GNOME Do: an Ontology-Informed, Search-Driven Command Interface for the GNU/Linux Desktop

David Siegel
djsiegel@seas.upenn.edu

Douglass Colkitt
colkitt@seas.upenn.edu

Faculty Advisor:
Kostas Daniilidis

University of Pennsylvania

Abstract

The typical computer user interacts with a number of different resources on her computer. These resources are accessed via many interfaces, including menus, location bars, icons, file browsers, and shortcut keys. We plan to consolidate these interfaces by creating an application that indexes items found in the user's desktop environment (e.g. documents, contacts, applications, multimedia) and lets the user search for these items, and perform common actions on these items (e.g. open, run, email, play). Our goal is to optimize our search using, among other techniques, information about items considered as members of type "ontologies" and as individual entities.

Of special note, our project will be free and open source, which means that all specifications, source code, documentation, and other project resources will be publicly available on the Internet for anyone to scrutinize at any stage in our development process. We will publicize our project and encourage others to participate by contributing bug reports, code, documentation, etc.

1 Introduction

A general observation can be made that experienced computer users often utilize keyboard-driven interfaces such as shortcut keys and command terminals, while less experienced users are often more comfortable with cursor-driven interfaces such as menus and buttons. Keyboard-driven interfaces allow the user to execute common tasks quickly; however, these interfaces tend to confuse inexperienced users due to a dearth of graphical representations. For example, in the case of shortcut keys, there is no graphical representation to accompany the pressing of key combinations. This may make it difficult for novice users to learn new shortcut keys or to have confidence that the key combinations they enter are correct. Keyboard-driven interfaces also intimidate novice users due to poor semantic associations between keyboard commands and names that the user is familiar with. For example, the keyboard shortcut for "paste" is Control-V, and the command-line program to "delete" or "trash" an item can be either *rm* or *rmdir*.

Our intent is to create an interface that takes advantage of the precision and expressiveness of the keyboard, while remaining intuitive enough to appeal to novice users. We will accomplish this by creating an application that indexes the items found in one's desktop environment, including documents, contacts, bookmarks, applications, notes, multimedia and more. We will then present graphical representations of these items to the user, allowing the user to search through and interact with these items using an interface that is keyboard-driven *and* graphical. In addition to user interface challenges, principle technical challenges facing this project include indexing and responsive searching of items in a user's desktop environment, and implementing appropriate techniques for dealing with items of changing relevancy to the user.

An additional goal is to develop our project as a free and open source application. This will require us to become familiar with tools, engineering practices, organizational techniques, and management skills that are completely new to us. We hope to build a strong community of contributers and users around our application so that, unlike the majority of completed senior theses that are relegated to trash cans or dusty filing cabinets, our work will continue to flourish, providing enduring value to perhaps millions of people. We also hope to get Penn underclassmen interested in using GNOME Do as the basis for future senior projects!

2.1 Related Work: Quicksilver

Quicksilver, a program produced by Blacktree Software, is the primary inspiration for our application; in fact, we have mimicked Quicksilver's user interface because we are familiar with Quicksilver and believe it is an excellent starting point. Quicksilver consists of an interface with two (optionally three) large icons: the first icon represents and item that the user has searched for, the second icon represents the action the user has selected to perform on that item, and the third icon represents an optional “indirect item,” which, if present, modifies the behavior of the action. Quicksilver also has a plug-in architecture that allows the application to be extended with new items and actions.

Our project differs from Quicksilver in that we will take a more thorough approach to maintaining working sets of searchable items, and to arranging the contents of these sets to account for changing item relevancies (see “Technical Approach”). Judging from a casual review of Quicksilver's source code[1] (note that Quicksilver's source code is undocumented, and contains years of antiquated code preserved in comments, so discerning exactly what is happening in any section of the code is difficult), Quicksilver simply maintains a list of all items, which gets filtered and sorted when searches are executed by the user. This results in an asymptotic running time of *O(n log n)* in the typical usage case, where *n* is equal to the total number of items indexed. In order to guarantee an upper bound on search time and thereby provide responsive search for the user, we will apply different indexing techniques (see “Technical Approach”).

Also, Quicksilver allows the user to manually specify “mnemonics” for items; for example, one might choose the mnemonic “thesis” for a file named “Pre-Execution via Speculative Data-Driven Multithreading.tex.” This way, one could simply type “thesis” to locate this item. From our experience, this feature is underutilized due to obscurity, cumbersomeness, and poor integration into Quicksilver's ranking algorithm. We hope to address this issue by creating an “alias” command that operates on items *within* the normal workflow of our application, and by capturing the relevancy information discussed in our “Technical Approach” section.

1 http://blacktree-alchemy.googlecode.com/svn/trunk/Quicksilver/Framework/Code/QSitemRanker.m, QSLibrarian.m

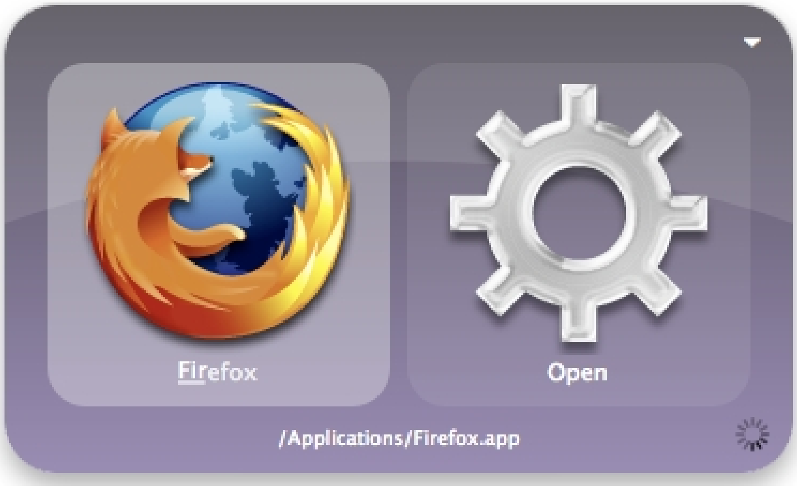


Figure 1: Quicksilver's "Bezel" interface

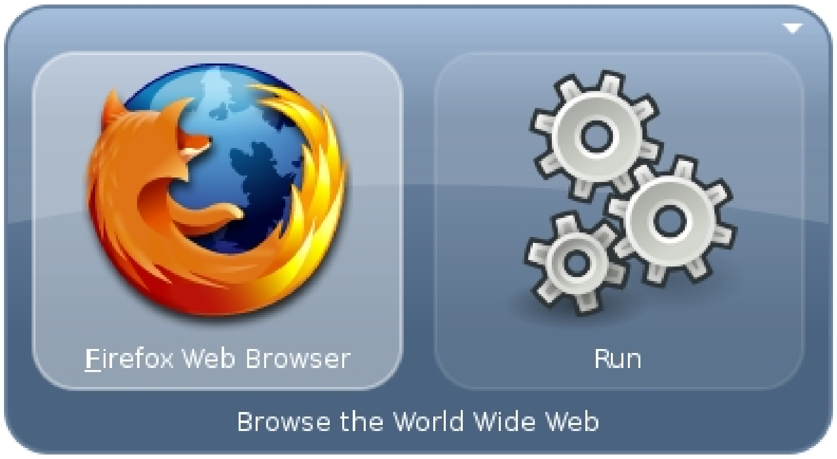


Figure 2: GNOME Do's "Classic" interface

Quicksilver has an extensive configuration interface through which the user can customize the behavior of the application. For example, one can benefit greatly from configuring Quicksilver to index the contents of one's Documents folder. This is done by opening the Quicksilver preferences window, navigating to the "Catalog" configuration section, then navigating to a subsection of that section, and clicking on a small button on the bottom edge of the window. This button reveals a menu containing the option to add a "File & Folder Scanner," which allows the user to navigate to her Documents folder in another window and finally choose that folder for “scanning.” After this, the user must navigate to a pane (a separate window attached to the side of the Quicksilver preferences window) and check a box indicating that the user would like to index the contents of her Documents folder. Next, the user is presented with a slider, with values ranging from 1 to infinity, which the user slides to indicate the depth to which she would like the contents of her Documents folder to be indexed (infinity indicates no depth limit). Obviously, requiring the user to complete this task adds tremendous complexity to Quicksilver, precluding all but technically savvy users from deriving even moderate benefit from the application. We will avoid this poor design by allowing users to browse for items not included in our index within GNOME Do's main application interface, and by applying the work of another group (see “Collaboration”) to intelligently determine which items need to be added to the index. In the ideal case, GNOME Do will use information derived from diverse sources (e.g. recently-opened documents lists, open files) to decide that a user is highly likely to be interested in the contents of her Documents folder, so GNOME Do will simply index those files automatically.

Finally, GNOME Do will be designed for GNU/Linux instead of Mac OS X, and the source for GNOME Do will be made freely available under the GNU General Public License (GPLv3). Quicksilver lacks documentation and until recently was not open source, which means that writing plugins for Quicksilver has been unnecessarily difficult. Presumably, this is improving now that Quicksilver's source code is freely available. Our system will provide third parties with complete source code and documentation from the beginning; in fact, we have already accepted fully implemented new features, patches, and plugins from around twenty contributors.

2.2 Related Work: GNOME Launch Box

GNOME Launch Box (GLB), developed by Imendio, is a Quicksilver-like launcher for GNU/Linux. Our original intent was to use GLB as a base for our application; however, the maintainers of GLB were unresponsive to patches we submitted, and they made an explicit statement that they were uninterested in developing GLB beyond its current form as a crude application launcher. Also, GLB is written in C, which we felt would bog us down in technical details, preventing us from making enough progress on GNOME Do during this academic-yearlong project.

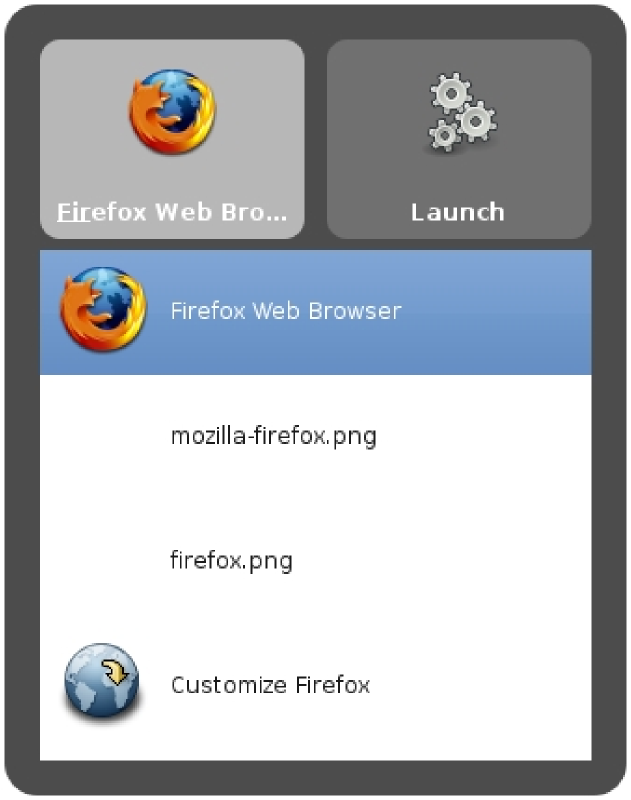


Figure 3: GNOME Launch Box

Our project will make up for GLB's shortcomings by using managed code in a common language runtime (CLR) which will give contributors greater flexibility in extending our application by allowing them to write plugins in any language supported by the CLR. We will develop our code using Mono, a free and open source implementation of Microsoft's .NET platform. Mono provides a C# compiler, a large collection of libraries, and a virtual machine for executing bytecode compiled from any supported language. Using these high-level tools, we will be able to focus more on the more interesting problems of search and user experience, and less on the nitty-gritty “gotchas” that we would encounter if we were to use C.

Another important difference between GNOME Do and GLB is the difference in indexing techniques. GLB has no indexing; instead, it queries each one of its “modules” (equivalent to what we call “plugins”) with the user's input, concatenates the results returned by each module in response to the query, and presents the results to the user. Deskbar Applet is another popular GNU/Linux search tool that takes this naïve approach. Our plugin API, on the other hand, requires that plugins publish searchable items up front so that GNOME Do can take full responsibility for searching and ranking items for the user.

2.3 Related Work: Entity Resolution

We encountered an interesting, unforeseen problem while working with multiple sources of contact data for items in GNOME Do: what should we do when we have duplicate contact items representing the same individual? How can we identify and merge these contact items into single identities? We learned from a Standford University paper entitled, "Generic Entity Resolution in the SERF Project," that this problem is referred to as "Entity Resolution" (ER) or "deduplication." Many of the techniques discussed in the paper, such as techniques for distributing large-scale matching and consolidation computations across multiple processors, were not directly applicable to our small datasets; however, this paper helped us frame our problem of deduplicating contact records.

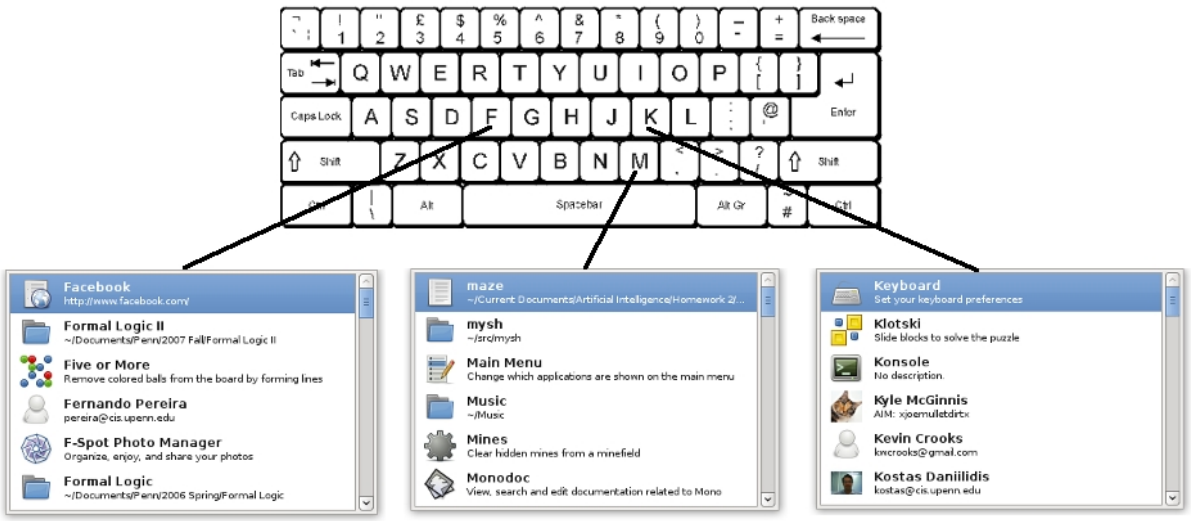
Our match criteria consist of contact attributes which, if identical, relate two contact records as duplicates of each other. These attributes include full names and email addresses. Due to our relatively small datasets, we are able to match duplicate contact records by hashing contact attributes and looking for collisions. To consolidate duplicates, we simply merge colliding records into a single, representative record.

3 Technical Approach

Our application maintains a type "ontology" modeling the resources users most frequently interact with: applications, bookmarks, documents, contacts, multimedia, etc. A separate ontology models the actions users perform on these items: launch, run, open, email, chat, etc. Each action contains information about the types of items it can be performed on. We have created a plugin system which allows our application to be extended with new items and new commands.

To address the problem of providing responsive search, we set out to learn appropriate indexing and caching techniques for presenting the user with a search interface that feels instantly responsive. GNOME Do can index a virtually unbounded number of items, and we would like GNOME Do to be portable enough to work in environments of varying requirements and capabilities, including personal computers as well as low-power mobile devices such as the OpenMoko phone platform. Our research revealed that in many applications similar to ours, the the limiting factor on search responsiveness is memory or cache technique-related (Xiao, Zhang, Kubricht); also, different sorting techniques have different performance depending on the working set size, memory architecture, and processor speed (Ibid). When using a prefix tree, it was found that sorting the data beyond a certain depth led to a tradeoff between greater speed, but less space efficiency and more cache misses (Sinha, Ring, Zobel). The prefix tree that our program uses to prepare results goes to a depth of one, mapping single character keypresses (e.g. 'a', 'g') to sorted lists of items.

In our experience, users will not tolerate much more than twenty milliseconds of sluggish search behavior before developing an unfavorable opinion about the performance or usefulness of a desktop search tool, so we formulated our searching strategy to have an upper limit on response time that is independent of the number of items indexed. Our sorting technique also takes into account the relevance of individual items to the user. This allows to prepare results that are most likely to be relevant to the user *before* the user initiates a search.



```
Facebook
http://www.facebook.com/
Formal Logic II
~/Documents/Penn/2007 Fall/Formal Logic II
Five or More
Remove colored balls from the board by forming lines
Fernando Pereira
pereira@cis.upenn.edu
F-Spot Photo Manager
Organize, enjoy, and share your photos
Formal Logic
~/Documents/Penn/2006 Spring/Formal Logic
```

```
maze
~/Current Documents/Artificial Intelligence/Homework 2/...
mysh
~/src/mysh
Main Menu
Change which applications are shown on the main menu
Music
~/Music
Mines
Clear hidden mines from a minefield
Monodoc
View, search and edit documentation related to Mono
```

```
Keyboard
Set your keyboard preferences
Klotski
Slide blocks to solve the puzzle
Konsole
No description.
Kyle McGinnis
AIM: xjoemuletdirtx
Kevin Crooks
kwcrooks@gmail.com
Kostas Daniilidis
kostas@cis.upenn.edu
```

Figure 4: The most relevant results for 'F', 'M', and 'K'.

When GNOME Do is started, it creates a 1-tier prefix tree that maps 26 letters of the English alphabet as found on a standard QWERTY keyboard to a list of items sorted by relevance, with a maximum sorted list length of 1000 items. When the user types the first letter in a search, GNOME Do retrieves the list of results predetermined to be most relevant for that keystroke (Figure 4). When subsequent letters are typed, the search algorithm takes the previous list of results and re-sorts the items by relevance with respect to the updated query, excluding items whose relevance drops to zero. Since the lists of prepared search results are capped at 1000 items, the number of items being searched and sorted is never greater than 1000, no matter how many items the program indexes.

GNOME Do's plugin architecture allows any item to have children. For example, a folder item has all the files contained in the folder as its children. This makes it possible for an item's children to exist outside the indexed "universe." For example, if a folder were only indexed 1-level deep, then a file contained in a subfolder of that folder would be an item that the user can discover outside of GNOME Do's searchable index. We use this capability to discover important features of the user's environment to index.

We have made some significant assumptions about how people use and refer to named resources on their computers. First of all, our search implementation relies on the assumption that people know the name of the item they are looking for. Our string scoring algorithm (we currently use Quicksilver's scoring algorithm for easy comparison with Quicksilver) takes advantage of this assumption, placing a higher weight on letters that appear in the beginning of the name or the beginning of words in the name. Also, by limiting the number of search results returned for any query to a maximum of 1000 items, we assume that users cannot name more than 1000 items in their desktop environment for any given character on their keyboard. We have found that these assumptions require slight changes in user behavior. For example, many users are quick to configure GNOME Do to index tens of thousands of files on their computer. We often have to explain to users that they are not actually interested in instant access to most of those files, and we suggest that they limit the index to files they open at least once a year—this is under a couple hundred files for most users, and fewer than a couple thousand files for even the most avid users. Also, many users have been trained by other search tools to search by content rather than by name, so they feel that searching by name is strange. We argue that searching by name makes sense for small datasets where the majority of the content is originated and therefore named by the user (e.g. contacts, documents, bookmarks). We encourage users to think more carefully about the names they assign to resources on their desktop. This makes it easier for users to find what they are looking for using GNOME Do, and has the nice side effect of making people more cognizant of and proactive about the organization of their desktop.

To address the issue of item relevancy, we had planned to implement a scoring system that does type-based, global prioritization of items considered as members of our type "ontology," and token-based, local prioritization of items considered as individual entities. By "global," we mean across all sets in the prefix tree. By "local," we mean specific to a single set in the prefix tree. The figure below illustrates the type heirarchy for ImageFileItem, a type representing an image file in the user's filesystem. Suppose one were to search with the query "por" for the item "Family Portrait.jpg," and perform the action "Rotate 90° Counterclockwise" on that item. GNOME Do will increase the relevance of each type on the path from ImageFileItem to the root of the item type hierarchy, IItem. This results in global, type-based prioritization because these type-based relevance scores are used to rank all items in the prefix tree. The relevance for the "Rotate 90° Counterclockwise" action will also be increased. If the next search is for "face" and the user has two items with equal string similarity for the query, such as a bookmark for "face.com" and an ImageFileItem for "face.jpg", the type-based prioritization will ensure that the image is considered more relevant than the bookmark.

As far as token-based, local prioritization goes, the "Family Portrait.jpg" item will have its individual relevance increased in addition to the relevance increase it accrues from being a member of the ImageFileItem type; however, this token-based relevance will only be considered for searches beginning with "p", as this was the initial keypress in the search resulting in the change in token relevance. Constraining the domain of token-based prioritization to individual working sets in the prefix tree increases differentiation among all working sets, thus increasing the number of distinct items indexed. For example, "Family Portrait.jpg" could easily find itself among the most relevant items in the sets F, A, M, P, O, R, etc., contending with other items for occupancy even though the user may never visit any set but P when searching for this item. The next time the user searches for “Family Portrait.jpg,” she may only have to type “p” (or “po”) because of the item's increased relevance along that search path.

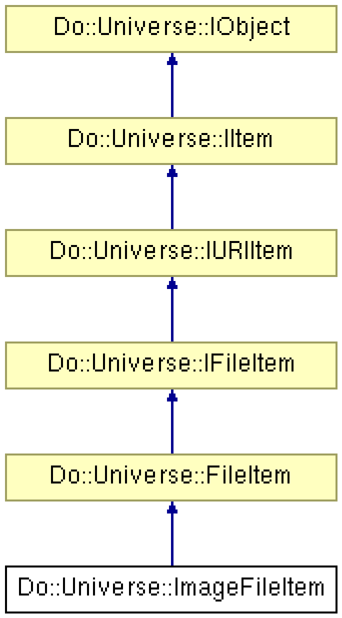


Figure 5: GNOME Do's type information about ImageFileItem

We had counted on work from another group that would give us relevance scores to implement these features with (see “Collaboration”). For unknown reasons, this work was either never completed or we were not told about it, so for the time being we have implemented a much simpler relevance scoring system that maintains two global histograms, one for items and one for actions. Every time an item or action is deemed relevant by some user action, the appropriate histogram is updated to reflect the increase in relevance. These histograms decay over time to allow new items and actions to become more relevant than old items and actions. Although this implementation is the simplest we could conceive of, we have received a large amount of positive feedback about the relevance predictions that GNOME Do is able make. This has led us to suppose that the type/token relevance implementation may not yield enough noticeable advantage to merit the increased complexity. In any event, we have created a modular ranking pattern that allows us alter GNOME Do's relevance algorithms by simply subclassing a single class, so it would be easy to experiment with alternate relevance scoring strategies.

Two great challenges we faced were learning to use the necessary tools and libraries (see “Resources Required”), and coordinating our efforts with the community in order to make this a successful open source project (see “Open Source Methodology”). To help bring us up to speed with these aspects of our project, Sean Egan, the lead developer of Pidgin Internet Messenger, has agreed to mentor us. Pidgin is an extremely successful open source project.

4 Open Source Methodology

An important aspect of our technical approach is the methodology by which we planned, organized, and developed our application. GNOME Do is an open source project, which means that all specifications, source code, documentation, and other project resources are available on the Internet for anyone to read or compile at any stage in our development process. This allowed us to collect feedback such as feature suggestions and bug reports from members of the community. This approach has been a double-edged sword, however: by inviting people to scrutinize and interact with us and our work, the quality and thoroughness of our project was made higher, but managing community interaction added a large, organizational dimension to our project that most senior design projects do not have to deal with. This is why we asked Sean Egan to mentor us. We hosted our project on Launchpad.net, a popular site for open source projects emphasizing community participation and collaboration. Our development page is http://launchpad.net/do. This page contains a detailed account of all of the planning and work that has gone into our project. Our project homepage is http://do.davebsd.com.

Here are some details about community participation for the seven-month duration of our project: at the end of the first semester spent working on the project, we had committed 100 revisions accounting for 9,000 lines of code. Four months later, we now have committed 335 revisions accounting for 24,000 lines of code. 162 bugs have been reported. We have received approximately twenty patches from external contributors, the first of which was contributed by Miguel de Icaza, VP of Developer Platform at Novell. Miguel founded the GNOME and Mono projects, and is one of the most influential members of the GNU/Linux desktop community. Miguel expressed great enthusiasm about our project in an email exchange and a post on his personal blog. GNOME Do was blogged about a few times by Jorge Castro on http://planet.ubuntu.com, a popular aggregation of blogs of Ubuntu developers. We have 182 people on our mailing list (up from 30 last semester). Chris Halse Rogers joined our project as our official Ubuntu package maintainer. Chris oversees an online repository containing binary packages of our project, keeping users updated with our newest releases. Our most recent release was 0.4.2, released on April 15, 2008. GNOME Do now has packages in every major GNU/Linux distribution, and is even installed by default in Foresight Linux and a few others. Shuttle, a boutique PC retailer, is now selling a line of low-cost Linux PCs that have GNOME Do running on them by default. Judging from our interactions with users and contributors, we are fairly certain that we have around 50,000 users at this point (up from a couple hundred last semester). We consider GNOME Do a remarkable success for seven months into our first open source project, after just recently switching to Linux and learning to use C#, Mono, and GNOME as we went along. This is a testament to the liveness and receptiveness of the free software community, and the flexibility and ease of use of tools like Bazaar, Launchpad, and most notably Mono.

5 Collaboration

As of last semester, we had barely begun to collaborate with Ian Cohen and James Walker's project for scoring items for relevance. We provided them with our project source code and enough information for them to start sharing data between their application and ours, but no significant collaborative efforts had taken place at that point. Ian and James had planned to develop, by the end of last semester, the item type ontologies which we would use in GNOME Do, but unfortunately they never delivered these. As discussed earlier, we were still able to implement historical relevance scoring. We understand that Ian and James were not obligated to work on a project as large in scale as ours, but we are utterly disappointed that they failed to collaborate in any sense whatsoever. They may have changed the direction of their project, but they never informed us of these changes and they failed to respond to emails sent to them in this regard.

6 Resources Required

We implemented GNOME Do on two Apple MacBooks running Ubuntu 7.10 “Gutsy Gibbon.” Software and services required for this project include: Mono (C# compiler, libraries, virtual machine), MonoDevelop (Mono IDE), autotools, Launchpad.net (project management), Bazaar (source code management), and assorted libraries (GTK+, GNOME, DBus). All of these components are free and open source software, so their greatest cost was the time it took to learn to use them.

7 Timetable

By the end of the fall semester, we planned to have accomplished:

- the symbolic application interface (like Quicksilver, GNOME Launch Box)

  Completed. Two additional symbolic interfaces have been created thanks to Jason Smith, an undergrad at the Western Michigan University. Jason has been the most prolific outside contributor.

- indexing and search of items and commands

  Completed.

- pre-fetching search results for apparently instantaneous search

  Completed.

- a version “0.5” public release, packaged for Ubuntu 7.10 and available as source for other GNU/Linux distributions

  Completed above and beyond our original expectations (online package repositories, volunteers maintaining packages). UPDATE: GNOME Do is now packaged for Debian, Fedora, OpenSUSE, Foresight Linux, Gentoo, and all other major GNU/Linux distributions.

- plugin API with thorough online documentation so that third parties can add items and commands to our application

  Completed. Tens of useful plugins have been contributed up to this point: https://wiki.ubuntu.com/GnomeDo/Plugins

- choose a free software license for our source code and explain our rationale for choosing that license

Completed; we have chosen the GPLv3 as our open source license. This license is endorsed by the majority of GNU/Linux projects.

By the end of the spring semester, we planned to have accomplished:

- the natural language interface, which will allow users to manipulate items and commands with a more free-form command grammar

  We feel that the natural language problem that we originally conceived and thought we would address is contrived. With command interfaces like the one we developed, it seems that a non-natural "command grammar" is preferred by computers and humans. We feel that this task would also make our project too wide in scope, as we have our hands full with the search, relevance scoring, and other features.

- improve upon search techniques and responsiveness developed earlier

  Completed.

- a core set of plugins providing access to applications, files and directories, web browser bookmarks, and address book contacts

  Completed with many additional plugins.

- conduct a usability study with both novice and advanced users to gather information for comparing the interfaces have developed

  Not completed. Our request to have CETS install the required software on lab computers was completely ignored.

- a version "1.0" public release, packaged for Ubuntu and available in source code for other GNU/Linux distributions

  Our current stable release is 0.4.0.1, and our next stable release, 0.8, will add online plugin repositories and superior plugin management via the Mono.Addins framework; graphical configuration options for plugins; and a lineup of fancy new plugins. We expect to finish 0.8 sometime this summer.

- consider package submission to the Ubuntu project so that our application will be included in Ubuntu repositories

  GNOME Do is included in Ubuntu 8.04's universe (community maintained) repository. This means that Ubuntu users can very easily install GNOME Do.

- work with Kostas Daniilidis to determine feasibility of porting project to a cell phone

  (Status as of last semester is unchanged) Due to two significant delays, we do not know when a stable release of the OpenMoko (GNU/Linux-based cell phone) platform will be available. Since it seems like we won't get our hands on the phone for at least another month, and once we get the phone we have to learn an entirely new toolchain for porting and developing GNOME Do, the feasibility of running GNOME Do on a cell phone by the end of the spring semester is low.

8 References and Works Cited

Source code of Banshee Media Player (2007) *GNOME Subversion source code repository.* Retrieved 01 Nov. 2007 <http://svn.gnome.org/viewvc/banshee/trunk/banshee/src/Core/Banshee.Core/>.

Reading the source code to Banshee helped answer a lot of our early questions about writing and organizing our Mono application. We borrowed parts of our logging and namespace design from Banshee. Banshee is perhaps the most high-profile Mono application, with many strong developers working on it, so we found the code to be of unparalleled quality.

Benjelloun, O., Garcia-Molina, H., Kawai, H., Eliott Larson, T., Menestrina, D., Su, Q., et al. (2007) Generic Entitiy Resolution in the SERF Project. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering.*

This article granted us exceptional insight into the problem of entity resolution, which we encountered for the first time when confronted with the task of deduplicating contact resources in a user's desktop environment. The article is clearly recent, and the bulletin in which it is published stands in high regard.

"GNOME Launch Box Architecture." (2007) *Imendio Developer Pages*. Retrieved 24 Sept. 2007 <http://developer.imendio.com/projects/gnome-launch-box/architecture>.

In addition to reading the entire C source of GNOME Launch Box, this document provided a brief, high-level overview of the relevant application architecture. We found this document to be a bit speculative, in that it alluded to features of GNOME Launch Box that have not yet been implemented; nevertheless, this source helped us determine the shape our project would take (and would not take) early on. This source was also a bit dated, although GNOME Launch Box is a stagnant project so we expected this would be the case.

Jitkoff, Nicholas. "Quicksilver: Universal Access and Action." (2007) *Google Video*. Retrieved 24 Sept. 2007 <http://video.google.com/videoplay?docid=8493378861634507068&q=google+tech+talk&total=668&start=0&num=10&so=0&type=search&plindex=8>.

An excellent presentation of the "philosophy" behind Quicksilver. This talk motivated much of the attention we payed to user behavior. Jitkoff talks a lot about the merits of universal resource access, and he also touches on the possbility of applying more advanced machine learning techniques to Quicksilver. We had hoped to explore some of these more advanced learning techniques through collaboration with Ian and James (see “Collaboration”).

"Quicksilver Documentation." (2007) *Blacktree Software*. Retrieved 22 Sept. 2007 <http://docs.blacktree.com/>.

Extremely useful in giving us a broad understanding of the requirements of our plugin system. This documentation also provided us with the string scoring implementation (which we ported from Objective-C). Although this source was authoritative since it originated from the author of Quicksilver, it is extremely rough and incomplete.

Xiao, L., Zhang, X., And Kubricht, S. A. (2000) Improving Memory Performance of Sorting Algorithms. *ACM Jour. of Experimental Algorithmics 5*, 3.

Exposed us to an excellent discussion on the memory-performance trade-offs that we would have to make throughout the development of our application. Well composed. Reputable source.