Inky: A Sloppy Command Line for the Web with Rich Visual Feedback

Robert C. Miller[1], Victoria H. Chou[1], Michael Bernstein[1], Greg Little[1], Max Van Kleek[1], David Karger[1], mc schraefel[2]

[1]MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{rcm,vikki,msbernst,glittle,emax,karger}@csail.mit.edu

[2]Electronics and Computer Science
University of Southampton
SO17 1BJ, United Kingdom
mc@ecs.soton.ac.uk

ABSTRACT

We present Inky, a command line for shortcut access to common web tasks. Inky aims to capture the efficiency benefits of typed commands while mitigating their usability problems. Inky commands have little or no new syntax to learn, and the system displays rich visual feedback while the user is typing, including missing parameters and contextual information automatically clipped from the target web site. Inky is an example of a new kind of hybrid between a command line and a GUI interface. We describe the design and implementation of two prototypes of this idea, and report the results of a field study.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Algorithms, Human Factors, Languages.

Keywords: command languages, natural language processing, web automation

INTRODUCTION

In the traditional taxonomy of user interface styles [11], most web sites offer a *menu-and-form* interface, which delivers important benefits to learnability, memorability, and error prevention. For frequent users, however, a menu-and-form interface can be inefficient, particularly if the interface is poorly designed and presents a long, complex form or a multi-step process. For these cases, a *command line* interface may be far more efficient, but at the cost of requiring the user to learn command names and syntax, putting more demands on the user's memory and making more chances for errors.

This paper presents Inky, a command line for the Web that aims to capture the efficiency benefits of typed commands while using novel techniques to mitigate the costs. Inky (short for Internet Keywords) is a popup window, invoked from the user's web browser, into which the user can type *keywords* expressing a command. Keywords may include command names, such as **reserve** or **email** or **search flights**, and command parameters, such as **9 am** or **vikki@mit.edu** or **New York**. The keywords are matched heuristically against a set of known commands using techniques presented in previous work [8,9] to produce a list of possible interpretations for the user to choose from. When the user chooses an interpretation and runs it, Inky fills in the menu-and-form interface for the corresponding web site.

Inky mitigates the classic pitfalls of command line interfaces in two ways: *sloppy syntax* and *rich feedback*. First, Inky commands have little or no syntax to learn. Keywords may be provided in any order, command names may be omitted or replaced with synonyms, and parameters may be entered in a variety of ways. The Inky interpreter makes its best effort to match the user's input against the available commands.

Second, Inky provides more rapid, richer feedback than a traditional command line. While the user is typing, Inky incrementally and continuously displays how it is interpreting the command. The interpretation also shows missing parameters to remind the user what other information can be included in the command, reducing the learning and memory burden.

In addition to textual feedback, Inky also judiciously incorporates graphical user interface widgets in its feedback. Some widgets are useful for parameter entry, such as a date picker for picking the day of a meeting, or a list of conference rooms for the meeting location. Other widgets provide *context* for the command that helps the user choose parameters and avoid errors. For example, Inky retrieves the user's own calendar and displays a snapshot of the relevant day and time. Inky thus represents a hybrid between a command line and a GUI interface.

Since Inky acts as a frontend shortcut for an existing menu-and-form system, the user does not need to write a complete command. If a command is invoked with important parameters missing, then Inky fills out as much of the form as it can from the parameters that were provided, and

leaves the user to complete the form and submit it. Inky also distinguishes between functions with persistent side effects (such as sending email or making a reservation) and functions without side effects (such as viewing a calendar or looking up a person). For commands with side effects, Inky normally leaves it to the user to submit the final form, unless the user presses Control-Enter to perform the side-effect immediately. Thus Inky covers a spectrum from simple bookmarks (**email**, then Enter, to go to the mail composition page) to partial commands (**email vikki@mit.edu,** Enter, to start composing an email to Vikki) to complete automation (**email vikki@mit.edu I'm running late**, Control-Enter, to send a quick email right away).

One important application of a sloppy command line is lightweight personal information management (PIM), such as entering to-do items, calendar meetings, and contact information. Previous work [1,12] suggests that the cost of starting, navigating, and entering data in PIM applications is one reason why users turn to Post-It notes or paper notebooks, despite the difficulty of search and retrieval that paper poses later. A command line with sloppy syntax allows PIM data to be captured quickly and efficiently, while filing it immediately in the appropriate application.

This paper describes our experience building and evaluating two prototypes. The first prototype, called simply Inky, uses sloppy syntax with textual feedback, and provides commands for several dozen websites. It was deployed in a small field study and a sample of commands invoked by those users is analyzed in this paper. The second prototype, called Pinky, focuses primarily on data entry for PIM web applications, such as Google Calendar and Remember the Milk. Pinky incorporates GUI widgets to help enter some arguments, and automatically shows relevant *web clippings* from the target web site to help the user complete the command. Both prototypes are implemented as Firefox extensions using Chickenfoot [2] to automate web sites.

The rest of this paper is organized as follows. First we survey related work in web automation and command lines. Then we describe the user interface and implementation of the Inky prototype, and report the results of a small field study. Finally, we describe recent work on the Pinky prototype, which includes GUI widgets and web clippings.

RELATED WORK

Most browsers support command-line-style shortcuts for web queries. For example, properly-configured Firefox bookmarks can be invoked by a keyword and one or more parameters. YubNub[1] extends this idea into a "social" command line for the Web, which allows users to contribute new commands to a shared repository. Unlike Inky, these are traditional command lines with rigid syntax and no feedback.

Many search engines effectively implement a sloppy keyword-driven command line, at least for information retrieval. For example, Google can display times (**Samoa time**), weather reports (**weather 02139**), arithmetic (**67/83**), and maps (**Monterey CA**). Google Calendar's Quick Add feature uses a similar approach to quickly add events to the calendar (**dinner with Michael Tues 7 pm**). Unlike these systems, Inky provides rich visual feedback while the command is being entered, and can be used to automate web sites that are not accessible to public search engines, such as an intranet conference room booking site.

Sloppy syntax has been used in other web automation systems, notably Koala/CoScripter [7] and Smart Bookmarks [6]. The command languages in these systems operate at a lower level than Inky: clicking buttons and links, filling in text boxes, and selecting from lists, checkboxes, and radio buttons. Inky commands aim at higher-level user goals, such as reserving conference rooms and sending email, which have a greater variety of commands and parameter types than other web automation systems must handle.

Others have considered how to solve usability problems in command line interfaces. Seminal work by Furnas et al. [4] showed that the names used for commands and objects vary widely across users, so unlimited aliasing offers the best hope for success. Inky follows this prescription by supporting synonyms. Usability improvements have been aimed at the Unix shell, notably Fishy[2], which adds syntax highlighting, multiple line editing, more autocompletion for command arguments, and better error messages. Systems like XMLterm[3] and LAPIS [10] move the Unix shell into the web browser, so that command line programs can display HTML or XML user interfaces for the user to interact with.

Command lines have also been built on top of desktop GUI applications, notably Quicksilver[4] and Enso[5]. These systems incrementally search for desktop applications, files, even dialog boxes and menu items. Quicksilver can be extended with plugins and user-written scripts. The continuous feedback provided by these desktop command lines inspired Inky's feedback.

For PIM data, our earlier Jourknow system [14] aimed to bridge the gap between lightweight text entry and structured information retrieval by managing the user's data as *information scraps*, but we found in subsequent studies that users are loath to abandon their current PIM tools. In contrast, Pinky provides lightweight text entry that puts the data directly into existing web applications.

[1] http://yubnub.org

[2] http://www.fishshell.org

[3] http://www.xmlterm.org

[4] http://www.blacktree.com

[5] http://www.humanized.com

USER INTERFACE

Pressing Control-Space in the web browser pops up the Inky window (Figure 1). This keyboard shortcut was chosen because it is generally under the user's fingers, and because it is similar to the Quicksilver shortcut (Command-Space on the Mac).

The Inky window has two areas: a text field for the user to type a command, and a feedback area that displays the interpretations of that command. The Inky window can be dismissed without invoking the command by pressing Escape or clicking elsewhere in the browser.

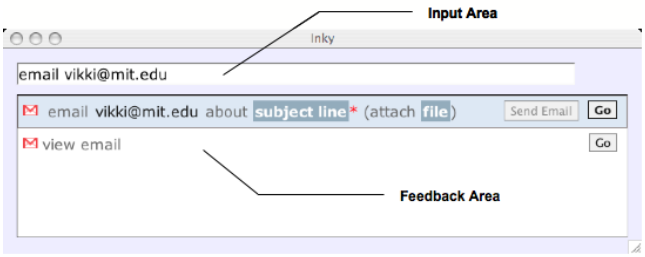

Figure 1: The Inky window.

Commands

A command consists of keywords matching a web site function, along with keywords describing its parameters. For example, in the command **reserve D463 3pm**, the **reserve** keyword indicates that the user wants to make a conference room reservation, and **D463** and **3pm** are arguments to that function.

To reduce the burden of learning and remembering syntax, the order of keywords and parameters is unimportant. For example, **reserve D463 3pm** and **3pm D463 reserve** will produce the same interpretation. Keywords that represent arguments to a function can be reordered and interspersed with keywords matching the function to be called. The order of the entered command matters only when two or more arguments could consume the same keywords. For example, in the command **reserve D463 3pm 1 2 2007** it is unclear if 1 is the month and 2 is the date or vice versa. In **find flights SFO LAX**, it is unclear which airport is the origin and which is the destination. In these cases, the system will give higher rank to the interpretation that assigns keywords to arguments in left-to-right order, but other orderings are still offered as alternatives to the user.

Commands can use synonyms for both function keywords and arguments. For example, to reserve D463 at 3pm, the user could have typed **make reservation** instead of **reserve**, used a full room number like **32-D463** or a nickname like **star room**, and used various ways to specify the time, such as **15:00** and **3:00**.

Function keywords may also be omitted entirely. Even without function keywords, the arguments alone may be sufficient to identify the correct function. For example, **D463 15:00** is a strong match for the room-reservation function because no other function takes both a conference room and a time as arguments.

The Inky prototype includes 30 functions for 25 web sites, including scheduling (room reservation, calendar management, flight searches), email (reading and sending), lookups (people, word definitions, Java classes), and general search (in search engines and ecommerce sites). Most of the functions included in the prototype are for popular web sites; others are specific to our university and our lab. Argument types specifically detected by Inky include dates, times, email addresses, cities, states, zip codes, URLs, filenames, and room names. Examples of valid commands include **email vikki@mit.edu Meeting Right Now!** (to send email), **java MouseAdapter** (to look up Java API documentation), **define fastidious** (to search a dictionary), **calendar 5pm meeting with rob** (to make a calendar event), and **weather cambridge ma** (to look up a weather forecast).

Feedback

As the user types a command, Inky continuously displays a ranked list of up to five possible interpretations of the command (Figure 2). Each interpretation is displayed as a concise, textual sentence, showing the function's name, the arguments the user has already provided, and arguments that are left to be filled in. The interpretations are updated as the user types in order to give continuous feedback.

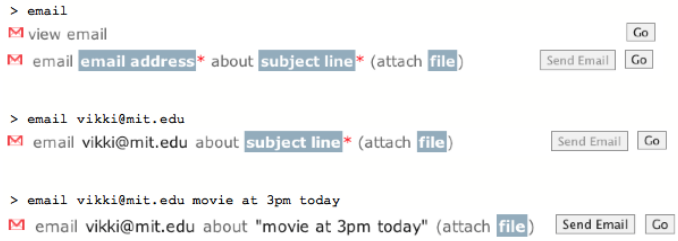

Figure 2: Feedback is shown continuously as a command is entered.

The visual cues of the interpretation were designed to make it easier to scan. A small icon indicates the website that the function automates, using the favicon image displayed in the browser address bar when that site is visited. Arguments already provided in the command are rendered in black text. These arguments are usually exact copies of what the user typed, but may also be a standardized version of the user's entry in order to clarify how the system interpreted it. For example, when the user enters **reserve star room**, the interpretation displays "reserve D463" instead to show that the system translated **star room** into a room number.

Arguments that remain to be specified appear as white text in a dark box. Missing arguments are named by a word or short phrase that describes both the type and role of the missing argument. If a missing argument has a default value, a description of the default value is displayed, and the box is less saturated. In Figure 3, "name, email, office, etc." is a missing argument with no default, while "this month" is an argument which defaults to the current month.

Figure 3: Feedback showing different kinds of argument feedback, including required (*room*), default values (*this month*), and rarely-used arguments (*repeats never*).

For functions with persistent side effects, as opposed to merely retrieving information, Inky makes a distinction between arguments that are required to invoke the side effect and those that are not. Missing required arguments are marked with a red asterisk, following the convention used in many web site forms. In Figure 3, the room and description are required arguments. Note that the user can run partial commands, even if required arguments are omitted. The required arguments are only needed for running a command in the mode that invokes the side effect immediately.

The feedback also distinguishes optional or rarely-used arguments by surrounding them by parentheses, like (repeats never) argument in Figure 3. It should be noted that this feedback does not dictate syntax. The user does not need to *type* parentheses around these arguments. If the user did type them, however, the command could still be interpreted, and the parentheses would simply be ignored.

Running a Command

Pressing Enter on a command runs the top-ranked interpretation by default. The arrow keys or the mouse can be used to select a different interpretation from the list, or the user can click the Go button next to the desired interpretation. When a command is run, the Inky window disappears, and Inky directs the browser to visit the target web site and fill in the form automatically.

Commands without side effects can be run with as few arguments as the user chooses to give. If the function is missing arguments, Inky relies on the fact that the website will either fill in appropriate defaults or prompt the user for required arguments when Inky tries to submit the form. By delegating these tasks to the website, Inky is able to use defaults that are stored by the web site. For example, AccuWeather.com uses an HTTP cookie to remember the last city used for looking up a weather forecast. By letting AccuWeather handle the default, Inky users can look up the weather in their usual area just by running the command **weather**.

A website's prompt may also include useful UI feedback and constraints that are not available in our textual prototype. For example, the command **travelocity SFO LAX** would start searching Travelocity for flights from San Francisco to LA, but Travelocity would prompt for departure and return dates with a custom calendar widget.

Functions with side effects can be run in two modes: *view* and *submit*. When a command is run in view mode, Inky fills in the form that will cause the side effect, but does not submit the form automatically. View mode is the default when the user presses Enter or clicks the Go button. For example, when the command **email vikki@mit.edu remember to buy milk today** is run in view mode, Inky creates an email composition window with the To and Subject fields filled in, and then turns control over to the user to review the composed email, edit it if desired, and press the Send button to send it. Arguments can be omitted from commands invoked in view mode, just as for commands without side effects, since either the web site can provide defaults or the user can fill in the remaining required arguments on the web site's form.

When a command is run in submit mode, Inky takes the final step of causing the side-effect to occur. Submit mode is selected by pressing Control-Enter, or by clicking the submit button in the desired interpretation. The submit button is labeled with the effect that it has, such as "Make Event", "Reserve Room", or "Send Email" (Figure 4). This button is disabled until all required arguments are provided to Inky, since Inky is taking responsibility for running the command. For example, typing **email vikki@mit.edu remember to buy milk today** and pressing Control-Enter will immediately send an email, with no further interaction. The command is run by automating the web site, however, so any confirmation pages or opportunities to cancel or undo would be visible.

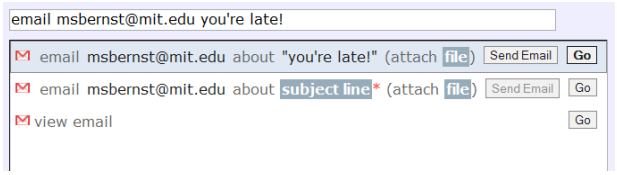

Figure 4: Commands can be run in either view mode (Go) or submit mode (*e.g.,* Send Email).

Separating commands with side effects from those without side-effects helps discourage Inky users from making errors that would be difficult to reverse. Since the default run methods always execute the command in a view mode, it is more difficult for a user to unknowingly cause a persistent side effect. However, by making it possible for users to run in a submit mode, Inky increases the efficiency of users who trust the system and want to commit to the side effect.

Self-Disclosure

One problem with a command line is poor visibility. The possible actions are not visible, as they are in menu-and-form systems, making it harder to learn what the system can do and how it is operated. Inky has two features that mitigate this problem. First, when the Inky window appears, before the user starts typing, the feedback area displays an example of a command. The example changes each time the window pops up, and whenever the user presses a key that doesn't change the text field, so the user can cycle through examples by pressing the arrow keys.

Second, when the user uses the web site interface for a function that Inky is capable of performing, Inky may display a reminder showing an Inky command that would have invoked the same function (Figure 5). This is a sim-

ple example of *self-disclosure* [3] that reminds the user of Inky's presence and capabilities. This kind of reminder risks annoying the user if it appears too often or indiscriminately, however. Inky displays the reminder probabilistically, with a probability that decreases each time the user sees the reminder for that function, and each time the user actually uses that function in Inky.

Figure 5: Inky reminds the user how to use it.

IMPLEMENTATION

Inky is a Firefox extension built on top of Chickenfoot [2] and created with Chickenfoot's extension packager. The system consists of the Inky window discussed in the previous section, which is implemented in HTML, CSS, and Javascript; a back-end keyword command interpreter implemented in Java; and XML files specifying the functions that commands can invoke and the types that arguments can take. Inky can be extended by adding new functions and types to these XML files.

The keyword command interpreter takes in a command from the user and returns an ordered list of possible interpretations of that command. The function and type definition files represent the database against which the user's command is matched.

Types

Function arguments have types, which help constrain the interpretation of commands. A type represents a class of objects that can be used or created by a function. Days of the week, airports, and conference rooms are all examples of types used in Inky.

Types are specified by an XML file. The type file in the Inky prototype includes string, email address, state, time, zip code, month, and day of the week, among others. The system can be extended with additional types by editing the XML file.

A type specification has three parts: a *recognizer* that detects an instance of the type in the user's command, and two *generators* that translate the matching text into Javascript code (for running a function) or human-readable English. For example, the *month* type has a recognizer that matches various ways of writing month names and numbers, a Javascript generator that produces the month's ordinal number, and a human-readable generator that returns a canonical name for the month, e.g. "May."

In its most general form, a type recognizer is a function that takes the user's input string (after tokenizing on whitespace boundaries) and returns all matching instances of the type, expressed as bit vectors over the token sequence. For example, a soft-drink recognizer given the string **I like coca cola more than pepsi** would return two bit vectors: 0,0,1,1,0,0,0 for the occurrence of Coca Cola and 0,0,0,0,0,0,1 for the occurrence of Pepsi. These bit vectors are used to account for which tokens were consumed by the type recognizer, so that the keyword command interpreter can ensure that different recognizers don't overlap.

The type recognizer function can be specified several ways in the XML file: as an enumeration of possible keywords (used, e.g., for month names), as a regular expression (e.g., for email addresses), or as a Javascript function if the type is complex.

Functions

Functions represent operations that Inky can perform on a web site. A function has several parts: a *recognizer* that detects the function's name in the user's input keywords; an ordered list of *arguments*; the *body* that actually runs the function; and a *generator* that displays a human-readable version of a partially-instantiated function.

A function recognizer is similar to a type recognizer, in that it takes the user's input and returns bit vectors for input tokens that match the function's name. A function recognizer acts as a set of synonyms for the function. For example, the recognizer for the *send-email* function recognizes send, attach, email, mail, and e-mail.

Each function argument specifies its type, whether it is required, whether it has a default, and how it should be displayed to the user before the argument is filled in. Often, this display name is simply the type name (such as *email address*), but it may also be more specific (*starting date*) or describe the default value (*today*), in order to help the user understand its role in the command and the effect of omitting it.

The body of a function is Chickenfoot code, which is Javascript augmented with some additional functions for automating web sites. When the function is run, this Chickenfoot code is evaluated in the current Firefox browser tab, and typically browses to a target web site and fills in forms with the function arguments.

A function generator takes a (possibly incomplete) list of bound arguments and returns a human-readable description of the function call as a string of HTML. These descriptions are displayed in Inky's feedback window as described in the previous section.

In the XML file defining a function, both the generator and the arguments are specified declaratively at the same time, using a template. Functions also have additional metadata, such as whether the function has a side-effect, the name of the side-effect (used to label the submit button), and a URL for the function's icon.

In addition to functions that represent web site operations, the XML file can also include functions that transform their arguments into a different type or a different value. These functions have a *return type*, so that the keyword engine can use them to construct nested function evaluations. (Top-level functions always have void return type, since they are run for their effect in the user's web browser, not

for transforming values.) The Inky prototype uses this technique to represent some natural ways to describe a date, such as *next month*, which transforms the value of the current month into the next month.

Keyword Interpreter

The keyword engine translates the user's command into a ranked list of interpretations. On initialization, it loads types and functions from XML files. When invoked, it takes the command string typed by the user, applies type and function recognizers to it, and uses a dynamic program to search for callable functions (with bound arguments) that best match the command. The resulting ranked list is passed back to the user interface to be displayed to the user and possibly invoked.

The keyword engine in Inky uses a variant of the bottom-up keyword matching algorithm described in [9]. Briefly, this algorithm uses dynamic programming to find the fragment of the user's input that best matches each type known to the system. The initial iteration of the dynamic program runs the type-matching recognizers to identify a set of argument types possible in the function. Subsequent iterations use the results of the previous iteration to bind function arguments to fragments that match each argument's types. Interpretations are scored by how many input tokens they successfully explain, and the highest-scoring interpretations for each type are kept for the next iteration. For example, three integers from a previous iteration may be combined into a date. The dynamic program is run for a fixed number of iterations (three in the Inky prototype), which builds function trees up to a fixed depth. After the last iteration, the highest-scoring interpretations for the void return type (the type used by web site functions) are returned as the result.

The Inky algorithm differs from that discussed in [9] in several ways. First, Inky uses bit vectors to represent matching command fragments. Inky does not allow two function or type recognizers to explain the same token in an interpretation, by ensuring that the bit vectors returned by the recognizers are disjoint. Thus, **find flight SFO** will not produce an interpretation that has SFO as both the departure and arrival airport.

The Inky algorithm also creates a separate bit vector for each recognized instance of a type or function, instead of splitting the score value between all matching tokens as in [9]. Although this decreases the efficiency of the system by creating many bit vectors to process (in the worst case, all possible subsets of the input tokens, if a type recognizer matches all of them), this change is essential for distinguishing multiple arguments of the same type. In practice, the commands and function trees are so small that exponential blowup doesn't hurt.

Finally, the Inky algorithm also includes a postprocessing step for tokens that were not used in the interpretation. Like Koala [7], Inky tries to account for these tokens by extracting arguments from them, particularly string arguments. The string type recognizer normally identifies strings only when they are explicitly quoted, as in **google "firefly tv show"**. To handle unquoted strings, the post-processing step locates the longest contiguous sequence of unused tokens, puts quotation marks around it, and resubmits the modified command to the keyword engine to see if a higher-scoring interpretation can be found using the new string argument. If so, that interpretation is used instead, and the postprocessing step is repeated until no more tokens can be consumed.

EVALUATION

The textual prototype was evaluated with a small field study. The purpose of the field study was to determine how Inky might be used in day-to-day activities and evaluate some of the decisions in its interface design.

The field study involved seven users, all members of our lab who regularly use Firefox, who used Inky over a period of approximately a week. The data gathered from the study shed light on learnability, accuracy, the importance of synonyms, the importance of order independence in commands, and the importance of suggestions.

For this field study, two different versions of Inky were released, in order to explore the effect of Inky's own feedback on the user's command ordering. The first version, InkyA, displayed all command feedback in the same standard order, with a command name first, followed by arguments in a consistent order. The same order was used for example functions displayed on startup and for the list of interpretations. The other version, InkyB, provided mixed feedback, sometimes with the command name at the front, and other times in the middle or at the end. The purpose of this variation was to investigate whether users mimicked Inky's feedback, or if there was instead a basic syntax users preferred regardless of how Inky prompted them. In every case, the command displayed was readable English.

Each user was directed to a website that gave instructions on how to download either InkyA or InkyB and install it. The website also stated the keyboard shortcut used to invoke Inky and that the user should type into the Inky textbox, but no other instructions were provided. Users were not told about the different versions of Inky, and command ordering was not mentioned in any of the instructions.

Inky logged all keystrokes entered into its textbox so that uncompleted or edited commands could also be observed. Over the course of the study, users typed 131 commands into Inky, and actually invoked 55 of these commands. The users were also interviewed after the study to capture their impressions.

Results

Inky was fairly learnable with minimal instructions. Out of the seven users, four later reported that they noticed the startup suggestions right away and learned commands through them. Nevertheless, all users successfully ran at least one command by trial and error. Only two of the sev-

en users ever ran a command in submit mode (which runs its side-effect automatically).

Alternative interpretations were used occasionally. Four of the seven users used the arrow keys to browse the interpretations. Out of the 39 commands executed by these four users, however, 35 were the top choice, and the other 4 were the second choice. These results suggest that Inky could display fewer suggestions to users without sacrificing accuracy. In post-study interviews, users mentioned that the suggestions often looked too similar, and it was often easier to change the command until the top suggestion was correct rather than visually scan the suggestions. It would be undesirable to eliminate multiple interpretations entirely, since the second-ranked interpretation was occasionally used, but Inky should decrease the number of suggestions made and differentiate them more clearly. Preliminary work in this direction is discussed later in this paper.

Out of the 131 commands collected in the study, 95 commands were correctly interpreted (i.e., the correct interpretation was one of the choices offered by Inky). Assuming every invocation of Inky was intended to issue a command, rather than merely explore it, this gives an overall accuracy rate of 73%. Much exploration did in fact occur: of the 36 commands that were not correctly interpreted, 21 were attempts to invoke functions that the Inky prototype did not provide, such as **find an apartment** and **order food**.

Of the correct commands, 16 used a synonym for a function keyword (i.e., a word that was defined in the XML files for that function but never appeared in the system's examples or feedback for that function). Of the 36 user commands that were not correctly interpreted by the system, 11 failed because they used a synonym that was not defined in the system. After adding the synonyms to the system post-study, those commands were interpreted correctly.

These results give strong evidence that synonym support is crucial to the system, which is well known from previous research on command languages [4], but rarely put in practice. With synonyms, the system could correctly interpret 81% of the commands in the study. Without synonyms, accuracy drops to 61%.

Since order independence plays a large part in the keyword engine implementation, it was useful to see if it actually improved the user experience. Requiring keywords and arguments to be provided in a specific order would make the command interpreter simpler and more predictable. The study collected 89 commands with useful ordering information. Commands that were a single word long or for functions that Inky does not support are not counted in this total. Of these 89 commands, 13 had an ordering that was inconsistent with the feedback Inky gave them. From users of InkyA, which always had the function keywords at the start and the arguments in a consistent order in the feedback, 57 out of 60 commands matched the order of the feedback. In the three remaining cases, the function keywords were reordered but parameters still appeared after the function keywords. To illustrate, one of Inky's examples is **see csail reservations tomorrow**. In one case, the user typed **csail see tomorrow** instead.

From users of InkyB, which had function keywords in different places and also reordered the arguments in different areas of feedback, 10 out of 29 commands did not match the order of the feedback. Seven of these commands had the function keywords at the start of the command, whereas the feedback Inky gave them had them in the middle or end of the command. Six of the commands had the arguments in an order that was inconsistent with the feedback. Three commands satisfied both conditions.

Although the size of the study is too small to establish statistical significance, the trend suggests that users may naturally put function keywords before argument keywords in a command. However, it should be possible for different function keywords to be reordered, and it is also important for Inky to support order independence among the arguments.

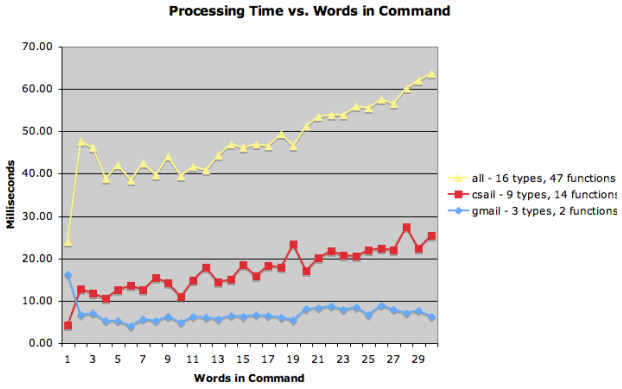

Figure 6: Keyword interpreter performance for different command lengths and database sizes.

Performance

The performance of the keyword engine was evaluated by running the commands acquired from the field study. On a 2GHz MacBook running Firefox 2 and Java 1.5, the average time to find the interpretations for a command was 15 milliseconds, and the maximum time was 55 milliseconds. These response times are more than adequate for running the keyword engine on every keystroke.

To determine how the keyword engine scales with very long commands, its performance was measured with commands of different lengths and with different databases of types and functions. Commands were generated by selecting 1 to 30 random keywords from the commands used in the field study. Figure 6 shows how processing time grows as a function of command length, for three different databases. In all cases, the response time is less than 70 milliseconds, which is adequate for the response to seem nearly instantaneous.

PINKY: INKY FOR PIM

Based on our experience with Inky, we are currently developing a second prototype, called *Pinky* (Personal Information Keywords) because its design focus is lightweight data capture for personal information management (PIM) applications.

Motivation

This work is motivated by our previous investigation into the prevalence of *information scraps* [1]: bits of information that fall between the cracks of personal information management tools. Information scraps are often scribbled on Post-it notes, on paper margins, buried in emails to oneself, or collected haphazardly in text files. Some information scraps exist because the user has no specific tool for managing them (*e.g.* guitar tabs or how-to guides). But other scraps are precisely the sort of data that PIM tools are designed to store. Our study of the scraps on knowledge workers' computers and physical desktops found that over 25% of scraps were to-do items or contact information, the two largest categories overall [1].

This result begs the question, what goes wrong? Why doesn't this information make it into a PIM tool? The cost of starting, navigating, and entering data in PIM applications is one reason why users turn to scraps, despite the difficulties of organization and refinding that information scraps will pose later [14]. Participants in our study reported the need for quick capture as a common reason for creating scraps. "If it takes three clicks to get it down, then it's easier to e-mail [a scrap to myself]," reported one participant. Another said, "Starting in Outlook forces me to make a type assignment, assign a category, set a deadline, and more; that takes too much work!" [1]

Starner et al. [12] found a similar effect for users of mobile PDAs and paper day planners. When prompted to schedule an appointment, almost half of PDA users and over half of day planner users wrote down information scraps (generally on bits of paper) rather than open up and navigate their calendars.

Our earlier Jourknow system [14] was designed to capture and manage information scraps, to catch these bits falling between the cracks of current tools and give them life. One finding from our studies of Jourknow, however, is that users are loath to abandon their current PIM tools, necessitating automatic synchronization between Jourknow and the universe of other PIM tools – a substantial engineering undertaking.

A Command Line for PIM

Pinky offers a possible solution to the lightweight capture problem. Pinky is a popup command line, like Inky, that allows the user to enter information quickly as text, then pushes the information into the appropriate PIM tool. By speeding up information capture, we hope to lessen users' need to create information scraps. The text is parsed using Inky-style keyword matching to extract PIM data (like to-do items, calendar events, or contact information), which is then filed immediately in the appropriate PIM tool. Since Pinky is based on Inky, it automates web-based PIM tools, including Google Calendar and Remember the Milk.

One consequence of focusing on fast data capture is that Pinky takes more responsibility for assisting data entry and providing contextual information, whereas Inky delegated these responsibilities to the menu-and-form interface it was automating. For example, if the user has trouble choosing a date parameter, the Inky philosophy would simply leave the date out of the command, and rely on the underlying web site form to display a calendar widget to make date picking easier. Pinky, on the other hand, aims to capture as much as possible *without* forcing the user to switch to the full PIM tool interface, so it incorporates a calendar widget directly in its own interface.

The rest of this section describes some of the new ideas embodied in Pinky: (1) using GUI widgets for choosing and changing arguments on the command line; (2) displaying relevant clippings from the backend web site while the user is entering a command; and (3) reorganizing the display of alternative interpretations to make them easier to scan and select.

GUI Widgets for Command Arguments

Sometimes arguments may be easier or faster to select from a GUI widget, such as a calendar picker, than to type. GUI widgets also inherently offer additional feedback that can reduce errors. For example, a calendar widget makes it obvious that April 12 is a Saturday, so it may not be a good day to schedule a work meeting.

The Pinky prototype incorporates three kinds of GUI widgets into its interface: people, places, and dates. The people widget () pops up an autocompleting list of people's names and email addresses, drawn from the user's email contacts. The places widget () has a similar list of relevant places, which for our environment are the rooms in our building. The date widget () is a conventional calendar widget. These three widgets are implemented in HTML and Javascript using the Yahoo User Interface library[6].

GUI widget buttons are incorporated into Pinky's feedback window, so that argument slots of the appropriate type (people, places, and dates) are automatically followed by the relevant button. Clicking the button pops up the widget to fill in the missing argument (or change the value already assigned to it by command parsing).

When an argument's value is set with a GUI widget, the command in the textbox automatically reflects the change as well, by either replacing the keywords that originally matched that argument or adding new keywords on the end. These keywords become *mandated variables*, which are prefixed by an argument name, as in **mtg 5pm**

[6] http://developer.yahoo.com/yui

with:emax at:G725. This syntax forces the keyword interpreter to use those keywords only for the specified argument. To avoid changing the user's command too drastically, this extra syntax is normally hidden, and mandated variables are shown by underlining them as in **mtg 5pm emax G725**. Clicking on the mandated variable expands it into its full syntax. The user can also directly type the syntax for mandated variables, but this requires the user to learn and remember names of argument.

Web Clips

GUI widgets like the calendar widget provide generic support for entering arguments accurately, reminding the user for example that April 12 is actually a Saturday and that msbernst@mit.edu is the right email address. For more personalized context, Pinky uses *web clippings* extracted from relevant web sites. These web clippings bring just-in-time information to the user, anticipating the user's information needs so that the user does not need to break off the entry to consult less efficient menu and form interfaces, or worse, choose not to record the information at all.

For example, when the user is scheduling a calendar event or reserving a conference room, Pinky automatically pops up a clipping of the user's calendar (Figure 7a) for that day, to help confirm the date and time of the meeting and avoid overbooking. When the user is sending an email, Pinky shows a clipping of recent emails exchanged with the intended recipient (Figure 7b). Clippings appear as satellites around the main Pinky popup window.

Pinky's clippings are reminiscent of WinCuts [13] for desktop windows, and Apple Web Clips[7] and Web Tracker [5] for web pages. Unlike these systems, however, Pinky extracts a clipping not by retrieving a single URL and extracting a snippet of HTML or a screenshot, but instead by automating a web application with Chickenfoot until it reaches the desired state. The resulting web page can thus be customized much more dramatically than with other tools; for example, by showing the calendar focused on the time under consideration, by displaying only e-mails exchanged with the person of interest, or by skinning the conference room schedule down to only the room under consideration rather than a large matrix showing all rooms.

To render a web page as a clipping, it is displayed in an HTML <iframe> element set to reasonable browsing dimensions (800x600), which ensures that the clipping is rendered in a familiar and readable way. The desired region in the page is located (*e.g.*, a single day in Google Calendar), and its bounding box is used to clip the <iframe> by positioning the frame appropriately inside a viewport element, a of the appropriate width and height.

Since the clipping is a live rendition of the underlying web application, it can update immediately when the user changes arguments in the command, like the day of a meeting. To interact with the web application directly, the user can click on the clipping, which expands the <iframe> to make the whole web page visible. Clicking away from the <iframe> shrinks it back to a clipping again.

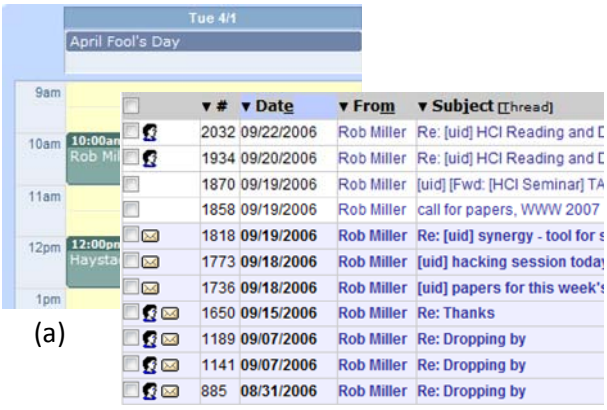

Figure 7: Web clippings of (a) the user's calendar and (b) webmail, displayed automatically when the user is entering a relevant command.

By automatically navigating web applications and showing appropriate clippings, Pinky helps bridge the gap between the user and the web application. For PIM data capture, clippings bring useful bits of the PIM tool out to the user, on demand, rather than requiring the user to find their own way into the tool.

Automatic clippings suggest another use for Pinky – not just a shortcut for data capture, but for *queries* as well. By typing a partial command, like **apr 17**, the user can immediately bring up a web clipping with useful information, in this case their calendar for that day.

Organizing Multiple Interpretations

One observation from the Inky user study was that the list of alternative interpretations was rarely used. Several study participants expressed the concern that the alternatives on the list often looked very similar, which made them hard to compare. For example, the top few choices may all be the same function, differing only in how the user's keywords are assigned to arguments. As a result, it often felt easier to change the command until the top suggestion was right, rather than visually scan the list of suggestions.

The Pinky prototype has a new feedback interface aimed at addressing this problem (Figure 8). The suggestion list is categorized by function, indicated by the tabs on the left, so that each function that matches the command appears only once. Within each command, the alternative parses for each argument are shown in a drop-down list under the argument, which the user can select.

[7] http://www.apple.com/macosx/features/safari.html

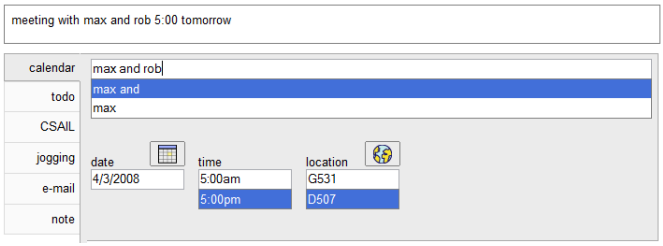

Figure 8: Pinky feedback window, grouping alternative command interpretations into tabs and alternative arguments into drop-down lists.

This interface allows the user to select the right interpretation by focusing on just one component at a time: first the function name (by picking a tab), then each argument (by picking from drop-down lists). Each choice may cascade to other choices, since the keyword interpreter does not permit two arguments to use the same keyword. For example, when the user uses the interface in Figure 9 to indicate that G531 is the location of the calendar event, G531 can be removed from the list of guesses for the title of the meeting. The new interface also incorporates the GUI widgets mentioned earlier (Figure 9 includes the calendar widget and the location widget).

This interface is still under development, and has yet to be tested, but we hypothesize that it will make alternative interpretations easier to understand and select.

CONCLUSION AND FUTURE WORK

We have presented Inky, a command line for the Web with sloppy syntax and rich visual feedback, and Pinky, which extends the idea to PIM data capture. Inky and Pinky exploit the fact that they are built on top of menu-and-form web interfaces to improve the usability of a command line without giving up its efficiency benefits. This is, in a sense, a reversal of history, since early graphical user interfaces were often façades built on top of existing command line interfaces. Now that graphical user interfaces are the standard, we believe it's desirable to build command lines on top of them, to provide fast data entry and reduced navigation, with the GUI still providing constraints and guidance where necessary.

One avenue of future work would be allowing end users to extend Inky with new web site functions, which currently requires editing the XML files. For example, Inky could be connected to a system that records web macros, like Smart Bookmarks [6] or Koala/CoScripter [7], so that the user could use the Inky command line for intranet web sites or very personalized tasks. A connection to CoScripter would be particularly interesting, because CoScripter's public wiki is evolving into a web-scale database of procedural knowledge, "how-to" scripts for a variety of web sites and tasks. From that perspective, Inky acts like a keyword search interface for procedural knowledge – but a search that doesn't just *find* the how-to script, but also fills in arguments and runs it. If CoScripter succeeds in building the database, a command line like Inky may be a convenient interface for exploiting it.

The Inky and Pinky prototypes are still under development, but will be released as publicly-available, open-source software during the summer of 2008.

ACKNOWLEDGMENTS

We thank members of the UID group who provided valuable feedback on this paper, and study participants for their time and comments about Inky. This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer under the T-Party project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

1. Bernstein, M., Van Kleek, M., Karger, D.R., and schraefel, mc. Information scraps: how and why information eludes our personal information management tools. *ACM TOIS*, to appear.
2. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. Automation and customization of rendered web pages. In *Proc. UIST 2005*, 163-172.
3. DiGiano, C. and Eisenberg, M. Self-disclosing design tools: a gentle introduction to end-user programming. In *Proc. DIS 1995*, 189-197.
4. Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The vocabulary problem in human-system communication. *CACM* 30, 11 (Nov 1987), 964-971.
5. Greenberg, S. and Boyle, M. Generating custom notification histories by tracking visual differences between web page visits. In *Proc. Graphics Interface 2006*, 227-234.
6. Hupp, D., and Miller, R. C. Smart bookmarks: automatic retroactive macro recording on the web. In *Proc. UIST 2007*, 81-90.
7. Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., and Kandogan, E. Koala: Capture, share, automate, personalize business processes on the web. In *Proc. CHI 2007*, 943-946.
8. Little, G., and Miller, R. C. Translating keyword commands into executable code." In *Proc. UIST 2006*, 135-144.
9. Little, G. and Miller, R. C. keyword programming in Java. In *Proc. ASE 2007*, 84-93.
10. Miller, R. C. and Myers, B. A. Integrating a command shell into a web browser. In *Proc. USENIX 2000*, 171-182.
11. Shneiderman, B and Plaisant, C. *Designing the User Interface*, 4th ed. Pearson/Addison-Wesley, 2005.
12. Starner, T., Snoeck, C., Wong, B., and McGuire, R. Use of mobile appointment scheduling devices. In *Proc. CHI 2004*, 1501-1504.
13. Tan, D.S., Meyers, B., and Czerwinski, M. WinCuts: manipulating arbitrary window regions for more effective use of screen space. In *Proc. CHI 2004*, 1525-1528.
14. Van Kleek, M., Bernstein, M., Karger, D.R., and schraefel, mc. GUI --- phooey!: the case for text input. In *Proc. UIST 2007*, 193-202.