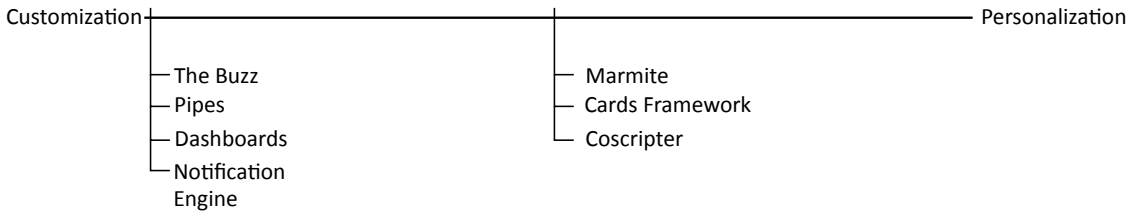

Figure 48: Personalization vs. Customization in Awareness Systems

The Buzz, in contrast, does not partition these approaches across the user and developer interfaces. Rather, it uses a combination of both general and specialized data gathering methods. Thus, a user can view the available gathering methods. If a specialized method is available for the task she wishes to perform, she can use that method. If, however, no specialized method is sufficient for her task, she may be able to use one of the general data gathering methods to gather her data. Allowing the user to choose this method, however, does potentially add complexity to the interface. If too many methods are available, selecting an appropriate method can in and of itself become a cumbersome task. Nonetheless, though this approach, The Buzz crosses the generality and specialization ends of this spectrum.

Thus, there is a tradeoff between the generality of the interface, which allows for greater flexibility, and the specialization of the interface, which allows the designer to focus more closely on concrete user goals. The different approaches may affect both the baseline usability of the system and how readily users may be able to transition between roles. As such, different systems take different approaches to supporting generalizable customization mechanisms.

6.3.2.2 Personalization vs. Customization

So far, we have considered the tailoring of the awareness system in terms of user customization—where the user explicitly expresses his or her desires to the software system. An alternative approach, however, is for the system to observe the user's interaction and to infer from those actions what information is relevant. That is, instead of the user explicitly stating his preferences (customization), the system infers them (personalization) [69, 94].

Under a customization approach, the user dictates her desires to the computer. The user is entirely in control of the process. With the control, the computer will only behave as instructed—any unexpected behaviors are the result of erroneous configuration. That is, the computer will only

do what the user tells it to do. This approach has the advantage that any unexpected behaviors are attributable to the user's configuration of the system rather than to a mysterious autonomous agent. The obvious drawback, however, is that the user must explicitly instruct the software on every action it should take; the system will not make a best effort to infer such actions.

There are, however, varying degrees of personalization. On one end of the spectrum, an interface may be controlled entirely through an agent. This agent might, for example, monitor the user's web browsing habits and infer that the user frequently visits the ESPN golf section and infer the relevant content on the various pages. On the opposite end of this spectrum, the same user would need to explictly specify that the awareness system should monitor certain pages in the ESPN golf section and define what content on the pages is relevant.

Intermediate approaches, however, can blend machine-learning and computer recognition approaches with explicit user customization. Although none of the systems highlighted in this chapter uses full-fledged agent-driven personalization, Marmite, the Cards Framework, and Coscripter do infer content based on what the user specifies. In both Marmite and the Cards Framework, the user browses to an instance of a web page and selects examples of relevant content on the rendered page. Using an approach from Sifter [52], the system uses each mouse click to infer which rendered item the user was referring to, which underlying HTML renders to that item, and whether that item is an instance of a class of similar items on the screen. Thus, if the user clicks on a single article headline on a page listing many articles, the system may infer that the user meant to include all of the article headlines in the listing.

Similarly, Coscripter [63] uses a programming by demonstration [29] approach wherein the user teaches the system how to perform a task by performing it herself. The system monitors the steps that the user performs and infers the actions necessary to perform it. (See Section 6.2.5 for more details.) In this way, Coscripter uses a combination of user-driven interaction with system inference. While not strictly personalization—the system does not learn and evolve—this approach enables the system to demonstrate a task rather than programatically defining it.

Both of these approaches offer their own relative benefits and disadvantages. Personalization can free the user from having to explicitly state his or her preferences to the system. The software simply monitors the user's actions and infers desires from them. The user need not rigorously define,

e. g., extraction rules, scheduling concerns, data dependencies, *etc*. In contrast, customization can allow the user to explicitly define complex behaviors that may potentially lie beyond the capabilities of the system to learn.

Consider, for example, a user who regularly commutes by subway, but drives to work on Fridays. Depending on the dimensions a personalization system uses for learning, the system may be unlikely to infer that it should show transit system information during rush hour Monday through Thursday, but traffic information on Fridays. Even if the system might be able to identify such patterns, it may take some time for the system to learn. Thus, the customization system could potentially offer the user greater control, but at the expense of some effort.

These approaches, however, are not mutually exclusive. For example, a personalization approach could integrate user customization by overriding various dimensions with a high weight. Alternately, the system could use personalization to complement the customization subsystem. Marmite and the Cards Framework use this approach by allowing the user to specify some elements manually, while the system infers others. A system could carry this further by, for example, inferring relevant widgets from user behavior. Thus, a user who frequently visits the Foxtrot comic strip might find the awareness system automatically subscribed to a Foxtrot feed, in addition to the feeds to which the user had already explicitly subscribed.

As it is implemented, The Buzz lies squarely on the customization end of this spectrum. The system provides no capabilities to infer behaviors based on the user's actions. Intermediate, inference-based extraction techniques could, however, be added into The Buzz fairly easily by developing new harvesters or scrapers that allow the user to, *e. g.*, use a programming-by-demonstration approach to define extraction rules. If such a harvester or scraper were added, the system could potentially extend toward the personalization side of this spectrum.

6.3.3 Data Presentation

The second half of the awareness data pipeline focuses on presenting the extracted data to the user. There are many approaches to visualizing information [19, 95] and to designing informative interfaces suitable for promoting information awareness. These approaches may be highly coupled to the particular data being conveyed, as in most Dashboard widgets, or they may be more general

in nature, as in the Notification Engine.

6.3.3.1 Awareness Interfaces

Awareness systems typically use ambient, peripheral, or notification-driven interfaces to convey their data. Although there is no strict consensus as to the precise definition of these terms, I will use the following distinctions in considering these interfaces. Ambient interfaces can be characterized by their use of subtle perceptual channels, which remain beyond the user's attention, but still convey data. Similarly, peripheral interfaces remain beyond the user's active attention under normal use, but are readily available to convey information should the user redirect her active attention. Finally, notification systems assert themselves in such a way as to attract the user's attention when various events occur.

Pousman and Stasko's taxonomy describes different dimensions relevant to the design of ambient and peripheral awareness interfaces [87]. Although these dimensions of the awareness interface presentation itself are beyond the scope of this document, each of these different approaches will influence the capabilities of the awareness system. An ambient interface, for example, is useful to promote relatively simple awareness dimensions, but is unlikely to be scalable to convey complex data without overloading perceptual channels. Furthermore, an ambient or peripheral awareness system will need to avoid distracting transitions [54] and make judicious use of animation [85] if it is to avoid unwittingly drawing the user's attention. As such, the high level class of the awareness system will influence the sorts of customizations that the system will need to support.

Although there are many approaches that these awareness systems have taken to convey individual data streams to the user, there are two techniques that have been used extensively to support the aggregation of multiple data streams: temporal multiplexing and spatial multiplexing. Under a temporally multiplexed approach, the system conveys different information sources and presentations at different times, as in slideshow-driven systems, including The Buzz [34, 111, 73, 89]. In contrast, spatially multiplexed systems depict multiple information sources simultaneously, as in at-a-glance informative dashboards [6, 96, 18, 45].

In both of these approaches, the content from each individual data stream is treated separately from the others. Although it is possible to merge content from multiple data sources in such a way

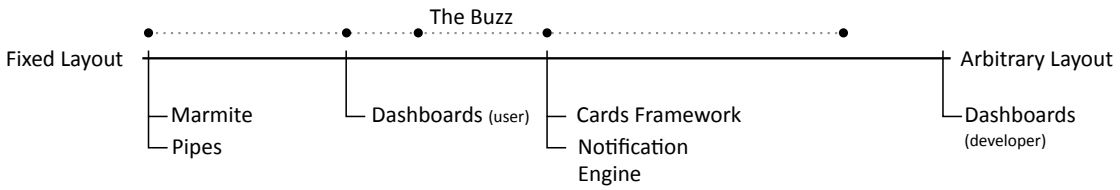

Figure 49: Customizing the Presentation.

that their presentation blends elements of each other, most general-purpose awareness systems use the multiplexing approaches. Such a blending typically requires more complex visualization techniques than are typically available in most general purpose systems. Furthermore, such integration would be relatively data-dependent, restricting the general-purpose methods that could be used. As such, current awareness systems tend to depict individual data streams independently, but combine them using one of these spatial or temporal approaches.

6.3.3.2 Customizing the Presentation

There are many approaches a system can take to enable the user to customize the presentation. Figure 49 depicts various approaches that have been taken. RSS readers, for example, used a fixed presentation style—the user cannot control how the data are presented. Similarly, the Dashboard widget subscriber typically is unable to control the presentation of the content. He may be able to control what data are shown (*e. g.* the particular stocks to display), but not how to do so. Still others allow the user to define any arbitrary rendering of the data, as is the case for the Dashboard widget developer. Most systems, however, provide intermediate approaches through which the user can express a more limited degree of control over how to render data to the screen, usually with less significant investment of effort.

These approaches can be broken down into three coarse categories: fixed presentation; template or layout based; and programming based. These three categories are rough clusters of different approaches that have been taken. As such, different systems and different user roles within those systems may demonstrate distinct interaction styles within the same cluster. Nonetheless, the class of customization approach used influences the kind of interactions the user will perform with the system and the control she will have over the view.

Fixed presentation approaches restrict the user's ability to tailor the presentation. The user may

be able to select properties of what to display, but not how to do so. For example, RSS readers allow the user to control what content to show, but the presentation is typically restricted to a small set of standard representations. Similarly, the Dashboard systems allow the user to subscribe to various widgets, but the user cannot control the presentation within an individual widget. RSS readers typically use this approach because the data are too heterogeneous. Although RSS feeds provide machine-readable structure for the title, author, and timestamp of various data, the data themselves are fairly free-form (though typically plain text or HTML). Widgets typically use this approach because they are *ad hoc* creations tailored specifically toward the data they depict. As such, the tailoring is delegated to a benevolent big brother: the widget developer.

To give the user additional control, various systems use template-based or layout-based approaches wherein the user can define the spatial placement of the various data. The Buzz, the InfoCanvas [96], Cards Framework [32], Notification Engine [43], and Dashboards [11, 75, 4, 6] use this approach. The Buzz provides users with a default layout template through which the user can adjust the regions on the screen in which different data will be drawn. Under the InfoCanvas, the user can define points or regions on the screen in which to depict different data elements using a particular rendering method. The Cards Framework lets the user layout data within individual cards and define collections of those cards. The Dashboards let the user create arbitrary layouts to assemble the various widgets, while the Notification Engine lets the user layout not only the composition of the various data sources but also the various data elements within a particular data source's presentation.

Although all of these systems use very different presentation styles, they all support the user at making similar sorts of customizations to those presentations: defining positions on the screen in which to draw data and optionally choosing parameters to control how the data are rendered to that particular location.

Finally, the Dashboards also give the widget developer full programmatic control over the content within the widget's region on the screen. The developer can create any arbitrary rendering expressible through the combination of JavaScript, HTML, and XML, although most widgets typically use relatively simple combinations of images and text. Few widgets actually define their own visualization techniques to render complex data.

Similarly, The Buzz allows plugin developers to write new visualizers. These visualizers give the programmer full control over the contents of an individual region. With such a plugin, a developer could define new representations of the data using Python code.

Although most of these systems fall primarily within one of these categories, it is important to observe that different interactions within the Dashboards fit within each of the categories. The user selecting widgets has no control over the presentation within the widget itself, but does have control over the composition of the collection of widgets. Furthermore, the developer creating a particular widget has the highest degree of control over the content within the widget itself, being granted essentially a blank canvas with which to work.

6.3.4 Beyond the Data Pipeline

So far, we have considered customization through the lens of the awareness data pipeline. This pipeline is useful in considering the various technical stages required to transform data from the publishers to information representations for the user. But these individual data streams are not always independent. Thus, the data pipeline does not reflect the challenges that arise when combining multiple data presentations into a single awareness system. This section surfaces above the data pipeline to consider some of the broader dimensions beyond individual streams.

6.3.4.1 Specifying Relationships

To better support the relationships between multiple data streams, the awareness system may need to be able to model aggregation, integration, and the mashing-up of the various data. In this section, we consider the effort versus expressiveness space for specifying such data relationships. Under this model, we examine various awareness systems that model the relationships between data in various ways and consider both the power of that expression and the relative effort required to make those kinds of expressions.

Figure 50 shows several different approaches that have been taken to model the relationships between data streams. Most awareness systems fall under the first grouping and do not provide any explicit support for the modeling of such relationships. Dashboard, the Notification Engine, and Coscripter treat individual data streams or processes as separate. The user can not express dependencies between data in one stream and the data or behavior of another. For example, the user

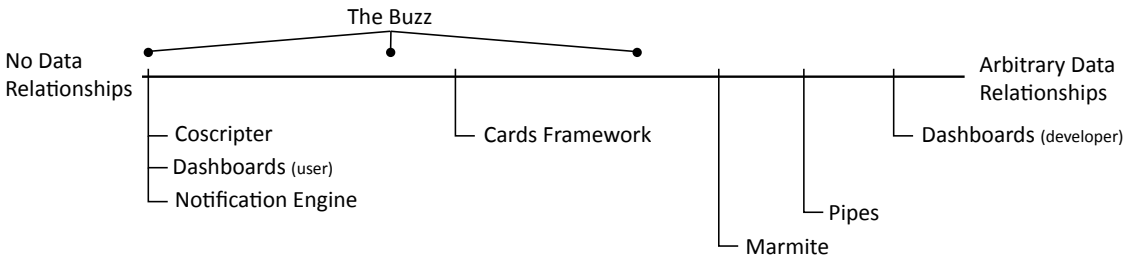

Figure 50: Expressiveness of data relationships.

could not create a stream that embeds the output of another, in part or in whole.

The Cards Framework, however, enables the user to identify relations between elements in the data. By drawing a connection from an element in one card to an element in another card, the user can specify that a second element derives from the first. When a new card is created, the first element can be propagated to the second in another card. Thus, a card that represents restaurant listings and another that represents restaurant reviews can combine their data into a composite card, with the two restaurant name fields from the two different data providers linked into a single view.

In this way, with a few relatively intuitive strokes of the mouse, the user can specify links between data in different sources and integrate data from multiple data sources. The relative simplicity of this style of interaction to the kinds of relationships the user can specify make this approach especially elegant. Furthermore, the incremental cost of providing this kind of interaction above fixed approaches is marginal. If the interface allows the user to examine the data prototypes, then specifying the relationships minimally alters the complexity of the interface. It well embodies the notion that a slightly more complex task should require only a slightly more complex interaction and interface.

The next step up on the expressiveness scale is embodied by systems such as Marmite, which allows the user to move beyond specifying simple relationships between the data. With Marmite, the user can describe linear data flows and transformations to apply to the data. Thus, in one step the system might be able to extract the name of a restaurant and it's reviews, while the next step might be able to transform the street address of the restaurant to a latitude and longitude suitable for placement on a map.

Moving farther along, Yahoo Pipes extends the data flows the user can specify to support the

Figure 51: Support for and encouragement of sharing/browsing content.

creation of nonlinear data flows. Creating these flows comes at the cost of a more cluttered interface, where the flows are not necessarily as straightforward for the user to follow. Instead, she must be able to trace the flow of data through branches and merges and even across into other Pipes. Nonetheless, the system does support linear data flows in a way that, were the two systems actually related, could be implemented in a way that negligibly alters the complexity of the interface when creating linear data flows. As such, although the transition between simply specifying relationships to specifying linear data flows is substantial, the transition from linear to branching data flows reflects only the complexity of the underlying data flow itself.

Finally, the Dashboards allow the user to define arbitrary widgets in code. As such, there is a significant hurdle to cross when transitioning to this style of interaction. Even simple programs present a significant barrier to all but the most skilled of users.

The Buzz, however, provides capabilities at various points along this space, depending on the role of interaction the user is exhibiting. While channels themselves cannot reference data from other channels, individual channels can potentially gather data from multiple data sources, depending on the configuration of the various harvesters and scrapers used. Through The Buzz's Wizard-like interface, the user can define such data relationships in a space that lies between that of the Cards Framework and that of Marmite. This interface does not support the same kinds of relationships definable by Marmite and Pipes, but it also poses a more constrained set of steps to the user, limiting the choices she must make to express herself.

6.3.4.2 Sharing and Browsing Content

Recall from Section 2.1 that many users rarely create their own customizations from scratch. Instead, many borrow customizations from others and use those as a starting-off point [66]. In the context of many systems, this sharing behavior takes place in an *ad hoc* fashion. The system itself

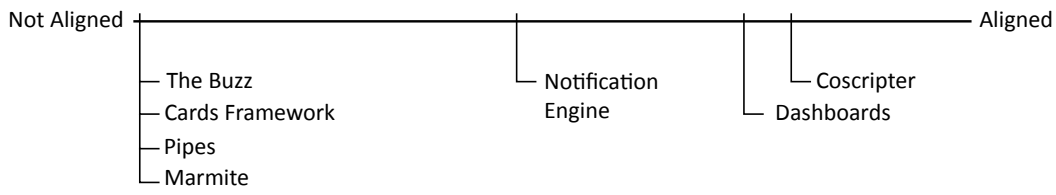

Figure 52: Alignment of customization task to user task.

provides no explicit support for sharing. Instead, users modify each others' configuration files and share them using established communication channels (*e. g.* email).

Nonetheless, because of the recognized patterns of sharing amongst users of customizable software, many systems do provide such explicit support. The various Dashboards provide support directly within the interface to browse available widgets that have been created and shared by other users. The Buzz extends this capability by allowing users to share their customizations from directly within the interface. (It also allows users to use external methods, such as email, to share their channels.) Coscripter further adds the ability to suggest relevant scripts based on the the user's browsing habits, in addition to allowing users to submit their scripts to a shared repository from within the Coscripter interface. In this way, the design of the system helps to lower one of the barriers to sharing. The user need not make a deliberate decision to leave the interface and package up an artifact for sharing. Instead, she can make a more impulsive choice. Finally, Yahoo Pipes not only allows such sharing within the interface, but also shares derived pipes by default—if a user wishes *not* to share a pipe, he must deliberately choose not to pass it around.

6.3.4.3 Customization-through-use

Although we have focused on customization as a deliberate user task, some systems provide for customization-through-use. As a user interacts with the system, that use itself provides the customizations to the content. Instead of being a meta-task, customization itself becomes the task. Compare, for example, the Notification Engine [43] and the Notification Collage [45]. Despite the similarities in their names and in the screenshots of the systems, the two focus on different problem domains. The Notification Engine supports the user at monitoring web sites for changes, while the Notification Collage promotes group activity awareness. As such, they also provide very different styles of interaction.

Under the Notification Engine, the user explicitly customizes the system, defining mechanisms to extract notifications from web pages. The user's primary goal is to receive notifications, but the user's task in interacting with the system is to configure the software to extract those notifications from the data publisher. In this way, there is an added layer of indirection between the user's primary goal and the task that he must perform with the software system.

Under the Notification Collage, however, those two tasks align. The user's goal in interacting with the system is to share awareness artifacts or to browse those artifacts that have been shared by others. As such, posting notifications directly accomplishes the first user goal, while viewing the current notification collage directly accomplishes the latter. In this sense, use of the system and customization of the system are the same.

Such alignment is not necessarily feasible in many awareness contexts. The Notification Collage is particularly able to leverage this kind of customization because of its focus on user-generated content. Nonetheless, this difference demonstrates the importance of this last dimension: the alignment of the customization task with the user's system-use task.

6.4 Customization Space Summary

Figure 53 presents an overview of systems in the awareness customization space. It breaks the space into five regions, or clusters, of systems that operate in a similar fashion. Within each of these regions, systems that are more similar to each other are drawn closer together. Systems that function within the same cluster but are drawn farther apart are less similar to each other.

The first of these clusters is comprised of "Demonstrative Interfaces." These are systems for which the user points out, or demonstrates, the various information entities in which she is interested. In the Cards Framework, Dapper, and Sifter, this interaction consists of clicking on entities in web pages, such as the title text of an article, the photo of the article author, or a comic strip. In this way, the user's customization action is primarily to indicate "This is what I want." The Cards Framework further allows the user to define relationships between data by drawing connections between their representations, hence pulling it farther from the pack. The Notification Engine similarly lets the user identify visual regions of web pages to control what content to extract. It, however, uses a different technique to accomplish this extraction, operating on the rendered output

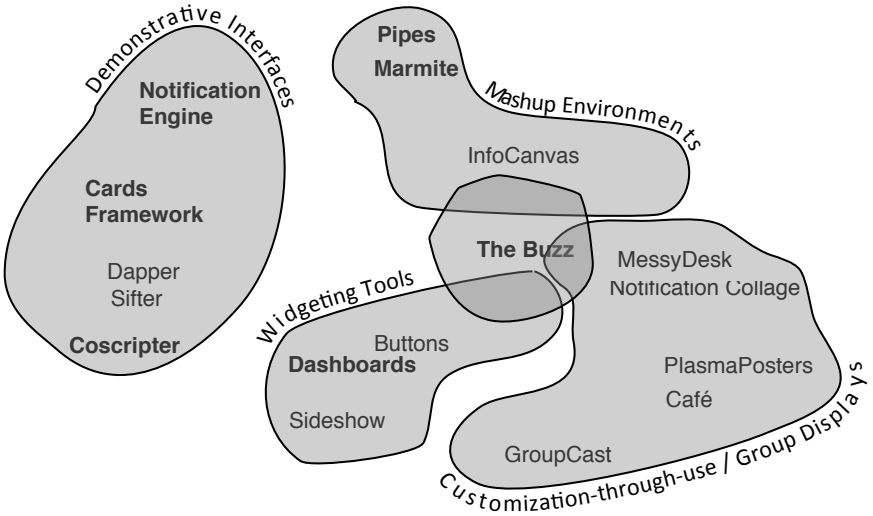

Figure 53: An overview of systems in the awareness customization space. Systems drawn closer to each other within a region are more similar than systems drawn farther apart. Those in bold are spotlighted in this chapter.

rather than making an inference as to which data was identified. Finally, Coscripter records the actions the user takes in the web browser in order to support replaying those actions. In this way, the user demonstrates a task once and the system records it. As such, while all of these systems use a demonstrative style and some even use a programming-by-demonstration approach, they are not all programming-by-demonstration tools.

The next cluster represents "Mashup Environments." These are tools whose explicit goal is to support users at integrating data from multiple sources into a single source. Both Pipes and Marmite accomplish this mashing up through their own visual programming interfaces. Through these interfaces, the user can construct a data flow, where data are gathered from their sources, connect to various operators that manipulate them, and aggregate them into a single stream. The InfoCanvas operates in a very different way. Instead of mashing up the underlying data, it mashes up their representations, creating an aggregate display. These representations use simple, user-selectable rules to govern how individual data entities are to be drawn. All of these systems focus primarily on integrating data from multiple sources and changing their data interfaces.

The "Customization-through-use" or "Group Displays" cluster represents two different ways of

looking at each of the systems in this space. They all share in common that the user's interaction with the system to perform any customizations is the same interactions the user would perform to accomplish his system-task goals. For example, the Notification Collage provides a space into which users can post various notifications—text, images, weblinks, *etc*. Through the user's normal interaction, he customizes the content and presentation of the display. These systems are also called "Group Displays" because they all happen to focus on supporting group collaboration or awareness. In most of these systems, any customizations that a user makes to his display are replicated to all of the other displays. In this sense, the individual user does not take ownership of the display.

"Widgeting Tools" focus on a bicameral style of customization, where users are divided into two camps: widget authors and widget users. The authors create new widgets, which represent a single kind of information or provide a simplified interface or shortcut to a common task. The widget users can use these created widgets to create their own collection of these pre-packaged information tools.

Finally, The Buzz overlaps with these last three regions because it shares many interaction styles in common with them. Through the way the user can combine harvesters and scrapers and can create layout templates to integrate data from multiple sources, The Buzz supports elements of the creation of mashup environments. Although The Buzz does not provide support for customization-through-use, its presentation style makes it amenable to use for group displays. Its capabilities with regard to supporting direct interaction are limited, but as a display space that can be shared among members of a group, it may be able to support shared community awareness and involvement. In fact, this very goal is at the origins of The Buzz [110]. The third region with which The Buzz overlaps is that of widgeting tools. Just as these tools support different users creating and sharing their artifacts, so too does The Buzz. The Buzz's primary distinction in this regard is that it attempts to support users at further refining their widgets (channels).

6.5 Dimensions Overview

Figure 54 depicts an overview of the dimensions described in Section 6.3. It presents the seven different dimensions using a parallel coordinates representation. For some of these dimensions, there is no "good" or "bad" meaning to be ascribed to a whether a particular system lies higher or

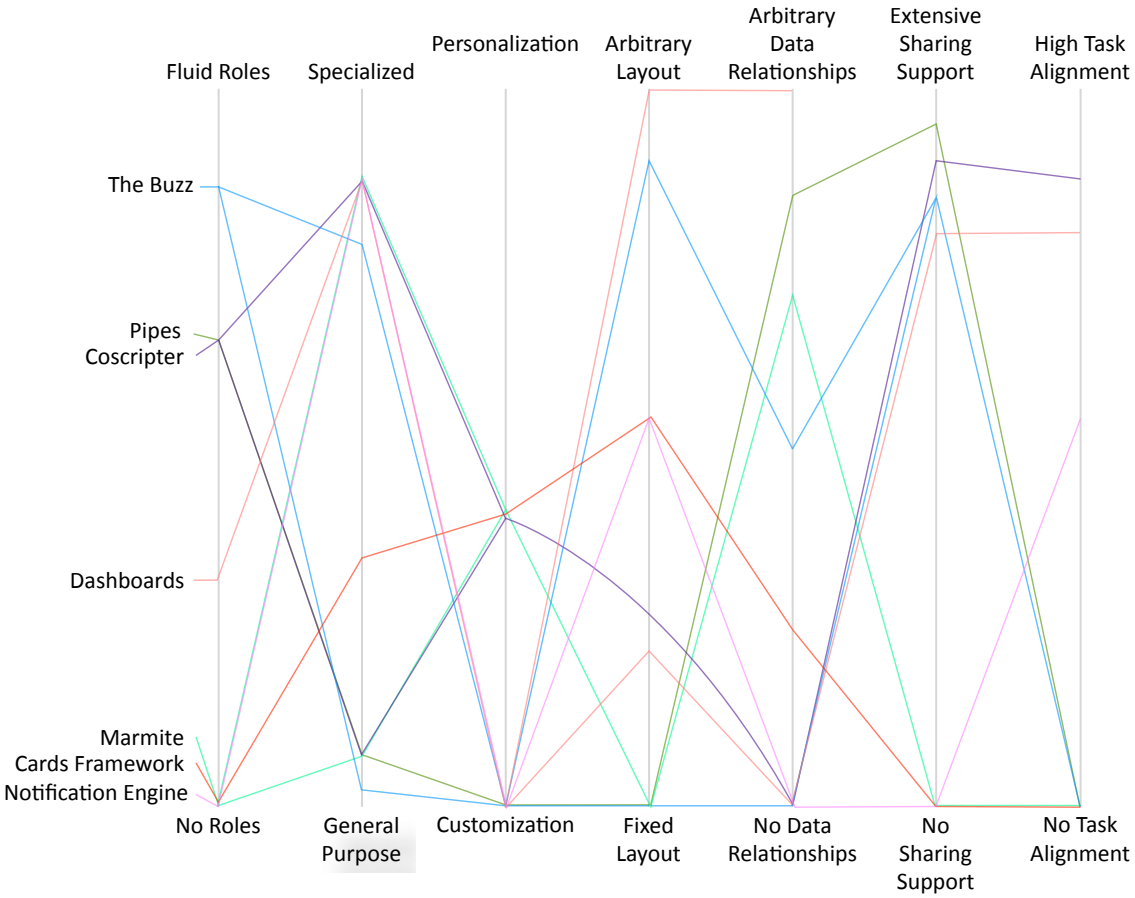

Figure 54: A summary of the information awareness customization space.

lower in the chart. Where there is a perceived better or worse in a particular dimension, the better position is oriented toward the top. However, the ideal position along the axis may depend upon the goals of the system.

Additionally, some of these systems may not occupy a fixed point on these dimensions. For example, both The Buzz and the dashboards support the expression of arbitrary data relationships or of no data relationships, depending on which style of interaction the user is in.

Along these axes, it is important to note that The Buzz is not unique in its position on any one dimension in particular. What does make it distinct, however, is the particular combination of positions it occupies. For example, on the Specialized *vs.* General Purpose axis, The Buzz occupies a range along the axis that depends not on what role the user is in but on what particular interface she is using for a particular task. For example, a user can create a channel from a Flickr photostream using a webcrawler to extract photos from Flickr web pages, using an RSS harvester to extract photos from an RSS feed, or using the dedicated Flickr harvester to gather photos with particular tags. Depending on which interface the user chooses, the system can provide a general-purpose or a specialized customization interaction. Thus, while other tools may also occupy the same points in the space, they do so in a very different way.

Furthermore, when we consider the various dimensions of the space, we can see that The Buzz ranks highly on most dimensions with the exception of the Customization *vs.* Personalization dimension. What this chart shows is that The Buzz occupies at least a reasonably good position across a large number of these dimensions, supporting a combination of customization styles that are not common among other systems. The Buzz provides support for fluid user roles, allowing a wide range of specialized or general customizations styles, allowing users a good degree of control over the presentation layout of their data, and with a high degree of extensive sharing support. By combining flexibility and generality, The Buzz is able to provide a "jack of all trades, master of none" approach to supporting customization.

6.6 Challenges and Limitations

Although the various systems described in this chapter take very different approaches to supporting customization and support different stages of the awareness pipeline, they all share in common the

notion that they help the user to access and repurpose data. They enable the user to create new uses and new representations of data made available by the various content publishers. This repurposing of data, however, presents significant challenges and limitations to the use of the data. By the very nature of repurposing data, the goals of the user or of the software system differ from the goals of the publisher.

These differences can manifest themselves as both technical and social challenges. Technically, the publisher may make the data available in a format that is undocumented or that does not necessarily align with the user's particular intended use. Socially, the publisher's business model might not be compatible with alternative uses of their data, leading many publishers to discourage such repurposing activities.

These differences in goals between the user and the data publisher present challenges and limitations for any system that aims to repurpose data, as is typically the case for information awareness tools. Section 6.6.1 discusses these challenges in general. Section 6.6.2 further discusses challenges that arise as a result of The Buzz's approach.

6.6.1 General Limitations

Information awareness systems typically repurpose the data that they convey. That is, they use data from a publisher in a fashion other than explicitly intended by the publisher. Publishers typically have a particular intended mode for the consumption of their data, such as viewing it through their web site. Some publishers will intentionally obfuscate these data so as to discourage data scraping or to hinder embedding the data elsewhere. Other publishers, in contrast, may actively encourage the repurposing of their data, explicitly providing these data in more accessible formats. Most, however, take neither approach.

There are various organizational reasons why a publisher may wish to discourage or encourage the use of their data. The Weather Channel's `weather.com`, for example, actively discourages data scraping. Different page loads to the same data may produce subtly different templates with alternate underlying HTML. These templates themselves are also frequently changed. These changes may not affect anyone accessing the data through a web browser—most people are unlikely to notice—but will confuse a web scraper that expects the data to be in a particular format.

The Weather Channel performs this obfuscation because their business model depends on the value of their weather data. For consumers, they rely on ad impressions on their web site. Any consumer use of their data that reduces this number of ad impressions will affect their revenue stream. In this way, The Weather Channel has a business interest in preventing tools from scraping their data.

The U.S. National Weather Service, however, makes their data readily available in various formats. Like The Weather Channel, weather data are available through a web interface. However, these data are also available in various formats. There are even web APIs for querying raw weather observations, forecast data, and radar imagery. One reason why the National Weather Service may so actively encourage the use of its data is that it is a government organization. Its operations are paid for by the taxpayers; consequently it operates as a public service. The Weather Channel, by contrast, needs to generate its own revenue. In the former case, the data are a public good; in the latter, they are a business asset.

Any meaningful information awareness application will need to repurpose information in some fashion, even if that repurposing merely involves collecting information as-is with other information. Such repurposing activities disalign the goals of the awareness application from the goals of the publisher. When the differences in alignment are minor, such repurposing may be relatively simple to accomplish, such as when a publisher makes available an RSS feed that includes exactly the content the awareness system needs. When the two goals are not well-aligned, however, such repurposing activities may be more difficult. Any awareness application, therefore, will face challenges relating to the relative alignment of its goals and the data publishers' goals.

In terms of the data pipeline, these common challenges arise at the front end of the pipeline: in gathering the data and transforming them into a suitable representation for the awareness application's purposes. The first of these challenges arises from addressing limitations: the system must be able to find the data it is to operate on. Different challenges then arise depending on whether the data are in a machine-readable semantic format, such as the National Weather Service APIs, in a format intended for human presentation, such as a typical web page, or if the system uses surface representations rather than the underlying data encoding. The next few sections focus on these general challenges.

6.6.1.1 Addressing Limitations

Regardless of how an awareness system is going to use the data, it must somehow be able to refer to it. If the data comes from a dedicated API, this task may be relatively straightforward—the API probably has well-defined entry points. When the system is repurposing data, however, such entry points may not be well-defined. For example, some content management systems (CMS) generate URLs for web pages that relate solely to the content and that may change whenever the dynamic content updates. For example, entries in the College of Computing faculty directory at one point had the format: `www.cc.gatech.edu/content/view/12/345/` where the numbers might change as the structure of the underlying data updated. In this way, these URLs are both meaningless to a human and are not persistent. Some changes to the underlying data change the URLs for the same content.

These addresses might also change according to a more meaningful format. For example, many online comic strips use a date-driven URL for the image containing the most recent comic. The web page for the comic strip typically exists at a fixed URL and updates the URL for the image embedded within the webpage whenever a new comic is published. In this case, the publisher's model implicitly expects that data consumers will visit the webpage. The fact that the image is not at a known URL is not a problem under the publisher's use-model. Addressing that image, however, requires either reverse-engineering the naming scheme for the image address, or adding a level of indirection to load the web page that embeds the image and then identifying the relevant image in that page. Both of these approaches are fairly fragile: the naming scheme may be complex or may change. The format of the embedding web page might also change.

Finally, some publishers intentionally obfuscate the entry points for their data. For example, Gas Buddy publishes listings of gas prices at local gas stations, helping people to find the cheapest gas in their area. To access their listings, the user must first load a "landing page." This page uses JavaScript in the browser to generate a unique URL and key for that particular user. Although this method is transparent to the user, it requires significant sophistication for an automated tool to be able to mimic the complex behaviors of a web browser. As a result, a system that does not support the specific behaviors of the web browser will not be able to access content that is protected in such

a way. Furthermore, CAPTCHA-driven systems [103, 104], which attempt to require a human to solve a problem, will make it difficult to automatically gather data without human intervention.

6.6.1.2 Limitations of Extracting Content from HTML

The web evolved around the transmission of data in HTML documents. As such, most information will be encoded within such web pages. While these pages are good at integrating multiple media into a single document for presentation to a human user, it is difficult to extract semantic meaning from those data. HTML is a presentation language, describing how to display data. It does not provide semantic information about the data being displayed. Thus, the document may describe that the author of an article should be drawn in a particular font face, but it does not indicate that the text drawn in that font face is the author of the article.

As such, awareness tools that extract content from HTML documents are limited by their ability to reverse engineer the underlying structure of the data elements within the page. Such parsers are often fragile, either being so strict as not to handle an extra HTML tag that might arise, for example if an HTML `<table>` is included within the body of an article that is otherwise normally encoded within a layout `<table>`. Or the parser might be too tolerant, accidentally extracting spurious content, for example when a field is omitted. Because many web pages are written by hand without any formal validation, many web pages involve broken HTML that does not conform to any of the numerous HTML standards. Most web browser include a "quirks mode" to handle such HTML, leading to web pages that appear to work in some web browsers, but which are not well-formed.

Because these pages are intended for human consumption, their layouts may frequently update as the publisher modernizes its looks or undergoes a rebranding campaign. As such, a data extraction template that was robust for a particular web site might suddenly fail when the publisher changes the presentation of the site.

Finally, many web pages use more complex markup techniques, such as embedding plugins or using dynamic code, such as written in JavaScript. These pages contain dynamic content that must be loaded from another source and which may consist entirely of executable code. Extracting data embedded within these plugins is frequently infeasible.

As such, there are many sorts of data in web pages that are difficult to extract using existing

web scraping methods. The data may be difficult to interpret or may involve executing potentially untrusted code or embedding a complete web browsing environment. Finally, these methods are fragile against changes, because no formal "contract" exists to describe the data format.

6.6.1.3 Limitations of Extracting Content from Semantic Formats

Machine readable semantic formats can help to ameliorate the problems with extracting data from web pages. These formats provide an encoding of the data that includes semantics of their meanings. For example, in an RSS feed, there is no presentation markup; instead, items such as the title, date published, article author, and article contents are included in the feed. As such, using an RSS parser, one can extract these various data elements from the document. The format is also extensible, so new elements can be added to a feed. Thus, an RSS feed containing weather data might also include fields for the forecast high/low temperature. The meaning of these fields, however, must be discerned by a human programmer.

These semantic formats, therefore, can be useful for identifying different data, but require specialized code to be able to handle the various data elements that are relevant to the system. A system that handles RSS, for example, may be able to automatically extract the basic data from an RSS feed, but might need a specialized extension to handle additional data within the feed. Furthermore, data that does not necessarily fit within the structure of RSS will require additional specialized handlers.

For example, many web sites provide web-based APIs to their data. These APIs provide rich access to the underlying data, but require a specialized interface to access those data. For each such API, some programmer must have written an interface to it. Thus, an awareness system that uses many such APIs is limited by its ability to communicate with so many different interfaces, potentially increasing the complexity of the underlying software.

6.6.1.4 Limitations of Using Surface Representations

The previous two sections describe limitations that affect systems that parse data from various formats, whether intended for human or machine presentation. Instead of using the underlying data format, however, an awareness system could use the surface representation: the artifact shown to the user. That is, instead of extracting data from the underlying HTML for a web page, the system can simply extract a clipping from a rendered web page. At its most naïve, this approach simply

involves rendering a web page and printing its contents to an image buffer. Thus, any data that the web browser is capable of displaying, the awareness system is capable of accepting as input.

While this approach solves many problems, it is not without its limitations. First, the naïve approach described above still requires timing problems to be addressed. For example, many web-pages take some time to load. While a web browser may be able to indicate when it has finished loading the collection of elements of a web page, plugins may indirectly load other data and present them through an animation. If the system screen-scrapes the rendered output too soon, it may end up capturing an intro-animation instead of the desired content. If it waits too late, the animation may have cycled on to other content. Without semantic understanding of the content, it may be difficult or impossible to appropriately guess the correct timing. Human configuration or intervention be able to help reduce this limitation.

In addition to timing limitations, most web pages include extraneous content to what the awareness system needs. For example, the web page may include navigational links and ads, in addition to the desired data. Although some methods do exist to handle some of these challenges [43], they suffer much of the fragility of web data scrapers, as described earlier. Small changes to the web page, whether by a change to the publisher's branding or by embedding an ad of a slightly different dimension, might have radical effects on the layout of the page. Any approach that involves reverse-engineering the data publisher's format, whether it uses the underlying data or the surface representation, will involve complexities and ambiguities that contribute to the frailty of the method.

Finally, the surface representation is useful for conveying the data as-is to the user. It does not, however, allow the system to infer underlying semantics from the extracted representation. If such semantics are necessary, the system must somehow augment its data extraction beyond using a simple image buffer, suffering the challenges and limitations of existing data extraction approaches. As such, while surface representation approaches can allow the system to handle a much richer set of data sources, it is not a magic bullet. There are still limitations on the kinds of data the system can handle, and the approach still limits the kinds of uses the system can make of those data.

6.6.2 Limitations of The Buzz

In addition to the general limitations that all information awareness systems face, the design of The Buzz introduces more specific limitations. The most significant of these limitations derives from its extensive decomposition of the data pipeline. The Buzz architecture breaks the stages of the data pipeline down into separate independent components and provides relatively little feedback capabilities. As such, the various stages are treated linearly: the output of one stage feeds into the next stage.

In this way, the behavior of a particular channel is determined solely by the combination of its inputs and the particular data gathered. Thus, a channel cannot adapt the data it gathers based on behaviors that occur at later stages of the pipeline. Ideally, for example, all behaviors relating to the presentation of the data will occur by one of the visualizers. However, if a data harvester is to short-circuit its data gathering based on the number of large images it has extracted from a web page, it must calculate image sizes in the data harvesting process. The process is not bidirectional. Feedback is limited to maintaining a snapshot of the previous state when a data gatherer operates.

Thus, a harvester can reuse previous data values if the same entity exists in a later harvesting operation. For example, if an RSS feed has three new entries and ten old entries (that the harvester had previously gathered), then the harvester need only fetch the three new entries. This limitation, however, prevents the harvester from changing its behavior based on operations that occurred in a later stage. For example, a harvester cannot modify its behavior to include more data or to exclude an item based on whether a user has clicked on it when it was being displayed. Such a feedback mechanism does not currently exist.

Additionally, The Buzz treats each channel as an independent data stream. The content or presentation of a channel cannot depend on the contents of another. Instead, each data gathering process and each presentation process for each channel is treated independently from those of all other channels. Thus, if there were a weather channel and a pollen forecast channel, it might make sense to merge those two data sources, such as by embedding the pollen forecast within the other weather forecast items in the display. However, the current design of The Buzz does not allow for such integration.

Hooks do exist to support such integration in a future implementation of The Buzz, however, by allowing an item in one presentation to reference the data in other channels. Nonetheless, even with such hooks, the current design does not support feedback from other channels, so a traffic channel could not update its behavior depending on data in a weather channel (for example to suggest routes that tend not to be backed up in rainstorms).

Finally, the presentation template approach used by The Buzz treats each individual region independently from the others. A region cannot resize itself or adjust the overall layout based on the content in it. If two regions are next to each other and one region contains a large image while the next region contains a small image, the two regions cannot share their allotted space. This approach simplifies the user's task of mapping data items to regions on the screen, but it also limits the kinds of layouts the user can create. Furthermore, while The Buzz does provide support for extending many of its components, the layout mechanism is not such a component. A developer can write a plugin to extend the rendering of data within a specific region, but not the layout of all of the data items within the collection of regions.

As such, The Buzz's extensive decoupling of its various components allows for a certain degree of flexibility within an individual component, but limits the interactions and influences those components can have on each other. Earlier components in the pipeline operate independently of later components in the pipeline due to the lack of a strong feedback mechanism, and channels operate independently of each other.

CHAPTER VII

DEPLOYMENT STUDY

In order to get a better sense of how people would use the software on their own, we conducted two pilot deployment studies within the College of Computing. The first of these studies involved deploying the software to a group of six members of the College of Computing community, some of whom had participated in the earlier interviews. The second study involved running the software on a large, public display in a heavily-trafficked lounge area.

Both of these studies were relatively small in nature, using only members of the local College of Computing community. Usage observations, therefore, can help to suggest how people might use the software on their own, but will be biased through the lens of a community of technologists. Because of this demographic, participants are more likely to possess a stronger degree of technical skill and will probably have a stronger proclivity to use it. Nonetheless, these observations can help to suggest real world usage patterns and to provide insight into how well the system is able to support its goals.

7.1 Desktop Deployment

Our goals in conducting these deployments were to gain insight into how our community would use the software capabilities provided. Would people customize their awareness experience, and if so, how? Would users stick to the default channel lineup, perhaps making minor adjustments to those channels? Would they browse available channels created by other users? Would they share their own?

We recruited six members of the College of Computing community. These participants were colleagues of the primary investigator on this project, including two members of this dissertation committee. As such, these participants had at least passing knowledge of the goals of research, and one participant was involved in design meetings. The remaining participants knew of The Buzz, but had not been involved in its design or creation.

Participants were asked to download and run the software. When the software was initially

launched after downloading, it presented the user with a brief overview of the system and included a reference to a web-based listing of available channels. Additionally, participants were encouraged to ask for help with the software if necessary, and to suggest any channels that they might like to see.

During the deployment, The Buzz collected log data on the participant's own machine. This log data recorded various user events, such as advancing or replaying slides, pausing the slide playback, configuring channels, and browsing or sharing channels. Each time a user performed one of those actions, the software recorded the event to a local file. By storing the file on the user's own computer, we aimed to mitigate participants' potential privacy concerns. Thus, although the software logged all events, the participant had to explicitly choose to send those log data to us.

Using these log data, we attempted to identify capabilities of the software that our participants used. Of the six users, all performed at least basic customizations to the system such as subscribing to or unsubscribing from channels. Additionally, five of the six users created derivative channels by modifying the configurations of existing channels. For example, one participant modified the "Your Photos Here" Flickr channel to display photos from the Atlanta Photo Group. Most of these configuration changes involved editing basic channel properties, such as the tags used in a Flickr query or the URL of an RSS feed.

Two users, however, created more complex customizations. For example, one user created a new channel from scratch using the web crawler to traverse his personal website and generate a collage of his wedding photos. Another user created a channel to gather his family vacation photos from Google's Picasa photosharing site instead of Flickr. He further shared that channel with other users of the system.

Although the software provided some limited programming capabilities in the form of writing pattern extraction rules, none of our participants created any channels that performed such complex extraction. The system designer, however, did create several channels using this mechanism. Specifically, the Digg channel used a custom extraction pattern to follow the link to dugg web pages, and the CS1315 and CS1316 channels used a custom pattern to extract "What the snake says" and "What the bean says," respectively, for these Python and Java-based courses. Although no users created any channels or examined the extraction patterns of any channels that used this capability,

one participant did subscribe to the CS1316 channel and another to the Digg channel. Log data suggests, however, that neither of these participants actually used the configuration interface on either of these channels. As such, it is not clear whether the participants did not use this capability because the need never arose, or if the need arose but they did not know that such a capability existed or how to access it.

Additionally, although a plugin capability existed, this capability was neither documented nor made known to the users. As such, no participants wrote such plugins to extend the capabilities of the software with new harvesters, scrapers, or visualizers. One participant, however, did enquire about the ability to write a custom visualizer, but did not seriously follow up on this capability.

As such, although log data shows that all users performed some form of customization to their channel lineup, the bulk of these customizations consisted of relatively basic modifications of the sort typically available through other awareness systems, such as the Dashboards. Some users, however, did use channels created by the designer of the software that would have been much more difficult to create using other means. For example, the "Digg", "What the snake says," and "What the bean says" channels would have involved creating a new widget from scratch using a separate IDE under one of the dashboard systems, whereas they required only writing a single pattern specified within the interface using The Buzz. In this way, The Buzz deployment suggests that, while the software itself may enable a user to create a broad class of customizations, few users appear likely to utilize those features. The bulk of customizations will likely, even amongst very technical users, involve basic customization of the software. The question does remain, however, whether more participants might have used these capabilities if the designer of the system had not been available to create those channels himself.

7.2 Lounge Deployment

The system also ran on a large public display in a busy hallway for over a year (see Figure 7). People walking by could glance at the screen and glean whatever happens to be showing at the time. In its configuration, the software focused on community-relevant content. Content came from within the College of Computing community, coupled with "value-added" content that was likely to be useful to members of the community. The primary content used an index of all of the

web pages on the College of Computing web server to generate collages of images found on the web server. The system then frequently displayed a collage assembled from a particular web page selected at random. Through these collages, users could potentially keep track of what's going on within a relatively large and diverse community that is spread across three different buildings. The system aimed to help members of the community keep aware of their peers' activities through their web postings. In this way, users could see vacation and baby pictures when people updated their websites. Or they could see figures from research project websites, potentially encouraging collaboration among people who might not have known they shared a common research interest.

In addition to data found by crawling the community web server, which includes a combination of research and personal web pages, the software also specifically indexed the research web pages within GVU Center, the organization within which the software is deployed. In this way, the collages depicted a higher quantity of research-related content to help showcase the work conducted within the community. Additionally, another channel gathered names, pictures, and research summaries from the GVU faculty and staff directory, showing dossiers on the various members of the organization. Through these channels, the software aimed to help promote awareness about the other members within a growing organization.

Finally, the software also provided support for user-submitted photographs via a special tag on Flickr.com. Any photographs tagged `4thebuzz` on Flickr would get indexed by The Buzz and appear in a collage in the lounge. Even though the software did not support walk-up-and-use interactions in the public environment, users still could share their content with the community.

The Buzz ran in this shared space for over a year. During this time, anyone who passed through the busy hallway lounge could view the collage, photo gallery, news article, weather forecast, or any other content that was shown on the screen. During this deployment, we informally observed use of the system by casually noting the behaviors of people as they walked past the display. These observations were typically performed when the observer himself walked by the display, during routine maintenance, or when the observer was using the shared lounge area. In these informal observations, people would often glance up at the screen and pause briefly while they read a news article or weblog posting that caught their attention.

7.3 Lounge Deployment Observations

Although The Buzz did not support direct interaction with the system when running in the public lounge, users were able to control what content appeared on the display and did adjust their own web presence specifically for the purpose of changing how their web pages were displayed by The Buzz. One user was a hobbyist photographer, who took some amazing photographs and shared them through a photo gallery linked from his personal web page. When his photos never appeared on The Buzz, however, he discovered that the web crawler used to index the College of Computing web server only indexed images that were stored on the local web server or were embedded on a web page stored on the local server. His photo gallery software, however, stored all of the photos on a remote web server and only linked to the images on the local server. As a result, he modified his photo galleries to embed the offsite images in a local HTML file, so that his photos would appear on The Buzz.

Another user also modified his web site, but this time to remove content from The Buzz. This user was an undergraduate student at Georgia Tech, who posted pictures from his parties on his web page. Although all of the pictures he posted were well within the University's Acceptable Use Policy, they depicted some drunken party antics. This student posted these photos in a gallery on his web pages so that he could share them with his friends. However, after he walked through the corridor and saw a large photo of himself in a potentially embarrassing situation projected on the wall in the lounge, he decided to remove those photos from his web site. Before he had seen his photos projected on the wall, he had thought that his photos were somewhat more private, despite being shared on a public web server.

In this way, The Buzz actually helped the user to structure the way he presented himself to others. By walking by the display, he was able to see that his partitioning of his different roles [41] had broken down. Pictures that he had intended only to share with a smaller group of his friends were available to anyone. While the public aspect of content on the web is well known, many people think that no one else will actually be looking at their content. As a result, it was not until he saw his photos projected large on the wall of the office corridor that he realized just how public content on the web was and took appropriate steps to change how he presented himself on the internet, at

least with regard to his personal pictures.

Similarly, a popular channel in The Buzz was the College of Computing Birthdays channel. On someone's birthday within the community, a collage would appear periodically throughout the day to wish that person a happy birthday. On several occasions, people commented that they really enjoyed the birthday channel and would describe occasions on which they had caught someone off guard by wishing them a happy birthday. They described it as helping them in a small way to help maintain those small social interactions to help maintain relationships. By wishing someone a happy birthday, it helps to reinforce a small sense of connectedness. Sometimes, however, someone would comment that they were disturbed by the fact that their birthdays were available to anyone passing by. Although many people do enjoy being wished a happy birthday, other feel some degree of invasion to their privacy.

Finally, six other users tagged their photos on Flickr with the special `4thebuzz` tag. These participants posted photos they had taken around Atlanta, vacation photos, images of their children, and even user-interface glitches in various software. This feature, however, was not widely advertised, so it saw relatively little use. As a result, many participants may not have known about that capability.

7.4 Deployment Conclusions

These two deployments suggest that The Buzz is able to support a range of customization styles. The desktop deployment shows how some users can create their own derivative channels and even share them with their fellow users. Although no users actually performed complex customizations using the software, they did subscribe to such customizations that were created by the system designer.

The lounge deployment shows another way for users to customize the awareness experience. Because the software was running in an environment in which the users of the system did not actually have access to the software itself, some were able to configure the experience by adapting their other content publishing practices. That is, some users changed their own practices in order to control the content that appeared on the display.

CHAPTER VIII

CONCLUSION

With increasingly abundant access to information and to the internet, and with decreasingly expensive display technologies becoming more pervasive throughout the environment, come new information awareness systems that offer the potential to help people better manage their attention. These awareness systems can help to calmly convey information to the user in the periphery, helping to avoid distracting the user and avoiding interruptions. These approaches offer the potential to help the user glean information casually throughout the day via these unobtrusive interfaces, rather than through explicit information foraging activities.

In order for these systems to effectively help the user to manage information overload, however, they must convey the right information. Because different people have their own unique needs and interests, many of these systems support customization in order to tailor their content to the individual user. Supporting such customization, however, is difficult. If a customization interface provides too fine a granularity of customizability, the interface may be overwhelming for the user. If the interface operates at too coarse a level, the user will be incapable of expressing her needs to the system.

To address these concerns, most systems focus on providing customization at an intermediate level. Others still might provide two forms of customization: one interface for end users and another for more advanced users (usually developers). In Section 6.2, we examined some of these approaches that have been taken to support users at customizing their awareness systems and the various approaches that they have taken to conveying information.

Throughout this document, we have explored the notion that it is possible to provide increased power and flexibility in customization interfaces, beyond what is provided by current systems, without requiring significant programming effort. In order to demonstrate this ability, we examined The Buzz in Chapters 4 and 5. The first of these chapters presented the user experience of The Buzz—what it does and how the user would make use of it. We explored the various interfaces that support

users at being about browse available channels created by other users of the software. We saw how the user can modify those channels to create his own derivative, one-off channels, or how he could create his own channel from scratch, and share any of these new channels with other users of the system, contributing back into the channel ecosystem.

In Chapter 5, we explored the underlying architecture of The Buzz. It's modular design enables various smaller components that perform relatively simple and straightforward tasks to be combined into a larger components. Through this mechanism, the various harvesters and scrapers can combine to handle a large variety of different data publishers' formats. Furthermore, the underlying plugin architecture allows developers to create new modules that can be applied to a variety of contexts.

Chapter 6 then explored the awareness application customization space using six other customizable awareness applications. With these systems as a lens, we extracted various dimensions that help to define this customization space, and helped to show how The Buzz is situated within this space. In this way, we characterized the customization space for information awareness applications.

In order to get a sense of how real people will use The Buzz, we conducted two deployment studies. Chapter 7 described these two deployments and examined how users actually made use of the customization capabilities of the software. These studies show that, even though the software provided extensive customization support, even amongst advanced computer users, these capabilities were rarely used. Instead, observed use resembled that of systems like the Dashboards [11, 75, 4, 6] and Yahoo Pipes [7], where users typically fell into one of two roles: end-user or developer. The participants fell under the class of end-users, while the system designer served as a channel developer.

Despite users typically falling into two classes, end-users did still perform more customizations than would be possible for an end-user of the Dashboards. In these systems, users must transition to a developer interface and write code in order to modify their information content streams. In the desktop research probe, we observed users who created new derivative channels that presented data from different information sources. For example, the participant who created a new channel for his wedding photos created a new channel from scratch using a web crawler to extract photos from his personal web gallery. This capability does not exist in the Dashboards without writing code;

Yahoo Pipes may be able to accomplish such a task depending on various implementation details of the underlying web gallery software, but even if such a task is possible, it would involve writing a visual program to process the data. In this way, even though most users' standard state was that of end-user, some users did dive into the role of tinkerers.

Even though these tinkerers did not use all of the advanced customization capabilities of the software, they did demonstrate the ability and desire to modify and create channels in ways that would have been difficult with other awareness tools. As such, The Buzz appears to enable end-users, tinkerers, and developers to use, modify, create, and share customizations across a wide swath of the information awareness customization space.

Throughout this analysis, we have been motivated by several research questions:

RQ1 Is it possible for an information awareness application to give users increased power in their customizations without requiring significant programming effort?

RQ2 Is it possible for an information awareness application to give users increased flexibility in how they create their customizations?

RQ3 What dimensions characterize the customization space for awareness applications?

RQ4 What kinds of customizations can users create with a powerful, flexible customizable information awareness application?

The first of these questions, RQ1, focuses on the relative power of a customization system that does not require significant programming effort. The first component of this question relates to *increased power* relative to existing customizable awareness interfaces. In Chapter 6, we charted the existing customization space and various related software systems that inhabit it. Taking these systems as indicative of the state of existing awareness applications, we showed that The Buzz is able to give users more power in their customizations than these systems do without significant programming effort.

In Chapter 6, we established this baseline against which to compare. The examples in Section 4.9 along with the analysis in Chapter 6 helped to demonstrate that the The Buzz does indeed

enable the creation of new channels that would not be possible using existing, non-programming-based, systems. While some of the software systems within this space may provide specific interactions that are beyond the capabilities of The Buzz, on the whole The Buzz is able to support a wider range of powerful customizations across the data extraction, transformation, and presentation space.

The second of these questions, RQ2, uses this same baseline to ask whether it is possible for an information awareness application to give increased flexibility in the creation of customizations. In Section 4.9.3, we used the NSF News example to examine how The Buzz provides significant flexibility to the user in how she can create new channels that align closely to her goals depending on both the effort she is willing to expend to customize the system and the complexity of the underlying data. This example, combined with the analysis in Chapter 6 demonstrates how The Buzz does provide for increased flexibility in customization relative to existing awareness tools.

RQ3 asks what dimensions characterize the information awareness customization space. In Chapter 6, we explored the various systems within the awareness customization space to identify the various dimensions of this space. These dimensions are depicted in Figure 54.

The last research question, RQ4 asks what kinds of channels users can create with an interface such as The Buzz. In Section 4.9, we demonstrated three typical examples of channels that a user could create. In particular, the NSF News (Section 4.9.3) and Digg (Section 4.9.2) examples demonstrate channels that require increased power, while the NSF News example further explores the utility of the increased flexibility of The Buzz. Furthermore, the deployment study in Chapter 7 demonstrates that real users in an in-the-wild setting did, to some extent, create their own new channels.

In this way, the answer to RQ1 and RQ2 is yes, with The Buzz demonstrating such a system. RQ3 is answered through the comparative analysis performed in Chapter 6. Finally, RQ4 is demonstrated through the examples in Section 4.9 and through the channels described in Chapter 7. With these research questions affirmed, and with these characterizations performed, the answers to these questions affirm the thesis statement that:

> It is possible for an information awareness application to enable end-users, tinkerers, and developers to use, modify, create, and share powerful, flexible customizations over

the content and presentation that the system provides. Such an application can provide more extensive customization capabilities than existing systems without requiring significant programming effort.

There are two primary research contributions that derive from this work. The first is a set of abstractions and components to support flexible and general customizations within this domain. Together, the design of The Buzz demonstrates the capabilities of this set of abstractions. The second of these contributions is an enumeration and a taxonomy of the important characteristics of the awareness application customization domain. Through this enumeration and taxonomy, we provide a characterization of this customization space, as demonstrated in the Comparative Analysis chapter.

Although The Buzz demonstrates increased power and flexibility over existing customizable information awareness applications, it does still have some significant limitations. First, there are various data gathering techniques described in the related work (Chapter 3) that provide more user-friendly techniques to support various kinds of data. In particular, the techniques used by the Cards Framework [32] and by Marmite [108] to allow the user to visually define data extraction patterns would greatly enhance the range of customization supported by The Buzz. These techniques could significantly reduce the number of cases for which it is necessary to hand-write extraction patterns. Similarly, the surface representation technique used by the Notification Engine [43] could satisfy many use cases in which no semantic representation of the data is necessary.

Not only would these techniques improve the power and reduce the complexity of many kinds of customization with The Buzz, but also they could integrate well into The Buzz's existing architecture. These techniques satisfy the role of a data scraper. For example, if The Buzz provided a data scraper for a visually-defined pattern extractor, the Digg (Section 4.9.2) and NSF News (Section 4.9.3) examples could use those extractions patterns in lieu of the regular expression-based pattern extractor, reducing a significant barrier to such customizations.

Additionally, while The Buzz does provide support customizing the presentation of channels, these capabilities are relatively crude. There is room for significant work to identify new techniques to allow users to define more complex representations of their data. In the terminology from North and Shneiderman's Snap-Together Visualizations work, The Buzz's non-programming interfaces

operate on Level 2. There is room to explore Level 3 interactions, where the user can control the presentation with finer granularity.

In addition to improving the technical capabilities of The Buzz's customization components, the configuration interfaces through which users create these customizations still offer significant room for improvement. These interfaces could be improved through simple refinements or more radical changes. A minor improvement would involve augmenting the harvesters, scrapers, and filters to tag the kinds of data they accept as inputs and produce as outputs. Through such changes, the interfaces could update to show only components that are relevant given a particular kind of data input or output. More radical changes could involve replacing the existing wizard-style interface with alternative approaches. For example, a direct manipulation interface may simplify certain kinds of customizations.

When creating these customizations, the existing interface provides limited debugging capabilities. The state of the underlying system and of the data are often hidden from the user, making errors difficult to detect or diagnose. Systems such as Marmite [108] and Yahoo Pipes [7] provide much more extensive support for monitoring data as they are processed. The End User Programming community has also produced numerous techniques that could help to improve these operations.

Finally, more study is necessary to identify how real users are able to take advantage of this increased power and flexibility. The deployment studies documented in Chapter 7 were limited in scale and in scope. They involved a small number of users within a very technically-oriented institution. As a result, the observations from that study are likely to skew in a particular direction. Conducting a larger study with a broader user population would likely provide some more insightful indications of how the broader user ecology might customize their awareness software.

Similarly, do the patterns of sharing and customization in the awareness domain resemble those from other domains? The preliminary observations from the deployment study suggest that they do, but they provided too limited a scope to create a larger sharing community. Do the same kinds of patterns of gardeners and gurus [40] emerge?

Finally, these awareness systems offer the potential to help people to manage information overload, but do they actually work? Do they improve awareness? Do they provide added distraction? In informal observations during The Buzz deployment, several users observed that their information

access habits had changed. For example, one participant who read a large number of weblogs, would frequently take explicit breaks throughout the day to launch his RSS reader. By several weeks into the deployment, however, he had stopped using his RSS reader and only viewed the weblog entries that happened to come up on The Buzz. Not surprisingly, he indicated that he was definitely reading far fewer weblog articles. Surprisingly, however, despite missing far more articles, he felt more informed, as if he were better keeping aware of those articles that he did see. Perhaps exposure to fewer articles enables him to better retain them. Or perhaps he only has a false perception of being more aware.

Regardless of the specific answers to these questions, it is clear that there is significant room for more study in the domain of customization and of information awareness applications. The Buzz provides an approach to provide more extensive customization by a broad variety of users in many different contexts. More work, however, is needed if we are ever to realize the dream of universal customization.

REFERENCES

[1] "Dapper." Retrieved June 13, 2007 from http://dappit.com.

[2] "Digg API." Retrieved August 11, 2007 from http://apidoc.digg.com/.

[3] "Flickr API." Retrieved August 11, 2007 from http://flickr.com/services/api/.

[4] "Google gadgets." Retrieved August 6, 2007 from http://www.google.com/apis/gadgets/.

[5] "Technorati." Retrieved November 26, 2006 from http://technorati.com/about.

[6] "Yahoo! Konfabulator." Retrieved June 13, 2007 from http://widgets.yahoo.com.

[7] "Yahoo! Pipes." Retrieved June 13, 2007 from http://pipes.yahoo.com/.

[8] "Yahoo! Search Web Services." Retrieved August 11, 2007 from http://developer.yahoo.com/search/.

[9] "RDF/XML syntax specification (revised)," W3C Recommendation, World Wide Web Consortium (W3C), February 2004. Retrieved August 11, 2007 from http://www.w3.org/TR/rdf-syntax-grammar/.

[10] "Apple, Inc. Automator," 2005. Retrieved June 13, 2007 from http://www.apple.com/macosx/features/automator/.

[11] "Apple, Inc. Dashboard," 2005. Retrieved June 13, 2007 from http://www.apple.com/macosx/features/dashboard/.

[12] AHLBERG, C. and SHNEIDERMAN, B., "Visual information seeking: tight coupling of dynamic query filters with starfield displays," in *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 313–317, ACM, 1994.

[13] AHLBERG, C., WILLIAMSON, C., and SHNEIDERMAN, B., "Dynamic queries for information exploration: an implementation and evaluation," in *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 619–626, ACM, 1992.

[14] BAILEY, B. P., KONSTAN, J. A., and CARLIS, J. V., "Measuring the effects of interruptions on task performance in the user interface," in *IEEE Conference on Systems, Man, and Cybernetics 2000*, pp. 757–762, IEEE, IEEE Computer Society Press, 2000.

[15] BAILEY, B. P., KONSTAN, J. A., and CARLIS, J. V., "The effects of interruptions on task performance, annoyance, and anxiety in the user interface," in *Proceedings of INTERACT 2001*, 2001.

[16] BOLIN, M., WEBBER, M., RHA, P., WILSON, T., and MILLER, R. C., "Automation and customization of rendered web pages," in *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 163–172, ACM Press, 2005.

[17] BURNETT, M., COOK, C., PENDSE, O., ROTHERMEL, G., SUMMET, J., and WALLACE, C., "End-user software engineering with assertions in the spreadsheet paradigm," in *ICSE '03: Proceedings of the International Conference on Software Engineering*, pp. 93–103, May 2003.

[18] CADIZ, J. J., VENOLIA, G., JANCKE, G., and GUPTA, A., "Designing and deploying an information awareness interface," in *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pp. 314–323, ACM Press, 2002.

[19] CARD, S. K., MACKINLAY, J. D., and SHNEIDERMAN, B., *Readings in information visualization: using vision to think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[20] CHANG, A., RESNER, B., KOERNER, B., WANG, X., and ISHII, H., "Lumitouch: an emotional communication device," in *CHI '01 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 313–314, ACM Press, 2001.

[21] CHERVANY, N. L. and DICKSON, G. W., "An experimental evaluation of information overload in a production environment," *Management Science*, vol. 20, pp. 1335–1344, June 1974.

[22] CHRISTENSEN, E., CURBERA, F., MEREDITH, G., and WEERAWARANA, S., "Web Service Definition Language (WSDL)," W3C Note, World Wide Web Consortium (W3C), March 2001. Retrieved August 8, 2007 from `http://www.w3.org/TR/wsdl`.

[23] CHURCHILL, E. F., NELSON, L., DENOUE, L., HELFMAN, J., and MURPHY, P., "Sharing multimedia content with interactive public displays: a case study," in *Proceedings of the 2004 conference on Designing interactive systems*, pp. 7–16, ACM Press, 2004.

[24] CHURCHILL, E. F., NELSON, L., and HSIEH, G., "Café life in the digital age: augmenting information flow in a café-work-entertainment space," in *CHI '06 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 123–128, ACM Press, 2006.

[25] CLARK, J. and DEROSE, S., "XPath (XML Path Language)," W3C Recommendation, World Wide Web Consortium (W3C), November 1999. Retrieved June 13, 2007 from `http://www.w3.org/TR/xpath`.

[26] COX, K., HIBINO, S., HONG, L., MOCKUS, A., and WILLS, G., "Infostill: A task-oriented framework for analyzing data through information visualization," in *Proceedings of the 1999 IEEE Symposium on Information Visualization Late Breaking Hot Topics*, pp. 19–22, October 1999.

[27] CRESCENZI, V., MECCA, G., and MERIALDO, P., "Roadrunner: automatic data extraction from data-intensive web sites," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 624–624, ACM Press, 2002.

[28] CUTRELL, E., CZERWINSKI, M., and HORVITZ, E., "Notification, disruption, and memory: Effects of messaging interruptions on memory and performance," in *Proceedings of INTERACT 2001*, 2001.

[29] CYPHER, A., HALBERT, D. C., KURLANDER, D., LIEBERMAN, H., MAULSBY, D., MYERS, B. A., and TURRANSKY, A., eds., *Watch what I do: programming by demonstration*. MIT Press, 1993.

[30] DAVENPORT, T. H. and BECK, J. C., *The Attention Economy: Understanding the New Currency of Business*. Cambridge, MA: Harvard Business School Press, June 2001.

[31] DEVICES, A., "Ambient Orb." Retrieved August 11, 2007 from `http://www.ambientdevices.com`.

[32] DONTCHEVA, M., DRUCKER, S. M., SALESIN, D., and COHEN, M. F., "Relations, cards, and search templates: user-guided web data integration and layout," in *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 61–70, ACM, 2007.

[33] DOURISH, P., LAMPING, J., and RODDEN, T., "Building bridges: customisation and mutual intelligibility in shared category management," in *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pp. 11–20, ACM Press, 1999.

[34] EAGAN, J. R. and STASKO, J. T., "The Buzz: Supporting user tailorability in awareness applications," in *CHI '08: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 1729–1738, ACM Press, 2008.

[35] FAABORG, A. and LIEBERMAN, H., "A goal-oriented web browser," in *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, (New York, NY, USA), pp. 751–760, ACM Press, 2006.

[36] FASS, A., FORLIZZI, J., and PAUSCH, R., "MessyDesk and MessyBoard: two designs inspired by the goal of improving human memory," in *DIS '02: Proceedings of the conference on Designing interactive systems*, (New York, NY, USA), pp. 303–311, ACM Press, 2002.

[37] FLORES, F. C., QUINT, V., and VATTON, I., "Templates, microformats and structured editing," in *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, (New York, NY, USA), pp. 188–197, ACM Press, 2006.

[38] FOGARTY, J., FORLIZZI, J., and HUDSON, S. E., "Aesthetic information collages: generating decorative displays that contain information," in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pp. 141–150, ACM Press, 2001.

[39] FUJIMA, J., LUNZER, A., HORNBÆK, K., and TANAKA, Y., "Clip, connect, clone: combining application elements to build custom interfaces for information access," in *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 175–184, ACM Press, 2004.

[40] GANTT, M. and NARDI, B. A., "Gardeners and gurus: patterns of cooperation among cad users," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 107–117, ACM Press, 1992.

[41] GOFFMAN, E., *The Presentation of Self in Everyday Life*. Anchor, 1959.

[42] GOMES, L., "Will all of us get our 15 minutes on a YouTube video?," *The Wall Street Journal*, vol. 248, p. B1, August 2006.

[43] GREENBERG, S. and BOYLE, M., "Generating custom notification histories by tracking visual differences between web page visits," in *GI '06: Proceedings of the 2006 conference on Graphics interface*, (Toronto, Ont., Canada, Canada), pp. 227–234, Canadian Information Processing Society, 2006.

[44] GREENBERG, S. and FITCHETT, C., "Phidgets: easy development of physical interfaces through physical widgets," in *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 209–218, ACM Press, 2001.

[45] GREENBERG, S. and ROUNDING, M., "The Notification Collage: posting information to public and personal displays," in *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 514–521, ACM Press, April 2001.

[46] GUDGIN, M., HADLEY, M., MENDELSOHN, N., MOREAU, J.-J., NIELSEN, H. F., KARMARKAR, A., and LAFON, Y., "SOAP version 1.2 part 1: Messaging framework (second edition)," W3C Recommendation, World Wide Web Consortium (W3C), April 2007. Retrieved August 8, 2007 from `http://www.w3.org/TR/soap12-part1/`.

[47] HAN, W., BUTTLER, D., and PU, C., "Wrapping web data into xml," *SIGMOD Record*, vol. 30, no. 3, pp. 33–38, 2001.

[48] HEINER, J. M., HUDSON, S. E., and TANAKA, K., "The information percolator: ambient information display in a decorative object," in *Proceedings of the 12th annual ACM symposium on User interface software and technology*, pp. 141–148, ACM Press, 1999.

[49] HUANG, E. M. and MYNATT, E. D., "Semi-public displays for small, co-located groups," in *Proceedings of the conference on Human factors in computing systems*, pp. 49–56, ACM Press, 2003.

[50] HUDSON, J. M., CHRISTENSEN, J., KELLOGG, W. A., and ERICKSON, T., ""I'd be overwhelmed, but it's just one more thing to do": availability and interruption in research management," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 97–104, ACM Press, 2002.

[51] HUDSON, S. E. and STASKO, J. T., "Animation support in a user interface toolkit: flexible, robust, and reusable abstractions," in *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 57–67, ACM, 1993.

[52] HUYNH, D. F., MILLER, R. C., and KARGER, D. R., "Enabling web browsers to augment web sites' filtering and sorting functionalities," in *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 125–134, ACM Press, 2006.

[53] IETF - THE INTERNET SOCIETY, "RFC 4287: The Atom syndication format," December 2005. Retrieved August 11, 2007 from `http://www.ietf.org/rfc/rfc4287.txt`.

[54] INTILLE, S. S., "Change blind information display for ubiquitous computing environments," in *Proceedings of the 4th International Conference on Ubiquitous Computing* (BORRIELLO, G. and HOLMQUIST, L. E., eds.), pp. 91–106, Springer-Verlag: Berlin, September 2002.

[55] ISHII, H. and ULLMER, B., "Tangible bits: towards seamless interfaces between people, bits and atoms," in *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 234–241, ACM Press, 1997.

[56] KELLAR, M., WATTERS, C., and INKPEN, K. M., "An exploration of web-based monitoring: implications for design," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 377–386, ACM Press, 2007.

[57] KELLEHER, C. and PAUSCH, R., "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, 2005.

[58] KHARE, R., "Microformats: the next (small) thing on the semantic web?," *IEEE Internet Computing*, vol. 10, no. 1, pp. 68–75, 2006.

[59] KO, A. J. and MYERS, B. A., "Debugging reinvented: asking and answering why and why not questions about program behavior," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*, (New York, NY, USA), pp. 301–310, ACM, 2008.

[60] LAENDER, A. H. F., RIBEIRO-NETO, B. A., DA SILVA, A. S., and TEIXEIRA, J. S., "A brief survey of web data extraction tools," *SIGMOD Rec.*, vol. 31, no. 2, pp. 84–93, 2002.

[61] LASSETER, J., "Principles of traditional animation applied to 3d computer animation," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 35–44, ACM, 1987.

[62] LERNER, R., "At the forge: Creating mashups," *Linux Journal*, vol. 2006, no. 147, p. 10, 2006.

[63] LESHED, G., HABER, E. M., MATTHEWS, T., and LAU, T., "Coscripter: automating & sharing how-to knowledge in the enterprise," in *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 1719–1728, ACM, 2008.

[64] LITTLE, G., LAU, T. A., CYPHER, A., LIN, J., HABER, E. M., and KANDOGAN, E., "Koala: capture, share, automate, personalize business processes on the web," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 943–946, ACM, 2007.

[65] LYMAN, P. and VARIAN, H. R., "How much information." Retrieved November 8, 2006 from `http://www.sims.berkeley.edu/how-much-info-2003`.

[66] MACKAY, W. E., "Patterns of sharing customizable software," in *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, (New York, NY, USA), pp. 209–221, ACM Press, 1990.

[67] MACKAY, W. E., "Triggers and barriers to customizing software," in *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 153–160, ACM Press, 1991.

[68] MACLEAN, A., CARTER, K., LÖVSTRAND, L., and MORAN, T., "User-tailorable systems: pressing the issues with buttons," in *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 175–182, ACM Press, 1990.

[69] MAES, P., "Agents that reduce work and information overload," *Communications of the ACM*, vol. 37, no. 7, pp. 30–40, 1994.

[70] MARLOW, C., NAAMAN, M., BOYD, D., and DAVIS, M., "Ht06, tagging paper, taxonomy, flickr, academic article, to read," in *HYPERTEXT '06: Proceedings of the seventeenth conference on Hypertext and hypermedia*, (New York, NY, USA), pp. 31–40, ACM Press, 2006.

[71] MATTHEWS, T., "Peripheral Display Toolkit: A toolkit for managing user attention in peripheral displays," Master's thesis, Computer Science Division, University of California, Berkeley, 2004.

[72] MATTHEWS, T., DEY, A. K., MANKOFF, J., CARTER, S., and RATTENBURY, T., "A toolkit for managing user attention in peripheral displays," in *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 247–256, ACM Press, 2004.

[73] MCCARTHY, J. F., COSTA, T. J., and LIONGOSARI, E. S., "UniCast, OutCast & GroupCast: Three steps toward ubiquitous, peripheral displays," in *Proceedings of the 3rd international conference on Ubiquitous Computing*, pp. 332–345, Springer-Verlag, 2001.

[74] MCCRICKARD, D. S., "Maintaining information awareness with Irwin," in *Proceedings of the World Conference on Educational Multimedia/Hypermedia and Educational Telecommunications (ED-MEDIA '99)*, (Seattle, WA), June 1999.

[75] MICROSOFT, "Vista Sidebar," 2007. Retrieved August 30, 2007 from `http://www.microsoft.com/windows/products/windowsvista/features/details/sidebargadgets.mspx`.

[76] MILLER, R. C. and BHARAT, K., "Sphinx: a framework for creating personal, site-specific web crawlers," in *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, (Amsterdam, The Netherlands, The Netherlands), pp. 119–130, Elsevier Science Publishers B. V., 1998.

[77] MILLER, R. C. and MYERS, B. A., "Outlier finding: focusing user attention on possible errors," in *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 81–90, ACM Press, 2001.

[78] MILLER, R. C. and MYERS, B. A., "Lapis: smart editing with text structure," in *CHI '02 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 496–497, ACM Press, 2002.

[79] MILLER, T. and STASKO, J., "The infocanvas: information conveyance through personalized, expressive art," in *CHI '01 extended abstracts on Human factors in computing systems*, pp. 305–306, ACM Press, 2001.

[80] MYERS, B. A., SMITH, D. C., and HORN, B., "Report of the "End-User Programming" working group," *Languages for Developing User Interfaces*, pp. 343–366, 1992.

[81] NARDI, B. A., *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.

[82] NARDI, B. A., MILLER, J. R., and WRIGHT, D. J., "Collaborative, programmable intelligent agents," *Communications of the ACM*, vol. 41, no. 3, pp. 96–104, 1998.

[83] NORTH, C. and SHNEIDERMAN, B., "Snap-together visualization: can users construct and operate coordinated visualizations?," *International Journal of Human-Computer Studies*, vol. 53, no. 5, pp. 715–739, 2000.

[84] O'REILLY, T., "What is Web 2.0," September 2005. Retrieved August 11, 2007 from `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

[85] PLAUE, C. and STASKO, J., "Animation in a peripheral display: Distraction, appeal, and information conveyance in varying display configurations," in *GI '07: Proceedings of the 2007 conference on Graphics interface*, Canadian Human-Computer Communications Society, 2007.

[86] POLSON, P. G., LEWIS, C., RIEMAN, J., and WHARTON, C., "Cognitive walkthroughs: a method for theory-based evaluation of user interfaces," *International Journal of Man-Machine Studies*, vol. 36, no. 5, pp. 741–773, 1992.

[87] POUSMAN, Z. and STASKO, J., "A taxonomy of ambient information systems: four patterns of design," in *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, (New York, NY, USA), pp. 67–74, ACM Press, 2006.

[88] RADEMACHER, P., "Housing maps," 2005. Retrieved March 27, 2007 from `http://www.housingmaps.com/`.

[89] RAMAKRISHNAN, S. and DAYAL, V., "The pointcast network (abstract)," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, p. 520, ACM Press, 1998.

[90] REAS, C. and FRY, B., *Processing: A Programming Handbook for Visual Designers and Artists*. Cambridge, MA: MIT Press, 2007.

[91] REDSTRÖM, J., SKOG, T., and HALLNÄS, L., "Informative art: using amplified artworks as information displays," in *Proceedings of DARE 2000 on Designing augmented reality environments*, pp. 103–114, ACM Press, 2000.

[92] ROBERTSON, G. G., D. AUSTIN HENDERSON, J., and CARD, S. K., "Buttons as first class objects on an X desktop," in *Proceedings of the 4th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 35–44, ACM Press, 1991.

[93] SHNEIDERMAN, B., "Direct manipulation: A step beyond programming languages," *IEEE Computer*, vol. 16, pp. 57–69, August 1983.

[94] SHNEIDERMAN, B. and MAES, P., "Direct manipulation vs. interface agents," *interactions*, vol. 4, no. 6, pp. 42–61, 1997.

[95] SPENCE, R., *Information Visualization: Design for Interaction (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007.

[96] STASKO, J., MILLER, T., POUSMAN, Z., PLAUE, C., and ULLAH, O., "Personalized peripheral information awareness through Information Art," in *Proceedings of UbiComp '04*, (Nottingham, U.K.), pp. 18–35, September 2004.

[97] SUGIURA, A. and KOSEKI, Y., "Internet scrapbook: automating web browsing tasks by demonstration," in *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, (New York, NY, USA), pp. 9–18, ACM, 1998.

[98] TUFTE, E. R., *Envisioning Information*. Graphics Press, 1990.

[99] U. S. NATIONAL WEATHER SERVICE, "National digital forecast database." Retrieved August 6, 2007 from `http://www.weather.gov/ndfd/`.

[100] U. S. NATIONAL WEATHER SERVICE, "National Digital Forecast Database (NDFD) WSDL interface," August 2004. Retrieved August 11, 2007 from `http://www.weather.gov/forecasts/xml/DWMLgen/wsdl/ndfdXML.wsdl`.

[101] VAN ROSSUM, G., *Python Tutorial*, ch. Appendix D. Python Software Foundation, September 2006. Retrieved January 23, 2007 from `http://docs.python.org/tut/tut.html`.

[102] VANDER WAL, T., "Folksonomy definition and Wikipedia," November 2005. Retrieved March 27, 2007 from `http://www.vanderwal.net/random/entrysel.php?blog=1750`.

[103] VON AHN, L., BLUM, M., HOPPER, N. J., and LANGFORD, J., *CAPTCHA: Using Hard AI Problems for Security*, vol. 2656/2003 of *Lecture Notes in Computer Science*. Springer-Verlag: Berlin, 2003.

[104] VON AHN, L., BLUM, M., and LANGFORD, J., "Telling humans and computers apart automatically," *Communications of the ACM*, vol. 47, no. 2, pp. 56–60, 2004.

[105] WEISER, M. and BROWN, J. S., "The coming age of calm technology."

[106] WINER, D., "RSS 2.0 specification," tech. rep., Berkman Center for Internet & Society at Harvard Law School, October 2002. Retrieved August 11, 2007 from `http://cyber.law.harvard.edu/rss/rss.html`.

[107] WONG, J. and HONG, J., "Marmite: end-user programming for the web," in *CHI '06 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 1541–1546, ACM Press, 2006.

[108] WONG, J. and HONG, J. I., "Making mashups with Marmite: towards end-user programming for the web," in *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, (New York, NY, USA), pp. 1435–1444, ACM Press, 2007.

[109] WORLD WIDE WEB CONSORTIUM (W3C), "Semantic web." Retrieved August 8, 2007 from `http://www.w3.org/2001/sw/`.

[110] ZHAO, Q. A., *Opportunistic Interfaces for Promoting Community Awareness*. PhD thesis, College of Computing, Georgia Institute of Technology, August 2001.

[111] ZHAO, Q. A. and STASKO, J., "What's Happening?: Promoting community awareness through opportunistic, peripheral interfaces," in *Proceedings of the Advanced Visual Interfaces Conference*, (Trento, Italy), pp. 69–74, May 2002.

[112] ZUR MUEHLEN, M., NICKERSON, J. V., and SWENSON, K. D., "Developing web services choreography standards: the case of REST vs. SOAP," *Decision Support Systems*, vol. 40, no. 1, pp. 9–29, 2005.