GNOME Do: an Ontology-Informed, Search-Driven Command Interface for the GNOME Desktop Environment

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 and programs on her computer, all of which are accessed by disparate means, including menus, location bars, icons, shortcut keys, etc. We plan to consolidate these interfaces by creating an application that indexes items in the user's desktop environment (documents, contacts, bookmarks, applications, multimedia, etc.) and lets the user search through these items and manipulate these items with commonly performed actions (open, run, email, chat, etc.). Our goals are to optimize our indexing techniques for instantaneous search using, among other techniques, information about items considered as members of type "ontologies" and as individual entities.

1 Introduction

Experienced computer users frequently utilize keyboard-driven interfaces such as shortcut keys and command terminals to perform common tasks quickly. Keyboard-driven interfaces such as these allow the user to execute more complex or precise actions more rapidly; however, these interfaces tend to confuse inexperienced users due to poor graphical representations of items—sometimes there is no visual interface, as in the case of shortcut keys. These interfaces also intimidate and alienate novice users due to unfamiliar item identifiers—for example, the shortcut for “paste” is Control-V, and the command-line program to “delete” or “trash” an item is *rm*.

Our intent is to create an interface that takes advantage of the precision and expressiveness of the keyboard, but is intuitive enough to appeal to novice users, while still remaining powerful enough to appeal to advanced users. We plan to consolidate the disparate interfaces previously mentioned into a single, unified, search-driven interface by creating an application that indexes the items found in one's desktop environment, including documents, contacts, bookmarks, applications, notes, multimedia, etc. We will then present graphical representations of these items to the user, allowing one to search through and interact with these items. Principle technical challenges facing this project include indexing and seemingly instant searching of items in a user's desktop environment, and implementing appropriate techniques for dealing with items of changing relevancy to the user.

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 and keyboard usage model 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 command. Quicksilver also has a plug-in library that allows the application to be extended to support new items and new actions.

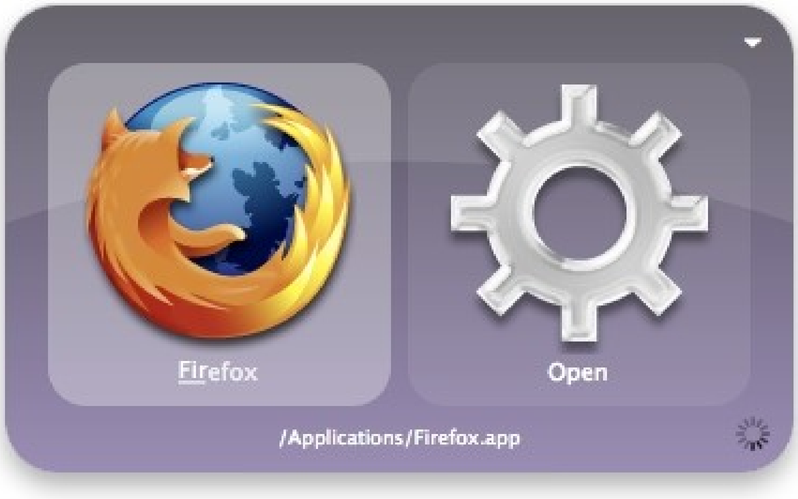


Figure 1: Quicksilver

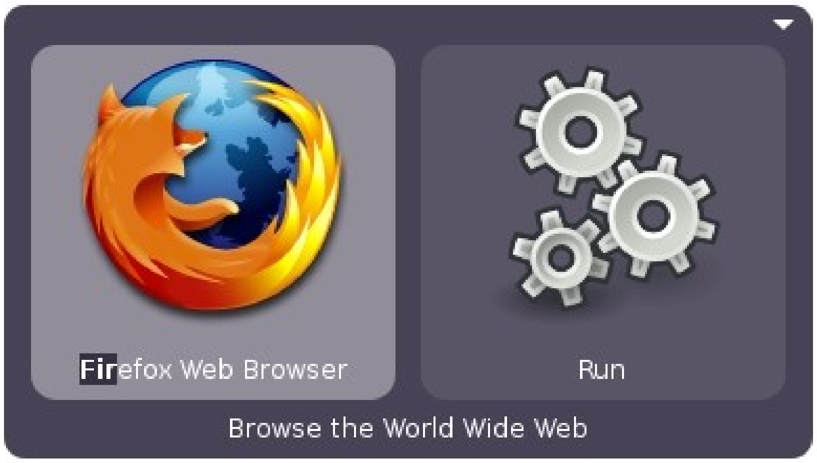


Figure 2: GNOME Do's Symbol Interface

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 type-level and token-level item relevancy (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

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

exactly what is happening in any section of the code is difficult), Quicksilver simply maintains a list of all searchable items, which is 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. In order to guarantee an upper bound on search time, and, more importantly, to provide apparently instantaneous search for the user, we will take a different approach (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 type/token relevancy details discussed in our "Technical Approach" section.

Quicksilver has many other interfaces 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 (historical user behavior data, 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 the first time GNOME Do is launched.

Finally, GNOME Do will be designed for GNU/Linux instead of Mac OS X, and the documentation and source for GNOME Do will be made freely available under the GNU General Public License (GPL). Quicksilver lacks documentation and until recently was not open source, which means that writing plugins for Quicksilver has been 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 had third parties write their own plugins and extend plugins that we had written ourselves.

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-year-long project.

Our project will make up for GLB's shortcomings by using managed code in a common language runtime (CLR) which will give third parties great 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, which is a C# compiler, collection of libraries, and virtual machine. Using these high-level tools, we will be able to focus more on search problems and user experience, and less on the nitty-gritty “gotchas” that we would encounter if we were to use C (this is due to our lack of C programming experience—we're not blaming C!).

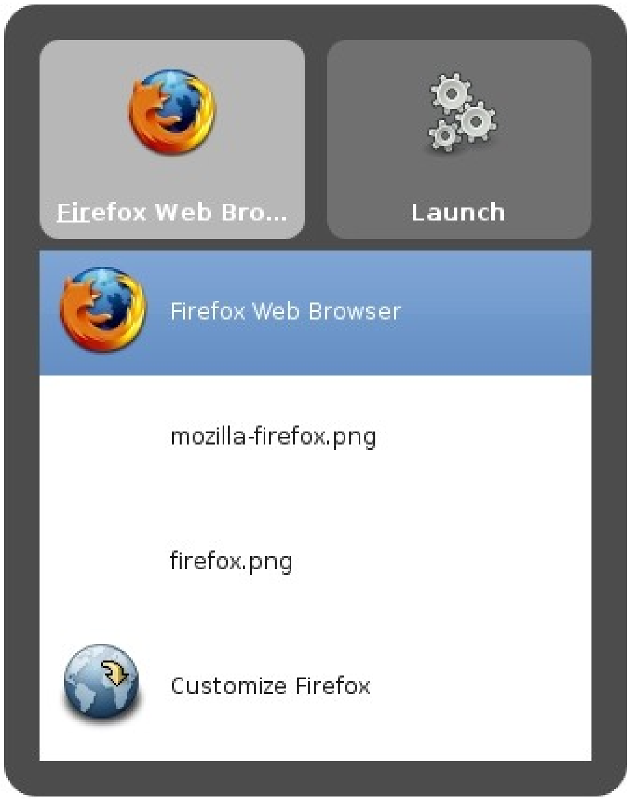


Figure 3: GNOME Launch Box

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 this 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, email addresses, and instant messaging handles. 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 the records into a single, representative record.

3 Technical Approach

Our application consists of a type “ontology” modeling the items people encounter and work with on a daily basis while using their computers: applications, utilities, bookmarks, directories, documents, movies, pictures, address book contacts, etc. A separate type ontology models the actions users perform on these items: launch, run, open, email, chat, etc. These types are substantiated by a plugin system which allows our application to be extended with new items and new commands.

To address the problem of providing apparently instantaneous search, we set out to learn appropriate indexing and caching techniques for presenting the user with a search interface that feels instantly responsive.

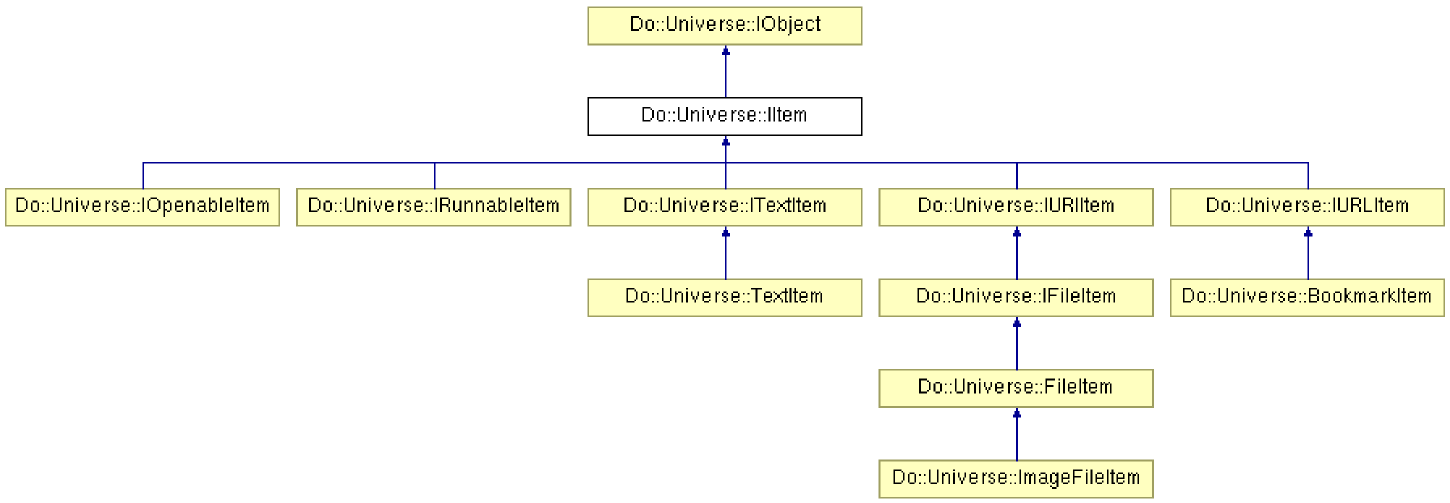
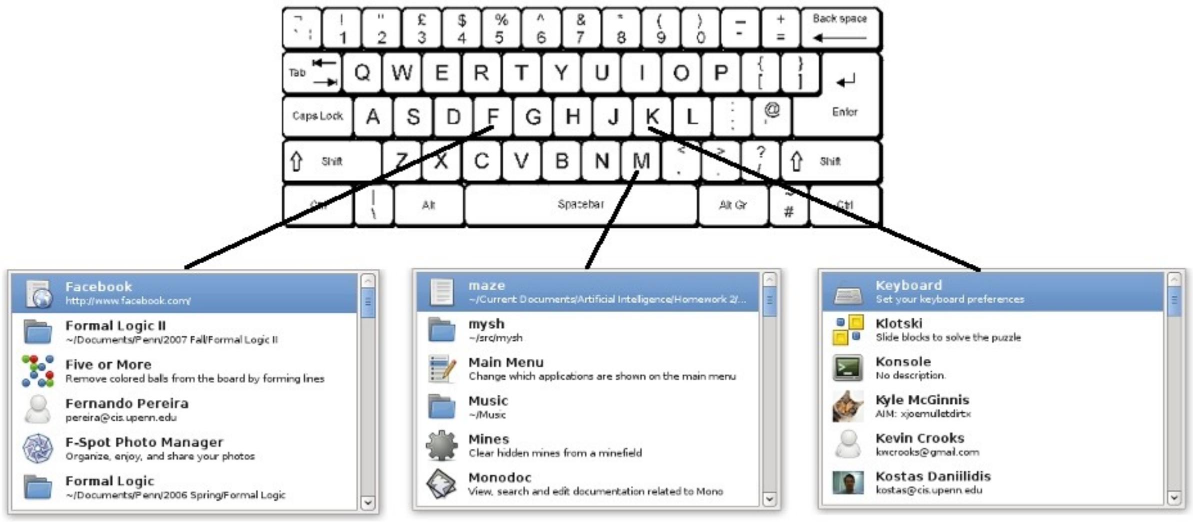


Figure 4: Hierarchy of built-in item types currently in GNOME Do.

GNOME Do can index a virtually unbounded number of items. 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 times 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. This semester we have focused on getting our application running with simple search techniques, but next semester we will focus on efficiency, particularly more efficient sorting techniques gleaned from our research. In addition we will explore ways to optimize our search and sorting techniques based on the state of user's desktop environment (see “Collaboration” section).

In our experience, users will not tolerate much more than a couple milliseconds of sluggish search behavior before developing an unfavorable opinion about the performance or usefulness of a desktop search tool, so we designed the searching algorithm to have an upper limit on runtime, independent of the number of items indexed. Our program contains a preliminary scoring system to determine an item's “relevance.” As of now, relevance only takes into account how closely the item's name matches the search string, but next semester we hope to integrate more factors into determining item relevancy (see “Collaboration” section).

When GNOME Do is started, it creates a 1-tier prefix tree that maps 26 single character strings (“f”, “m”, “k”, etc.) 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 (see “Figure 5”). 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 any items whose relevancy scores drop to zero. Because the length of the lists of prepared search results is capped at 1000 items, the set being searched and sorted is never larger than 1000, no matter how many items the program indexes.



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

The user interface communicates with UniverseManager, the class that stores the universe of objects and the search method, through an intermediate class called SearchContext that stores all the relevant state while the user is conducting a particular search. The SearchContext stores a stack of previous contexts, and every time a new letter is added to the search string, the current SearchContext is copied and pushed onto the stack. When the user deletes a character from the search string, the context stack is popped, allowing speedy “backwards search.”

GNOME Do's item API allows any item to have child items. 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 to a depth of 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 hope to use this capability to discover important features of the user's environment to index. The SearchContext also allows for setting “dependencies” or type constraints in the search. For example, SearchContexts can be limited to only search actions supported by a specified item, or items supported by a specified action.

Our search implementation assumes 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 in certain positions in the name string of the item—it assumes that letters that appear in the beginning of the name or the beginning of words in the name should be given higher weight. We plan to re-examine the string scoring algorithm next semester to account for more name features like capitalization, other token delimiters, and character classification (letters vs special characters). Also, by limiting the search results list length to 1000 items, we assume that users cannot name more than 1000 items in their desktop environment for any given character on their keyboard.

To address the issue of item relevancy, we plan 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; this set corresponds to the initial letter pressed in a search resulting in a changed relevancy score. The figure below illustrates the inheritance tree 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 execute the command “Rotate 90° Counterclockwise” on that item. GNOME Do will increase the relevancy score 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 relevancy scores are used to rank all items in the prefix tree. The relevancy for the “Rotate 90° Counterclockwise” command will also be increased. If the next search is for “face” and I have two items with equal string relevancy for the query, such as a bookmark for “facebook.com” and an ImageFileItem for “face.jpg”, the type-based prioritization will ensure that the image is considered more relevant than the bookmark.

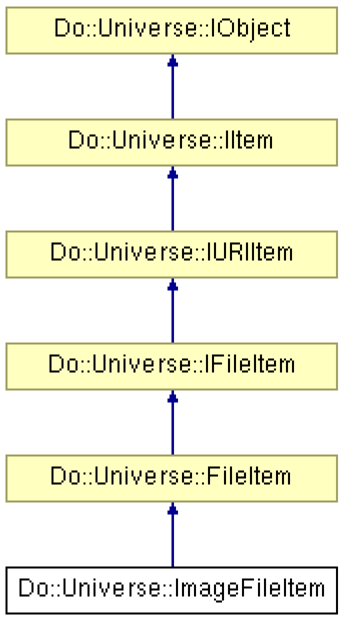


Figure 6: GNOME Do's type information about ImageFileItem

As far as token-based, local prioritization goes, the "Family Portrait.jpg" item will have its individual relevancy score increased in addition to the relevancy increase it accrues from being a member of the ImageFileItem type; however, this token-based relevancy score will only be considered for searches beginning with "p", as this was the initial keypress in the search resulting in the change in token relevancy. 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, and R, 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 relevancy. We are awaiting work from another group that will give us relevancy scores to implement these features with (see "Collaboration").

Two great challenges will be learning to use the necessary tools and libraries (see "Resources Required"), which neither of us is familiar with, 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 maintainer of Pidgin (formerly "GAIM") 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 will plan, organize, and develop our application. GNOME Do is an open source project, which means that all specifications, source code, documentation, and other project resources are publicly available for anyone to read or compile at any stage in our development process. This allows us to collect feedback such as feature suggestions and bug reports from members of the community. We are hosting our project on Launchpad.net, a popular site for open source projects emphasizing community participation and collaboration. Launchpad is the home to the Ubuntu project, the particular GNU/Linux distribution we will be developing our project on. Our project page is http://launchpad.net/gc. By the end of the year, this page should contain a detailed account of all of the planning and work that has gone into our project. This approach is a double-edged sword, however: by inviting people to scrutinize and interact with us and our work, the quality and thoroughness of our project should be higher, but managing community interaction adds a large, organizational dimension to our project that most senior design projects do not have to deal with. This is why we have asked Sean Egan to mentor us.

Here are some details about community interaction and participation with our project during the first three months of development: a screencast demonstrating GNOME Do was posted on Google Video on October 26th, and it was been viewed 1,500 times in the first month. 25 bugs have been reported (http://bugs.launchpad.net/gc). We have committed 100 revisions to our release branch, which contains 9,000 lines of code. We have received approximately 5 patches from external contributors, one of whom was 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 also expressed great enthusiasm about our project in an email exchange. GNOME Do was blogged about by Jorge Castro on http://planet.ubuntu.com, a popular aggregation of blogs of Ubuntu developers. We have about 30 people on our mailing list, which contains 250 messages. Richard Harding has joined our project as our official Ubuntu package maintainer. Richard has prepared an online repository containing versioned binary packages of our project, keeping users updated with our newest releases. Richard has reported that more influential members of the Ubuntu project have expressed great interest in getting GNOME Do packages included in the next release of Ubuntu, Hardy Heron. We have no way of knowing how many people are currently using GNOME Do, but judging from our interactions with users and contributors, we are fairly certain that we have at least a couple hundred users at this point. We consider GNOME Do a remarkable success for a project in its infancy.

5 Collaboration

We have barely begun to collaborate with Ian Cohen and James Walker's project for scoring items for relevancy. We have 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 have taken place. Ian and James had planned to develop, by the end of this semester, the item type ontologies which we would use in GNOME Do, but unfortunately we do not expect that they will deliver these. We are confident that we will still be able to implement historical relevancy scoring if Ian and James's work does not materialize or other complications arise.

6 Resources Required

We will implement 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)~~ (we now build with autotools for greater control and compatibility), 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 will be the time it takes to learn to use them.

7 Timetable

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

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

  This is almost complete—we have not added the modifier (third) item pane to our interface because we decided it was more important to implement the browse feature first, as the browse feature is essential for learning about new (non-indexed) items from the user.

- 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).

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

  This should have been prioritized after *development of our "core" set of essential plugins, because we need to create plugins ourselves* before *we can instruct others on the matter. We have created open, open URL, email, chat, play, append to play queue, define, open location in terminal, and execute in terminal commands; and we have created item representations for applications, bookmarks, files, folders, multimedia (music albums, artists, and songs), contacts (with email and instant messaging attributes), and raw text.*

- 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 plan 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, relevancy, and other features.*

- improve upon search techniques and responsiveness developed earlier

  > *This will occupy a significant portion of our time for the rest of the year.*

- 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

  > *We would still like to do usability testing like this, although we don't plan on developing a second interface—we would like to focus on improving our current interface, and focus on refining and expanding upon other features.*

- a version "1.0" public release, packaged for Ubuntu and available in source code for other GNU/Linux distributions
- consider package submission to the Ubuntu project so that our application will be included in Ubuntu repositories

  > *This is well on the way to becoming a reality.*

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

  > *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

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.*

"Documentation." (2007) *MonoDevelop*. Retrieved 24 Sept. 2007 <http://www.monodevelop.com/Documentation>.

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

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>.

Luke, Zettlemoyer, and Michael Collins. (2007) "Online Learning of Relaxed CCG Grammars for Parsing to Logical Form." *MIT CSAIL*. Retrieved 23 Sept. 2007 <http://people.csail.mit.edu/mcollins/papers/relaxed.pdf>.

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

Sinha, Ranjan, Ring, David, and Zobel, Justin. (2006) Cache-Efficient String Sorting Using Copying. *ACM Jour. of Experimental Algorithms 11*.

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