SEARCH INTERFACES FOR INTEGRATING CROWDSOURCED CODE SNIPPETS WITHIN DEVELOPMENT ENVIRONMENTS

by

Douglas Wightman

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

January 2013

Abstract

In this thesis we report on the design and evaluation of interfaces to support crowdsourced programming tasks. We present WordMatch and SnipMatch: two programming tools that can incorporate crowdsourced source code. The design of these tools is informed by an investigation of crowdsourcing; specifically, Crowdsourced Human-Based Computation (CHC) systems, which organize tasks performed by humans. Recommendations include methods to obtain and maintain users who are highly motivated to participate and methods to improve task performance. WordMatch, a novel programming environment for specifying direct answers for search queries, builds on this work by introducing a parameterized search interface that can be easily understood by end users. In a laboratory study, we found that people with basic computer literacy could be taught to create complex direct answers with minimal training. Finally, evaluations of SnipMatch, a search interface for curated source code snippets, demonstrate that features from WordMatch are applicable to general programming tasks. Participants in our longitudinal study reported that SnipMatch was an effective tool for reducing context switching and as a memory aid.

Acknowledgements

Roel Vertegaal, thank you for your support. You allowed me to pursue my own path. It’s been a pleasure to work with you. Joel Brandt, Zi Ye, Scott Bateman, and Carl Gutwin, your feedback and encouragement also had an enormous impact on this work. Thank you, this would have been a lonely pursuit without you. James Cordy and Pat Martin, I am grateful for your guidance and many perceptive questions and comments. Melissa, Mom, Dad, Anne-Marie, and Spencer, thanks for listening.

Statement of Originality

I, Douglas Wightman, certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with standard referencing practices. Earlier versions of some parts of the work reported in this thesis have previously appeared as [132, 133, 134].

[illegible]

Douglas Wightman

January 2013

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis describes Crowdsourced Human-Based Computation (CHC) systems for programming tasks. These systems are programming tools that outsource the provision of aspects of their functionality to humans. For example, two of the primary contributions of this work are search interfaces that display results curated by many programmers.

Two high-level hypotheses led me to conduct this research: (1) context-sensitive search interfaces, built into the development environment, could make programming substantially easier, and (2) crowdsourcing could be more efficient if the tasks involved programming. My interest is at the intersection of these opportunities: effectively, crowdsourced programming.

Consider ReCAPTCHA [130]: this tool engages online users to digitize old editions of the New York Times, one word at a time. What if, instead of typing in the letters shown in the image, users made minor modifications to pattern recognition algorithms? This approach could improve the algorithm for future tasks and slowly reduce dependence on humans. While a robust design for such a tool does not yet exist, progress has been made. There are many simpler applications of crowdsourced programming, and tools for two of them are studied in detail in Chapters 3 and 4.

In this work, we focus on search interfaces with crowdsourced components. To demonstrate how search interfaces with curated code snippets can reduce programming time, we designed and evaluated two tools. First, a programming environment named WordMatch was designed to support end user programmers completing tasks in a relatively narrow domain. Findings from building and evaluating WordMatch were then applied to create SnipMatch, a tool for more general programming tasks. We found that both tools significantly reduced programming time, for certain tasks. Participants found SnipMatch particularly compelling; professional programmers now use it regularly. Motivations and contributions from this work are described in detail in the rest of this chapter.

1.1 **Motivation**

The primary motivation for this research was to reduce the time required to locate and use code snippets. Programmers often use general Internet search engines, such as Google, to locate code snippets [14, 16]. There are several costs associated with this approach that this research seeks to mitigate: (1) when the search engine is not integrated into the IDE, considerable context is lost (for example, the programming language and imported libraries), (2) changing from the IDE to the browser is a task switch – this interruption takes time and affects mental load [59], and (3) code snippets found on the Internet often require modification before they can be inserted into one's existing code [16]. Prototype systems for two use cases were built and studied: (1) a structured editor for creating direct answers, and (2) an Eclipse plugin for sharing code snippets. Direct answers – specific, factual answers that sometimes appear at the top of Internet search results, are a limited domain to begin this inquiry. We then expanded the focus to investigate code snippets for general applications.

1.1.1 **Direct Answers**

Several Internet search engines include direct answers above the search result listings for some search queries. These direct answers can be sufficient to answer a user's query. Google includes direct answers for unit conversions ("24 inches to cm"), census information ("Canada population”), and current weather information (“weather Palo Alto”), among others [44]. Google purchased Metaweb (metaweb.com), the company that created and maintains Freebase, to improve its direct answers [43].

Bing (bing.com) has sourced data for direct answers from both Metaweb and Wolfram Alpha. On Wolfram Alpha, a search for “distance to the moon” produces the current and average distances to the moon. Similar to Freebase, Wolfram Alpha sources some of its data from volunteers. While data is sometimes located or verified by volunteer curators, Google, Bing, and Wolfram Alpha rely on employees to create direct answers. None of these systems allow end users to add direct answers. This limitation restricts potential usage scenarios and the rate at which direct answers can be produced. End users might also decrease the cost of adding and maintaining direct answers.

1.1.2 **Integrating Code Snippet Search into the IDE**

Programmers routinely search online for snippets to integrate into their source code [14, 16, 79]. This find-and-integrate behavior provides quick access to Application Programming Interfaces (APIs), libraries, and algorithms [14, 145]. Even when an API is well understood, snippets provide productivity gains: the time and effort required is often less than writing the code from scratch [16].

Despite the benefits of snippet use, substantial time can still be required to locate and integrate code snippets. Programmers often must locate and combine multiple code snippets, rename or typecast variables, and manually locate and include dependencies [14, 35, 79]. After this complex integration has been performed, a code snippet may be discarded if it contains errors or otherwise does not work as expected [15]. Even a small decrease in the time required to perform this common task – finding and integrating snippets – could cause a qualitative change in behavior [45].

1.2 **Problem Statement**

Thesis Statement: Development environments can be enhanced with search tools for integrating crowdsourced code snippets.

Structured editors are software tools that are customized based on editable content. For example, Microsoft Word, which can correct common word misspellings, is a structured editor. IntelliSense [57], which can be used to quickly find relevant methods and variables, is another example. In software development environments, structured editors can prevent syntax errors and guide users through the process of selecting operations [20, 123].

Structured editors can also locate and integrate code snippets [14]. Two structured editors with snippet search and sharing features were designed and evaluated: one for direct answers and a second for general programming tasks. This progression allowed the work to begin with a simple, restricted use case before advancing to the broader task that was more difficult to control and required a longitudinal study in addition to lab studies. The goal of this research is to use crowdsourcing to reduce the time required to perform programming tasks. Specifically, we sought to answer the following questions:

- How should a search engine for crowdsourced snippets, integrated into an IDE, filter and rank snippets?

- How will programmers use and share crowdsourced code snippets?
- Will programmers be able to complete programming tasks in less time using a crowdsourced snippet search system integrated into the IDE than by searching on the Internet with a web browser?

The direct answers structured editor is designed for end users and accessible in standard web browsers. It is intended to be used by web site owners or maintainers, to create, update, and share direct answers embedded within their web sites, and to collaborate on direct answers for shared sites (for example, to provide an approach that an Internet search engine could adopt). This work includes evaluations that study the efficacy of direct answers in the wild and produced by this system.

The second structured editor is a plugin developed for the Eclipse IDE. This plugin allows users to search for and integrate crowdsourced code snippets using a sloppy syntax [91]. The plugin uses code context to filter and rank code snippets. Further, a markup language allows code snippet creators to specify additional rules to govern snippet integration.

1.3 **Contributions**

This research presents a novel approach to sharing code snippets. Specifically: methods to implement snippet search, sharing, and integration, and methods to engage users to share code snippets. This is an early attempt to utilize crowdsourcing principles to enhance programming tools. We demonstrate that it is a viable approach for certain tasks and provide a foundation for further study, in particular:

Design recommendations: Before building and evaluating new tools, we first studied prior work in two relevant domains: Crowdsourced Human Computation (CHC) systems, which organize tasks performed by humans, and end user programming. An understanding of the interaction between these two domains is critical to the design of programming tools with crowdsourced components. Design recommendations were derived from an analysis of CHC systems, grouped into four distinct classes, and existing end user programming paradigms. Even the aspects of this work that are primarily intended for experienced programmers were informed by the study of end users programmers, as an understanding of how to create tools for this audience provided a baseline for comparison. Recommendations include methods to obtain and maintain users who are highly motivated to participate and methods to improve task performance.

Environment for programming Direct Answers (WordMatch): We created a novel end user programming environment for specifying direct answers to simple, factual questions written as search queries. We found that people with basic computer literacy could be taught to create complex direct answers. We also identified simple lexical pattern matching as an effective mechanism for generating natural language direct answers. Third, we demonstrated that our participants' direct answers can significantly reduce search time compared to standard interfaces.

Interface for finding and integrating curated code snippets within software development environments (SnipMatch): This interface features a novel search algorithm that ranks, filters, and customizes snippets based on the code context in the development environment. It also includes a lightweight markup for specifying integration instructions for code snippets.

SnipMatch Evaluation: We developed insights about how code snippet search tools are used by performing a lab study, a longitudinal study, and interviews. We observed that SnipMatch was regularly used by professional programmers – who were not incented to participate – to reduce context switching and as a memory aid. Participants reported that including snippet arguments in the search box was particularly effective for the two most common usage scenarios: shortcuts and quick reference.

1.4 **Organization of the Thesis**

This thesis begins with a study of crowdsourcing systems and opportunities to engage participants in programming tasks, in Chapter 2. The results from the study are then applied to inform the design and evaluation of two development tools for integrating crowdsourced code snippets: (1) WordMatch (Chapter 3), a custom structured editor to enhance the direct answers that sometimes appear at the top of Internet search results, and (2) SnipMatch (Chapter 4), a plugin for the Eclipse IDE. Chapters 3 and 4 each include multiple subchapters with evaluations specific to these systems. Finally, Chapter 5 reviews our contributions and outlines topics for future research.

Chapter 2

Crowdsourcing Programming

2.1 Crowdsourcing Human-Based Computation

Human-based computation is the technique of outsourcing steps within a computational process to humans [67]. Alex Kosorukoff, who coined the term, designed a genetic algorithm that allows humans to suggest solutions that might improve evolutionary processes [67]. His description of human-based computation includes a division of labor between humans and computers. Labor is divided into two roles: selection and innovation. Selection refers to the selection of tasks and innovation to the performance of a task. A human or a computer can act as a selection agent or an innovation agent. For example, electronic systems that administer GRE tests could be considered selection agents and the human users innovation agents. Human-based computation can also involve multiple humans.

Crowdsourcing is the practice of outsourcing tasks to a group of humans [138]. In some cases, computers may be used to administer crowdsourced tasks, creating human-based computation systems. Many online systems (websites and other online applications) feature this form of human-based computation. In this chapter, examples of crowdsourced human-based computation (CHC) are grouped into four distinct classes using two factors: the users' motivation for completing the task (direct or indirect) and whether task completion is competitive. These factors were selected to highlight differences in usage behavior and implications for recruitment and retention. Image labeling games, news aggregation websites, and Wikipedia [135] are non-competitive CHC examples in which the users' primary motivation for participation is to perform the task itself (direct motivation). reCAPTCHA [107], online surveys, and websites for volunteers can be non-competitive CHC systems in which the users' primary motivation for participation is not the task itself (indirect motivation). Mechanical Turk [86] and InnoCentive [56] are competitive examples with users who are indirectly motivated to participate. Google search ranking and Yahoo! Answers [144] are competitive CHC examples with users who are directly motivated to participate.

Following an analysis of the different types of CHC systems, programming paradigms relevant to the design of interfaces for CHC participants are described and compared. Many CHC systems include custom user-interfaces that participants must use to perform tasks. Since most CHC participants have limited programming experience, the focus of this secondary inquisition is end user programming. The chapter concludes with suggestions for applying structured editor and programming by demonstration concepts to the design of CHC systems.

2.2 **Crowdsourced Human-Based Computation**

2.2.1 **Non-Competitive Direct Motivation Tasks**

2.2.1.1 Image Labeling Games

Most of the images that are publicly available on the Internet are not associated with keywords describing their depictions. If image keywords were available, they might be used to improve algorithms for image search and the filtering of inappropriate content. The ESP game [129] is an online game designed to label images with keywords. Humans, who play the game for enjoyment, enter the image keywords.

The ESP Game groups players into pairs, shows them both the same image, and awards points when the players type the same word on their keyboards. Every 2.5 minutes, points are awarded and a new image is displayed. The players are not able to communicate directly with one another, hence the game of the name. The game may indicate that certain words are taboo, meaning that points cannot be earned by entering these words.

Once a certain number of pairs of players have entered the same word for an image, the ESP game will notify future players that encounter the image that this word is not to be used to describe it. This feature forces players to enter different keywords, providing different labels for images. The accuracy of the labels generated by participants playing the ESP game was evaluated. The results indicated that 85% of the keywords associated with images “would be useful in describing” them [129].

The ESP game uses a number of techniques to encourage players to label images accurately. Players are allowed to skip images. This feature is also used to determine when an image is finished being labeled. The rationale is that players will skip an image if all of the keywords are taboo. Players’ IP addresses are required to be different. The authors suggest that cheating might be further curtailed by requiring players to use different words to describe each image during a particular session playing the game. This might further reduce opportunities for player collusion. In their 2004 paper, the creators of the ESP Game estimated that 5,000 individuals playing continuously could label each of the 425 million images indexed by Google in 31 days [129]. The authors note that popular online games websites, such as Yahoo! Games, feature many games that appear to have more than 5,000 individuals playing at the same time. The 31-day estimate is for labeling each of the images with a single keyword. In six months, the authors anticipate that each image could be labeled with six keywords.

Phetch is a game that produces natural language descriptions for images [131]. Natural language descriptions can be more useful than keyword lists, particularly for describing complex images. Natural language descriptions may be used to describe images to individuals who are blind. In each round, one player, called as the Describer, is shown an image. The Describer provides a textual description for the image to help the other players, who are called Seekers, select the image from among many different images using a search engine. Points are awarded to the Seeker who finds the image. The authors estimated that 5,000 individuals playing the game continuously could create captions for all of the images indexed by Google in ten months.

2.2.1.2 News Aggregation Websites

Slashdot [117] and Digg [32] are two examples of news aggregation websites. These websites maintain lists of user-submitted stories. Stories typically consist of a web link and a short text description. Moderation systems filter the user-submitted stories, removing duplicate entries and ranking them. Slashdot inspired the Digg moderation system. Digg now receives more than forty million unique monthly visits [31]. The traffic generated from a link that appears on the main page of Digg or Slashdot is often sufficient to overload the web server that is linked.

The Slashdot moderation system consists of users and moderators. Users submit stories. Moderators assign descriptors to comments. Descriptors include: normal, offtopic, flamebait, troll, redundant, insightful, interesting, informative, funny, overrated, and underrated. Each descriptor is associated with numeric values. A score for the comment is determined by calculating the sum of the scores associated with the assigned descriptors. This same information is also used to generate a characterization of the comment, such as “20% insightful, 80% funny”. The user who submitted the comment has their karma value (a measure of their reputation on the site) updated based upon the comment score. Moderators are assigned semi-randomly. The algorithm selects a subset of the users who view comments associated with a new story the opportunity to act as a moderator for that story. Meta-moderation is a process whereby users are selected to review the correctness of eight to ten moderation decisions. Meta-moderation assigns scores to moderators, affecting their karma values.

Digg allows users to vote for stories. Votes are called diggs. Users can also vote to “bury” a story. Stories with a large number of diggs (typically over one hundred) appear on the main page of the website. Stories with a very large number of diggs (typically over one thousand) appear in a special section of the main page that is updated less frequently. This provides increase exposure for stories with a large number of diggs. uSocial advertises the opportunity to pay a fee to have a story appear on the main page for Digg [118].

2.2.1.3 Wikipedia

Wikipedia is an online, user-moderated encyclopedia [135]. Any individual with Internet access can act as an editor, modifying encyclopedia articles even if they have not created a user account. The accuracy of the encyclopedia is maintained by a set of policies and guidelines that are enforced by volunteer editors who act as moderators.

Wikipedia's policies and guidelines include written rules for resolving disputes and for working with other editors. Articles are expected to be written from a neutral point of view, contain only verifiable information, and not include original research. Each Wikipedia article has an associated discussion page that acts as a forum for editors to organize and debate revisions.

Editors can place notices on articles to advertise that they do not appear to follow policies and guidelines. Editors can also indicate which articles they are planning to work on next, to avoid overlap. By creating a watchlist, an editor can quickly survey recent changes to articles that are of interest. Editors receive commendation in a number of different forms. Some editors have user pages on Wikipedia to which other editors can post messages or award them virtual badges. An internal peer-review process is also used to label some exemplary articles as "Featured Articles".

Some editors are provided with access to restricted technical tools. These editors are also called administrators. Administrators are able to reverse edits to articles and remove editors.

Administrators are also called upon to mediate disputes between editors. Editors can request a comment from an administrator, or request arbitration, depending upon the severity of the dispute.

Wikipedia has many policies in place to address vandalism. Individuals who have not created an account are not able to edit certain flagged articles, due to high rates of vandalism. IP addresses that have been used by vandals are also regularly banned. Some high profile articles require an editor to have a certain "edit count" (a numerical measure of their editing experience) before they are permitted to make modifications. Edit count refers to the number of revisions that an editor has made that have not been reversed.

A history flow visualization of Wikipedia edits has been developed to help researchers identify editing trends [128]. This visualization has been used to identify four patterns of cooperation and conflict: vandalism and repair, anonymity versus authorship, negotiation, and content stability. Vandalisms are categorized mass deletion, offensive copy, or phony copy. The authors indicate: “there is no clear connection between anonymity and vandalism” [127]. As of May 2003, anonymous editors had conducted approximately 31% of the edits to Wikipedia. The negotiation pattern refers to sequences of conflicting edits in which two or more editors effectively revert each other’s revisions. Finally, the authors also note that most edits consist of insertions or deletions, rather than moving text within articles.

Novice Wikipedia editors primarily locate information and fix mistakes [19]. They often begin editing in order to fix a perceived shortcoming in an article. Experienced Wikipedia editors, often referred to as Wikipedians, typically view themselves as members of a community. Experienced editors who are not administrators often still perform administrative roles. For example, answering novice editors’ questions at the help and reference sections within Wikipedia.

2.2.1.4 Analysis

These examples demonstrate that computer systems can be used to coordinate many humans, each performing a small task. Further, the humans who perform these computer-mediated tasks are primarily motivated by the task itself. They are not provided with compensation, beyond acknowledgement of the work they have performed.

The task does not need to be of great importance to a human for it to be performed. It seems likely that the easier the task is to perform, the less important the task can be and still be performed. Novice Wikipedia users are often enticed to edit articles when they find a mistake and then notice the “Edit this page” link. The accessibility of the task is sufficient to engage participation.

The Wikipedia and Slashdot user communities demonstrate that users can moderate themselves. Moderation can require a higher level of engagement than performing a task that is more directly applicable to an individual. Experienced users have a higher level of engagement with these websites. This is a reason to nurture experienced users.

Rewards and performance tracking may be effective methods to foster a sense of identity in a community of users. By helping users identify with the community, they may be encouraged to continue to participate, increasing the number of experienced users.

It can also be important to design systems to filter out intentionally inaccurate user submitted information. The ESP game verifies the accuracy of keywords by comparing the keywords created by different pairs of users who viewed the same image. Digg compares stories to other stories that have previously been submitted. If the match is exact, the story is not accepted. If there is a partial match, the submitted is prompted to review similar stories and verify that there is a difference. The submitter’s privileges may be restricted if the story is later flagged as a duplicate by other users. The Wikipedia system’s moderation tools allow both users’ accounts and IP addresses to have restricted access privileges.

The benefit to the user can be different than the benefit to the owner of the system. The ESP game is an example of such a system. This approach requires designers to address potentially divergent interests between the users and the system owner. It is also expected that this approach would be more difficult to scale to more complex tasks. More complex tasks might be more difficult to translate into fun games. However, designers may also find that there is a happy medium for many tasks, in which users may be willing to perform useful tasks that they also find moderately enjoyable to complete.

Designers who are evaluating whether to create a non-competitive direct motivation task might want to consider:

- The difficulty of the task
- The accessibility of the task for the humans who might be willing to complete it
- Methods to filter inaccurate user-submitted information
- Rewards that might be offered to encourage user-moderation

2.2.2 **Non-Competitive Indirect Motivation Tasks**

2.2.2.1 reCAPTCHA

A CAPTCHA is a Completely Automated Public Turing test to tell Computers and Humans Apart [130]. CAPTCHAs are used to verify that a user is human. Google, Yahoo, and many other providers of free email accounts require individuals to complete a CAPTCHA as part of their sign up processes. This step provides some confidence that a human, rather than a machine, is completing the sign up process. It also makes it difficult to create a large number of email addresses at once, which increases the cost of sending spam email messages. Most CAPTCHAs are images of words or phrases that have been distorted so that computer vision algorithms will be unlikely to be able to correctly identify the text.

reCAPTCHA is a web service that harnesses the human mental effort that is required to decipher a CAPTCHA [107]. Each reCAPTCHA image consists of two images that have been distorted so that they are difficult to read. One of the images is a word that is known to the system. The other image contains a word that is not known to the system. The word that is known to the system acts as the control. If a user does not enter this word correctly, they are considered to have failed the CAPTCHA. The word that is not known to the system is a word that is being transcribed. The reCAPTCHA system compares the text that different users entered. If a number of users have entered the same text for an unknown image, this text is considered to be the transcription for that image. In an evaluation, reCAPTCHA was found to be more than 99% accurate. A standard Optical Character Recognition algorithm was found to be 83.5% accurate on the same data set.

The images are presented in random order. This prevents users from being able to determine which word is the control. Users are also able to indicate that they would like to try a different set of images. This allows users to opt-out, rather than entering arbitrary text if they are unable to identify a word. It also provided an indicator that an image may be unreadable. These features may reduce inaccurate transcriptions.

If six individuals opt-out on an image, it is considered unreadable, and removed from the set of words to be transcribed. In an evaluation, only 4.06% of the images required six or more users to enter the same text for the transcription to be accurate. A post-processing step corrects text for predictable human errors. These errors include transposing letters, incorrect capitalization, failing to enter a space between the two words, extra letters accidentally appended to words, and Non-English characters.

reCAPTCHA is used on over 40,000 websites and has transcribed over 440 million words [107]. CAPTCHAs are necessary on many different websites. By creating reCAPTCHA as a web service that can be embedded on any website, the designers have managed to harness to the mental effort of a large number of individuals. These individuals are willing to perform this task for the indirect benefit of verifying that they are human.

2.2.2.2 Online Surveys with Participation Incentives

Online surveys are a standard method to gain insights from prospective customers and other target audiences. Many online surveys provide individuals with participation incentives. These surveys are examples of non-competitive indirect motivation tasks. Search engine listings, banner advertisements, and newsgroups are common methods to attract survey participants.

One study found that search engine listings were significantly more successful than banner ads and newsgroups at attracting individuals to complete a survey for a chance to win a mobile phone [105]. The authors found that their newsgroup postings were sometimes considered unwelcome even if they were not off-topic. Less intrusive approaches that are only likely to be noticed by interested parties are recommended. Of course, these approaches may also amplify effects from non-random sampling.

Advertisements that mentioned the incentive were three times more likely to attract a participant. However, even when the incentive was mentioned, the highest response rate was less than one in five thousand. This study also found that females were significantly more likely to respond than males.

Another study, which tracked participants' email addresses, found that 4% of the responses were from duplicate email addresses [85]. Researchers have also found evidence that individuals completing mailed print questionnaires may be more motivated but will not necessarily do a more thorough job [104].

2.2.2.3 Volunteers

Many websites facilitate the exchange of information between users. Some websites facilitate the organization of volunteers. One example is a website that was created to support the completion of an image search task for a missing aviator named Steve Fossett [136]. A second example is a website that was created by the The Guardian to crowdsource the identification of government expense documents that might be of interest to the general public [124].

More than 50,000 volunteers helped search for Steve Fossett by viewing and flagging satellite images of the 17,000 square-mile area in which his plane is believed to have crashed [137]. The website which hosted this distributed search task was built using Amazon's Mechanical Turk web service [86]. Each of the more than 300,000 images was viewed over ten times. The satellite images on the website were updated as new images became available. This online search was ultimately unsuccessful. Afterwards, one volunteer said "It was so exciting and new when we started it and it seemed like it could really help them, but eventually it was disheartening, and I realized I had no idea what I was actually looking for". A Major in the Civil Air Patrol said that the online search “added a level of complexity that we didn't need, because 99.9999 percent of the people who were doing it didn't have the faintest idea what they're looking for” [136].

Following media coverage of a major expense account scandal, The Guardian, a national British newspaper, downloaded over two million printed documents and receipts that the government made available online. The Guardian paid one software developer for one week to develop an online system that would allow Internet users to volunteer their time identifying receipts that might be of interest to the general public [124].

The website allows users to search for documents by postcode, or Member of Parliament or constituency. Users click one of a set of buttons after viewing an image of a document. The buttons allow the user to indicate the document type (claim, proof, blank, or other) and whether it may be interesting (not interesting, interesting, interesting but known, investigate this!). The main page features a progress bar and statistics about the number of documents that have been reviewed.

More than 20,000 volunteers have reviewed the documents using this system [125]. In the first 80 hours, over 170,000 documents were reviewed.

2.2.2.4 Analysis

reCAPTCHA demonstrates that tasks which humans perform can be modified to provide additional human computation data. The Internet allows for easy integration of web services. There may be many other opportunities to harness existing human computations. For example, Internet users who use social networking sites spend many hours browsing photos of their friends. This browsing data could be used to rank interest in the images.

It is likely that only a small percentage of advertising impressions will be effective. Providing an incentive to prospective online survey participants may be an effective method to attract more people. For these reasons, when possible, it may be more effective to use other approaches to entice participation. For example, building the task into an existing task, encouraging free word-of-mouth advertising by creating a community around performance of the task, or integrating the task an as optional activity on web sites that may attract people who are interested in performing similar tasks.

The volunteer examples illustrate that humans are willing to work together to perform computer-mediated tasks that may help them achieve a goal. Some people seem to have significant trust in the capacity of CHC systems to solve problems. This trust may not yet always be warranted, but compelling applications exist. The relatively low costs required to create a CHC system, along with the strong interest from the general public to participate in solving problems, may enable many more applications in the near future.

Designers who employ indirect motivation may approach web site owners or provide incentives to entice users to complete tasks. A web site owner may have sufficient leverage to convince a large number of individuals to complete the task. It is also possible to convince a large number of users to perform a task by providing an incentive. Incentives can be monetary or physical prizes, or more altruistic outcomes that may appeal to potential volunteers. Incentives can be an effective method to engage a large number of participants in a short period of time.

Designers who are evaluating whether to create a non-competitive indirect motivation task might want to consider:

- Existing tasks that might be modified to also achieve a desired CHC goal
- Providing an incentive
- Tasks that can be associated with major media events may be more likely to attract large audiences
- Response rate may improve if less intrusive advertising approaches are used
- Monitoring the results: people may participate even if their actions are not effective

2.2.3 **Competitive Indirect Motivation Tasks**

2.2.3.1 Mechanical Turk

Mechanical Turk is a web service that provides a marketplace for Human Intelligent Tests (HITs) [86]. A Human Intelligence Test is a task that humans can perform more cost-efficiently than computers. HITs include image and audio processing and subjective rating and ranking tasks. Mechanical Turk is advertised as being well suited to handle photo and video processing, data verification, information gathering, and data processing tasks.

Several companies outsource aspects of this business to Mechanical Turk. CastingWords [21], an audio transcription company that has been employed by the Wall Street Journal, hires and manages freelance transcribers and editors using Mechanical Turk. Most HITs pay users less than 5 cents USD. Amazon, the company that owns and operates Mechanical Turk, is paid a 10% commission on each HIT.

Amazon provides APIs for companies to integrate their online systems into Mechanical Turk. This allows the process of creating and approving HITs to be automated. Once a user has accepted a HIT, they are typically given a short period of time (less than an hour) to complete it. Creators can specify qualifications that a user must have before they can accept a particular HIT. Qualifications are earned by completing online tests called qualifiers. Users can complete qualifiers on the Mechanical Turk website. These tests allow users to demonstrate their capacity to perform specific tasks. For example, there are qualifiers that test users' abilities to transcribe audio, to label images with keywords, and to write Java source code.

The creator of a HIT has the right to decide whether or not to pay the user who accepts the HIT, regardless of the user's performance completing the task. A HIT that has been indicated by its creator to be successfully performed is called an approved HIT. Online message boards have been created to provide experienced Mechanical Turk users, often called turkers, with venues for rating companies that post HITs [127]. Turkers also use message boards to warn each other about scams.

Turkers warn each other about HITs that require personal information to be entered. HITs requesting users to click on the banner advertisements displayed on particular web pages have also been created. Spammers have created HITs that require users to sign up for email accounts and then share their passwords. Turkers have also indicated that HITs have been created for the specific purpose of completing CAPTCHAs.

Over 100,000 users from over 100 countries have performed HITs. There is data indicating that there are more female users than male users, and most users are under the age of 40 [103]. Forum posts and articles written about turkers indicate that most users earn significantly less than minimum wage by performing HITs. Despite the relatively low pay, most Mechanical Turk users are located in the United States [103].

Reasons for participation vary. Turkers who have been interviewed have cited many different reasons for participating. These reasons include: the ease of performing the tasks while in distracting environments, the ability to earn some money in one's spare time at home, and preference to work rather than watch television in their free time [113].

Little et al. evaluated models for HITs that are iterative steps in the refinement of a solution [75]. Tasks included improving the descriptions of images, improving a letter to (better) convey an outline, deciphering handwriting, and sorting images. Instructions requested users to complete small improvements towards goals. Goals included "make it succinct", "use the present tense", and several others [75]. The paper discusses the possibility of automating the selection of different goals based upon other HITs that moderate the progress made between the iterative steps.

The authors were initially concerned that turkers might try to game the system. Their first evaluations include review HITs that required a majority of the users to agree that tasks had been completed properly. This review process did not prove to be necessary. Subsequent evaluations instead only required users to have a 90% approval rating on previously performed HITs.

Once submitted to Mechanical Turk, review HITs were generally completed in 7-15 minutes and improvement HITs in 15-30 minutes. Many of the results from the evaluations were included in the paper, indexed by iteration number. Although the quality of the results was not formally evaluated, the iterative improvement method appeared to be effective.

2.2.3.2 InnoCentive

InnoCentive is a web service that provides a marketplace for companies to outsource research and development work to individuals [56]. Similar to Mechanical Turk, companies, called seekers, post problems and individuals, called solvers, post solutions to the problems. Innocentive collects a fixed fee once a solution is selected. Proctor & Gamble and Eli Lilly are companies that have posted multiple problems on InnoCentive. Awards for solving problems are typically between $10,000 to $100,000 USD. InnoCentive has paid out over $3.5 million in awards to over 300 solvers.

Once solvers have submitted solutions to a problem, the seeker evaluates the solutions and selects the one that best meets the stated criteria. Seekers are typically given 2-6 months to evaluate solutions. InnoCentive reserves the right to "audit the decision making process of the Seeker on any specific awards where there might be an issue or question around payment" in order to ensure a fair outcome is reached [56].

InnoCentive supports four different types of problems: ideation, theoretical, reduction to practice (RTP), and electronic request for proposal (eRFP). Ideation problems are brainstorming problems to come up with new market opportunities, such as a product or service. The seeker received a non-exclusive perpetual license to use all of the submitted solutions. These solutions are typically two pages in length.

Solutions to theoretical problems are typically awarded larger payments than ideation solutions. In most cases, the terms of the agreement require the solver to transfer or license the intellectual property rights. Solvers are provided with technical evaluations of their submissions regardless of their selection by the seeker.

RTP problems require a precise description and evidence of solution optimality. These solutions typically require the most time to prepare and have the largest cash awards. eRFP problems do not have cash awards. Terms are directly negotiated between seekers and solvers.

2.2.3.3 Analysis

Mechanical Turk features a very different form of competition than InnoCentive. A Mechanical Turk users' performance is often measured by the number of HITs that he or she has had approved and rejected. Many HITs require users to have completed a certain number of HITs successfully and to have a minimum ratio of approved to rejected HITs. Users scan the Mechanical Turk website, racing each other to be the first to accept easy or profitable HITs. Competition is primarily on time to find and time to complete HITs. Most users appear to be motivated by the opportunity to perform a useful task in their spare time, rather than, for example, watching television, and to earn a relatively small amount of money.

The process of producing and accepting an InnoCentive solution is complex. Solutions can require technical evaluation by experts and months to prepare. Competition is not on the basis of time to find or complete, but instead on the quality of the product. Most users are motivated by the opportunity to earn a considerable amount of money and to receive credit for having performed a task of significant value to a large corporation.

Mechanical Turk demonstrates that a large number of individuals are willing to perform simple tasks for less than minimum wage. Many Mechanical Turk tasks are created by dividing larger tasks into smaller tasks, moderating fulfillment of the smaller tasks, and then combining them to create a solution to the larger task. It is possible that this approach might also yield useful results for problems that are submitted to InnoCentive. On Mechanical Turk, multiple users could be working on different aspects of the problem at the same time. For some problems, this approach might also significantly reduce the time required to receive an acceptable solution.

Designers who are evaluating whether to create a competitive indirect motivation task might want to consider:

- High paying or low paying tasks
- Opportunities to reduce large tasks to sequences of smaller tasks
- Costs of paying users
- Opportunities to get results without paying users

2.2.4 **Competitive Direct Motivation Tasks**

2.2.4.1 Google Search Ranking

Google search results appear in a specific order, determined by a web page ranking algorithm. The ranking of a web page is partially dependent on the number of other web pages that link to it, and the rankings of the web pages that link to it [17]. HTML links from web pages that are highly ranked are more influential in determining the ranking of the web pages to which they link.

The ranking of a web page in Google's search results is often important to the web site's owners. Web sites that sell products or feature advertising, among others, have a strong incentive to attract as many visitors as possible. Companies create, or remove, HTML links to improve search result rankings [141]. For these reasons, Google search ranking is a competitive, direct motivation task.

Many techniques have been developed to improve search result rankings. 302 page hijacking and link farms are two examples of search engine index spamming. 302 page hijacking can trick web crawlers into believing a particular web address displays the same content as another web address. Since Google removes pages that contain duplicate content from its results, 302 page hijacking can cause a legitimate web address to be replaced with an illegitimate address. A link farm is set of websites that include web links to each other. This technique can inflate the rankings of each of the pages. Some embodiments of this technique disguise the exchange of links by having only some of the set of websites link to any specific other website. Automated programs have been developed to create and maintain link farms [141].

2.2.4.2 Yahoo! Answers

Yahoo! Answers is a web site that allows users to ask and answer each other's questions [144]. A user can only ask a few questions before he or she must answer some questions in order to be allowed to ask more questions. The specific number of questions that must be answered is dependent on the quality of their answers, as points are awarded based upon other users' grading of the answer. In the default cases, 2 points are earned for answering a question and 5 points are taken away for asking a question.

When a question is created, it is categorized using a fixed, three-level hierarchy. The categories cover a broad range, including makeup and mathematics. Users who ask or answer questions in specific categories will be prompted to answer questions in those categories. The creator of a question selects one of the answers as the best answer. That answer is awarded additional points. The creator can also increase the points that are awarded to the chosen answer by offering some of his or her points. This may increase the number and quality of answers. All users who have at least 250 points can vote either for or against any other users' answers. These votes, and other factors, such as how often the user visits the site, determine the number of points they are given [143].

Each user is assigned a level, depending upon his or her number of points. A user's privileges increase as their level number increases. For example, a user with 5,000 or more points is not subject to any daily limits on the number of questions asked or answered.

Adamic et al. found that Yahoo! Answers users who focused their answers in fewer categories tended to have answers selected as the best answer more often [2]. Categories favoring factual answers were found to be more likely to have fewer and shorter answers. Users participating in these categories were also found to be unlikely to both ask and answer questions in the same category.

Liu and Agitchein found that as Yahoo! Answers has grown the complexity of the questions which are asked has increased [78]. They also found that users are becoming more likely to be passive, voting on each other's answers rather than answering questions. Further investigation would be required to determine the overall effect on the quality of the answers. Bouguessa et al. have taken a step in this direction by creating an algorithm to identify authoritative users [12]. These authoritative users were demonstrated to "contribute significantly to the generation of high-quality content".

2.2.4.3 Analysis

When users are directly motivated to be competitive, it may be especially important that there is a robust heuristic for gauging the quality or accuracy of task performance. Indirectly motivated users of competitive systems are not primarily motivated to compete. Users of competitive direct motivation systems may be more likely to consider competition to be their task.

Google search results are subject to carefully researched and organized collusion among webmasters. Web links that are created for the purpose of manipulating search result rankings, rather than directing web site visitors to related content, can be considered noise on the signal that is interpreted by PageRank. The success of Google suggests that the search algorithm is sufficiently robust to filter out most of the noise in this signal.

Yahoo! Answers allows the creator of a question to indicate which of the answers is best. If the purpose of the system is considered to be answering each user's questions to their satisfaction, the selection of a best answer (or decision not to select) may be a highly robust heuristic. Of course, it is possible that the user has unknowingly selected an inaccurate or less accurate answer. One advantage of the voting system is to allow other users to help the question creator select from among the answers. Users might collude to vote up an inaccurate answer, however, the selection decision is still entirely within the control of the question creator.

In the case of Google search, the robust heuristic is an algorithm. The algorithm appears to be effective because it models the primary reason that web links appear on most web pages. Most web links are created to provide web site visitors with access to related content. In the case of Yahoo! Answers the heuristic is a moderation system. This heuristic appears to be effective because it is relatively easy for a question creator to judge content quality.

Designers who are evaluating whether to create a competitive direct motivation task might want to consider:

- How collusion may be controlled
- If there is a robust heuristic for measuring quality/accuracy
    - If the heuristic is an algorithm, the accuracy of the use case model
    - If the heuristic is user moderation, how difficult it may be to judge quality

2.2.5 **Classification Analysis**

2.2.5.1 Direct and Indirect Motivation

Designers who employ indirect motivation may approach webmasters or users with incentives to increase user participation. CHC direct motivation examples can also use these methods, but it may be difficult to formulate direct motivation tasks such that indirectly motivated users will be effective participants. For example, the quality of Wikipedia articles would likely differ if users were paid to contribute. A considerably more complex moderation system might be required to prevent collusion.

Indirect motivation tasks may require different moderation. The Wikipedia, Digg, Slashdot, and Yahoo! Answers moderation systems are reliant upon experienced users. Users who participate in indirect motivation tasks may be less likely to be concerned with the community of users, as their primary reason for participation is an incentive. The moderation systems for most of the direct motivation tasks are optional. Users are not forced to moderate if they are not concerned about the community or the quality of its output. When moderation is required, the quality or the accuracy of the moderation may also differ between indirect and directly motivated participants. Tasks that use Mechanical Turk, an indirect motivation example, often feature multiple levels of moderation. Further, HIT creators have the right to reject users' submissions without explanation.

The success, and relatively inexpensive costs, of operating direct motivation tasks are a compelling argument for their use. Building the task into an existing task, encouraging free word-of-mouth advertising by creating a community around performance of the task, or integrating the task an as optional activity on web sites, may attract people who are willing to perform the task.

Indirect motivations, including incentives, may be an effective alternative when a larger number of participants than would otherwise be likely to perform the task are required within a particular period of time. Depending upon the task, indirect motivation may also require less effort to implement, as the user experience may not need to be enjoyable for participation to occur.

2.2.5.2 Non-Competition and Competition

Competition can be a useful task feature. Competition on Mechanical Turk decreases the time before HITs are completed. Most of InnoCentive's tasks are inherently competitive. The quality and number of answers to questions on Yahoo! Answers is at least partially dependent on the competitive nature of the task.

Systems that feature competition between users require robust heuristics for measuring the quality or accuracy of the users' contributions. The heuristic may include an algorithm or a moderation system. One approach to the design of heuristics is to create models of the system use cases. Some users may attempt to exploit the heuristic. The heuristic must be able to extract the signal that it is intended to interpret from the noise generated by exploitation attempts.

Tasks that are not inherently competitive or that are reliant on experienced users are particularly vulnerable to be negatively affected by competition. Introducing competition to a non-competitive task can reduce the sense of community between users. Wikipedia and Digg are examples with non-competitive tasks and moderation systems that rely on experienced users. Moderation systems that are reliant on experienced users will also be more prone to manipulation by collusion if the task is competitive.

2.2.5.3 Motivation Interaction with Competition

Competitive indirect motivation tasks may be improved by conversion into competitive direct motivation tasks. These tasks can be significantly less expensive to operate, as no incentive may be required to encourage participation. The quality of users' contributions may also increase, as they will be more likely to be concerned with system performance.

In some circumstances, competitive direct motivation tasks may be improved by conversion to indirect motivation tasks. Designers who do not have a robust heuristic for measuring the quality or accuracy of user contributions, or webmasters who find that the heuristic is ineffective after the task has been created, might transform their tasks to be indirect motivation tasks. A new moderation system to control the distribution of the incentive can be introduced. If the indirect motivation incentive is sufficiently compelling, collusion may be reduced.

It can be difficult to directly motivate users to perform complex tasks. InnoCentive, which features highly complex tasks, provides incentives that are commensurate with the difficulty of the tasks. By dividing complex tasks into a large number of easier tasks, it may be possible to encourage communities of users who are highly motivated by a task to complete it using a non-competitive direct motivation system.

2.3 **End User Programming**

I am interested in developing CHC systems that include programming tasks. There are many programming tasks that might be crowdsourced. For example: writing scripts to automatically extract content from the web or algorithms to recognize events in data streams. As described in the preceding sections, many tasks can be crowdsourced. Since many crowdsourcing participants are likely to have limited programming experience, I focus on end user programming paradigms. In this section, end user programming systems are described and their relevance to the design of user interfaces for CHC participants is considered.

2.3.1 **Programming By Demonstration**

Time and effort is required to learn programming languages and write source code. Programming by demonstration (PbD) allows a user to specify operations without learning a formal programming language [5]. PbD has been used for end user software development because it does not require textual programming and can hide program logic from the user [99]. With PbD, the user demonstrates a sequence of operations for the computer to repeat. Pattern recognition algorithms can be used to generalize the sequence of operations, allowing them to be applied to different data sets without further demonstration. For example, a user can train Exemplar [50] to recognize a sensor-based interaction, such as shaking an accelerometer, by performing the interaction once.

Exemplar is a PbD tool for authoring sensor-based interactions [50]. Exemplar models sensor signal patterns, demonstrated by users, so that similar signal patterns can be recognized. For example, Exemplar has been used to augment a bicycle helmet with lights; sensors were set to blink the lights when tilting was sensed. Results indicate that Exemplar reduces the time required to build prototypes and that it facilitates experimentation. Exemplar extends the functionality of d.tools [48], a statechart-based visual design tool for prototyping information appliances. Product designers, with no prior programming experience, have been able to quickly develop interactive hardware prototypes using Exemplar and d.tools.

Bergman et. al. created DocWizards, a PbD wizard for authoring interactive documentation [70]. DocWizards allows a user to document a sequence of actions, required to complete a task, by performing them. Documenting a sequence of actions by performing them can be less time consuming, and less error prone, than writing them down as text. DocWizards records the actions using application hooks. Users can replay the actions, either stepping through them one at a time, or with all actions performed at once. This can reduce the effort required for the task to be completed. DocWizards was designed to provide an alternative to traditional, non-interactive manuals, which can be difficult to navigate and understand. DocWizards uses visual overlays, such as a red circle around a button that will be clicked, to indicate the next action that will be performed. Users can also customize this interactive documentation. For example, instead of clicking a button to have the next action performed, a user can manually perform a different action, and have it saved as an alternate path within the wizard. This allows users to extend the documentation as different cases are encountered.

Koala [76] is a programming-by-demonstration system that records user interactions within web browsers. Recorded interactions are stored as pseudo-natural language scripts that can be read by humans and machines. For example, the following three lines might appear in a Koala script that searches Google for information about the Human Media Lab: “go to http://google.com”, “type human media lab into text box”, and “click search button”.

When a user modifies a script, Koala transforms the edited pseudo-natural language instructions into syntactically correct formal programming statements. This is accomplished using a revised version of the Keyword Commands [74] keyword interpretation system, described in more detail in the following section. The interpretation of an instruction is presented visually during execution. For example, a transparent green rectangle would appear around the search button if the next instruction would cause it to be clicked.

Users can click a button in the Koala interface to bold the words within an instruction that were used to generate the formal programming statement. This can simplify the process of revising instructions that are not correctly interpreted. For some pseudo-natural language instructions, Koala can also provide a list of alternate interpretations for the user to choose between.

Scripts can also be generalized to work with different user-specific data. When user-specific data is required, the Koala interface prompts the user to assign values to unassigned variables. For example, a script to re-order office supplies might include a variable named “email address” to which an order receipt should be sent. However, users are not able to modify or extend the set of instructions that Koala can interpret. It can also be difficult to determine how to properly phrase an instruction so that it can be interpreted, since Koala features neither suggestions nor autocomplete.

Eager is a PbD system for automating repetitive tasks [4]. Eager runs in the background in the HyperCard environment, monitoring the user’s actions. The user does not indicate when a task that can be automated is initiated. When Eager detects repeated sequences of actions that it can automate, it notifies the user by displaying a special icon on the desktop. If the user acknowledges Eager’s recognition of a repetitive task, Eager confirms the task by stepping the user through the sequence of actions that it has detected. This is accomplished by visually highlighting the next button to be pressed or underlining text to be inserted. For example, Eager can detect and repeat the mouse and keyboard events required to copy and paste the subject lines from unread emails into a text document. Eager can determine that different actions have the same effect. This is important for the detection algorithm because users often do not perform repeated tasks exactly the same each time. For example, there are multiple ways to move a card in a HyperCard stack. Once a user confirms that Eager has recognized the repeated task, the performance of the task can be quickly completed by either automating the completion of each occurrence of the task individually or all at once.

Many other PbD tools detect and automate repeated actions. Witten's Predictive Calculator [139] recognizes repeated patterns of operations performed on a calculator. SMARTEdit [69] automates text editing tasks, such as address formatting, by generalizing recorded macros. Eager, Predictive Calculator, and SMARTEdit do not directly support conditional actions or nested loops. These cases are more difficult to passively detect, requiring more domain knowledge or many repetitions for recognition.

PERIDOT [98, 94] allows users to specify conditional actions by demonstrating multiple repetitive actions. PERIDOT is a tool for creating UI elements, including menus and scrollbars. Repetitive actions are inferred as interactions. Inferences are immediately displayed to the user for confirmation. PERIDOT introduces *active values*, which are similar to variables, and can immediately update dependent objects. To keep the tool easy to use, all interactions associated with an active value can be removed together. Repetitive actions can also be re-demonstrated. Users are intended to demonstrate many, simple repeated actions, with PERIDOT detecting and the user then confirming the intended interaction. For example, to create a scrollbar, a user might first create a rectangle within a rectangle, and confirm with PERIDOT that the inner rectangle should be evenly nested within the outer rectangle. PERIDOT allows users to construct complex, conditional interactions without explicitly specifying program logic. However, the demonstration and confirmation process can be tedious, and PERIDOT is only able to recognize a limited set of possible interactions.

GAMUT [84] introduces new objects and metaphors to enable more interactions to be recognized. With GAMUT, a user can highlight an object to indicate that it is associated with an interaction, providing a hint for the inference algorithm. Users can also *nudge* GAMUT to "Do Something" (infer an interaction) or "Stop That" (infer a different interaction). Question dialogs are displayed to resolve contradictions and to determine whether objects that are not highlighted should be included in an inference. *Temporal ghosts,* faded overlays of prior environment states, are used as guide objects to clarify interactions. Lists and random events are supported using a deck-of-cards metaphor. Users can display the cards on the screen, use them to control interactions, and shuffle the deck to create random effects. While these features require the user to be more proactively engaged in specifying inferences, they considerably extend the set of interactions that tools such as PERIDOT can achieve. Assuming new objects and metaphors can be easily taught and applied, they may reduce time and effort requirements. These features reveal more of the program logic to the user, blurring the line between demonstration and programming.

Programming by example (PbE), a term often used interchangeably with PbD, can also hide program logic from the user. PbE tools accept output examples, provided by the user, and produce sequences of operations that can generate the output [39]. The operations can then be executed with different data, to produce different output. For example, consider a user wishing to obtain a result from performing an SQL query on a relational database. Using a Query by Example (QbE) tool [80], a user can obtain a result by entering a few exemplary fields in a table representing the search result. The tool then extrapolates source code in a query language, such as SQL, to extract the desired results from the partial result listing.

2.3.1.1 Analysis

In this section, the use of PbD in CHC systems is considered with respect to two general types of programming tasks: (1) automating common computer tasks (e.g. writing a script to scrape data from a web site) and (2) recognizing a pattern in a data set (e.g. detecting events in a data stream).

PbD is often used to document a sequence of actions by performing them. The actions can then be executed on demand. This can be easier than writing a script from scratch, which can require learning a programming language. Since Koala is integrated into the browser, users can record tasks by performing them as they might normally. Koala also supports post-hoc modifications by representing the action sequences in a human readable script. These features allow Koala to be used to perform many web automation tasks. Many individuals with experience using computers, but no programming experience, can be trained to use Koala in a few minutes.

GAMUT includes tools for script modification, including the ability to infer relationships between actions and data structures. However, it was designed for use in a limited environment that restricts its direct application to common tasks. It might be possible to extend Koala, or similar tools, to support GAMUT logic and data structures. For example, Koala could be extended to recognize repeated actions, such as selecting every value in a row of values.

The relatively minimal training requirements for many PbD tools make them attractive for CHC systems. If a task requires predominantly user-interface manipulation actions that participants already perform on a regular basis, a PbD tool could be quick to learn and use. Lowering barriers to participation can increase the number of users who will complete a crowdsourcing task. It is also particularly important for non-competitive direct motivation CHC systems, which often rely on the task being engaging to attract participants.

Exemplar and Query by Example demonstrate how tools can be used to recognize patterns in data sets. In the case of Exemplar, the sample output is used to create a recognizer, and in the case of Query by Example, the output is used to retrieve other database search results. No prior programming experience is required. Image labeling, and other common crowdsourcing tasks, can also benefit from generalized sample output. Thousands of participants can create output samples that cover many edge cases, for example, to provide a secondary prediction source when existing character or scene recognition algorithms fail. A participant might create the sample output by sketching the letters in the word on top of the image. Such a tool might replace CAPTCHAs: providing not only recognition for a single occurrence of a word but also a model that can be used to detect other occurrences. A tool for this task might even be able to verify the accuracy of the model without using a control. Instead, the tool could test the precision a recognizer generated from the sample output.

2.3.2 **Structured Editors**

Structured editors are software tools that are customized based on editable content. For example, Microsoft Word, which can correct common word misspellings, is a structured editor.

IntelliSense [57], which can be used to quickly find relevant methods and variables, is another example. In programming environments, structured editors can prevent syntax errors and guide users through the process of selecting operations [123, 20].

MENTOR [34] is one of the earliest structured editors for programming. The MENTOR Pascal editor allows no exceptions for breaking syntax. This restricts how programmers can use the system, since modifications cannot temporarily leave a section of the source code incorrectly formed. The dangling else problem, which occurs when it is unclear whether an else statement is associated with an outer or inner if statement, is addressed by including an else statement with each if statement.

The Cornell Synthesizer [123] avoids the dangling else problem with templates. Templates are predefined character patterns with placeholders. For example, "condition" and "statement" in the following template are placeholders: "DO WHILE (condition); {statement} END;". Templates can also be added within templates. Users can enter text or templates in place of the placeholders. In this way, source code modification is guided by the grammar. Expressions and assignments can be inserted as text. The compiler is automatically invoked immediately after text entry to detect errors. The Cornell Synthesizer warns the user when an error occurs and highlights the text, but does not require that it be corrected before other operations are performed. It also includes run-time debugging features for tracing program execution with visual highlighting of the current line of execution and controlling the rate of execution.

Gandalf, which originally referred to a specific structured editor, now refers to a set of software development environments [46]. The structured editors for these environments are all derived from the ALOE (A Language Oriented Editor) generator [39, 87], which produces editors from language descriptions. One advantage of ALOE editors is that they allow for multiple program views by supporting alternative mappings of ASTs into visual representations. The GNOME [41] family of structured editors are all Gandalf software development environments.

The GNOME [41] structured editors include a family tree editor, an editor for the Karel programming language, which has no nested loops or variable declarations, a Pascal editor, and a Fortran editor. These editors were designed to incrementally develop a novice's programming skill.

The Pascal and Fortran editors have semantic checking for common errors, including invalid variable declarations and conditional expressions. These editors include in-context help for each editor function. They also support associating text comments with parts of the AST. Unlike prior editors, the text cursor movement could also be structural or textual. For example, it is sometimes faster to click textually than move structurally. The next cursor position movement was also changed to follow control flow rather than text location. Finally, templates are organized into levels so that rarely used templates are not always visible. However, these editors are not without shortcomings. Rigid rules and the limited support for transformations can require multiple syntax tree levels to be removed when making minor modifications [41].

The GENIE structured editors, also developed at Carnegie-Mellon, build upon the GNOME editors. MacGNOME [90] and ACSE [102] are two GENIE editors. MacGNOME enhances mouse interactions: the smallest structure that contains the mouse cursor will be selected when the mouse is clicked. Code structures can also be navigated by holding down and dragging the mouse. The parser supports code-completion: typing "for" then pressing enter will create a loop structure. MacGNOME also has relaxed syntax constraints. Teaching programming with MacGNOME, instructors have found it useful to include text comments in place of statements and conditions.

The ACSE (Advanced Computing for Science Education) project [102] includes a programming environment designed to enhance science education. It introduces a new view called the "volume view" that can display graphics, movies, and interactive simulations. This visual and audio content can be affected by modifying program elements. Students who were encouraged to investigate the programming of prebuilt, interactive biology applications were found to receive higher test scores on the material [101].

Alice [25, 26] is a python programming environment for controlling the behavior of 3D objects. It has been used to support teaching programming at several universities. GUI tools and programming can manipulate 3D objects and handle user actions. Similar to ACSE, a custom view displays a graphical representation of the current state of the system: in this case, a 3D environment. This view can also be used to directly manipulate the placement of objects. Alice2 [20] extends Alice, The Alice2 editor includes five sections: scene window (drag in, position, and resize objects), object tree (object selection), object details (customizable parameters), animation editing area (main programming area), behaviors area (associate programming with events). Alice2 implements a drag and drop system for specifying source code by directly manipulating [116] logic structures. When a template is dropped into the source code view, menus with valid parameter choices appear. Scratch [110], an introductory programming environment for children, features a similar drag and drop structured editor. Scratch templates include visual indicators, similar to the indentations on puzzle pieces, to describe how they can be ordered. For example, the visual representations for loops include special indentations to indicate that statements can be nested within them.

Whyline [64] allows programmers to debug their programs by asking “why did” or “why didn't” questions. These questions are used to generate a visualization of the relevant source code dependencies. Software engineers typically use breakpoints, code-stepping and print statements to debug. The Whyline uses event selection and visualizations to reduce accessibility barriers for hypothesis evaluation. In a controlled study, Whyline was found to reduce debugging time by a factor of 8. The bugs encountered by the participants were also created by the participants, as the participants were not given code to work from, but only instructed to complete particular tasks. The design of the Whyline appears to be extensible to other programming languages and also to include additional features, such as the ability to ask questions that relate multiple objects ("semantic differencing").

The authors recognize that “questions about complex Boolean and numerical expressions give equally complex answers”. It is unclear how this problematic issue might be overcome. The visualization, which interactively reveals the execution path, may require significant expansion in order to reveal the information necessary to validate a hypothesis.

Barista [66] is a Java programming environment with interaction techniques for combining text editing with visual code representations. For example, rather than collapsing a section of code when it is not currently of primary importance, double-clicking will reduce the font to half the previous size, allowing more code to be viewed. Barista also introduces new views, embedded within the source code view. Some source code fragments are displayed in easier to read formats when the text cursor is not upon them. For example, Java expressions representing math operations are transformed to standard mathematical notation. Barista was built using the Citrus

[65] toolkit for creating structured editors. Structured editors are created by specifying classes of data to be edited and then views for each class.

Blueprint [13] is a development environment plugin that augments Internet keyword search queries with the language and framework used in the development environment. Snippets are automatically extracted from web pages and can be browsed within the plug-in. While keyword matching can be ineffective for locating many general code structure or program logic patterns, it is easy to use and can match comments, variables, dependencies, and other lexical features. Blueprint filters search results by language and framework to fit the context of the code.

2.3.2.1 Code Snippet Integration

Tools that support the integration of source code snippets can sometimes reduce the effort required to write complex code. Eclipse Templates [38] is a code snippet search and integration tool for the Eclipse development environment. Eclipse Templates provides a simple lookup system for inserting predefined code snippets, called templates. New templates can be created by the user and can include a markup with customization instructions to match existing code. For example, a template can include placeholders for variables or suggest unused variable names, and can automatically import libraries if they are missing from a project.

Many different types of modifications can be required to integrate a code snippet. EUKLAS [35] highlights source code errors and suggests corrections. EUKLAS can detect and correct missing JavaScript parameters, function and variable definitions, and imports. Eclipse Quick Fix [37] provides similar functionality for Java, and can also correct some unhandled exceptions. These tools allow programmers to quickly resolve many simple errors.

HelpMeOut [49] presents suggestions for correcting compiler and runtime errors. Relevant compiler error suggestions are found by examining the source code line referenced in the error. Relevant runtime exception suggestions are found by examining the stack trace. Suggestions are manually created, shared, and voted on by programmers.

2.3.2.2 Sloppy Interpretation

A sloppy interpreter is a structured editor that is customized based on text typed by the user. Sloppy interpreters have lenient syntax requirements [74]. Inky [91] interprets keyword searches as web tasks and allows the user to click a button to perform them. The interpreter attempts to fit the command line text to *functions* that have *arguments* of different *types*. The command line text is tokenized on whitespace boundaries then a type recognizer identifies token types. Unused text is grouped into longest contiguous sequences, surrounded by quotes, and matched with optional arguments. For example, the top search result for the query "johnd@gmail.com leaving the office now" might be "email johnd@gmail.com about "leaving the office now". The keyword interpreter tokenizes the query, detects token types (e.g. email addresses), matches the tokens with predefined functions, and returns an ordered list of results. Koala [76], described in the programming-by-demonstration section, also allows users to write scripts to automate web tasks by typing sequences of commands. The Koala interpreter can recognize the commands "type UIST into search field" and "click search button". Koala built on Greg Little's previous work translating keyword commands into source code [73, 74]. Snippets are created by nesting methods, extracted from open source projects, with names similar to the search keywords. Unfortunately, this algorithm is often not practical because it can create semantically incorrect code.

2.3.2.3 Search Interfaces

Search interfaces [51, 140] can be tailored for specific tasks. Prior work includes systems to support data analysis [16, 88] and programming [13, 120]. Programming search interfaces can enhance web search results [52, 120] and, most relevant to this work, locate code snippets [13, 112]. While not specifically designed for end user programmers, these systems could be relevant to the design of future tools. Snippet search interfaces query code repositories to find keyword [13, 38, 42] and structural [7, 82, 112, 126] matches. Structural search can include building an internal representation of the program, while keyword matching is limited to text analysis. Google code search [42] locates occurrences of keywords in source code files. Uncommon, domain-specific search keywords can improve the relevance of results from this large repository.

Blueprint [13] is a development environment plugin that augments Internet keyword search queries with the language and framework used in the development environment. Snippets are automatically extracted from web pages and can be browsed within the plug-in. While keyword matching can be ineffective for locating code structures or program logic patterns, it is easy to use and can match comments, variables, dependencies, and other lexical features.

Structural snippet search interfaces perform static analysis to determine how code functions. For example, structural search interfaces can locate snippets that create an object of a certain type from other objects. Locating a Method Invocation Sequence (MIS) is a common search task when programming with an API [112, 126]. PARSEWeb [126] ranks code fragments to be incorporated into MISs by frequency of use and length. Structural search is a narrow domain. However, it can be difficult to ensure that snippets do not contain errors [82] and to differentiate between similar structural features: for example, frameworks and class libraries [7].

2.3.2.4 Analysis

PbD tools may be difficult to use if the actions required to complete a task are not familiar to the user. Unlike tasks that predominantly involve demonstrating a sequence of common user-interface interactions, the time and effort required to complete some crowdsourced tasks consists primarily of determining (or creating) the actions to perform. Many competitive CHC tasks belong to this category. For example, Yahoo! Answers tasks often involve locating information on the Internet and then summarizing findings. Many Mechanical Turk tasks are also primarily dependent on mental actions, rather than common user-interface interactions. This may become a distinguishing feature of competitive CHC tasks. Tasks that require more skill (mental actions) may require stronger incentives and CHC systems reliant on these tasks might benefit from competitive markets.

Structured editors that provide environments for experimentation and testing could be well suited for these tasks – that is, tasks that require relatively more mental effort and the use of complex actions. For these tasks, actions may need to be selected from large sets of possible actions, and it may be necessary to spend considerable time structuring their order and interactions. ALICE, and other structured editors that can be used by novice programmers, might provide a good starting point from which to build tools for these tasks.

While some training is likely to be required, it may be possible for the training to be automated. Mechanical Turk includes a feature for testing prospective participants' skills. Only participants who pass a qualification test are eligible to work on certain tasks. If training is automated, it could be offered as a free service to prospective participants, and participants who have received positive feedback for completing prior tasks might even be compensated to complete the training.

A CHC system could also feature multiple roles for participants. For example, participants with minimal programming experience might be able to request that participants with extensive programming experience create logic or data structure components. For example, if the task required the creation of a script that can recognize human faces in images, a participant with minimal programming experience might request a component that could normalize the lighting in the image or convert it to black and white. These components – possibly expressed as methods in a novice programming language – could be similar to code snippets. Participants might use a search interface with sloppy interpretation to locate existing code snippets. Further, tools for building crowdsourcing systems, similar to TurKit [75], might be created to support these crowdsourced programming tasks.

It might make sense for crowdsourced programming tools to include unit-testing functionality. This might, for example, provide a participant with a performance indicator and ideas for improvement. Further, participants might be allowed to use structured editors to modify scripts created by other participants. For example, a participant that significantly improves recall or precision by modifying an existing script might be compensated. Character recognition and machine translation, two tasks that are often crowdsourced, can have many edge cases. Small modifications to existing scripts may sometimes be sufficient to accommodate previously unaddressed edge cases.

2.4 **Summary**

Four different classes of CHC tasks, with varying motivation (direct or indirect) and competition (competitive or non-competitive), have been described and compared. Considerations for designers and opportunities for future work have been identified. In particular, methods to improve task performance by transforming complex tasks into many simple tasks should be investigated. Methods to encourage and support CHC contributions from users who are highly motivated to participate may also provide substantial improvements.

Following the study of CHC systems, relevant end user programming tools have been described and design opportunities for future systems considered. Features of programming by demonstration tools are can be relevant to CHC tasks that require common user-interface interactions. Structured editors could be useful when participants are required to perform less familiar actions. Several examples describe how tools for CHC systems can be designed to facilitate crowdsourced programming. It may be possible for a thousand participants, each with limited programming experience and organized by a CHC system, to jointly create an algorithm that can solve an open problem in computer science.

Chapter 3

WordMatch: Enhancing Search Results with Direct Answers Created by End Users

This chapter describes the design, implementation, and evaluation of an end user programming environment that can be deployed as a CHC system. The environment was designed to support the completion of a direct motivation, non-competitive task: adding direct answers to search engines. We chose direct motivation because, as noted in the last chapter, users are more likely to be highly motivated to contribute and this approach does not require the provision of an incentive. We are interested in engaging highly motivated users because they are more likely to create more direct answers, which benefits all users, and to be active community members in future embodiments. While we did compensate study participants, we expect that users of the tool will not require compensation, since it can reduce the difficulty and time required to perform the task. Finally, we consider this task to be non-competitive because, in its current embodiment, users are assumed to be working together to improve a shared search system; for example, employees of a movie rental company building a search engine for their customers. The studies in this chapter demonstrate that end users can complete a programming task of practical importance that can include crowdsourced components.

3.1 **Finding Answers to Factual Questions on the Internet**

Search engines on the World-Wide Web (such as Google or Bing) have become standard tools for completing information-seeking tasks. Although these search engines make it possible to gather many different types of information through a single interface, their very generality means that they do not always fit the specific needs of particular types of tasks.

Search engine queries typically return a list of documents. While this provides several avenues for the user to continue their search, this presentation is not optimal for some information needs. In particular, the results provided by a search engine can be a poor fit for simple information needs, such as *fact-finding* queries. As we will describe, Google, and other Internet search engines, have recently implemented features to address this problem. In fact-finding tasks, users need an answer to a simple factual question, such as “who won the women’s gold medal in hockey at the 2010 Olympics?” or “what is the average rainfall in the Sahara?”.

These questions have definitive factual answers, but current search engines do not provide simple answers to these simple questions. For example, a user interested in the question above might enter a query like “average rainfall Sahara” into a search engine – but instead of the answer to their question, they would receive a list of web pages. Unfortunately, a substantial amount of work is often required to obtain an answer from the list of documents that are returned from a search query. The user must navigate one or possibly several hyperlinks, wait for pages to load, skim through the pages to find relevant sections, and then read the content to find the answer to their question.

The time and effort needed to complete this process seems out of proportion with the simplicity of the information need. Each time a search interface forces a user to carry out several actions in order to find out a simple fact, it decreases productivity and increases frustration (“I just want to find out who won the gold medal in luge!”).

This problem – the amount of work needed to find the answer to a simple question – has been recognized before, and some tools do exist (or have been proposed) to provide faster and more direct answers to simple factual queries. Recently, online services that provide *direct answers* to search queries have appeared. Direct answer services, including Google's Search Features [44] and Wolfram Alpha (wolframalpha.com), provide answers based on keyword search. For example, on Google, the text "10 U.S. dollars = 7.421 Euros" appears above the document search results for the query "10 USD in Euros". The appearance of direct answer services suggest that there is a perceived need for this type of specialization. Direct answers can reduce the effort required to obtain factual information by immediately providing an answer to a keyword query.

Although each of these tools has had at least limited success, all of the current approaches suffer from one main problem: direct answers cannot be added by end users. We consider individuals who are motivated to add direct answers, and who don't necessarily have previous programming experience, to be our end users. End users are not able to contribute direct answers to existing search systems, nor are they able to create their own direct-answer search system. Google does not provide any method for direct answer contributions, and Wolfram Alpha approves only a limited number of users to help source and verify the data it uses. Both of these tools are slow to evolve and provide answers for only a limited set of domains. However, like Wikipedia editors, end users could add direct answers if given the right tools. Our research shows that there are many simple questions that direct answers can answer, that end users can make these contributions with a reasonable amount of effort, and that these contributions could significantly benefit existing direct answer systems.

In this chapter we describe a new approach, embodied in a system called WordMatch, that allows end users to add direct answer capabilities to search systems. WordMatch users create *answer patterns*, from which the system can produce natural language direct answers for many questions. Answer patterns can build upon previously created answer patterns, greatly extending the number of possible direct answers that can be created. Answer patterns are stored in a central database and are used to produce direct answers for keyword searches. Our approach makes it possible for trusted curators to add direct answers, for many domains, to search systems. Although WordMatch does not yet support contributions from untrusted volunteers, this work is also a step towards that goal.

We performed four evaluations of the WordMatch approach and system. We first demonstrate in a Wizard-of-Oz study that factual questions would be well served with a query interface that provides direct answers. Second, we show that many factual questions can be answered using answer patterns. Third, we show that everyday users can create answer patterns with very little effort. Finally, we demonstrate that the WordMatch system, with answer patterns created by study participants, outperforms standard Google search results for questions about movies. Participants did not participate in more than one study, and none of the studies used subsamples.

Our work makes three main contributions. We show that end users can easily create direct answer search interfaces. We also identify simple lexical pattern matching as an effective mechanism for generating natural language direct answers. Third, we show that the direct answers available in WordMatch can significantly reduce search time compared to standard interfaces. This approach has the potential to dramatically improve the support that search tools provide for the task of finding simple answers to simple questions.

3.2 **Related Work**

WordMatch relies on three main areas of previous work: research into Internet search behaviors, task interruption literature, and existing solutions for finding factual answers.

3.2.1 **Search Behavior**

A large proportion of Internet searches are conducted to find factual answers [18, 111]. Broder [18] conducted a user survey to determine the prevalence of different categories of search tasks. Survey results indicated that 39% of the searches were informational, meaning that the user's goal is to find information on a web page. This percentage was thought to be low due to ambiguity in the survey questions. A subsequent query log analysis found that 48% of the searches appeared to be informational.

Kellar et al. [61] examined search behavior during a week-long field study, and found that in 18.3% of the searches the user was looking for specific facts or pieces of information. The average time required for these searches was 8 minutes. In theory, direct answers could reduce search time to the time required to notice and read the direct answer on the first search results page.

Drori found that keywords in context, presented along with document title and other identifiers, reduce search time [54]. The control conditions in our first and fourth studies include similar identifiers, as implemented by Google.

3.2.2 **Task Interruption**

Searches are often conducted to find an answer that is needed as part of a primary task. For example, a journalist might need to look up a politician's term in office while writing an article. The immediacy of a direct answer limits the negative impact of task interruption. Corragio defines an *interruption* as an "externally-generated, randomly occurring, discrete event that breaks continuity of cognitive focus on a primary task" [27]. A *self-interruption* occurs when the event is internally initiated. Jin and Dabbish [59] identify an *inquiry self-interruption*, in which a user seeks additional information to inform their completion of a primary task. Jin suggests that users with inquiry self-interruptions may be frustrated or sidetracked if the desired information is difficult to locate. Long or complex search processes may require large switches in work context, making it more difficult to return to the primary task [22, 28, 92]. The studies of Speier *et al*. suggest that as primary tasks become complex, the ability to accurately complete interruptive tasks decreases. Also, these are perceived more negatively, suggesting that reductions in the time and cognitive requirements of interruptions is desirable [119].

3.2.3 **Finding Factual Answers on the Web**

Social Question-and-Answer Systems. Answers to factual questions can be found on social question-and-answer systems (Q&A systems), which use the shared expertise of willing participants to answer questions in a domain [24]. Ackerman's Answer Garden [1] allowed questions to be asked in a central database where they were answered, categorized and accessed by others in an organization. On the Web today there exist many different Q&A systems that allow a user to ask a larger community of answerers, such as Aardvark [54] (vark.com), ChaCha (chacha.com), and Yahoo! Answers (answer.yahoo.com). While Q&A systems have the benefit of providing a wider variety of answers and points of view [47], this is not necessarily a desirable trait when answering factual questions. We focus the rest of our discussion on Q&A systems' two main limitations: the time to receive an answer; and knowing if an answer will be found at all.

Response time on social answering systems tend to be long. Hsieh and Counts [55] found that the average time to receive any answer on Microsoft's Live QnA was 2 hours, 52 minutes, and that the time to get the answer that the questioner deemed as "best" was 4 hours and 18 minutes. A study of Yahoo! Answers by Zhang et al. [147] found that answers could take between 23 minutes (for questions from "low expertise" users) to about 9 hours (for questions from high-expertise users) to get a reply.

In addition to the issue of the time it takes to receive an answer on social answering systems, there is the question of whether or not the question will be answered at all. Dearman and Truong's literature survey [30] found that 5 – 53% of questions do not receive answers [2, 47, 55, 58, 106, 121], and even if an answer is given it doesn't necessarily mean the answer will be sufficient, as the quality of answers can be highly variable [47, 58].

3.2.3.1 Semantic Web Systems for Answering Questions

The Semantic Web provides a vision of disparate applications interoperating to make decisions appropriate to the current needs of a user [115]. As a step towards realizing this vision, some work has been done to create large knowledge bases of interlinked facts.

DBpedia and Freebase are systems that leverage structurally consistent data created in Wikipedia (and other publicly available sources) to allow for sophisticated queries of factual information [6, 10]. For example, in Wikipedia all articles about cities typically have a template. City templates are essentially a form that editors can fill out with common facts that relate to cities (e.g., the country where the city is found, and the population of the city). This allows queries such as “list all cities in China, with a population of at least 2 million people” to be issued. Freebase additionally allows users to sign up and become knowledge base editors. Anyone can sign up and begin entering assertions that describe a *topic* (e.g. a person, place or thing) that conform to the schemas that have already been created. Schemas define what *properties* the topic has and what data type they can take on (e.g. movie stars can have annual income, provided in thousands of dollars per year). Users can create their own schemas (e.g. a schema describing a movie star), but cannot edit schemas created by other users (a restriction to avoid breaking previously entered assertions on topics, based on a schema).

The main limitation to these semantic web based systems is that they remain a relatively heavyweight solution in terms of how contributors may add and specify facts and retrieve answers. Processes for entering facts are multi-step, requiring the user to learn existing schemas, specify new properties, and then enter appropriate facts. Issuing queries that can answer questions based on entered facts often involves using a complicated language similar to SQL (e.g. MQL for Freebase, or SPARQL for DBpedia). These do not approach the ease-of-use of keyword search approaches.

3.2.3.2 Direct Answers

Several Internet search engines include direct answers above the search result listings for some search queries. These direct answers can be sufficient to answer a user’s query. Google includes direct answers for unit conversions (“24 inches to cm”), census information (“Canada population”), and current weather information (“weather Palo Alto”), among others [44]. Google recently purchased Metaweb (metaweb.com), the company that created and maintains Freebase, to improve its direct answers [43].

Bing (bing.com) has sourced data for direct answers from both Metaweb and Wolfram Alpha. On Wolfram Alpha, a search for “distance to the moon” produces the current and average distances to the moon. Similar to Freebase, Wolfram Alpha sources some of its data from volunteers. While data is sometimes located or verified by volunteer curators, Google, Bing, and Wolfram Alpha rely on employees to create direct answers. None of these systems allow end users to add direct answers. This limitation restricts potential usage scenarios and the rate at which direct answers can be produced. End users might also decrease the cost of adding and maintaining direct answers.

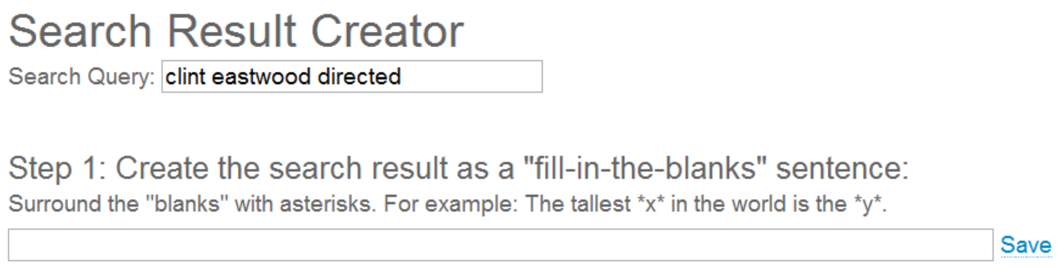
3.3 **WordMatch: A System for Direct Answers**

The WordMatch system allows end users to add and search for direct answers. WordMatch can be integrated with other search systems: for example, a WordMatch answer could be embedded at the top of a regular search result page.

Users add direct answers to WordMatch by uploading files containing facts, and then creating *answer patterns* for those facts; direct answers are produced from the answer patterns. Answer patterns are represented in a simple constraint-based language, and WordMatch provides a structured editor to guide users through the process of creating answer patterns.

3.3.1 **Structured Editor**

Users upload data files and create answer patterns with the structured editor. In this section, we introduce the structured editor by reproducing steps performed by one of our pilot participants, whom we will refer to as Melissa. The process starts with a search query. Melissa enters "clint eastwood directed" into the WordMatch search box, then presses the enter key to perform the search. The search results page indicates that no search results were found. There is, however, a link indicating that she can "click here to create more results for this search query". Clicking on this link brings her into the structured editor (See Figure 1).


Search Result Creator

Search Query: clint eastwood directed

Step 1: Create the search result as a "fill-in-the-blanks" sentence:

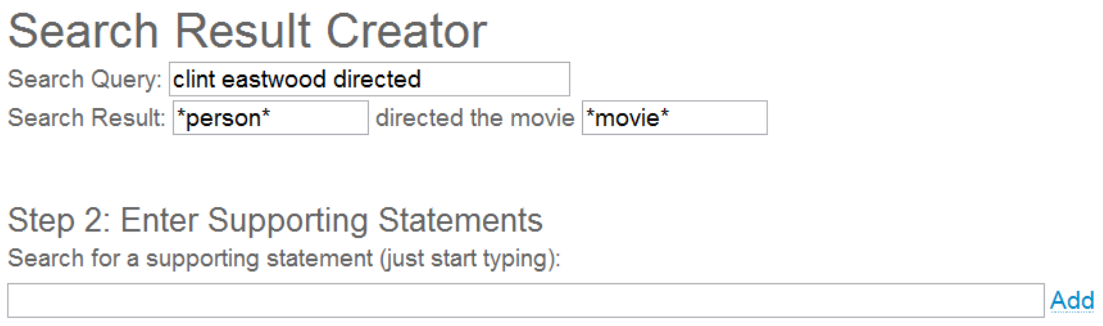
Surround the "blanks" with asterisks. For example: The tallest *x* in the world is the *y*.

Figure 1. Entering the structured editor, after a search with the query "clint eastwood directed".

3.3.1.1 Loading Data from an Excel Table

Melissa wants to add the direct answer "Clint Eastwood directed the movie Bronco Billy", and other direct answers that have the form "*person* directed the movie *movie*", where *person* and *movie* are variables, to WordMatch. She has already uploaded an Excel table named *directorimdb* to WordMatch. This table contains information about the directors of movies. It contains columns labeled *name* and *film*. She will use the editor to create an answer pattern that imports the data from these columns to produce direct answers.

Melissa's first step is to enter the format of the direct answers that she wants to add to WordMatch. She types "*person* directed the movie *movie*" into the empty search box in Figure 1, then clicks to save. She now sees the screenshot that appears in Figure 2 (note that the variable names are not meaningful, and she could have used any names instead of *person* and *movie*).


Search Result Creator

Search Query: clint eastwood directed

Search Result: *person* directed the movie *movie*

Step 2: Enter Supporting Statements

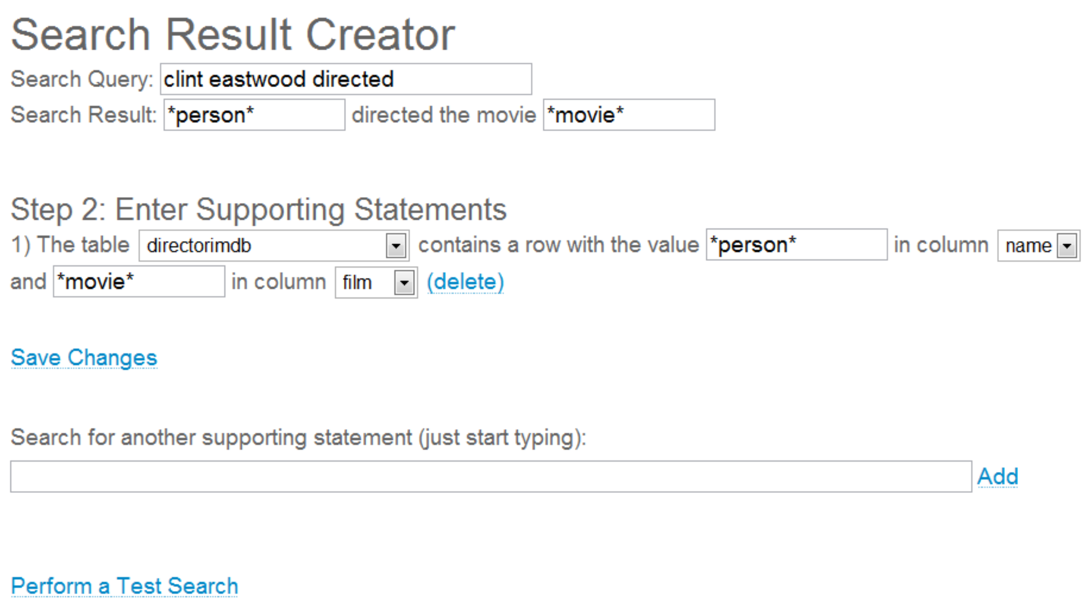
Search for a supporting statement (just start typing):

Figure 2. Editor before adding a supporting statement.

For the direct answers produced from this answer pattern to be accurate, the variables *person* and *movie* must have particular values. The structured editor includes predefined supporting statements for associating columns of data from a spreadsheet with the variables defined in the patterns. This allows Melissa to assign column *name* from the sheet *directorimdb* to the variable *person*, and the column *film* to the variable *movie*.

In Figure 3, Melissa has already selected the appropriate values from the drop boxes, and also edited the textboxes to use the variable names from the search result. This answer pattern is complete. Melissa can click the link to perform a test search, then click once more save the answer pattern.

Once saved, Melissa types "clint eastwood directed" in the WordMatch search box (not pictured in the figures) to verify that her direct answers have been added. A list of direct answers appears, including “Clint Eastwood directed Bronco Billy”. Since her Excel table also included information on other actors, she types in “depp directed”. Another list of direct answers appears, also produced from her answer pattern.


Search Result Creator

Search Query: clint eastwood directed

Search Result: *person* directed the movie *movie*

Step 2: Enter Supporting Statements

1) The table directorimdb contains a row with the value *person* in column name and *movie* in column film (delete)

Save Changes

Search for another supporting statement (just start typing):

Add

Perform a Test Search

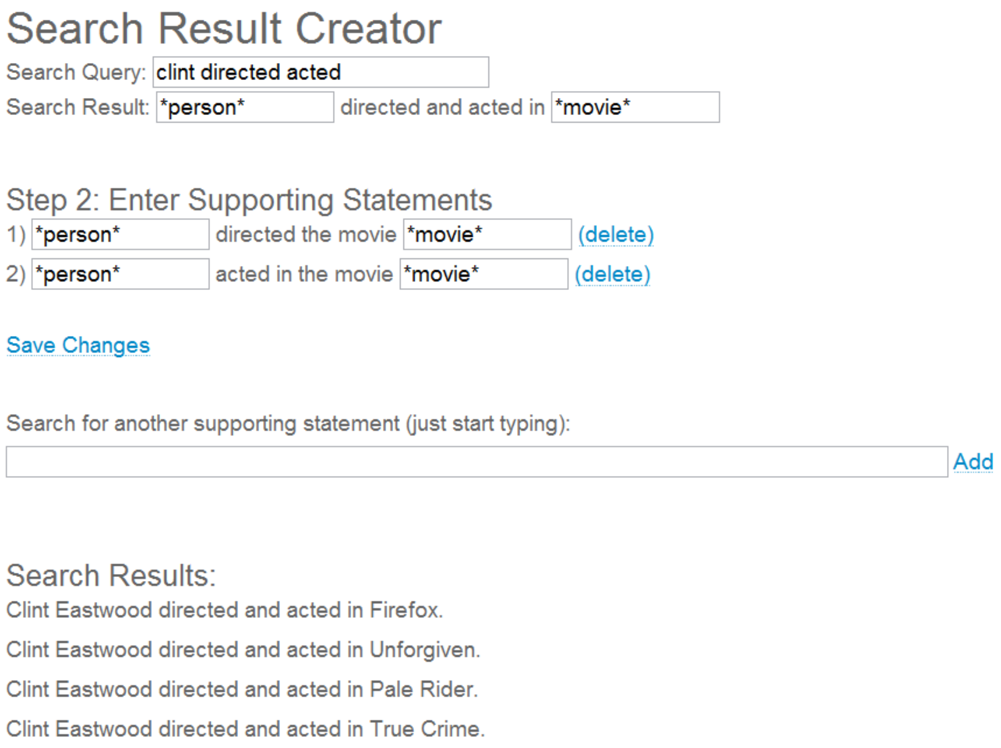
Figure 3. Editor after adding a supporting statement.

3.3.1.2 Using Answer Patterns as Supporting Statements

Melissa now wants to add direct answers that state which movies Clint Eastwood has both directed and acted in. These will have the form “*director* directed and acted in the movie *movie*” As we have seen, she has already created an answer pattern for “*person* directed the movie *movie*”. We will assume that she has also already created the answer pattern: “*person* acted in the movie *movie*” in the same way that she created the director pattern. To create this next answer pattern, Melissa will use both of these previously created answer patterns as supporting statements. As Melissa starts typing words that appear in these answer patterns (e.g. “directed” or "acted") into the *add supporting statements* text box, these answer patterns appear for her to select. Figure 4 shows the completed answer pattern. By allowing users to build upon previously created answer patterns, the set of patterns that can easily be created is considerably extended.

3.3.1.3 Additional Supporting Statements

In the previous example, Melissa created an answer pattern that listed one movie in each direct answer. Since direct answers are usually displayed one at a time, it can also be useful to list multiple movies in one direct answer. WordMatch has additional supporting statements for manipulating lists, math calculations, and other basic operations.



Search Result Creator

Search Query: clint directed acted

Search Result: *person* directed and acted in *movie*

Step 2: Enter Supporting Statements

1) *person* directed the movie *movie* (delete)

2) *person* acted in the movie *movie* (delete)

Save Changes

Search for another supporting statement (just start typing):

Add

Search Results:

Clint Eastwood directed and acted in Firefox.

Clint Eastwood directed and acted in Unforgiven.

Clint Eastwood directed and acted in Pale Rider.

Clint Eastwood directed and acted in True Crime.

Figure 4. Editor after testing the supporting statements.

3.3.2 **Searching for Direct Answers**

WordMatch currently features two search modes: *single answer* and *all answers*. When a search query is entered, WordMatch creates a list of all the direct answers that include each of the terms in the search query. This list of direct answers is then ordered by a relevance algorithm (we use simple ordering by sentence length, but more sophisticated schemes could easily be used). If the search mode is *single answer*, then only the first direct answer in the list is displayed; otherwise, all answers are presented.

3.3.3 **Implementation**

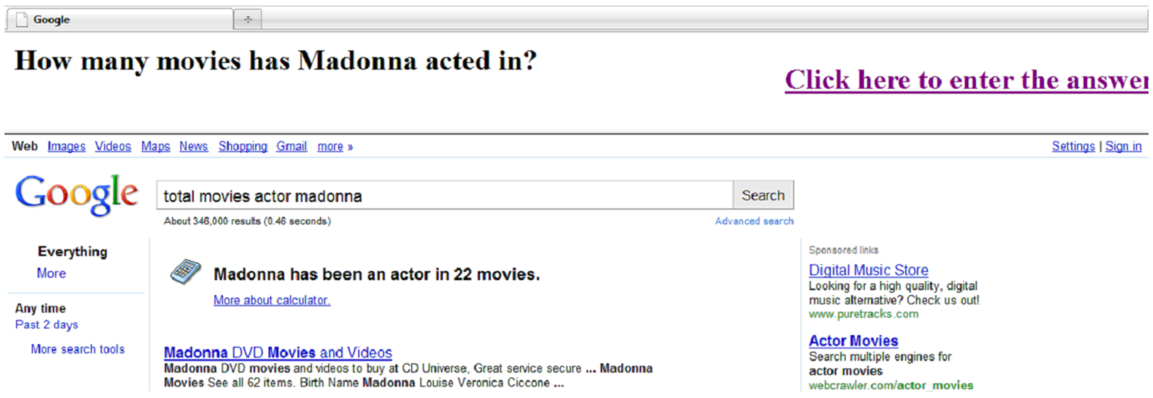
WordMatch was implemented using Apache Server, MySQL, PHP, and C++. A computer running Windows 7, with a 2.2 GHz processor and 4GB RAM, can generate search results in less than a second for any search query performed in our studies. The movie data used in our studies was extracted from Freebase, IMDB, and DBpedia, and formatted in spreadsheets.

3.4 **Study 1: Can Direct Answers Reduce Search Time?**

We carried out a small study to determine whether direct answers, embedded above Google search results, can decrease search time. This Wizard-of-Oz study measured time spent, and logged the actions performed by the user to find answers to a preset list of questions.

3.4.1 **Design and Procedure**

Participants were asked to find answers to provided questions on the Web, using the study interface. For each question (trial), the study interface initially displayed a Google search page (google.com). When an answer was found, the participant clicked the "Click here to enter the answer" link and entered the answer in a text field, completing the trial. Each participant received 16 questions; 8 questions in each of two conditions. In the first condition a direct answer would appear, embedded above Google search results (see Figure 5). This answer was preset to appear regardless of the query; mimicking the case where the system perfectly understood the information seeking need. The second condition was a control condition, where the experimental system did not provide a direct answer to the query; search results were returned as they normally would with Google. Questions were presented in a pseudo-random order, and were balanced over participants.


Google

How many movies has Madonna acted in?

Click here to enter the answer

Web Images Videos Maps News Shopping Gmail more » Settings | Sign in

total movies actor madonna — Search

About 346,000 results (0.46 seconds) — Advanced search

Everything / More / Any time / Past 2 days / More search tools

Madonna has been an actor in 22 movies.
More about calculator.

Madonna DVD Movies and Videos
Madonna DVD movies and videos to buy at CD Universe, Great service secure ... Madonna Movies See all 62 items. Birth Name Madonna Louise Veronica Ciccone ...

Sponsored links
Digital Music Store
Looking for a high quality, digital music alternative? Check us out!
www.puretracks.com

Actor Movies
Search multiple engines for actor movies
webcrawler.com/actor_movies

Figure 5. Experimental apparatus for the first study.

3.4.2 Apparatus and Participants

The experimental apparatus was a PHP script running on a remote web server. The script was responsible for displaying experiment specific pages, injecting direct answers into the Google search results, and recording participant actions.

The eight participants (3 female, 5 male) who participated in the study were graduate or undergraduate students, aged between 20 and 40 years. All reported using Google as their primary search engine.

3.4.3 **Results**

Extreme values were removed (values outside 3.0*IQR), 3 of the 64 trials in each condition. These values occurred in the direct answer condition when the participant did not notice a direct answer, and in the condition without direct answers when the participant had particular difficulty finding an answer. In one trial, the experimenter noted that the answer was clearly visible on one of the visited pages, but was not seen by the participant.

Direct answers dramatically reduced the amount of time needed for the search tasks. Completion time in the direct answer condition was more than six times lower (mean 9.2 sec., s.d. 3.8) than in the Google condition (mean 58.6 sec., s.d. 17.6). Not surprisingly, this difference was significant (Wilcoxon signed rank test, Z=-2.521, p<.01).

Direct answers also halved the number of pages that participants needed to inspect. When no direct answer was shown, participants needed 2.04 visits to a Google search results page before finding an answer (s.d. 0.48). This includes cases in which the user was able to find the answer by scanning the blurbs that Google provides below result links. In contrast, participants in the direct answer condition needed only 1 visit to find the answer. A Wilocoxon signed rank test showed direct answer required significantly fewer visits to the results page (Z=-2.521, p<.05).

3.4.4 **Interpretation**

This simple study demonstrates strong potential for time and effort savings when direct answers are included with search results. On average, participants saved 50 seconds when a direct answer was shown, and required half as many visits to a search results page. In a few cases, users needed to spend much longer to find a direct answer and visited search results many more times. Given that a large portion of searches are performed to find specific facts [61], it seems appropriate that effort be spent extending the existing solutions for directly answering search queries.

3.5 **Study 2: Do People Ask the Kinds of Questions That WordMatch Can Answer?**

3.5.1 **Design and Procedure**

To obtain a corpus of questions for use in subsequent studies, we sent an email to a graduate student mailing list, simply requesting "ten questions about movies", without further directions. We selected this topic domain because we wanted a familiar, limited domain for which participants could easily come up with ten questions. As a secondary goal, we were also interested to informally confirm that the submitted questions could be answered by answer patterns created with WordMatch. We classified answer patterns in the following categories: "simple", if the answer requires four or fewer supporting statements, and "complex" otherwise. We chose this method because we expected that many end users can create answer patterns with a small number of supporting statements in under five minutes. Of the 9 respondents, one was female. All respondents were university students, aged 20 to 27.

3.5.2 **Answer Pattern Categorization**

Questions were informally categorized to obtain a rough approximation of the percentage that could be answered by simple answer patterns. For each question, we considered whether it could likely be answered by an answer pattern with four or fewer supporting statements. We allowed for these four supporting statements to be spread across multiple answer patterns, if necessary (e.g. if, as in the list creation example, another answer pattern would need to be created first). For example, an answer pattern that could produce a direct answer for one of the questions received, "Of which brand is the leather jacket of Neo (Matrix)?", would likely require more than four supporting statements; unless it relied on another existing, highly similar, answer pattern. When considering whether a specific supporting statement would be used in an answer pattern, we asked ourselves if another, more general supporting statement would be more likely to exist.

Although this analysis was highly subjective, in almost all cases, we found the categorization was obvious. Most questions were either very simple ("Who is the lead female actress in Marie Antoinette?") or complex (like Neo's jacket). We excluded an open-ended question and subjective questions from the categorization. Overall, we found that direct answers for 68 of the 81 questions, or approximately 75% of the total questions submitted, could be produced from simple answer patterns. If answer patterns were created for these 68 questions, they could also produce many other direct answers, answering thousands of questions.

While limited by a small set of participants and a specific domain, these findings motivate this work, providing some limited, informal confirmation that WordMatch direct answers might answer a large number of factual questions. This is not surprising. The categorization served primarily to confirm prior findings. According to Kellar [61] and Broder [18], factual questions constitute a substantial portion of all Web searches. It is possible that direct answers might answer a significant fraction of total Internet searches.

3.6 **Study 3: Can Users Create Answer Patterns?**

3.6.1 **Design and Procedure**

To better understand how end users add direct answers to WordMatch, we conducted a study in which participants created answer patterns. Answer patterns were created for 7 questions: "When was the movie Groundhog Day released?", "Who won the Oscar for Best Actor in 2010?", "Is Battlefield Earth a Space Opera?", "Did Clint Eastwood star in the movie Bronco Billy?", "Who is directing Scream 4?", "What upcoming movies is Wes Craven working on?", and "How many westerns has Clint Eastwood starred in?". The questions were selected from the questions submitted by respondents in Study 2. We chose questions that would require participants to create answer patterns of varying difficulty. The questions were given to participants in an order of increasing difficulty. We grouped the answer patterns into three levels of difficulty. A *basic* answer pattern does not require comparisons to be made between sentences (see Figure 3). An *intermediate* answer pattern includes at least one comparison between its sentences (see Figure 4). *Advanced* answer patterns require the creation of a list. The seven questions required the participants to create three basic, two intermediate, and two advanced patterns. After a participant completed all the patterns at a difficulty level, a questionnaire was given.

Participants were provided with an Excel document containing six worksheets. The worksheets contained tables with the movie information necessary to complete the study. For example, the worksheet named *bestmovies* contained information on films that have won the Academy Award for Best Picture, and included the columns: *filmname*, *filmid*, *producer*, and *year*. Participants were required to use these tables to create their answer patterns. During training, participants uploaded the Excel document to WordMatch. WordMatch automatically identifies the document format and imports the table data.

Participants completed a training session, during which they followed written instructions to create four answer patterns. They then created four additional answer patterns without instruction. Instructions were provided in a printed document; the experimenter answered any questions.

For each participant, we recorded each attempt to create each answer pattern, the time required for each attempt, and whether the participant successfully created the answer pattern. The system logged an attempt every time the participant clicked on the "Test Search" link (see Figure 3).

3.6.1.1 Apparatus and Participants

Participants accessed WordMatch remotely through a web browser (Google Chrome). Ten participants (3 female), aged 19 to 26, participated in this study. Six participants indicated that they were familiar with spreadsheets, including spreadsheet formulas. The four remaining participants indicated that they had little or no familiarity with spreadsheets. Each participant indicated that, aside from spreadsheets, they had no programming experience. On average, participants spent about 37 minutes training.

3.6.2 **Results**

The average number of attempts (including the successful attempt) was less than 3 for each of the 7 answer patterns (see Figure 6 for a breakdown by rule difficulty). We observed that users spent most of their time thinking. On average, users were able to fix all of their mistakes, after a failed attempt, with a single further attempt.

All the errors that occurred during the completion of the basic and intermediate answer patterns were logic errors. Examples of logic errors include failing to match the variable names and failing to include all the necessary supporting statements.

	Time (minutes)	S.E.	Attempts
Basic tasks	2.2	0.4	1.8
Intermediate tasks	3.5	0.8	2.2
Advanced tasks	3.8	0.9	2.3

Figure 6. Average time and attempts required for task completion.

For the advanced answer patterns, average completion time was 3.8 minutes (s.d. 0.9 minutes). The creation of these answer patterns involved copying, then modifying, a previously created sentence in a textbox. Semantic and syntax errors only occurred for the final two (advanced) answer patterns, which required lists to be made. In the 70 tasks completed, there were only 2 semantic errors. Both errors involved adding supporting statements to access a table instead of creating a list. 6 syntax errors occurred, 3 for each of the two advanced answer patterns. These involved incorrectly written supporting statements. For example, a participant incorrectly recalled the format of an answer pattern he had already created, typing "*actor* acted in the western *wildcard*" instead of "*actor* starred in the western *wildcard*".

One participant was unable to complete the advanced answer patterns. This person's actions demonstrated that they did not understand how to create lists. This participant's completion times for the basic and intermediate patterns were similar to other participants.

3.6.3 **Interpretation**

These results support our hypothesis that end users will be able to add direct answers. Participants successfully created 68 of the 70 answer patterns. Answer pattern creation times are reasonable for many usage scenarios. With minimal training, an end user creating content for a movie rental website could create twelve answer patterns in an hour, potentially adding thousands of direct answers in the process. Also, observations that participants spent most of their time thinking suggest that more practice might further reduce completion times.

Although training required a non-negligible amount of time, spending less than an hour to train an employee or volunteer curator to add direct answers is reasonable. Training time could also have been decreased. During training, answer patterns were created quickly. The average completion time during training was 4.6 minutes. This includes time spent reading instructions and asking questions. Finally, about half of the training time was spent learning to create advanced answer patterns; training which often might not be required.

This version of the interface has no error reporting, which, if included, might further reduce completion times. Error reporting could have highlighted many of the syntax and logic errors, and guided users through the process of correcting many of their mistakes. For example, most of the logic errors involved forgetting to use the same variable name in different sentences. The system could highlight variables that are only referenced once, and provide auto-completion to guide users towards selecting appropriate variables. The interface for list creation is a weakness of the current design. All of the errors that occurred when creating lists were due to mistakes in the writing of answer patterns that were already in the system. Requiring participants to use an auto-completion interface, similar to the interface for adding supporting statements, could prevent these errors.

3.7 **Study 4: Does the WordMatch System Reduce Search Time?**

3.7.1 **Design and Procedure**

In this final study, we evaluate WordMatch with a search task. Study 1 used a Wizard-of-Oz method to simulate a system that embedded direct answers into search results. In this study, we replicated Study 1, but replaced the simulated answer system with a working WordMatch system.

Participants were asked to use Google to search for answers to 7 questions. We used the 7 questions from Study 3. To add the direct answers for these questions to WordMatch, we used answer patterns created by our participants in Study 3. For each question, participants in this study were asked to "find an answer you think is likely to be correct". No other directions were provided.

As in the first study, participants formulated their own search queries. Also as in the first study, there were two conditions: direct answers and no direct answers. The condition varied between questions (trials), according to a balanced, random ordering. Whether a direct answer could be shown for a question was varied by question, within participants, and balanced between participants. Each participant answered each of the 7 questions.

When a direct answer was to be shown, the experimental apparatus queried a WordMatch API with the participant's Google search query. If WordMatch did not return a direct answer, none was shown. After each answer was found, we asked the participant to complete a brief questionnaire.

3.7.1.1 Participants

Ten participants (2 female), aged 20 to 31, participated in this study.

3.7.2 **Results**

Showing a direct answer decreased the average time to answer the question, for each question. On average, direct answers reduced the time to locate the answer by 43 seconds (314% faster). Trials in each condition were averaged over each participant. Completion time in the direct answer condition was lower (mean=13.7 seconds, s.d.=5.4, mdn=14) than the no direct answer condition (mean=56.8 seconds, s.d.=65.8, mdn=27). This difference was found to be significant using a Wilcoxon signed rank test (Z=-2.366, p<.05). On average, participants conducted 1 search on Google when a direct answer was shown (mean=1.26, s.d.=0.44), and 2 searches when a direct answer was not shown (mean=1.71, s.d.=1.03). A Wilocoxon signed rank test showed that the direct answer condition required significantly fewer visits to the results page (Z=-0.707, p<.05).

When a direct answer was not shown, all participants followed at least two search result links, and 5 participants conducted three or more Google searches, before finding answers to the questions "How many westerns has Clint Eastwood starred in?" and "What upcoming movies is Wes Craven working on?".

One participant did not trust the direct answers, but for three of the questions, after his initial search queries, he reformulated his search queries to include the information in the direct answers. That is, rather than ignoring these direct answers, he incorporated them into his search queries to verify their accuracy. During debriefing after completing the study, this participant, and most other participants, indicated that they thought the direct answers were included by Google. This participant indicated that he was uncertain that he could trust a sentence without a source reference that appeared on Google.

Questionnaire data revealed that two participants failed to notice the direct answer on two occasions, but each noticed it before completing the task. A direct answer was displayed, and provided a correct answer to the trial question, for 84% (37/44) of the search queries in the direct answers condition. Direct answers were not shown 3 times due to misspelled search terms; in each case the user then corrected the spelling (either manually or by clicking on a spelling correction link provided by Google), and the direct answer appeared. A direct answer that did not answer the question was shown for 4 search queries. Each of these 4 queries was ambiguous. For example, to find an answer to the question "When was the movie Groundhog Day released?", one participant entered the search query "hog released", and WordMatch returned the direct answer "The Hedgehog was released in 2009".

3.7.3 **Interpretation**

The results support our hypothesis that WordMatch can significantly reduce the time and effort required to find answers to questions. For search systems that are known to retrieve results from untrusted data sources, user confidence in the accuracy of direct answers should be addressed. This could be done in a number of ways, for example, including icons that indicate a degree of confidence in the direct answer, based on the reviews of the answer pattern and/or the creator of the answer pattern.

3.8 **Discussion**

There are several main findings from our three evaluations of WordMatch. In Study 1, we found that showing a direct answer above document search results can dramatically reduce the time and effort required to find the answer to a question. In Study 2, we found evidence to support our claim that direct answers, and more specifically answer patterns, can be used to answer a large number of the questions that people currently answer using existing Internet search engines. In our third study, we demonstrated that our structured editor allows end users to create answer patterns quickly. In our final study, we showed that direct answers, produced from answer patterns created by end users, can significantly decrease the time required to answer questions with Google.

3.8.1 **Deployment Considerations**

WordMatch is reliant on end users who are willing to create answer patterns. Small businesses with websites and custom search systems, and large businesses with internal search systems, are potential users that can benefit from WordMatch with only one or two individuals creating answer patterns. While there is also evidence to suggest that volunteers would be willing to create answer patterns for large-scale, general-purpose search systems, WordMatch has not yet been publicly deployed.

Large-scale, general-purpose search systems, introduce three challenges that are beyond the scope of this work: (1) establishing user confidence in the accuracy of direct answers, (2) determining when it is appropriate to display a direct answer, and (3) deciding between multiple direct answers that might be displayed. We believe that our current implementation of the WordMatch system would be suitable for many niche search systems that have a relatively small number of answer patterns and individuals creating answer patterns (e.g., the many public and private search systems hosted on company websites).

Direct answers are ideally factual answers and can be produced from an answer pattern that also produces many other direct answers. Although answer patterns can be used to dynamically generate direct answers, as underlying data changes, end users might still be required to update or maintain the fact base.

We have shown that WordMatch direct answers can be effective for a common knowledge domain (movies), for which there are many online data sources. We believe that WordMatch would be even more useful for esoteric domains, which have fewer search results on Internet search engines. We also expect that the potential benefit of this technology will increase as the current trend to make structured data available online continues.

3.8.2 **Lessons for Designers of Future Direct Answer Systems**

The key lessons from designing and evaluating WordMatch are as follows:

1. *Direct answers can decrease the search time required to answer simple, factual questions.* Direct answers can also compliment document search results.

2. *There are many factual questions that can be answered by answer patterns.* Since a single answer pattern can also answer many questions, there appears to be an opportunity to use answer patterns to add many useful direct answers.

3. *End users can quickly learn a constraint-based language for simple, lexical pattern matching.* This finding may be applicable beyond direct answers, enabling users to easily complete tasks involving natural language sentences.

4. *Large-scale, general-purpose search systems should include methods for users to verify the accuracy of a direct answer*. However, even if a user lacks confidence in a direct answer, they may still benefit from its appearance, performing additional searches to verify its accuracy.

3.9 **Summary**

We have demonstrated that people with no programming experience can use the WordMatch system, and that WordMatch direct answers can decrease the time and effort required to find answers on Google. Unlike prior closed, centralized direct answer systems, WordMatch allows end users to add direct answers. With WordMatch, end users create simple natural language patterns that produce direct answers to keyword queries. A large number of Google search queries can be answered with a single sentence, rather than a lengthy list of possible answer sources.

Chapter 4

SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization

This chapter describes the design, implementation, and evaluation of SnipMatch, a search interface for finding and integrating code snippets from within the development environment. SnipMatch builds upon WordMatch in several ways. In addition to SnipMatch being derivative in design, the WordMatch evaluations demonstrated the efficacy of the approach, leading to an investigation that resulted in the production of this tool. In particular, studying WordMatch we observed that (1) search queries with parameters can be easily understood and manipulated by novice programmers, (2) search interfaces can enhance the process of writing source code by reducing syntax requirements and proposing alternatives, and (3) there are effective mechanisms for ranking and filtering user-generated results that include parameters. SnipMatch is also a direct motivation, non-competitive CHC task. We expect that the non-competitive nature of this task encourages snippet sharing. The use of direct motivation simplified our evaluation; our costs were minimized and not compensating our longitudinal study participants also strengthened the evidence of commitment to use the tool. This chapter begins with an introduction to the problem and a rationale for the development of a search interface to enhance general programming tasks.

4.1 **Locating and Integrating Code Snippets**

Programmers routinely search online for snippets to integrate into their source code [14, 16, 52, 81, 120]. This find-and-integrate behavior reduces the time to leverage Application Programming

Interfaces (APIs), libraries, and algorithm implementations [16, 145]. Even when an API is well understood, snippets provide productivity gains: the time and effort required is often less than writing the code from scratch [14].

Despite the benefits of snippet use, substantial time can still be required to locate and integrate code snippets. Programmers often must locate and combine multiple code snippets, rename or typecast variables, and manually locate and include dependencies [16, 35, 79]. After this complex integration has been performed, a code snippet may be discarded if it contains errors or otherwise does not work as expected [15]. Even a small decrease in the time required to perform this common task — finding and integrating snippets — could cause a qualitative change in behavior [45].

Existing code snippet solutions fall into one of two broad categories: those that leverage a carefully curated set of purpose-built snippets [38, 100], and those that either synthesize new snippets or mine relevant snippets from large repositories [14, 16, 82]. There is a simple trade-off associated with choosing one of these two approaches: curated collections yield higher-quality snippets, but are less complete. Mined or synthesized snippets can cover a much broader set of use cases, but may sometimes yield irrelevant, difficult to understand, or incorrect code.

This research is driven by two insights:

First, given metadata about integrating a snippet into existing source code, we can build much more powerful tools for snippet retrieval and adaptation. In particular, we can use information provided in the search query, such as local variable names, to semi-automatically tailor snippets and facilitate further adaptation. We can also leverage local code context (e.g. variable types, imported libraries) to rank and filter results in relation to compatibility with the existing code. Unfortunately, gathering rich metadata about a snippet is difficult because users are usually focused on saving time when they are interacting with snippets. When a person is *creating* a snippet, however, she has already decided to invest time now in order to save time later.

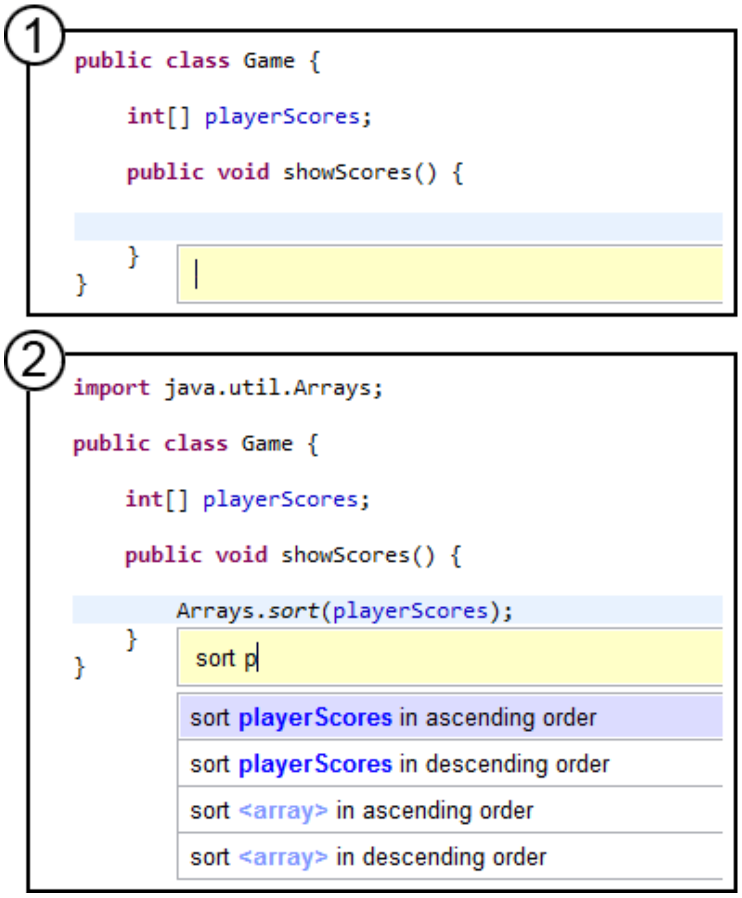
Second, when a person creates a code snippet from scratch, she is willing to spend a reasonable amount of effort to make that code snippet configurable and easy to integrate. While creating a snippet, the author might add metadata using a snippet markup language. This is a viable method to obtain search and integration metadata: for example, variables to be renamed and preconditions to verify compatibility with the user's existing code. To explore these insights, we built and evaluated *SnipMatch*, a search interface for finding and integrating curated code snippets.

This work offers three contributions:

1. A search algorithm for curated code snippets that leverages code context — Curated snippets are ranked, filtered, and customized based on the code in the development environment. SnipMatch builds on prior work in Integrated Development Environment (IDE) search [14, 37, 37, 73, 91]. The SnipMatch search algorithm extends the use of code context beyond the current programming language and framework to enhance the ranking of shared, curated code snippets. Our search algorithm uses the following features of the programmer's source code to rank and filter prospective snippet results: variable types and names, the cursor position within the abstract syntax tree, program logic, and code dependencies. We selected these features because they can be used to determine how closely results match the existing code. Results that more closely match the features of the existing code – that is, make use of existing variables and require fewer modifications – are ranked higher.

2. A lightweight markup for specifying integration instructions for code snippets — Other tools suggest error-correction source code modifications [35, 37, 49]: for example, prompting the user to rename variables after a snippet has been inserted. These tools require human intervention because the intended use for a snippet can be ambiguous. Did the programmer intend to use the code snippet "as is", or only part of it? Should the snippet, or the existing code, be modified when there is an error? An integration tool requires more information to answer these questions.

3. Insights about how code snippet search tools are used, derived from the implementation and evaluation of SnipMatch. We implemented SnipMatch, a snippet search plug-in for the Eclipse IDE. To better understand how snippet search tools change the way people program, we conducted a comparative laboratory study with 16 participants and a public deployment with 93 programmers. We observed that SnipMatch was used to reduce context switching and as a memory aid. Participants reported that including snippet arguments in the search box was particularly effective for the two most common usage scenarios: shortcuts and quick reference.



```
public class Game {

    int[] playerScores;

    public void showScores() {

    }
}
```

```
import java.util.Arrays;

public class Game {

    int[] playerScores;

    public void showScores() {
        Arrays.sort(playerScores);
    }
}
```

sort p

- sort playerScores in ascending order
- sort playerScores in descending order
- sort <array> in ascending order
- sort <array> in descending order

Figure 7. The SnipMatch plug-in for the Eclipse development environment. A keyboard shortcut opens the search window (1) at the programmer's cursor position. Search results (2) are updated as the query is typed. The search query also affects integration: the local variable playerScores is included in the snippet (2). Snippet integration is previewed within the existing code.

4.2 **Related Work**

4.2.1 **Search Interfaces**

Search interfaces [51, 93] can be tailored for specific tasks. Prior work includes systems to support data analysis [17, 88], web page revisitation [3, 122], and programming [14, 81].

Programming search interfaces can enhance web search results [52, 81] and, most relevant to our work, locate code snippets [14, 112]. Locating snippets can be time-consuming: in one study, 19% of programming time was spent looking for source code on the Internet [16].

Snippet search interfaces query code repositories to find *keyword* [14, 38, 42] and *structural* [37, 53, 82, 112, 126] matches. Structural search can include building an internal representation of the program, while keyword matching is limited to text analysis. Google code search [42] locates occurrences of keywords in source code files. Uncommon, domain-specific search keywords can improve the relevance of results from this large repository. Blueprint [14] is a development environment plug-in that augments Internet keyword search queries with the language and framework used in the development environment. Snippets are automatically extracted from web pages and can be browsed within the plug-in. While keyword matching can be ineffective for locating many general code structure or program logic patterns, it is easy to use and can match comments, variables, dependencies, and other lexical features.

Structural snippet search interfaces perform static analysis to determine how code functions. For example, structural search interfaces can locate snippets that create an object of a certain type from other objects. Finding a Method Invocation Sequence (MIS) is a common API search task [112, 126, 53, 142]. PARSEWeb [126] ranks code fragments to be incorporated into MISs by frequency of use and length. S^6 [109] allows programmers to include simple test cases and contracts with keyword searches. However, it can be difficult to ensure that snippets do not contain errors [82] and to differentiate between similar structural features: for example, frameworks and class libraries [37]. Further, Jungloid [82] and most other structural search interfaces are less effective when the programmer does not know the names of relevant classes, methods, or types.

SnipMatch is designed to provide more precise results and result rankings than these alternatives, but at a cost: someone must associate additional data with a code snippet for its search ranking and automatic integration to be improved. Snippet creators can also specify alternate search result wording to make snippets easier to find, mitigating the vocabulary problem [40]. SnipMatch allows programmers to enhance the accessibility of snippets that are important to them.

4.2.2 **Snippet Integration**

Many different types of modifications can be required to integrate a code snippet. EUKLAS [35] and Eclipse Quick Fix [37] highlight source code errors and suggest corrections. HelpMeOut [49] presents suggestions for correcting compiler and runtime errors. These tools can be useful for fixing snippet integration mistakes. Rather than fixing errors, SnipMatch attempts to prevent them from occurring.

The Codelets system [100] allows authors of example code to use a simple markup language to indicate which parts of an example are fixed and which parts the user should edit. Editable regions can be given names that are then referenced in custom user interfaces for configuring the example code. In SnipMatch, we extend this idea of providing users with a lightweight markup language for annotating snippets. In particular, we add syntax for expressing type information, integration instructions, and external dependencies. Additionally, we leverage this semantic information to make our code search more powerful.

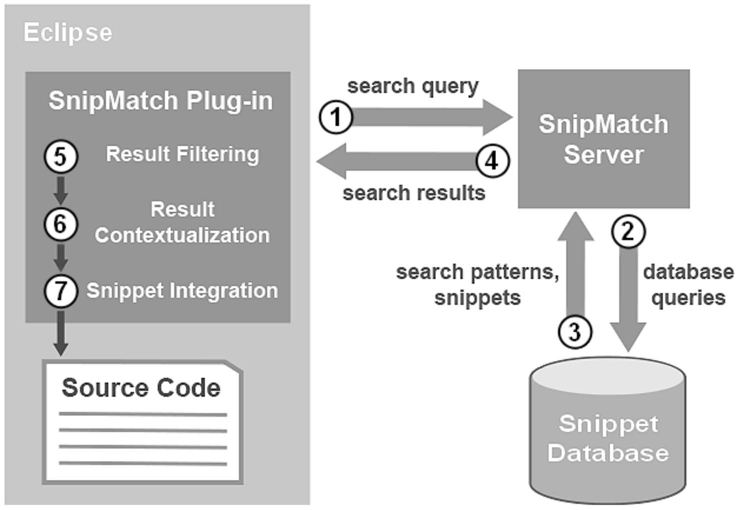


Figure 8. SnipMatch snippet search overview

Blueprint extracted over 100,000 code examples from blogs, forums, and human-created documentation available on the Internet [13]. In all of these cases, people expressly curated and published these snippets with the intention of sharing. SnipMatch is also dependent on programmers' willingness to curate and share snippets.

4.2.3 **Sloppy Interpretation**

Sloppy interpreters have lenient syntax requirements [73, 74, 77, 91]. Similar to Inky [91], SnipMatch matches search queries with pre-defined snippets. This prevents semantic errors while preserving the primary benefits of sloppy interpretation: minimal wording and syntax requirements.

4.3 **Enhancing Snippet Search With Code Context**

Programmers can use SnipMatch to find and integrate Java snippets from within the development environment. Pressing Ctrl-Enter while editing source code opens the SnipMatch search interface.

The top search result is previewed inline as the programmer types in the search box. Figure 7 shows the result of typing “sort p” in the search box. The array playerScores is included in the top search result because this snippet has a *search pattern* with a parameter.

The SnipMatch search algorithm matches search queries to search patterns. A search pattern is a text description of the effect of the code snippet, interspersed with placeholders for snippet parameters. For example, this is the text description for the search pattern shown in Figure 7: “sort <array> in ascending order”. “<array>” refers to a snippet parameter. To prevent type mismatches, the parameter type can also be specified in the search pattern. Snippets can have multiple search patterns.

When a snippet is submitted to SnipMatch, a lightweight markup language can be used to specify where search pattern parameters will appear in the snippet source code. This markup can also be used to import dependencies as required and define preconditions for snippets to appear in the search results. After results are returned from the central server, the Eclipse plug-in filters and customizes the snippets based on the markup and the code context in the development environment (Figure 8).

SnipMatch snippets can be private or public. Public snippets are available to all users, while private snippets appear only in the snippet creator’s search results. To ensure correctness and maintain a consistent level of snippet quality, system moderators review snippets that are added to the public repository.

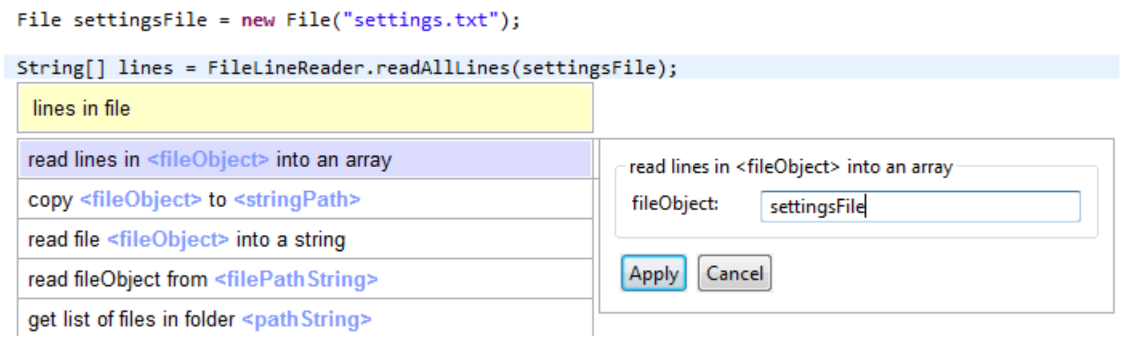
4.3.1 **Design Rationale**

We are interested in reducing the time and effort required to both *locate* and *integrate* code snippets. As described in the introduction, snippet integration can be complicated and time-consuming. Previous approaches for supporting snippet integration include methods that prompt the user with options [35, 37, 49] and methods that generate snippets from keywords based on rules [73, 74]. Although helpful to highlight problems that require attention, user prompts still require code comprehension, decision-making, and manual input prior to testing. Generative methods frequently create semantic errors.

One viable alternative that does not require manual input at the time of integration is for programmers to annotate snippets *when they are created* with additional machine-readable markup language. This information can then be used by the search interface to provide enhanced search ranking and snippet integration. Programmers already create Eclipse Templates for personal use, and share and comment on code snippets online [44, 81]. We believe that programmers will similarly contribute search patterns and integration markups if they will improve snippet usability. As with Wikipedia, relatively few individuals need to contribute for the system to be useful for many.

4.4 **Implementation**

The SnipMatch plug-in was written in Java using the Eclipse Plug-in Development Environment (PDE). The plug-in communicates with the server through HTTP requests. Server-side, the search algorithm is written in C++ and PHP, and snippets are stored in a MySQL database. During the evaluation, our server was located in the United States.



```
File settingsFile = new File("settings.txt");
String[] lines = FileLineReader.readAllLines(settingsFile);
```

lines in file

read lines in <fileObject> into an array
copy <fileObject> to <stringPath>
read file <fileObject> into a string
read fileObject from <filePathString>
get list of files in folder <pathString>

read lines in <fileObject> into an array

fileObject: settingsFile

Figure 9. The argument editor (right window) is displayed if the user presses Enter before all arguments have been specified. This structured view prompts the user with text fields for each search pattern parameter. The user's code is modified as the text field is completed.

4.4.1 **Snippet Search**

Search Window. The search window (Figure 7) has two main components: a static text field for the search query input, and a dynamically resizing region below for search results. The ordered list of search results is updated as the contents of the text field are modified. This provides the user with instant feedback, facilitating snippet discovery and step-wise refinement of search queries. To increase readability, search results have basic syntax highlighting; different font colors are used for keywords, arguments, and placeholders for missing arguments.

Argument Editing. If a user attempts to insert a snippet with missing arguments, the argument editor dialog appears to the right of the search window (Figure 9). The dialog presents a structured view of the snippet parameters and includes text fields for entering the arguments. Changes to the arguments are immediately reflected in the source code. The editor can also be manually invoked for any highlighted result by pressing Ctrl-Enter.

Tab Completion. Since search results are updated as the search query is typed, the user can refine the search query based on the results. For example, the search query "read" might return some results that begin with "read lines from file", and other results that begin with "read character from keyboard". The user can narrow the results to only include those pertaining to file operations by changing the search query. Tab completion facilitates this process. When the user presses Tab, the search query text is autocompleted up to the next parameter in the currently highlighted result. For example, if the search query is "read", and the highlighted result is "read lines from file <filePath>", pressing Tab will change the search query to "read lines from file ".

Snippet Icons. A small group of icons is docked in the lower right corner of the currently highlighted result (Figure 11, page 108). Most of these icons are buttons that allow the user to send feedback about the result. Feedback options include rating, flagging, and commenting. All user feedback is logged for manual analysis. A yellow warning icon appears when the snippet includes changes to the source code that are omitted from the preview, such as helper classes. Resting the mouse over this icon reveals a summary of all the hidden changes.

4.4.2 **Snippet Submission**

The SnipMatch Eclipse plug-in includes an interface for adding and editing code snippets. This interface allows users to modify search patterns, and includes features to facilitate the addition of integration markups to the snippet code. SnipMatch uses the Java type hierarchy to recognize standard and user-created parameter types that appear in search patterns.

4.4.3 **Search Results**

There are three different types of search results: *in-order, unordered, unsigned*. After search results are found, they are ranked, filtered, and shown to the user in a single list.

4.4.3.1 *In-order.*

An in-order result is a snippet with a search pattern that begins with exactly the same sequence of characters as the search query. For example, the result "read from file <filePath>" is an in-order result for the search query "read from".

4.4.3.2 *Unordered.*

A result is unordered if the search pattern does not begin with exactly the same sequence of characters as the search query. For example, the result "read from file <filePath>" is an unordered result for the search query "file".

4.4.3.3 *Unsigned.*

An unsigned result is a snippet that does not yet have a search pattern, but whose code contains one or more tokens from the search query. These snippets can be retrieved not only from the SnipMatch snippet database, but also from external snippet repositories.

4.4.4 **Result Ranking**

Results are first ranked by type. In-order results appear first, followed by unordered. Unsigned results appear after all other results. In addition to being ranked by type, each result type uses a different set of ranking criteria.

4.4.4.1 *In-order.*

In-order results are first ranked by the number of search query tokens matching each search pattern. Next, results are ranked by the number of missing arguments. Results with fewer missing arguments are ranked higher.

4.4.4.2 *Unordered.*

Unordered results are ranked based on term frequency–inverse document frequency (tf-idf) [60]. Tokens from the search query are matched against search patterns. Search patterns that contain more query tokens or query tokens that appear less frequently in other search patterns are ranked higher. User feedback (e.g. votes and frequency of selection) will be incorporated in the future.

4.4.4.3 *Unsigned.*

The same ranking criteria as for unordered results are applied, except that the search query is matched against the snippet code instead of the search patterns.

4.4.5 **Result Filtering and Contextualization**

Since the SnipMatch server has limited information about the user's code, the client modifies the returned search results before they are shown to the user. This is completed in two stages: result filtering and result contextualization.

4.4.5.1 *Filtering.*

Since the SnipMatch server does not have access to the Java type hierarchy, some search results may contain incompatible types. The client performs a type compatibility test on each search

```
${import(java.io.File)}
${import(java.io.BufferedReader)}
${import(java.io.FileReader)}
${import(java.io.IOException)}
${import(java.util.ArrayList)}

${helper}
class FileLineReader {
   public static String[] readAllLines(File file) {
      ArrayList<String> lines = new ArrayList<String>();
      try {
         BufferedReader br = new BufferedReader(new FileReader(file));
         String line = null;

         while ((line = br.readLine()) != null) {
            lines.add(line);
         }
         br.close();
      }
      catch (IOException e) { return null; }
      return lines.toArray(new String[lines.size()]);
   }
}
${endHelper}

String[] ${freeName(lines)} = FileLineReader.readAllLines(${fileObject});
```

Figure 10. Integration markup

result and filters out the results that fail the test. For example, the query "for x" may return a result that tries to create a for loop using a hypothetical local integer x. However, if the client detects a local variable x of another type, this search result will be removed. The client also filters results that are incompatible with the existing code. For example, if integration markup preconditions are not met.

4.4.5.2 *Contextualization.*

If the user omits an argument, or does not finish typing an argument, the client analyzes the user's source code to find variables within the current scope that can serve as arguments. This list of possible substitutions is used to create variations on the original, incomplete search result. These client-generated results are presented to the user instead of the original. In Figure 7, the client detects a compatible local variable (playerScores) and uses it to complete the server's search results. After variables are matched with parameters, the snippet is customized according to the integration markup.

4.4.6 **Snippet Integration**

Instructions for integrating a snippet into the user's source code are expressed using a lightweight markup embedded within the snippet code. The SnipMatch markup is similar to the markup used in Eclipse Templates [38], with several additions. Eclipse Templates is a built-in feature in Eclipse that allows users to create and insert code "templates" (snippets). Like SnipMatch, it allows snippets to be integrated into the source code by taking into account local variables and adding missing import statements. Unlike SnipMatch snippets, each Eclipse Template can only be found through its single-word name. Parameters, helper classes, and precondition checking are not supported. Figure 10 shows the snippet code associated with the search pattern "read lines in <fileObject> into an array". This snippet includes two additions to the Eclipse Templates markup: a search result argument (<fileObject>) and a helper class. In figure 10, ${fileObject} indicates where to insert the argument for the <fileObject> parameter. The helper class is delimited by ${helper} and ${endHelper}. Helper classes are inserted in the current source file.

Helper classes are an optional feature that can make code easier to read and reduce redundancy. For example, a programmer might prefer to read text from a file by calling a method of a helper class instead of having the method code appear inline. SnipMatch maintains a record of inserted helper classes so that future snippet insertions will not create duplicate classes.

Another addition to the Eclipse Templates markup is support for integration preconditions. The SnipMatch markup includes the terms ${startPrecondition} and ${endPrecondition}. The code between these two terms is expected to be a Java method named preconditionTest. Before the snippet is shown to the user, the preconditionTest method is executed in a secure sandbox that prevents access to other local processes or communication with other machines. The return value of the method is a Boolean that determines whether or not the user's code meets the necessary conditions for the snippet to be integrated. This method accepts three arguments with information about the source code file currently open in the IDE. The first argument is an instance of the Eclipse JDT ASTParser class. This argument contains a copy of the abstract syntax tree that can be traversed to perform validations. A copy of the source code and the current cursor position in the file are also included as arguments – for example, so that a snippet creator can alternately write a simple grep when string matching is sufficient.

A graphical interface for specifying preconditions without writing code could be created. For now, video tutorials on the SnipMatch website and an interface for selecting markup within the snippet editor provide usage instructions and examples.

4.4.7 **Benefits of Client-Server Search Processing**

Allowing both the client and the server to process search results has privacy and efficiency benefits. It mitigates privacy concerns, since source code is not sent from the client to the server. Computation is distributed between the client and server, with the client handling the computationally expensive filtering and integration steps that are dependent on the user's code. Server results can be cached because they are not specific to the user's code.

4.4.8 **Extensibility**

Extending SnipMatch to support other imperative programming languages is straightforward. In addition to Java, we have tested SnipMatch with C++, PHP, and JavaScript. The search algorithm and markup were not changed. The Eclipse plug-in was modified to extract variable names and types from these different ASTs. Type consistency checking was disabled for the dynamically typed languages, but search pattern parameters were still supported. Wrapper classes have been created to simplify the process of extending the plug-in for third-party developers.

4.5 **Evaluation**

We conducted three studies: a lab study, an analysis of usage logs following the public deployment of SnipMatch, and interviews with programmers who have used SnipMatch. The lab study was conducted to gather initial feedback and assess ease of use. We then made SnipMatch publicly available and analyzed our logs to gain additional insights and confirm external validity. Finally, we interviewed five SnipMatch users to learn more about their needs and search behaviors.

4.5.1 **Study 1: Evaluating SnipMatch in the Lab**

4.5.1.1 Method

Two sets of 8 computer science students were recruited for this study. Each participant was given two programming tasks to complete. Participants in the first set were given a brief (5-minute) SnipMatch tutorial and asked to use SnipMatch instead of searching online for code snippets.

Participants in the second set were allowed to search online and were not trained to use SnipMatch.

To avoid priming our participants with search keywords, we explained the tasks by showing images. For the first task, we showed two images. The first image depicted two file folders, one full of files and the other empty. It was described as the “before” image. The second image contained the same two file folders with the all of the files now in the other folder. Participants were asked to “write a program to change the current state of the system to the second image”. They were also shown that the folder that contained the files in the first images existed on the computer and was currently full of files. For the second task, we showed one image, depicting a simple Graphical User Interface (GUI) for moving files from one folder to another. This GUI included two text fields, labeled “Source folder” and “Target folder”, and a button labeled “Move files”.

At the time of the study, SnipMatch included 29 snippets and 55 search patterns. The snippets included all of the functionality required to complete the tasks. The larger number of search patterns indicates that some snippets were discoverable through more than one search pattern. Approximately half of the snippets were created specifically for this study. Drawing from textbooks and our experiences teaching computer science, we attempted to include snippets to support most standard file input and output operations and many basic GUI features. To narrow the scope of our investigation to features specific to SnipMatch, we did not include search results from external code repositories, such as Google Code Search [42].

We anticipated that participants using SnipMatch would need to perform a minimum of 7 snippet integrations: 2 for the first task and 5 for the second. This calculation was based on the assumption that participants had memorized the method calls required to complete our tasks. It also assumed that participants do not use SnipMatch for basic programming statements (e.g. to create a for loop) and that they prefer to copy and paste their code rather than use SnipMatch repeatedly.

After completing the tasks, each participant using SnipMatch filled out a questionnaire. We logged all SnipMatch searches and the times required by the server and the client to generate the results.

We recorded web search queries and web pages visited for the participants who did not use SnipMatch. After both tasks were completed, we asked these participants about their programming behavior and experiences working with code snippets found online.

4.5.1.2 Results

Programming with SnipMatch. All of our participants successfully completed both tasks. On average, participants completed the first task in 12 minutes (*s.e. 1.5*) and the second in 28 minutes (*s.e. 3.3*). On average, participants opened the SnipMatch search window 14.2 times (*s.e. 2.2*) and selected a snippet to integrate 74% of the time. Participants performed many exploratory searches to test the capabilities of the system. Most also used SnipMatch to write the loop required in the first task and to create the multiple labels and textboxes for the second task. Verbs were popular search terms (copy, create, move, open, read).

Two participants mentioned that they did not need to remember syntax. One of these participants explained the process of writing code using SnipMatch as “search to figure out how to do something, resulting in the creation of an object, then search again, including this object, to continue using it". Several participants indicated that they were interested in using SnipMatch to create private snippet repositories.

The mean rating for the question "I would use this tool on a regular basis if it was available in my preferred development environment" is particularly encouraging (μ=4.75 on 5 point Likert Scale). Participants also consistently gave SnipMatch high marks as an efficient alternative to online search (μ=4.75) and for ease of use (μ=4.5).

Programming without SnipMatch. All participants successfully completed the first (file i/o) task. Two participants were unable to complete the second (GUI) task within an hour. Including only data from tasks that were completed, on average participants performed 14 online searches (10 min., 18 max.), 7 for each task. On average, participants viewed 9 (5 min., 12 max.) non-search engine web pages for the first task and 15 (8 min., 21 max.) for the second task. Participants completed the first task in 25 minutes (*s.e. 2.5*) and the second task in 37 minutes (*s.e. 3.6*). All searches were performed on Google, with the exception of four performed on Stack Overflow.

4.5.1.3 Discussion

Programming with SnipMatch. Study participants understood how to use the tool and were able to use it effectively. We were surprised that all participants successfully completed both tasks. One participant had never programmed in Java and GUI programming can be difficult. Most participants chose to use SnipMatch even when it wasn't necessary. For example, they used it to create the loops and as an alternative to copying code from elsewhere in the file.

One participant began the first task by creating String variables for the directory paths. With the variables in scope, he then typed his first search query and the top search result previewed the exact code required to complete the first step for this task: creating an array of File objects for the files in the directory identified in one of his Strings. Several participants regularly took advantage of this context-sensitivity, adopting a search-based code writing behavior in which objects created from prior searches were included as keywords in future searches.

Some results were not as accurate. Participants requested that synonyms be added to the system. In particular, it was suggested that words referring to the same abstract concept in different programming languages might be interchangeable (e.g. form and frame) for search purposes. Also, preconceived ideas regarding command lines initially biased some participants towards selecting snippets that included string arguments instead of object arguments. This was overcome once SnipMatch displayed results that included local object variable names.

Participant feedback indicates that they were comfortable including arguments within search queries. The inline snippet preview provides code context for the arguments. We observed several participants reading the preview as they entered arguments.

The combination of natural language search results and inline previews was sufficient for participants to understand most snippets before integration. Although snippet integration can be undone with a single undo operation, this option was rarely used.

The average server response time was 64ms (*s.e. 2.0*). The Eclipse plug-in then spent 172ms (*s.e. 15.9*), on average, customizing the server results. While the code is not optimized, we believe that these numbers reflect the substantial offloading of computationally expensive operations to the client. Operations specific to the programmer's existing source code are performed client-side. This preserves privacy. With this approach, it is also possible to cache all server responses.

SnipMatch can be provided to a large number of users at relatively low cost, similar to hosting static HTML pages.

Programming without SnipMatch. Only one participant who did not have access to SnipMatch completed the tasks in less time than the slowest participant who used SnipMatch (14 minutes, 19 minutes). These numbers are encouraging, but this comparison has many limitations, including the small number of snippets that were in the SnipMatch database. While we expect that SnipMatch will continue to perform well as the number of snippets increases – since search patterns are short, precise, and can be refined as required – in this study, we are more interested in the differences in search behaviors.

On average, participants performed exactly the same number of search queries (14). However, participants using SnipMatch spent less time finding and integrating snippets. Without SnipMatch, participants were not able to directly select snippets from the search result list. Participants opened and viewed many additional web pages (24 on average) listed in the search results. These web pages contained Java examples, tutorials, and class documentation.

Most participants opened multiple tabs in the web browser and flipped between tabs, comparing examples. When asked about this behavior, participants explained that they were checking for differences in dependencies, attempting to verify that the code would operated as expected, and determine which of the examples would be easiest to integrate. With SnipMatch, participants often did not dwell on the search results. They typically inserted snippets with little hesitation, then experimented inline. We believe that this was partially due to the smaller time commitment required to insert snippets with SnipMatch. Viewing the snippet inline, participants could also benefit from the error and warning highlighting provided by the IDE.

Without SnipMatch, participants frequently revisited search result pages and selected alternate links rather that performing additional searches. After completing the tasks, 4 of our 8 participants indicated that they had difficulty coming up with alternate wording that would "make a difference" to the search results shown. They explained that adding additional words to their search queries often did not improve the search results. SnipMatch users may be less likely to experience this problem, since search queries are matched against search patterns, rather than whole documents, for *in-order* and *unordered* results.

4.5.1.4 Limitations to the Participant Recruitment Method

While we were careful to recruit participants from the same population (computer science students at our university) and using the same recruitment channel (our faculty mailing list), the first set of participants was recruited before the second set of participants. We initially intended only to obtain first use data, then decided to expand the study. Between sets, participants had similar programming experience (first set: 3-10 years, second set: 2-10 years), age (20-29, 21-26), and gender balance (1 female, 2 female). No statistical tests were performed on the study results.

4.5.2 **Study 2: Deployment To 93 Programmers**

4.5.2.1 Method

We made the SnipMatch plug-in publicly available to gain additional insights and verify external validity. We were specifically interested in better understanding usage in the wild. Will programmers use SnipMatch? How often will programmers use SnipMatch? We were also interested to determine the types of search queries and snippets that are most frequently used.

To obtain users, we created a webpage and embedded links to share it on Facebook, Twitter, and Google+. We then publicly announced SnipMatch on academic and industry mailing lists. With user consent, we logged all search queries and snippet insertions conducted during the first three weeks of public deployment. Before the announcement, we increased the number of SnipMatch snippets (72 snippets, 111 search patterns).

4.5.2.2 Results

93 programmers performed 516 searches: 345 resulted in snippets being inserted into the existing source code, and 171 were cancelled. Ten programmers used SnipMatch on more than four days. Four programmers used SnipMatch during each of the three weeks. These programmers performed 22-54 search queries. Five programmers created snippets. Two of these programmers are among those who have used SnipMatch on the largest number of days. Programmers created snippets for logging, class creation, object and value comparison, and debugging tasks. To preserve external validity, these results exclude usage by all individuals associated with the research and development of SnipMatch.

4.5.2.3 Discussion

Programmers are often eager to try new tools, but long-term retention and integration into daily practice is substantially less common. As a comparison point, Blueprint had a 1% user retention rate over a five-month span (where “retention” was measured by *any* use of the tool five or more months after initial installation) [14]. In the unlikely case that SnipMatch can maintain even a moderate portion of its current 4% (4/93) rate of *weekly* use, we will consider the tool to be highly successful.

When we conducted phone interviews (described in the following section) we were surprised to learn that at least two of our regular users are professional programmers, and that they are using SnipMatch in the workplace. Programmers with no direct interest in our work are regularly using SnipMatch while creating commercial software. In particular, this professional interest demonstrates need for a snippet search and insertion tool.

The snippets that print values and perform type conversions were among the most frequently inserted, along with several of the user-created snippets. These popular print and conversion snippets have search patterns with parameters. Some of the frequently used user-created snippets also have parameters and other features that require the snippet markup. This indicates that programmers were able to teach themselves to perform searches and use the markup.

4.5.3 **Study 3: Interviews with Professionals**

4.5.3.1 Method

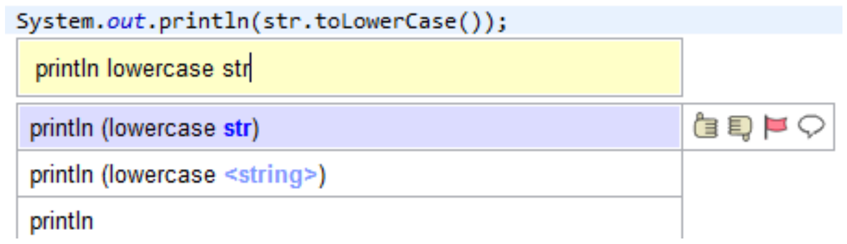
We sent an email to each of the 49 SnipMatch users who registered for an account, requesting that they participate in a 15-minute phone interview. 5 of our users, all professional programmers, agreed to be interviewed (all male, two living in the United States, one in Germany, and two in India). Two of these individuals are among the four programmers who have used SnipMatch during each of the three weeks of public deployment. We asked our interviewees to describe situations when they had used SnipMatch. We also examined their usage logs and spoke with them about searches they had performed. Then we asked if they had any difficulties using SnipMatch or ideas for improving the tool.

4.5.3.2 Results and Discussion

Common Usage Scenarios. Our interviewees described two common usage scenarios: shortcuts and quick reference. All interviewees used SnipMatch as a typing shortcut. For example, typing “convert”, “log”, or “println”, along with arguments, in the search box. In addition to saving time by reducing keystrokes, one programmer reported that this helped him to “focus his attention”, “staying in flow with the code”. Three programmers described specific situations when they had inserted snippets that they could not have written from memory, including design patterns and API calls. One programmer explained that SnipMatch was particularly useful for his Android development, which he found to require many “hard to remember structures”. Another mentioned writing snippets so that he would not need to memorize how to call JDBC methods. These programmers did not memorize how to type the search queries to insert these snippets – they trusted that they could find them quickly.

SnipMatch complements libraries. Each call requires some boilerplate code: minimally, importing a library and passing arguments. SnipMatch makes this easier, and shifts the balance towards reuse over code duplication. Further, with SnipMatch, programmers are able to help each other maintain good coding practices by curating the search results: if a snippet duplicates code instead of importing it, users can rate, flag, or comment to provide feedback.

Suggestions for Improvement. The programmers in India both requested that we reduce the server response time. In addition to promising to add a geographically proximate server, we proposed to modify the client to cache most frequently used snippets. This will also allow SnipMatch to function offline.



```
System.out.println(str.toLowerCase());
```

Figure 11. A search result with a nested snippet.

One programmer requested that search results for recently added snippets appear when the search box is empty. He was concerned that users might otherwise not discover them. For example, if a user had previously searched for image manipulation snippets and hadn't found any, he thought they might not think to search again for some time. We proposed to implement his suggestion and also to update the client to display results from other snippet databases (i.e. *unsigned* results) below SnipMatch results.

A programmer who uses SnipMatch regularly requested a method for specifying Boolean expressions that could be varied from within the search box. He indicated that he didn't want to have to create snippets for each combination. We then told him about the nesting feature that we are building, not yet released for public use. He thought it would be sufficient for his purposes.

4.6 **Inserting Multiple Snippets With One Query**

During the lab study, participants spent substantial time manually integrating snippets. For example, most participants in both studies inserted one snippet to retrieve a list of the files in a directory and a second to move files between directories. The participants then modified the snippets to interact. To reduce the time and effort required to integrate snippets, we designed an extension to SnipMatch that allows programmers to insert multiple snippets with a single search query (Figure 11).

For example, if file1 is a Java File object, SnipMatch can recognize that the search query "move file1 to file1 parent directory" can be satisfied by combining snippet results that match the following search patterns: "move <x> to <y>" and "<z> parent directory". In this example, <x>, <y>, and <z> represent search pattern parameters. To reduce ambiguity in the search results, parentheses are inserted: "move file1 to (file1 parent directory)".

To implement this SnipMatch extension, we wrote a context-free parser that interprets search pattern parameters as nonterminal symbols and generates production rules from search patterns. We also added a new, optional, field to store the snippet return type. The left hand side of each production rule is a nonterminal symbol representing the return type of the snippet. This allows programmers to specify, by writing in-order search queries, how multiple snippets are to be combined.

Nested search results – search results that are produced by this extension – are first ranked using the *in-order* result ranking criteria. Next, the results are ranked by nesting depth. Results that have fewer levels of nesting are ranked higher. The ranked list of *nested* results appears below the *in-order* results and above *unordered* results in the search result list.

When performing a nested search, programmers will likely not type the entire query at once. Instead, they might add additional words to a shorter query as results appear. For example, in an informal evaluation of this extension we observed a programmer building up to the search result "read file (lowercase <string>) into a string". He first typed "read file", then remembered that his code could be storing the file name in the wrong case, and decided to modify the search query to "read file lower" to verify that SnipMatch supports this nesting. While this extension has not been formally evaluated, we believe that it is a useful starting point for future investigations.

4.7 **Summary**

We have documented initial user experiences that demonstrate how search patterns and the snippet markup can improve a snippet search interface. Results from our lab study and public deployment suggest that SnipMatch can be an effective tool for certain tasks. Professional programmers, found to be using SnipMatch in the wild, reported that SnipMatch was useful as a memory aid and reduced context switching.

Unlike prior search tools for curated snippets, results are ranked and filtered based on both a snippet markup and existing code in the IDE. SnipMatch users can include local variables in search queries to pass them as arguments, customizing curated snippets from the search box.

Installation instructions and a quick start tutorial are available at http://snipmatch.org. We have released SnipMatch as open source. The source can be downloaded from the Eclipse Foundation's code repository.

Chapter 5

Conclusion and Future Work

We presented two search interfaces for integrating shared code within development environments. The first interface, WordMatch, is an end user programming environment for specifying direct answers to simple factual queries. Once created, an answer pattern – used to construct an answer – will be suggested for use within other answers and can be shared. Participants with no prior programming experience were able to create complex answers that can significantly reduce search time with minimal training. SnipMatch, the second interface, is a plugin for the Eclipse IDE. Snippet search results are ranked, filtered, and customized based on the current code context and the search query. SnipMatch was found to significantly reduce programming time for certain tasks. Findings from a longitudinal study and interviews with professional programmers include that SnipMatch is commonly used to reduce context switching and as a memory aid.

This work is a step towards crowdsourced programming – that is, crowdsourcing the completion of programming tasks. These search interfaces are CHC systems that crowdsource a critical aspect of their functionality: the source code available for use. CHC systems with programming tasks – crowdsourced programming systems – may be used to solve a variety of problems. For example, it may be possible to design an alternative to ReCAPTCHA that will not be indefinitely reliant on human input: the task could instead be a programming task to improve an optical character recognition algorithm. This is only one of many possible applications beyond improving programming environments, the primary focus of this work. Algorithms for other tasks that humans can perform with relative ease but remain challenging for computers might also be approached with crowdsourced programming systems. But even focusing only on software development, there is potential for further application of this work.

Given the substantial amount of time people spend working with computers, it's important that we try to reduce human-computer communication bottlenecks. We may further shift the divide between "computer users" and "programmers" as we make software development tools easier to use and more accessible. Language remains a substantial communication bottleneck. Often, less time is required to instruct a human to perform a computer task than to perform the task. We instruct each other in natural languages but interact with computers using programming languages or interfaces built with them. SnipMatch, and other tools that allow us to interact using words from natural languages, can reduce the time and skill required to perform computer tasks. Keyword search, and search for structured phrases, is useful not only for information retrieval but also to direct or control computer operation.

In this chapter we present suggestions for future work. We first examine several possibilities for extending WordMatch, including the integration of semantic web systems. We then consider opportunities to enhance or extend SnipMatch.

5.1 **Future Work**

5.1.1 **WordMatch**

5.1.1.1 Semantic Web Integration

WordMatch users need access to data sources before they can add direct answers. DBpedia, Freebase, Data.gov, Data.gov.uk, and many more of the largest online data sources for public information make their data accessible in the RDF format. RDF data is represented using subject-object-predicate tuples. In preparation for future work, we developed a set of special answer patterns for WordMatch that allow users to retrieve data from RDF data sources without understanding the RDF data representation. We chose to use the terms label and value to describe the relationships between the RDF subject, object, and predicate terms. For example, our set of special answer patterns for accessing RDF data includes: “In DBpedia, *x* has the label *y*” and “In DBpedia, the label *x* has the value *y* for *z*”. In our current design, to avoid naming collisions between different RDF ontologies, the set of prefixes is restricted and prefixes are not visible to users.

Using these special answer patterns, a WordMatch user can produce the direct answer “Marlon Brando was born on 1924-04-03”. This is because the special answer pattern “In DBpedia, the label *x* has the value *y* for *z*” can produce the direct answer “In DBpedia, the label birthdate has the value 1924-04-03 for Marlon Brando”. The current implementation produces direct answers by executing, in real-time, remote SPARQL queries on the DBpedia servers. These queries use the user’s search terms to find relations that satisfy an answer pattern. Although this approach is very inefficient, it serves as a proof-of-concept. Using these special answer patterns, end users can add direct answers that are dynamically produced from the petabytes of public data available from semantic web data sources. It would be interesting to publicly deploy this technology – for example, as a mobile phone application – and study usage patterns and user feedback.

5.1.1.2 Executable Commands

Many systems have been created to support executing commands written in natural language. Spotlight for Mac OS X, Windows 7 search, Inky [91], and Koala [77], are a few examples. None of these systems were designed to support users creating new executable commands. There are several reasons why users might want to create commands, including to create alternate wording for existing commands so that they can be more easily found when searching, and to create command "mashups".

WordMatch could be used to enable users to create executable commands. We developed a prototype implementation by programming special answer patterns that execute commands. These special answer patterns are called executable answer patterns. As with any answer pattern, they can be included as supporting statements in new answer patterns. When a direct answer is produced from an answer pattern that has an executable answer pattern as a supporting statement, the system provides a link that the user can click to execute the action(s) associated with the executable answer pattern. For example, if the executable answer pattern "Change the screensaver delay to *x* minutes" is included as a supporting statement in the answer pattern from which the direct answer "Change the screensaver delay to 30 minutes" is produced, clicking on the link to execute the action associated with this direct answer should change the screensaver delay to 30 minutes. A user, looking to update the screensaver delay on their computer, might enter the query "screensaver 30 minutes", and be prompted with a link to perform the action. Further investigation could include initial user experience studies and analysis of applications for this technology.

5.1.2 **SnipMatch**

5.1.2.1 Additional studies

The Eclipse Code Recommenders team is currently working to integrate SnipMatch into the Eclipse IDE. This should yield several opportunities for further study, including: additional snippet usage data, interface usage data, and user feedback. This version of SnipMatch moves all of the logic to the client: the client directly accesses the snippets, and it can be synchronized with remote code repositories.

We will log requests to access the default SnipMatch code repository and we anticipate an optional setting that allows Code Recommenders to track usage data client-side. For example, for some of our users, we may be able to log search queries and snippets selected for integration. Given that this deployment will be to a much larger audience and over a longer time span, this data seems likely to foster new insights. In particular, we are interested to explore methods to motivate contributions: for example, we might perform a study to determine if information about the number of times others have integrated your snippets affects the quality or quantity of snippets a user submits. We will also investigate the search result feedback: for example, user-submitted ratings, flags, and comments could be incorporated into the ranking algorithm and exposed in the interface.

Eclipse committers have already begun providing feedback and making feature requests. We have received several requests for features to support publishing code repositories. These features would make it easier to find repositories maintained by different teams. It will be interesting to explore methods to rank snippets based on retrieval location. We've also had several requests for additional UI components to allow missing parameters to be filled in once the code has been inserted. Users want to use SnipMatch for more than just search; also as a programming interface. We expect that there will be demand for studies comparing alternate interface designs for different use cases.

5.1.2.2 Search suggestions

The method and interface for nesting multiple snippets might be even more useful if it were extended. For example, it might be possible to disambiguate many search queries when brackets are not present. As the number of snippets accessible from within SnipMatch increases, similar techniques may make it possible to effectively create new snippets on demand, perhaps without users even being aware that no individual snippet was sufficient to match some search queries. There is likely considerable room for new search interfaces that construct snippets on-demand from existing snippets.

5.1.2.3 Commands

Similar to WordMatch, there are opportunities to use SnipMatch to execute commands. Potential applications include a new set of command-line tools for the IDE and a new interface for users to perform general computing tasks. A search interface that supports parameters may be useful for executing a variety of IDE actions. For example, in the case that a user is not familiar with available shortcuts to perform the same task using alternate methods. Similar to sharing snippet code, code to execute might be shared, or plugins for the IDE.

A search interface for general computing tasks could also be constructed. One embodiment might be a replacement for Spotlight, the Mac OS X application. Similar to the IDE tool for executing commands, this tool could include parameter support and crowdsource the code to execute. Similar to SnipMatch, this interface might decrease the time required to perform search tasks. Responses to search queries could be suggested and codified by users: a command-line replacement for the Internet age.

5.2 **Summary**

Search interfaces for integrating source code within development environments can support end user programming and reduce programming time. This is a step towards crowdsourced programming; since the source code is crowdsourced, these search interfaces are CHC systems with programming tasks. We look forward to one day replacing current crowdsourcing systems – which will always require human input – with new systems that use programming tasks to improve existing algorithms.

Chapter 6

References

1. Ackerman, M. S., McDonald, D. W. *Answer Garden 2: merging organizational memory with collaborative help*. In Proceedings of the ACM conference on Computer Supported Cooperative Work Companion (CSCW ’96), pp. 97-105. 1996.

2. Adamic, L. A., Zhang, J., Bakshy, E., and Ackerman, M. S. 2008. *Knowledge sharing and yahoo answers: everyone knows something*. In Proceedings of the International World Wide Web conference (WWW ’08). ACM, New York, NY, pp. 665-674. 2008.

3. Adar, E., Dontcheva, M., Fogarty, J., and Weld, D. S. *Zoetrope: Interacting with the Ephemeral Web*. In Proceedings of the ACM symposium on User interface software and technology (UIST '08), pp. 239-48, 2008.

4. Allen Cypher. *EAGER: programming repetitive tasks by example.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '91). pp. 33-39. 1991.

5. Allen Cypher. *Watch what I do: programming by demonstration.* MIT Press, Cambridge, MA, USA, 1993.

6. Auer, S., Bizer, C., Lehmann, J., Kobilarov, G., Cyganiak, R., and Ives, Z. *Dbpedia: A nucleus for a web of open data.* In Proceedings of the International Symposium on Wearable Computers (ISWC ’07), 2007.

7. Bajracharya, S., T. Ngo, et al. *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. In Companion to OOPSLA: ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 681-82, 2006.

8. Bajracharya, S., T. Ngo, et al. *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*. In Companion to OOPSLA: ACM SIGPLAN, pp. 681-82, 2006.

9. Bolin, M., Webber. M., Rha, P., Wilson, T., and Miller, R. C. *Automation and Customization of Rendered Web Pages.* In Proceedings of the ACM symposium on User interface software and technology (UIST '05), pp. 163-172. 2005.

10. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. *Freebase: a collaboratively created graph database for structuring human knowledge.* Proc. SIGMOD, pp. 1247-1250, 2008.

11. Borras, P.; Clement, D., Despeyrouz, Th., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. PSDE 1989. 24, pp. 14–24. 1989.

12. Bouguessa, M., Dumoulin, B., and Wang, S. 2008. *Identifying authoritative actors in question-answering forums: the case of Yahoo! Answers.* In Proc. 14th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining. 2008. KDD '08. ACM, 866-874

13. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R. *Example-centric programming: integrating web search into the development environment.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '10). pp. 513-522, 2010.

14. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R. *Example-Centric Programming: Integrating Web Search into the Development Environment.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '10). pp. 513-522, 2010.

15. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., Klemmer, S. R. *Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover*. IEEE Software, 26 (5), pp. 18-24. 2009.

16. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R. *Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '09). pp. 1589-1598, 2009.

17. Brin, S. and Page, L. *The anatomy of a large-scale hypertextual Web search engine.* Computer Networking ISDN Syst. 30, 1-7 (Apr. 1998), 107-117. 1998.

18. Broder, A. *A taxonomy of Web search*. SIGIR Forum, 36(2), 3-10. 2002.

19. Bryant, S. L., Forte, A., and Bruckman, A. *Becoming Wikipedian: transformation of participation in a collaborative online encyclopedia*. In Proc. 2005 international ACM SIGGROUP Conference on Supporting Group Work (GROUP '05), pp. 1-10. 2005.

20. C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch. *Alice2: Programming without syntax errors*. In Proceedings of the ACM symposium on User interface software and technology (UIST '02), 2002.

21. Casting Words. http://castingwords.com/. Retrieved on September 1, 2012.

22. Cellier J.M., & Eyrolle H. *Interference between switched tasks*, Ergonomics, 35, 1, pp. 25-36. 1992.

23. Cheyne, T. L. and Ritter, F. E. *Targeting audiences on the Internet.* Communications of the ACM 44, 4, pp. 94-98. 2001.

24. Chi, E. H., *Information Seeking Can Be Social,* Computer, v.42 n.3, pp.42-46, March 2009.

25. Conway, M., Pausch, R., Gossweiler, R., and Burnette, T. *Alice: A rapid prototyping system for building virtual environments*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '94), pp. 295–296, 1994.

26. Conway, S., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K. 2000. *Alice: lessons learned from building a 3D system for novices.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '00), pp. 486-493, 2000.

27. Corragio, L. *Deleterious Effects of Intermittent Interruptions on the Task Performance of Knowledge Workers: A Laboratory Investigation*, unpublished doctoral dissertation, University of Arizona, 1990.

28. Cutrell, E.B., Czerwinski, M., & Horvitz, E. *Effects of instant messaging interruptions on computing tasks*, In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '00), 99-100. 2000.

29. David Canfield Smith. *Pygmalion: A Creative Programming Environment*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. 1975.

30. Dearman, D. and Truong, K. N. *Why users of yahoo!: answers do not answer questions.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '10), 329-332. 2010.

31. Digg Site Ranking. http://www.alexa.com/siteinfo/digg.com. Retrieved September 1, 2012.

32. Digg. http://digg.com. Retrieved September 1, 2012.

33. Dontcheva, M., Drucker, S. M., Salesin, D., and Cohen, M. F. *Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout*. In Proceedings of the ACM symposium on User interface software and technology (UIST '07), pp. 61–70, 2007.

34. Donzeau Gouge, V.; Huet, G., Kahn, G., and Lang, B. (July, 1980). *Programming environments based on structured editors: The Mentor experience*. INRIA Research report no. 26. 1980.

35. Dörner, C., Myers, B. A. EUKLAS, Plug-in for Eclipse IDE. http://www.cs.cmu.edu/~euklas

36. Drori, O. *How to Display Search Results in Digital Libraries-User Study*. Proc. Workshop on New Developments in Digital Libraries, pp. 13–28. 2003.

37. Eclipse Quick Fix. http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F. Retrieved September 1, 2012.

38. Eclipse Templates. http://www.ibm.com/developerworks/opensource/library/os-eclipse-plugin-templates/index.html. Retrieved September 1, 2012.

39. Feiler, P. H., Medina-Mora R. *An Incremental Programming Environment*. IEEE

Transactions on Software Engineering SE-7(5), September 1981.

40. Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The Vocabulary Problem in Human-System Communication. In *Communications of the ACM* 30, 11 (Nov 1987), 964-971. 1987.

41. Garlan, D. B., & Miller, P. L. 1984. GNOME: *An Introductory Programming Environment Based on a Family of Structure Editors.* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

42. Google Code Search. http://www.google.com/codesearch. Retrieved September 1, 2012.

43. Google Inc., Deeper understanding with Metaweb. http://googleblog.blogspot.com/2010/07/deeper-understanding-with-metaweb.html. Retrieved September 1, 2012.

44. Google Search Features. google.com/help/features.html. Retrieved September 1, 2012.

45. Gray, W. D. and D. A. Boehm-Davis. *Milliseconds Matter: An Introduction to Microstrategies and to Their Use in Describing and Predicting Interactive Behavior.* Journal of Experimental Psychology: Applied 6(4). pp. 322-35, 2000.

46. Habermann, A. Nico; Notkin, David. 1986. *Gandalf: Software Development Environments*. IEEE Trans. Software Eng. 12 12 (12): 1117–1127.

47. Harper, F.M., Raban, D., Rafaeli, S., and Konstan, J.A. *Predictors of answer quality in online Q&A sites.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '08), 865-874. 2008.

48. Hartmann, B., Klemmer, S., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher,. A., and Gee, J. 2006. *Reflective physical prototyping through integrated design, test, and analysis.* In Proceedings of the ACM symposium on User interface software and technology (UIST '06). pp. 299-308. 2006

49. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. *What Would Other Programmers Do? Suggesting Solutions to Error Messages*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '10). pp. 1019-1028, 2010.

50. Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. 2007. *Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07). 145-154, 2007.

51. Hearst, M. *Search User Interfaces*, Cambridge University Press, Cambridge, UK, 2009.

52. Hoffmann, R., Fogarty, J., and Weld, D. S. *Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers*. In Proceedings of the ACM symposium on User interface software and technology (UIST '07), pp. 13-22, 2007.

53. Holmes, R., Murphy, G. C. *Using structural context to recommend source code examples.* In Proc. ICSE pp. 117-125, 2005.

54. Horowitz, D. and Kamvar, S. D. 2010. *The anatomy of a large-scale social search engine*. Proc. WWW, 431-440.

55. Hsieh, G. and Counts, S. *mimir: a market-based real-time question and answer service*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '09), 769-778. 2009.

56. InnoCentive. http://innocentive.com. Retrieved September 1, 2012.

57. IntelliSense. http://en.wikipedia.org/wiki/IntelliSense. Retrieved September 1, 2012.

58. Janes, J., Hill, C., and Rolfe, A. *Ask-an-expert service analysis*. J. Am. Soc. Inf. Sci. Tech. 52, 13, pp. 1106-1121. 2001.

59. Jin, J. and Dabbish, L. A. 2009. *Self-interruption on the computer: a typology of discretionary task interleaving.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '09), pp. 1799-1808. 2009.

60. Jones, K. S. *A Statistical Interpretation of Term Specificity and Its Application in Retrieval*. Journal of Documentation 28: 11, 1972.

61. Kellar, M., Watters, C., and Shepherd, M. *A field study characterizing Web-based information-seeking tasks*. J. Am. Soc. Inf. Sci. Tech. 58, 7, pp. 999-1018. 2007.

62. Kelleher, C., Pausch, R. *Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers*. ACM Computing Survey, 37(2):83–137, 2005.

63. Kittur, A., Chi, E. H., and Suh, B. 2008. *Crowdsourcing user studies with Mechanical Turk.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '08). ACM, New York, NY, 453-456. 2008.

64. Ko, A. J. and Myers, B. A. 2004. *Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '04), pp. 151-158. 2004.

65. Ko, A. J. and Myers, B. A. 2005. *Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data*. In Proceedings of the ACM symposium on User interface software and technology (UIST '05), pp. 3-12. 2005.

66. Ko, A. J. and Myers, B. A. 2006. *Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors*, In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '06). 2006.

67. Kosorukoff, A. (2001) *Human based genetic algorithm*. IEEE Transactions on Systems, Man, and Cybernetics, SMC-2001, pp. 3459-3464, 2001.

68. Kuznetsov, S. 2006. *Motivations of contributors to Wikipedia.* SIGCAS Computing Society 36, 2 (Jun. 2006), 1.

69. Lau, T., Wolfman, S. A., Domingos, P., and Weld, D. S. *Learning repetitive text-editing procedures with SMARTedit*. In Your wish is my command. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA pp. 209-226. 2001.

70. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. *DocWizards: a system for authoring follow-me documentation wizards.* In Proceedings of the ACM symposium on User interface software and technology (UIST '05), pp. 191–200, 2005.

71. Lee, U., Liu, Z., & Cho, J. (2005). *Automatic identification of user goals in Web search*. Proc. WWW 2005, pp. 391–400. 2005.

72. Leibenluft, J. *A librarian's worst nightmare: Yahoo! Answers, where 120 million users can be wrong*, Slate Magazine 2007.

73. Little, G. and Miller, R.C. *Keyword Programming in Java.* In Proc. ASE. pp. 84-93, 2007.

74. Little, G. and Miller, R.C. *Translating keyword commands into executable code.* In Proceedings of the ACM symposium on User interface software and technology (UIST '06), pp. 135-144, 2006.

75. Little, G., Chilton, L. B., Goldman, M., and Miller, R. C. 2009. *TurKit: tools for iterative tasks on mechanical Turk*. In Proc. ACM SIGKDD Workshop on Human Computation (Paris, France, June 28 - 28, 2009). 2009.

76. Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E., and Kandogan, E. 2007. *Koala: capture, share, automate, personalize business processes on the web.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07). pp. 943-946. 2007.

77. Little, G., Lau, T.A., Cypher, A., Lin, J., Haber, E.M., Kandogan, E. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07). pp. 943-946, 2007.

78. Liu, Y. and Agichtein, E. 2008. *On the evolution of the Yahoo! Answers QA community*. In Proc. 31st Annual international ACM SIGIR Conference on Research and Development in information Retrieval (Singapore, Singapore, July 20 - 24, 2008). SIGIR '08. ACM, New York, NY, pp. 737-738. 2008.

79. M. Kim, L. Bergman, T. Lau, and D. Notkin. *An Ethnographic Study of Copy and Paste Programming Practices in OOPL.* In *Proc. ESEM*. pp. 83–92 , 2004.

80. M. Zloof. 1975. *Query-by-example: the invocation and definition of tables and forms.* In Proceedings of the 1st International Conference on Very Large Data Bases (VLDB '75), pp. 1-24. 1975.

81. Mamykina, L., Manoim, B., Mittal, M., Hripcsak, G., Hartmann, B. 2011. *Design Lessons from the Fastest Q&A Site in the West.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '11)*,* pp. 2857-2866, 2011.

82. Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. *Jungloid Mining: Helping to Navigate the API Jungle*. In Proc. PLDI, pp. 48-61, 2005.

83. Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. 2000. *Alice: lessons learned from building a 3D system for novices*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '00), 486-493. 2000.

84. McDaniel, R. G. and Myers, B. A. 1999. *Getting more out of programming-by-demonstration*. In Proc. of the SIGCHI Conference on Human Factors in Computing

Systems: the CHI Is the Limit. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '99), pp. 442-449. 2009.

85. McDaniel, S.W. and Rao, C.P. *An investigation on respondent anonymity's effect on mailed questionnaire response rate and quality*. Journal of the Market Research Society 23, 3, pp. 150–160, 1981.

86. Mechanical Turk. http://mturk.com. Retrieved September 1, 2012.

87. Medina-Mora, R. *Syntax-Directed Editing: Towards Integrated Programming Environments* (PhD Thesis, Department of Computer Science). Pittsburgh, PA: Carnegie Mellon University. 1982.

88. Medynskiy, Y., Dontcheva M., and Drucker, S. M. *Exploring Websites through Contextual Facets*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '09). pp. 2013-22, 2009.

89. Miller, P., & Chandhok, R. 1989. *The Design and Implementation of the Pascal Genie*. Proceedings of the 1989 ACM Computer Science Conference.

90. Miller, P., Pane, J., Meter, G., and Vorthmann, S., *Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University*, ILE, 4, 2, pp. 140-158. 1994.

91. Miller, R. C., Chou, V. H., Bernstein, M., Little, G., Van Kleek, M., Karger, D., and Schraefel, M. *Inky: A Sloppy Command Line for the Web with Rich Visual Feedback.* In

Proceedings of the ACM symposium on User interface software and technology (UIST '08), pp. 131-140, 2008.

92. Miyata Y., & Norman D. *Psychological issues in support of multiple activities,* In: User Centered Systems Design, Hillsdale NJ: Erlbaum, pp. 265-284. 1986.

93. Morville, P., and Callender, J. Search Patterns: Design for Discovery. O'Reilly Media, Inc., 2010.

94. Myers, B. A. 1986. *Creating dynamic interaction techniques by demonstration.* In Proceedings of the SIGCHI conference on Human factors in computing systems and graphics interface (CHI '87), pp. 271-278. 1987.

95. Myers, B. A. 1986. *Visual programming, programming by example, and program visualization: a taxonomy.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '86), pp. 59-66. 1986.

96. Myers, B. A., Burnett, M. M., Wiedenbeck, S., and Ko., A. J., *End user software engineering: CHI 2007 special interest group meeting.* In extended abstracts of the SIGCHI conference on Human factors in computing systems (CHI EA '07). ACM, New York, NY, USA, 2125-2128. 2007.

97. Myers, B. A., Weitzman, D., Ko, A. J. and Chau, D. H. 2006. *Answering Why and Why Not Questions in User Interfaces*. ACM Conference on Human Factors in Computing Systems, Montreal, Canada, April 24-27, 2006.

98. Myers, B. *Peridot: creating user interfaces by demonstration*. MIT Press, Cambridge, MA. 1993.

99. Nardi, B. A., *A Small Matter of Programming: Perspectives on End User Computing.* Cambridge, MA: MIT Press. 1993.

100. Oney, S., Brandt, J. Codelets: Linking Interactive Documentation and Example Code in the Editor. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '12), 2012.

101. Pane, J. F. *Assessment of the ACSE Science Learning Environment and the Impact of Movies and Simulations* (School of Computer Science Technical Report CMU-CS-94-162). Pittsburgh, PA: Carnegie Mellon University, 1994.

102. Pane, J. F., & Miller, P. L. *The ACSE Multimedia Science Learning Environment*. In Proc. 1993 International Conference on Computers in Education, pp. 168-173, 1993.

103. Panos Ipeirotis, *Mechanical Turk: The Demographics*. http://behind-the-enemy-lines.blogspot.com/2008/03/m echanical-turk-demographics.html. Retrieved September 1, 2012.

104. Pitkow, J. E. and Kehoe, C. M. 1996. *Emerging trends in the WWW user population*. Communications of the ACM 39, 6 (Jun. 1996), pp. 106-108. 1996.

105. Pitkow, J. Recker, M. *Using the Web as a survey tool: Results from the Second WWW User Survey 1995*; http://www.igd.fhg.de/www/www95/papers/79/survey/survey_2_paper.html.

106. Rafaeli, S., Raban, D.R., and Ravid, G. How social motivations enhances economic activity and incentives in the Google answers knowledge sharing market. I. J. Knowledge Learning, 3, 1, pp. 1-11. 2007.

107. *reCAPTCHA: Human-Based Character Recognition via Web Security Measures Luis von Ahn*, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum (12 September 2008) *Science* **321** (5895), 1465.

108. Holmes, R., Murphy, G. C. *Using structural context to recommend source code examples*. In Proceedings of the 27th international conference on Software engineering (ICSE '05), pp. 117-125. 2005.

109. Reiss, Steven P. Semantics-based code search. ICSE 2009.

110. Resnick, M., Maloney, J., Monroy-Hernande, A., Rusk, N., Eastmond, E., Brennan, K. *Scratch: programming for all*. Communications of the ACM 52, 11, pp. 60-67. 2009.

111. Rose, D.E., & Levinson, D. *Understanding user goals in Web search.* Proc. WWW 2004, pp. 13-19. 2004.

112. Sahavechaphan, N. and Claypool, K. *XSnippet: Mining for Sample Code*. In Proc. of OOPSLA, pp. 413-30, 2006.

113. Salon, *"I make $1.45 a week and I love it"*. http://www.salon.com/tech/feature/2006/07/24/turks/. Retrieved September 1, 2012.

114. Scaffidi, C., Myers, B., and Shaw, M. *Topes: Reusable abstractions for validating data.*

ACM/IEEE International Conference on Software Engineering, 1-10. 2008.

115. Shadbolt, N., Berners-Lee, T., Hall, W. The Semantic Web Revisited, Intelligent Systems, 21, 3, 96-101, 2006.

116. Shneiderman, B. *Direct manipulation: a step beyond programming languages*. IEEE Computer 16(8) (August 1983), 57-69. 1983.

117. Slashdot. http://slashdot.com. Retrieved September 1, 2012.

118. Smart Company*, Facebook sends uSocial cease and desist warning.* http://www.smartcompany.com.au/web -20/20091123-facebook-sends-usocial-cease-and-desist-warning.html. Retrieved September 1, 2012.

119. Speier, C., Valacich, J. S., and Vessey, I. The effects of task interruption and information presentation on individual decision making. Proc. ICIS '97, pp. 21-36. 1997.

120. Stylos, J. and Myers, B. A. *Mica: A Web-Search Tool for Finding API Components and Examples.* In Proc. VL/HCC, pp. 195-202, 2006.

121. Su, Q., Pavlov, D., Chow, J.-H., and Baker, W.C. *Internet-scale collection of human-reviewed data*. Proc. WWW 2007, pp. 231-240. 2007.

122. Teevan, J., Cutrell, E., et al. *Visual Snippets: Summarizing Web Pages for Search and Revisitation.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '09), pp. 2023-32, 2009.

123. Teitelbaum, T. and Reps, T. *The Cornell program synthesizer: a syntax-directed programming environment.* Communications of the ACM 24, 9. 1981, 563-573

124. The Guardian, *Investigate your MP's expenses.* http://mps-expenses.guardian.co.uk/. Retrieved September 1, 2012.

125. The Guardian, *MPs' expenses – what you've found so far.* http://www.guardian.co.uk/politics/blog/2009/jun/ 19/mps-expenses-what-you-ve-found. Retrieved September 1, 2012.

126. Thummalapenta, S. and Xie, T. *PARSEweb: A Programmer Assistant for Reusing Open Source Code on the Web*. In Proc. ASE. pp. 204-13, 2007.

127. Turker Nation Message Board System. http://turkers.proboards.com/. Retrieved September 1, 2012.

128. Viégas, F. B., Wattenberg, M., and Dave, K. 2004. *Studying cooperation and conflict between authors with history flow visualizations.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '04). CHI '04. pp. 575-582. 2004.

129. von Ahn, L. and Dabbish, L. 2004. *Labeling images with a computer game.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '04), pp. 319-326. 2004.

130. von Ahn, L., Blum, M., Langford, J. *Telling Humans and Computers Apart Automatically.* Communications of the ACM 47, 56, 2004.

131. von Ahn, L., Ginosar, S., Kedia, M., Liu, R., and Blum, M. 2006. *Improving accessibility of the web with a computer game.* In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '06), pp. 79-82. 2006.

132. Wightman, D. *Crowdsourcing human-based computation.* In Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries (NordiCHI '10), pp. 551-560. 2010.

133. Wightman, D., Bateman, S., Gutwin, C., Vertegaal, R. WordMatch: Enhancing Search Results with Direct Answers Created by End Users. *In submission.*

134. Wightman, D., Ye, Z., Brandt, J., Vertegaal, R. *SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization.* In Proceedings of the ACM symposium on User interface software and technology (UIST '12). 2012.

135. Wikipedia. http://wikipedia.org. Retrieved September 1, 2012.

136. Wired, *50,000 Volunteers Join Distributed Search for Steve Fossett.* http://www.wired.com/software/webservices/news/2007/09/distributed_search. Retrieved September 1, 2012.

137. Wired, *Online Fossett Searchers Ask, Was It Worth It?* http://www.wired.com/techbiz/it/news/2007/11/fossett_search. Retrieved September 1, 2012.

138. Wired, *The Rise of Crowdsourcing.* http://www.wired.c om/wired/archive/14.06/crowds.html. Retrieved September 1, 2012.

139. Witten, I. *A Predictive Calculator,* In Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, MA, 1993.

140. Woodruff, A., Faulring, A., Rosenholtz, R., Morrsion, J., and Pirolli, P. *Using Thumbnails to Search the Web*. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '01). pp. 198-205, 2001.

141. Wu, B. and Davison, B. D. *Identifying link farm spam pages.* In WWW '05. ACM, New York, NY, pp. 820-829. 2005.

142. Xie, T., Pei, J. *MAPO:Mining API Usages from Open Source Repositories.* In *Proc. MSR.* pp. 54-57, 2006.

143. Yahoo! Answers Scoring System. http://answers.yahoo.com/info/scoring_system. Retrieved September 1, 2012.

144. Yahoo! Answers. http://answers.yahoo.com/. Retrieved September 1, 2012.

145. Yeh, R. B., Paepcke, A., and Klemmer, S.R. Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces. In Proceedings of the ACM symposium on User interface software and technology (UIST '08), pp. 111-120, 2008.

146. *Your wish is my command: programming by example.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

147. Zhang, J., Ackerman, M. S., Adamic, L., and Nam, K. K. *QuME: a mechanism to support expertise finding in online help-seeking communities.* In Proceedings of the ACM symposium on User interface software and technology (UIST '07), pp. 111-114. 2007.