A NEW SPEECH ENABLED APPLICATIONS INFRASTRUCTURE FOR THE INFOPAD

by

Anoop Kumar Sinha

Memorandum No. UCB/ERL M98/3

14 January 1998

COVER PAGE

A NEW SPEECH ENABLED APPLICATIONS INFRASTRUCTURE FOR THE INFOPAD

by

Anoop Kumar Sinha

Memorandum No. UCB/ERL M98/3

14 January 1998

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A New Speech Enabled Applications Infrastructure for the Infopad

By

Anoop Kumar Sinha

Masters of Science
Engineering-Electrical Engineering and Computer Sciences
University of California Berkeley

Abstract

A new speech enabled applications infrastructure for the Infopad is proposed and demonstrated. The architecture allows use of arbitrary recognizer engines and eliminates the dependency on a specific recognizer or specific recognizer API. The Nuance recognizer system and TI Dagger system are used as test speech recognition engines. The infrastructure allows easy speech activation of Java programs. A Java Virtual Keyboard, a Java Personal Information Manager, and a Java Graph Editor were created using the infrastructure. These demonstration programs give some insight into the capabilities and limitations of speech as an input mechanism for portable multimedia devices, such as the Infopad.

Table of Contents

List of Figures

I. Background: Infopad User Interface

The Infopad is as a standalone, wireless network terminal, connected through high bandwidth wireless link to a basestation. This basestation is in turn connected via wire to a Sun Unix workstation. All of the Infopad computation is performed on the Sun workstation, only the user interface output and the user interface input are performed at the actual terminal itself. The Infopad is in essence an X-terminal, but one that is portable as well as wireless.

The Infopad has no keyboard or mouse. It only has pen and speech as input. An infrastructure, called Infonet, has been developed to handle these forms of input across the wireless Infopad link [Nara96b]. Tools have been developed to translate pen input to mouse press and drag input for control of menus and windows [Nara96a]. Extensions to the Infonet system have also enabled audio input and output at the Infopad terminal, specifically using a modified version of the AudioFile (AF) client-server system originally developed by DEC [AF].

A. Pen Input

In its present state, the pen is best used for manipulation tasks rather than input tasks, pointing and dragging rather than pen input. One reason for this is that an integrated handwriting recognizer is not available for the Infopad. Handwriting recognition systems are available, such as the QuickPrint system by Lexicus and the handwriting recognition system developed by Narayanaswamy as part of his doctoral thesis [Nara96a]. There is no handwriting recognition system integrated into the Infopad

for use across all applications, just as there is no handwriting recognition system integrated into the X-windows environment.

The original handwriting recognizer for the Infopad and much of the pen input infrastructure were written by Narayanaswamy [Nara96a]. Essentially this system accesses the handwriting recognition functionality through a specific API, with favorable integration with applications written in the Tcl/Tk scripting and user interface language. Applications must be compiled with the Narayanaswamy's recognizer library. This infrastructure works and has been used to create a few demonstrable programs such as a SPICE input editor and a handwriting input widget [Nara96a].

The standalone QuickPrint system works as follows. Individual letters are drawn into a small window. The individual letters are recognized and then can be sent to any given window as X-windows KeyPresses that the user designates by pointing. This system is clumsy and ineffective, not only because of the multiple steps involved in inputting a simple word, but also in the ineffectiveness of the recognizer.

B. Speech Input

The original speech interface for the Infopad is similar in architecture to Narayanaswamy's handwriting input system. The audio input mechanism is essentially a modified version of the AudioFile server system and is quite generic in allowing audio input and output. The original speech recognizer, written by Burstein, is a fairly sophisticated speaker dependent, trainable recognition system [Burs96]. Like the handwriting system, this recognizer requires use of Burstein's API and compilation with

Burstein's libraries for the speech system. This system has been demonstrated applications such as the Spice editor and a speech input widget [Burs96].

In the Infopad application environment, dependency on any specific speech recognizer API or library is not ideal. Dependency on a specific API restricts the user to a specific speech recognizer, since there is no standard speech recognizer API. Since there are many speech recognition engines available of differing performance and suitability, the dependency on a specific API is not favorable. Furthermore, integration with Tcl/Tk, as in the original Infopad UI systems, is less favorable than integration with the Java AWT for instance, given that Tcl/Tk is less mainstream than Java.

For these reasons, the Infopad user interface infrastructure is not complete. This project concentrates on a new speech enabled applications infrastructure for the Infopad. The handwriting recognition system was pushed out of the scope of this project.

II. Motivation and Goals

One of the primary goals in this project is to take the next step in the advancement of the speech user interface infrastructure of the Infopad. The main goal is to allow easier development of speech recognizer enabled programs, easier in this case including enabling Java based applications. A secondary goal for this project is to create a speech recognition architecture, which was not only programming environment independent, but also speech recognizer independent. The recognition system needs to be available to programmers through a well-defined interface. Furthermore, the system should support the available commercial, speaker independent speech recognizers, such as those from TI and Nuance that were available for this project.

The handwriting recognition infrastructure was not pursued in this project. It is a work that is left for improvement for other developers, and is work that was placed out of the scope of consideration for this project.

A. New Programming Environment

As described later in this report, the end architecture was developed to be integrated with the Java programming environment, making it readily available for use by Java programmers. The Java programming environment was chosen for a number of reasons. The primary reason was its recent wide availability and popularity. The Java programming environment, together with the virtual machine on different reference platforms, provides a lightweight, functional subsystem, which gives us the opportunity to quickly create applications. In many ways, the basic user interface capabilities included in the Java Abstract Window Toolkit (version 1.0.2) are the ones that are most

required by developers for the Infopad. By its nature as a stand-alone handheld device, the Infopad application level does not require particularly complex access to different platform-specific capabilities. Another advantage of Java is its support for network capabilities, such as sockets. This network support proved useful in the final architecture of the speech recognition system.

It is interesting that the Java language was originally developed for use in lightweight standalone devices [Flan96]. This seems to make it even more appropriate for use as the underlying environment for the new speech recognition system architecture for the Infopad.

B. New Speech Recognition Architecture

For the new speech recognition architecture, the process in this project was to start with a speaker-independent speech recognition engines available from TI and Nuance and then use their available API's to it and develop a speech recognition system that would work with the Infopad audio system. In this process, it was also an important consideration to develop a speech recognition interface architecture, which would allow a program developer to access the speech recognition capabilities without too much application customization.

The end solution for this problem was to eliminate dependence on the speech recognizer API's by wrapping a speech recognition server system around the speech recognizers. In the actual application, a client module accesses the speech recognizer server through a socket. A diagram of the architecture follows:

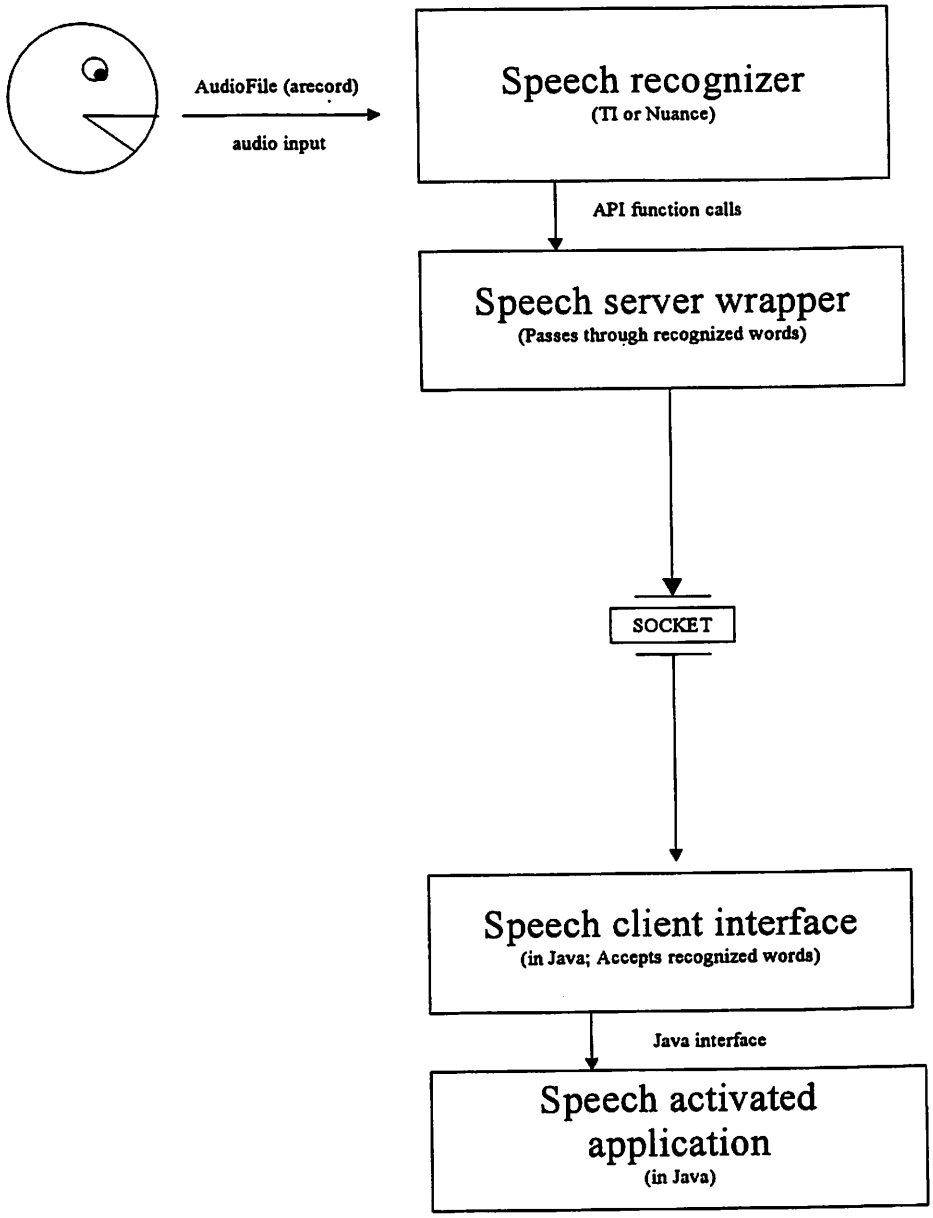

Figure 2-1. Speech Enabled Application Architecture

In the diagram above, the audio input goes directly to the speech recognizer system. This system then passes on the recognized words to the speech recognizer server wrapper, through the recognizer's API. The wrapper then talks through a single point of connection, a socket, to the client program. The only things that pass through that socket are the recognized words. The client application, which has the option of using the client wrapper module, can then take those recognized words and translate them into desired result actions.

As is readily seen from the architecture diagram, it is possible to use any speech recognition engine for a given client application. For each recognizer, the speech server wrapper needs to be rewritten to include the hooks to the speech recognition engine API. But with that done, the client application does not need to change.

For this project, the above architecture is demonstrated for two applications. The first application is a simple personal information manager (PIM) which shows some of the potential advantages of speech input in personal information creation and retrieval. The other application is a speech-activated graph layout application, which highlights the ability of speech to be used even for spatial creations. The design and implementation of both of these applications is included in further sections of this report. Both of these applications use the speech recognizer independent infrastructure, and both applications have been tested with the TI and the Nuance recognizers.

It is possible for an application developer to use this application architecture from languages other than Java. For instance, in 'C', the developer must perform the manual socket connection operations, but having done that, he will have access to the infrastructure.

III. Speech Recognizer Comparison

For this particular project, the TI Dagger Speech recognition system (TISR) and the Nuance Recognition System (Nuance) were used. Both of these recognizers are speaker independent and implemented in software only. Both require audio input and a user-specified grammar. They return a result based on the specified recognition grammar.

The most important step in using these speaker independent recognizers is writing a useful speech recognition grammar. The grammar limits the recognizer's words search space, and a good grammar is critical for good performance and accuracy of these recognizers, especially because they are speaker independent. The grammar used for the applications in question is discussed in the next section of this report.

At the outset, one of the strongest limitations in the accuracy of these recognizers is the fact that they are both speaker independent and not trainable. The trainable recognizer written by Burstein tends to improve in performance for a given user given training. Neither the TI nor the Nuance recognizer has this capability and recognition tends to be either good or horrible depending on whether the user's voice matches the voice models used by these recognizers.

Before considering the actual use of these recognizers for the Infopad, it is useful to highlight the differences between the two recognizers. The differences between the two recognizers help us understand the performance differences in the Infopad speech recognition system, as well as the potential and limitation of speech as an input mechanism for the Infopad.

A. TI Speech Recognizer

The TISR recognizer is smaller; it uses less memory and uses less disk storage space. It runs as a standalone program on a machine, the engine itself simply taking in audio input and returning a result as well as statistical parameters for that result. The algorithms are hidden, but there are some customizable parameters for the speech system. However, there are fewer customizable parameters than for the Nuance system.

The TI recognizer core functionality offers little more than grammar specification and a built-in speech model for recognition. It is speaker independent, and there is no way to adapt or learn from recognition mistakes. It is a fairly good recognizer, but has serious recognition problems when used in noisy environments. It also has serious problems with voice standards that do not match its speech models. These deficiencies were addressed by changing the grammar to improve recognition accuracy.

Its turnaround time is suitable for our applications, with good end-pointing and recognition results being returned in at most a second from the end of end-pointing, provided the load of the machine it is running on is low. However the initialization for TISR takes approximately ten to fifteen seconds on a low load machine.

B. Nuance recognizer

The Nuance recognizer is undoubtedly superior in performance. Architecturally, the Nuance recognizer always runs in the client, server mode. The server process is the core recognizer engine and is hefty in memory requirements and CPU usage. The recognition system has been optimized for noisy environments, in particular the default

speech models are for a telephone line [Nuan]. This being the case, it almost always performs better than the TI Recognizer.

Any Nuance client is lightweight, since all of the computing is going on in the Nuance server. Once a server is started, the client takes less than a second to start, connect and initialize. When recognizing, the Nuance recognizer is also perceptibly faster than the TI recognizer is. No doubt the Nuance event model, which uses callbacks to pass back the recognition result, accounts for some of that performance increase from the threads-based TI recognizer. For the Nuance recognizer, it is also possible to distribute the recognition load, putting the hefty server process on a fast server and using lightweight clients.

Through testing and usage, it is clear that the Nuance recognizer is superior and should be used whenever possible. Perhaps the only case where it cannot be used is if a Nuance license is not available for the machine in question.

C. Implementation Details

In the Infopad cluster of machines, the TI recognizer is available on all machines whereas the Nuance recognizer is only available on badlands.eecs.berkeley.edu.
In order to use each system, a user must go through the following steps:

1. Write a grammar:

- TI: the grammar is in standard Backus Naur Form, specifying individual words or sentences [Tisr]. The grammar used is in Appendix A.

- Nuance: the grammar can be BNF but also can include extensions, such as allowing optional words in a given sentence, allowing multiple repeated words. [Nuan]

2. Compile the grammar with an audio model:

- TI: "tisr_compile"; the grammar is compiled using a built-in audio model, which has a 10,000 word vocabulary. Words not in the vocabulary are automatically split into phonemes, and a utility creates a pronunciation model for those words.
- Nuance: "nuance_compile"; words not in the model files vocabulary are returned to the user with a request for manual, phonetic breakdown. The user must specify the pronunciation of the words that Nuance does not have in its vocabulary.

3. Test the compiled grammar file

- TI: requires using a sample program, which implements the TISR API to read in the compiled grammar.
- Nuance: requires using a sample program, which implements the NUANCE API to read in the complied grammar.

4. Use the grammar in the specified application.

Both TISR and Nuance come with their own 'C' level API. At the core, both recognizers go through the following operations: initialize, read in the compiled grammar file, start listening, endpoint each phrase and return the text result, clean-up by freeing memory and quit. However, Nuance performs those operations using a set of individual Nuance-defined Event callbacks. TISR's approach is more sequential, with the use of a separate thread to return recognized words to the user which recognition is being performed. These two API's are not compatible and not interchangeable. To use one or

the other in their specified form would limit the user to one recognizer or the other. This incompatibility is not so much a difference in functionality, but rather a lack of standardization.

For this project, one of the main goals was to make the Infopad system recognizer independent. One of the first tasks was deciding how to take TISR and Nuance and put standard wrapper code around them so that clients could access them in individual ways. By using a common wrapper, the recognizer independence was limited by the least common denominator of the functionality of the two recognizers. Between TI and Nuance, this meant that the common wrapper could not use some of the scoring probabilities returned by the Nuance recognizer, nor could it use the Nuance routines for reading input directly from a microphone.

Least common denominator API functionality seemed best accompanied by least common denominator grammar functionality. Hence, the advanced features in the Nuance grammar specification were not used. Rather, the grammars were specified using BNF. With standard BNF, it was easy to write a program, `Gramm`, which translated the TISR grammars to the Nuance grammars, ensuring the same accepted words no matter which client application was used.

The best model for this system was to implement a generic server wrapper around the speech recognizers, rather than rely on a specific API to fold into a written application. The speech recognizer server wrapper creates and maintains a socket to which clients connect. The speech recognizer wrapper just passes recognized words through that socket.

This model is appropriate for our task for many reasons. The first is that the audio inputted into the recognizer comes from an AudioFile server (AF) in the Infopad model. The client microphone audio passes to an AF server, which then redirects the audio to an "arecord" client. This allows us to simultaneously set-up the audio input, speech recognizer and wrapper in a single line function call using pipes:

For TI:

```
> arecord -e lin | tisr_parse -params tisr_files.prm
```

For Nuance:

```
> arecord -e lin | rstest -package pim2_n
```

arecord –e lin: acquires linear encoded speech input

tisr_parse –params tisr_files.prm: tisr_files.prm is the compiled grammar file

rstest –package pim2_n: pim2_n is the directory with the compiled grammar file

These function calls setup the recognizers and wrapper. At the end of these function calls, these programs set up a socket and wait for a connection to that socket from a client. Once that connection is made, the speech recognizer wrapper passes the recognized words through that socket to the client.

The methodology described above creates a speech recognition server out of any speech recognizer. The speech recognizer server model suits our application much better than internally compiled speech recognizer API function calls, since the speech recognizer can easily be changed with no change in functionality of the code.

Furthermore, clients simply need to talk appropriated to the speech recognition server socket. They can be written in any language that supports sockets.

For this project, clients were implemented using Java, which has built-in socket support. As will be described later in this report, using Java and its programming abstractions and constructs on the client side with the speech recognition server wrapper provides a superb architecture for writing speech activated applications for the Infopad.

IV. Design of the speech recognizer grammar

The grammar performance in the speech recognizer system is perhaps the single biggest factor in the performance of this speech recognition system -- performance meaning the accuracy of the word recognized compared to the word desired. This is the most important factor here, because if a given word is consistently recognized incorrectly or simply never recognized correctly, the functionality enable by that word will never be able to be accessed by speech.

In our speech recognition systems, the grammars can specify one word or a phrase of words. A phrase of words gives the impression of being naturally connected speech, when in fact the recognizer is treating the phrase as its own entity. Hence, "please send an e-mail to this person," is actually said in one breath rather than as seven individual words. This is in fact is obviously more natural for the user, even though it does lead to the need to specify a large number of possible input phrases.

A. Handling words that sound the same.

The speech recognizer grammar greatly influences the overall accuracy of a speaker-independent speech recognition system. Here we introduce the term "orthogonal grammar," by which we mean a speech recognizer grammar where the words or recognizable phrases do not contain similar words or similar phonemes and hence are more easily phonetically distinguishable. For instance, having the two phrases "hello" and "hello there" in the grammar leads to potential conflict because a user could be starting the phrase "hello there" even though the recognizer returns "hello". "Hello" and "hello there" in the same grammar makes the grammar less orthogonal.

Though it is desirable to have an orthogonal grammar in a speaker independent recognition system, it is clearly not practical, especially if the system is to be designed for natural speech input, speech input that a user would say in everyday speech. This is one tradeoff that must be made in the design of a speech recognition system, which uses speaker independent recognizers. In fact, the compromise must be made regardless of whether the system is speaker dependent or independent, and a point that should be noted in any speech recognition system design.

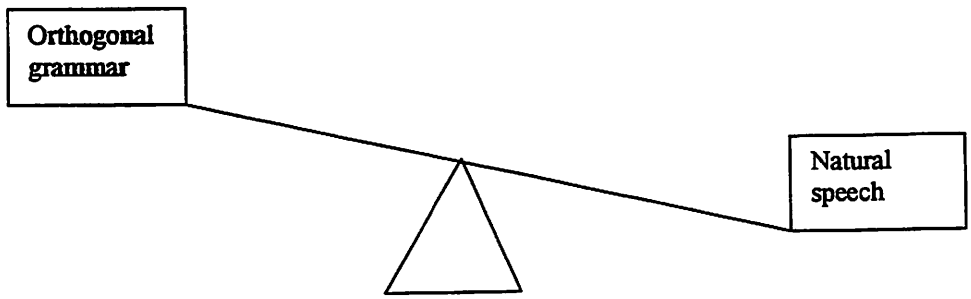

Figure 4-1. Orthogonal grammar vs. natural speech tradeoff

For our specific case, the grammar tradeoffs involved needing to include many phrases that were started by the same words. Some specific phrases that ended up in the final grammar design include "switch to contact" versus "switch to calendar", "year-up" versus "year-down". There is also a whole set of commands that are started by the same phrase and simply end in a different number or name. "Please send and e-mail to <name>" can take any one of "bob", "kevin", "group", etc. The set of commands that deals with specifying a time, such as "meet at <number> o'clock" or "go to <month> <day> <year>", all rely on sufficient phonetic distance between the numbers and terms in question. Luckily, most numbers are fairly phonetically distinct and also most

recognizers are built with models that easily distinguish numbers, one of the initial tasks of a speech recognizer. The distinction between "seven" and "eleven" often trips up the recognizer. Also, numbers that are prefixed with a tens specification, such as "twenty-" or "thirty-", can be easily confused, especially if the speaker does not enunciate.

There were also cases where there were many phonetically similar words, even if the words do not start with the same phrase or even look similar. As an example, in the International Phonetic Alphabet, "alpha" representing "a" and "papa" representing "p" are quite similar in their phonetic representation. This means that the recognizer confused them quite readily. The solution in this case was to extend "alpha" to "alphabet," and extending it to three syllables distinguished "papa" from "alphabet". Initially, "alpha" was consistently recognized as "papa" by the recognizer, in essentially 100% of the cases. After changing the phrase to "alphabet", the recognition problems essentially were eliminated.

This highlights an interesting point about the design of grammars and the ensuing performance with these speaker-independent recognizers. A small change in the grammar vocabulary can change the recognition accuracy from near 0% to near 100%. This also follows from the fact that speaker independent recognizers are fairly deterministic. There is no dynamic learning involved and hence if the word is said the same way, the word will be recognized in the same way. One viable solution to this is, as we have done, change the grammar so that non-orthogonal conflicts do not exist. This is why the grammar design is critical for performance of the recognizer.

A wider implication of this fact is that it is not necessarily useful to talk about performance numbers in this particular speech recognition system. Performance numbers

such as 99% accurate or 5% accurate are of dubious value since those numbers are so much at the two extremes for these particular recognizers.

Another wider implication of this exercise is to show that this consistently accurate or consistently inaccurate speech recognition system performance is one of the most critical problems for a speech recognizer. A user can experience extreme frustration using the speech recognition system if that user is unable to access a given behavior because he is unable to say a given word. To have "papa" recognized each time "alpha" is said means that spelled words will have "p's" in their spelling as opposed to "a's"!

B. Natural language in the grammar

One general trend that was discovered in the design of the grammar was the trade-off between the grammar orthogonality and the closeness to natural language of the required input speech. If a change needed to be made in the grammar so that the given phrases would be more recognizable, then the speech would often times change from being most natural to being less natural.

In the design of the grammar, it was also not possible for the grammar to include every possible variation of a given phrase. One of the goals in the design is to pick the most natural expression of a phrase so that a user has a high probability of choosing it when spoken. But greater accuracy will be achieved if the user is trained and is familiar with the specific variations of the phrases that are recognized in the limited grammar system.

As an example of this, there are an incredible variety of possible phrases that can be said for a greeting. Only some of them can be picked for inclusion in this grammar:

```
<GREETING> ---> hello | good-bye | see ya | take care | sincerely
```

What this leads to is a potential that the user will say some other greeting, such as "salutations" or "later", and the recognizer will be unable to recognize their phrase. A further problem with this system is that the recognizer will not only be able to recognize the correct phrase. It will not know whether it has the correct phrase or not, as it simply sees and inputted stream of phonemes and tries to recognize them as best it can.

In this particular project, these concerns were addressed by carefully considering the trade off between how natural the grammar is versus how well the grammar performed. In designing some of the possible inputted phrases, adding "please" to the front of the phrase changed the performance of the recognition search and improved the grammar accuracy. This allowed the introduction of a set of "natural language" commands, all of which needed to start with "please" to distinguish them from other commands that are more direct command based.

C. Future of Speech Recognizer Grammars

The prospects are, however, that this will improve. There will be continued improvement in speech recognition algorithms that will allow recognizers to more finely distinguish between similar sounding words. This will also permit the added value of allowing more natural phrases to be specified as input. More likely than not, in the next generation of speech recognition systems, the natural feel of the grammar will improve and the restrictions on the grammar will decrease.

An alternative approach which somewhat by-passes the entire issue of specifying long phrases in a grammar involves preparing a speech recognition system that tries to

segment and then recognize individual words no matter what phrases are said. This system will not have to deal with the issue of a user manually specifying natural phrases, provided the vocabulary size is large enough. But it will have added problems of segmenting words as well as problems distinguishing words with slurred syllables.

V. Speakable X-keyboard

One of this project's first test applications implemented using the TI and Nuance recognizers is the "Speakable X-keyboard." The goal of this application is to allow the user to speak a word representing a letter and then have the corresponding letter appear on the screen in the window with the current input focus. This methodology eliminates the need for a keyboard, though it still necessitates the use of a mouse to change the input focus or change the input position.

The vocabulary chosen for this process was the International Phonetic Alphabet [IPA], and modifications were made to it. The modifications included making words that sounded similar to the recognizer more orthogonal either by adding syllables, in the case of "alphabet" instead of "alpha", or changing the structure of the word for a given letter, as in "go-golf" instead of "golf". The modified vocabulary was chosen after experiment and iteration, by pronouncing the 26 words in the alphabet and seeing which were generally clearly recognized and which were not.

The basic steps in this Speakable X-keyboard are as follows: say the word, have the program recognize the word, translate that word to the appropriate letter, and then echo that letter to the screen. The drawback in this approach is that X-windows system has a built in security model, which makes it impossible to generate fake keyboard events for windows that do not explicitly allow them [Young]. In order to accept fake keyboard events, an xterm, for instance, needs to have the X-resource "*xallowkeypress" property set to true, and an emacs window likewise has to have the "x-allow-keypress events" member variable set to true [Shil97a]. The "Speakable X-keyboard" was tested for those

two cases, but for some other applications, the X-keyboard as implemented would not work.

For the emacs window and the xterm, using this Speakable X-keyboard conclusively showed that this Speakable keyboard approach was unacceptable. The input was completely too slow; there was a high probability of inaccuracy and the need for correction; and the interface was simply not friendly. The Speakable X-keyboard allowed input at approximately the order of 5 words per minute, given the latency in recognition, the need for correction, and the length and number of spoken words needed to specify a given written word. This is a magnitude slower than a standard typing input speed of 50 words per minute, and intolerably slower than the average speaking speed of 200 words per minute. The high inaccuracy in this method is partly a function of the performance of the recognizer on the modified International Phonetic Alphabet vocabulary, and unavoidably a function of the speaker's ability to speak at the proper rate for accurate recognition and to avoid slurring individual words representing letters. The task of speaking a word to represent a letter does not represent a user-friendly input, especially given that speakers are familiar with spelling words using the sounds representing the letters, "ay", "be", etc., not different words represent the letters.

It is quite clear from the implementation of the Speakable X-keyboard that it is not a good input method. It might be a more natural method if speech recognizer had the capabilities to distinguish and recognize the 26 different sounds representing letters. But the recognizers used in this experiment do not have that capability. It is unlikely that any recognizers would have that capability just given the phonetic similarity of the letters with the long "e" sound: "b", "c", "d", "e", "g", "p", "t", "v", "z".

As far as speech input is concerned, discrete word input is not a viable input methodology here, because discrete word input involves specifying a large, unwieldy vocabulary for the recognizers used in this project. Furthermore, it is hard to cover all of the required spoken words. Added to these problems is the fact that these speaker independent recognizers are not designed for large vocabulary, discrete word entry.

In the end, the Speakable X-keyboard showed that letter by letter, spoken word input is not a particularly good method for Infopad users.

VI. Java Virtual Keyboard

The work done for the Speakable X-keyboard was not entirely abandoned, because even though it has poor performance, the approach does provide a generic input methodology. The Speakable X-keyboard approach is a useful backup in the case that there is no other way to input individual letters. A Java Virtual Keyboard was chosen as the appropriate method to input individual letters in the new infrastructure for Java applications for the Infopad.

The Java Virtual Keyboard avoided the issue of X-event security by simply creating keypresses for "appropriate" Java AWT text boxes and text fields. The "appropriate" boxes are those widgets that are created using the Java classes SpeakableText and SpeakableTextField that were defined for this project. SpeakableText is a subclass of the java.awt.Text, and SpeakableTextField is a subclass of java.awt.TextField.

In addition to being responsive to the Java Virtual Keyboard, these sub-classed text fields allow speech-controlled focus, allowing the user to say the name of the text field, for instance "name text field", and have input focus shift to that text field.

Input focus is one of the seminal problems in speech interface design. It is clear that in a joint mouse and speech environment, the mouse can be used for input focus. However to fully enable hands free operation, if desired, it is useful to provide a methodology for changing input focus not only by mouse but also by specific speech commands.

The other difference between the Speakable X-keyboard and the Java Virtual Keyboard is that the Java Virtual Keyboard has an on-screen representation. In a keyboard-absent environment, having a familiar on-screen keyboard representation ensures that the user can input any text that they normally would input via the keyboard. Especially in the Infopad's case where the speech or handwriting recognition systems are not mature enough for arbitrary word input, the on-screen representation proves to be very useful.

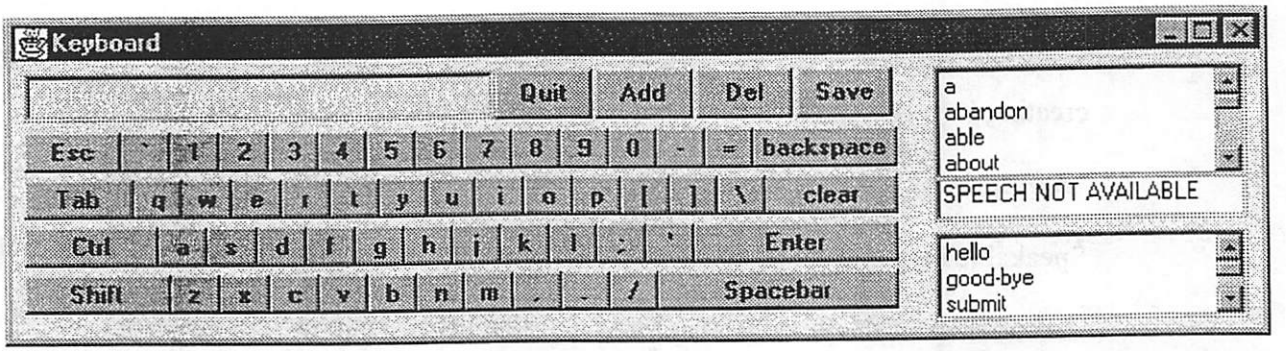

Figure 6-1. Java Virtual Keyboard

Specific design choices were made for this particular graphical user interface representation. The first choice, the size of the Java Virtual Keyboard window, followed from the 640x480 pixel size restriction of the Infopad screen. The buttons were chosen and sized to be as big as practical with the additional choice that the total window was to be 640x100 in size. The graphical user interface was chosen to have a text box representing the word that would be forwarded to the destination text box. This dual representation makes it clear to the user that they are using the input virtual keyboard.

The Java Virtual Keyboard itself has an upper right hand list box with a list of 2,000 most common words. As the user types using the Virtual Keyboard, the highlight moves to the word started by the word presently being typed in the input box. The user can complete the word by selecting it from the list box. This technique, often called "autocompletion," can speed up the input of words. The rate at which it speeds up input heavily depends on the usefulness of its preloaded vocabulary and the amount that it is used. An end user can extend the vocabulary in this particular autocompletion system simply by pressing the "Add" button for a word that he fully finishes typing in the Java Virtual Keyboard. Likewise words can be removed from the autocompletion using "Del." The modified word list can and must be manually saved off to disk using "Save".

In an effort to conserve the window space, the Java Virtual Keyboard box was also designed with the lower right text box, which lists the speakable words given the current program focus. The application program using the Java Virtual Keyboard is responsible for loading and deleting words to that list box, through an interface. The list provides a way for the programmer to give the user a list of possible commands to say.

One important point is that the commands listed in the speech-input box can be activated not only by speech, but also by selecting them by mouse. This provides an alternative way to activate commands even if speech is not available or perhaps if the speech recognition was inaccurate.

The text field above the list box shows the word that was recognized by the speech recognizer and the command that is sent to the application. So long as the speech recognition system is running properly, that speech input box will always show what the speech recognizer recognized.

Architecturally, this Java Virtual Keyboard also enables the programmer to avoid dealing with the plumbing associated with connecting to the speech recognition system. The Java Virtual Keyboard implementation has in it all of the code needed to connect to the speech recognizer and retrieve words and forward those words through an interface to the end level applications that use it. In the architecture that was described earlier in this report, the Java Virtual Keyboard sits on the client side of the socket of the speech recognizer server wrapper. The Java Virtual Keyboard awaits words from that speech recognition server, and passes those along to the application.

In end-user experiments, this Java Virtual Keyboard fulfilled its specified tasks and functionality well. In applications that were written and are described later in this report, the Virtual Keyboard, with its speech input list box and the ability to discretely input any word, is extremely useful if not critical for the various applications. For a user just starting with one of the speech activated applications, it also provides that list of speakable vocabulary. For certain text inputs for name, address, or e-mail address, for instance, the Java Virtual Keyboard is the obvious input method.

Originally, the Infopad's user interface was not planned to rely on an on-screen keyboard. The original plan was to enable all user-interface input through handwriting recognition or speech recognition, which are arguably more natural inputs than a keyboard. The problem with relying on handwriting and speech input alone is the immaturity of the handwriting and speech infrastructure for handling generic applications. In the case of handwriting input, the commercial input widget that is installed in the Infopad is not an integral part of the system and is not widely used. Speech activation required and still requires custom applications. In the end, it is clear

that in the User Interface design for the Infopad, a Virtual Keyboard is necessary, if only because the speech input and the handwriting input technologies are not yet up to par.

VII. Java Personal Information Manager (PIM)

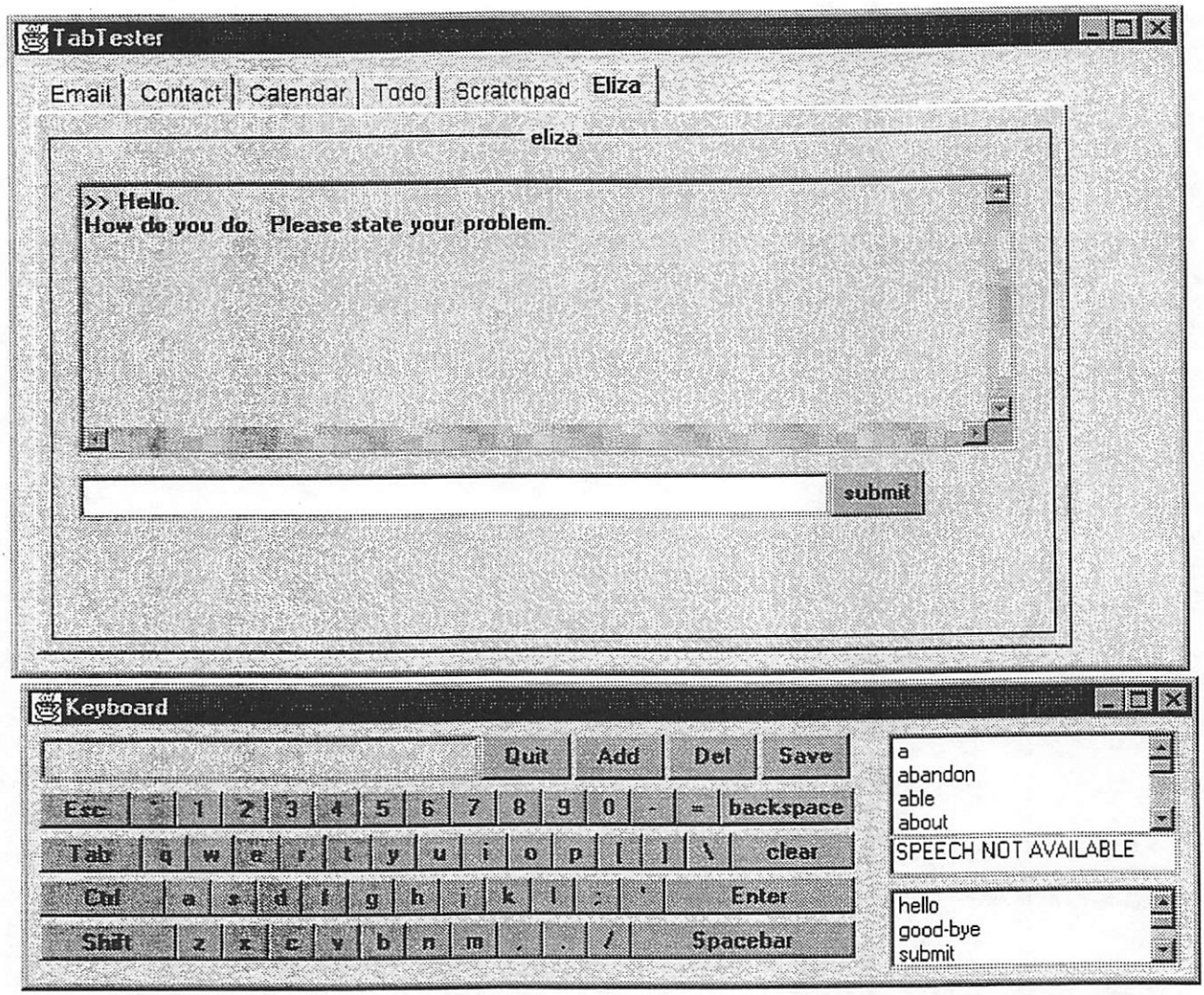

Figure 7-1. Java PIM

One of the two applications written using the Java Speech infrastructure for the Infopad was the speech-activated Personal Information Manager (PIM). A personal information management application was chosen for a number of different reasons. The first is that it is an application of great potential value to an end user if speech activated.

Personal Information Management is an area in which a computer organizer can help tremendously in keeping track of e-mail, contact information, calendar information, and to-do list. It is also an application, which lends itself very nicely to speech activation. The tasks that a PIM can control are similar to the tasks that a secretary often takes care of. Giving instructions about appointments, inputting address information, and sending messages by speaking to a PIM can be like talking to a personal computer secretary.

Personal Information Management is one of the areas that speech recognition is likely to have a strong impact when speech recognition technologies become ubiquitous. Personal handheld devices already perform PIM functions. It is likely that these PIM devices will continue to become smaller in form factor. Below a certain size a keyboard is impractical, and it is likely that they will rely on speech input and possibly output in their use.

Though certainly much bigger and more functional than a tiny handheld PIM device, the Infopad provides a good testing ground from a User Interface perspective for the value of speech input for PIMs. The PIM designed for this project was not meant to be functional but not over-featured. It was meant to perform basic functions, such as the e-mail, calendar, contact management mentioned before, but was not meant to be a production level system.

Given this as the case, it was possible to experiment with some of the potential interactions between different PIM elements using complex phrases for speech input. One example of this is sending an e-mail directly to a contact list member using a natural sentence: "please send an e-mail to this person." The interaction, as it worked itself out in development, is natural, even if not all variations of the phrase are available.

Examples of these natural language phrases are given in the detailed explanations of each element of the PIM.

One non-standard element is also included in the PIM application. A speech-activated implementation of the ELIZA program was developed to see show how it is already possible to enable speech input for a computer Rogerian psychologist [Eliza]. Though ELIZA might not be incorporated into a PIM, it certainly fits the analogy of a conversation with a secretary. The ELIZA implementation was modified from public domain code [Eliza].

This PIM is designed to be completely hands free, if desired. A pen is not needed for any element. Especially in conjunction with the Java Virtual Keyboard, however, pen use makes use of the PIM more efficient and easier. Perhaps the single most common area where a pen is desirable is for input focus, for specifying the text box that is the current focus of Java Virtual Keyboard input. The various text fields can be specified by speech by saying their label or by whatever element that they represent; however a simple pen touch in that element is a more efficient way of specifying the proper input focus.

The underlying element that was chosen to enable multiple functionality required in the PIM was the tabbed panel widget. This widget allowed us to create six different functions in this PIM which were easily integrated. The most important benefit of the tabbed panel widget is that it saves real estate, which is quite a premium on the Infopad's small screen. A tabbed element metaphor seems to be an advisable general methodology for the programs written for the Infopad's small screen. The tabbed widget that is used, as well as many of the other elements and graphical widget classes, came from Symantec's Visual Café Pro 1.0 Java authoring tool.

Switching between tabs by voice involves the "switch to" set of commands where "swith to" is followed by one of "e-mail", "contact", "calendar", "todo", "scratchpad", or "eliza". A switch in panels automatically updates the list of sayable elements. The update of the sayable elements is controlled at the application level via an interface to the Java Virtual Keyboard. A good consequence of the tab-enabled approach is that it automatically leads to high rejection for commands that should not be enabled when a given PIM element is in the foreground. For example, though the recognizer might recognize "please send an e-mail to this person" based on speech input, that command will have no effect if we are in the "To-do" tab. Though this does lead to the need to repeat mis-recognized commands, at least it does not lead to the wrong behavior.

In regards to accuracy, "please" precedes complex natural language phrases, which must be said in a single breath. If recognized at the beginning of the phrase, "please" signals the recognizer to move down a certain tree in the grammar which includes the more complex commands. The single breath is a requirement because the recognizer is actually recognizing a phrase as a single element rather than a set of discrete words. In this implementation, all complex phrases start with "please," increasing the accuracy of the recognizer as well as the politeness of the input.

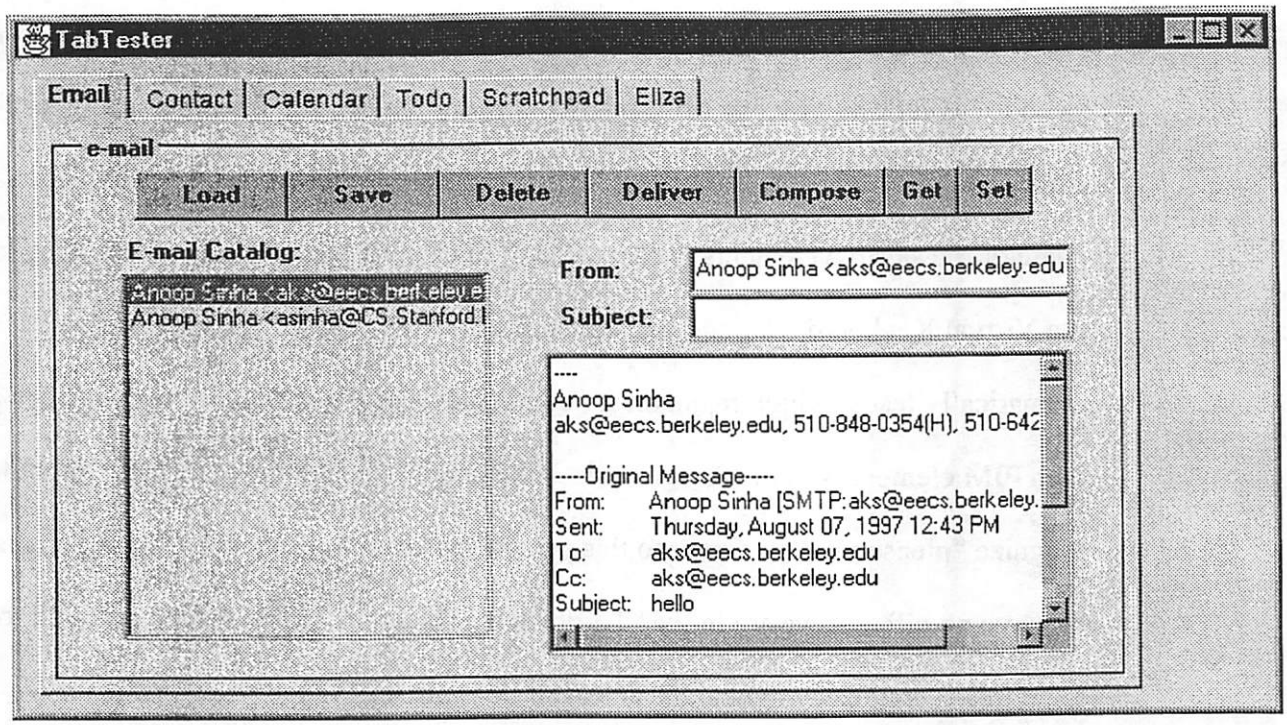

Figure 7-2. PIM E-mail

The E-mail client that was written is extremely simple yet fully functional for basic notepad memo type tasks. The program is hard-wired to connect to the Infopad cluster mail server, zabriskie.eecs.berkeley.edu. The first time the user request to "Get" e-mail, either by pressing the "Get" button or by the "get my e-mail" voice command, the program requests the username and password, both most easily inputted via the Java Virtual Keyboard.

Saying "next" and "previous," scrolls through messages that show up in the e-mail catalog. If the user desires to use the e-mail application completely hands-free, he can specify input focus by calling out "to text field", "subject text field", or "message text field", for the three input text fields, and then input letters by speech. Saying "letter"

followed by the modified International Phonetic Alphabet word for that letter creates a letter.

The most interesting commands to the e-mail tab are those that involve natural language input and do not show up in the sayable words list box. Those commands include, "add this person to my contact list", which takes the e-mail address and name of a person from the message currently selected and creates an entry for that person in the contact manager. Likewise, the command, "please send an e-mail to Bob", looks in the contact manager for a person with the name Bob, pulls out the e-mail address and puts that e-mail address in the "To:" text field. To assist in composing messages quickly and efficiently, it is possible to tell the PIM to "please say hello". This will write add a line with "hello" in the message text box. "Please say meet at 4 o'clock" will add the phrase "meet at 4:00" to the e-mail to enable assisting in the scheduling of meetings.

Each of these natural language commands is appealing to an end user, who can see them as spoken commands between individuals. These commands do not cover all potential ways of saying various phrases, for the clear reason that it is too difficult to specify all such commands. Furthermore, specifying too many variations would tax the recognizers. Ultimately, the included language commands can be read from the grammar specification elsewhere in this report.

The longer complex phrases are much more likely to be used, if available, in speech input form compared to the shorter command phrases, such as "load", "deliver". These longer complex phrases seem to be the are the area in which speech recognition has the greatest potential, even if the complex parsing routines that must be used to decipher the commands.

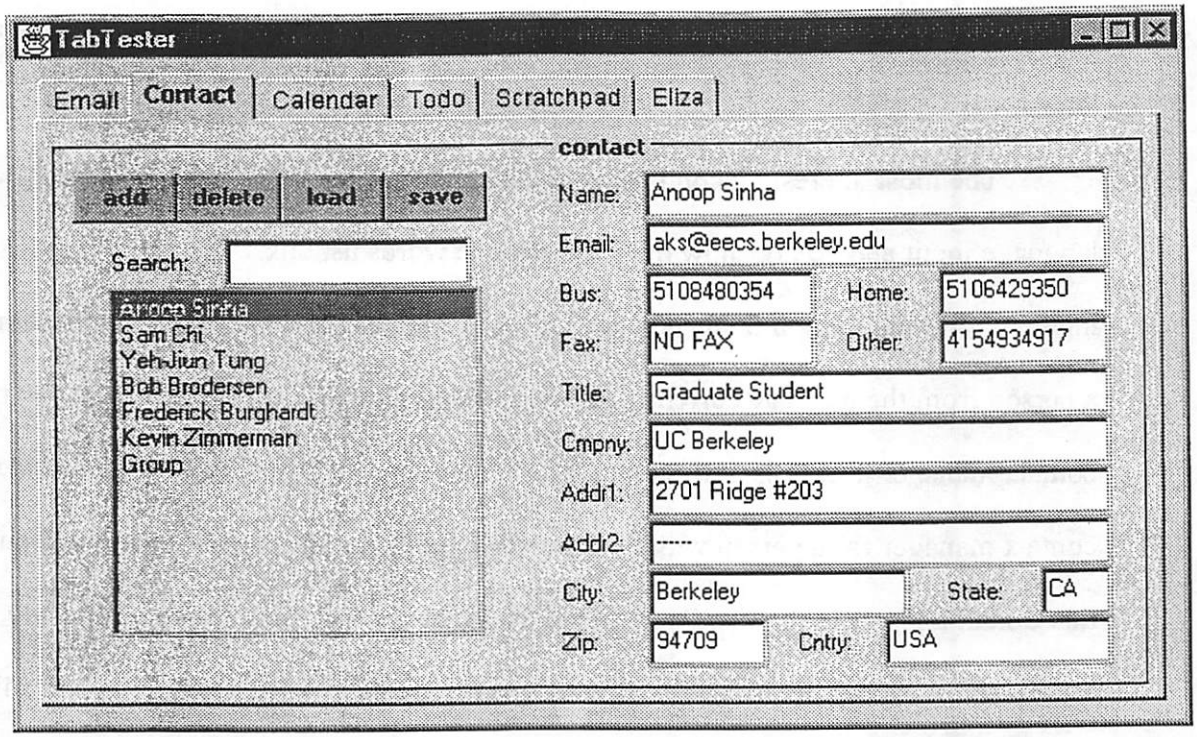

Figure 7-3. PIM Contact

The contact manager element is quite similar in structure to the e-mail program. The commands "add", "delete", "load" and "save" carry over but having different meaning when the contact manager is the active tab panel. An added distinction is the search text field, which can be used much like the autocompletion of the Java Virtual Keyboard. Each letter added to the search text field will move the highlight to the desired element. Another distinction is that "please find anoop" will perform the search completely and move the highlight to the name that starts with "anoop". One natural language phrase available in the contact manager is "please send an e-mail to this person." This phrase creates an e-mail sent to the e-mail address of the currently selected person.

The contact manager itself is a difficult element to completely automate using speech. The first problem is that the vocabulary of names is limited by the names that are included in the grammar. This set of names in the test grammar is arbitrary and based on the elements in the stored in the default contact manager file. Adding a new element to the contact manager also necessitates changing the speech grammar in order to be able to use that name in voice commands. Furthermore names are often quite distinctive and often mispronounced. This being the case, names are almost always most likely to be entered using the Java Virtual Keyboard rather than voice.

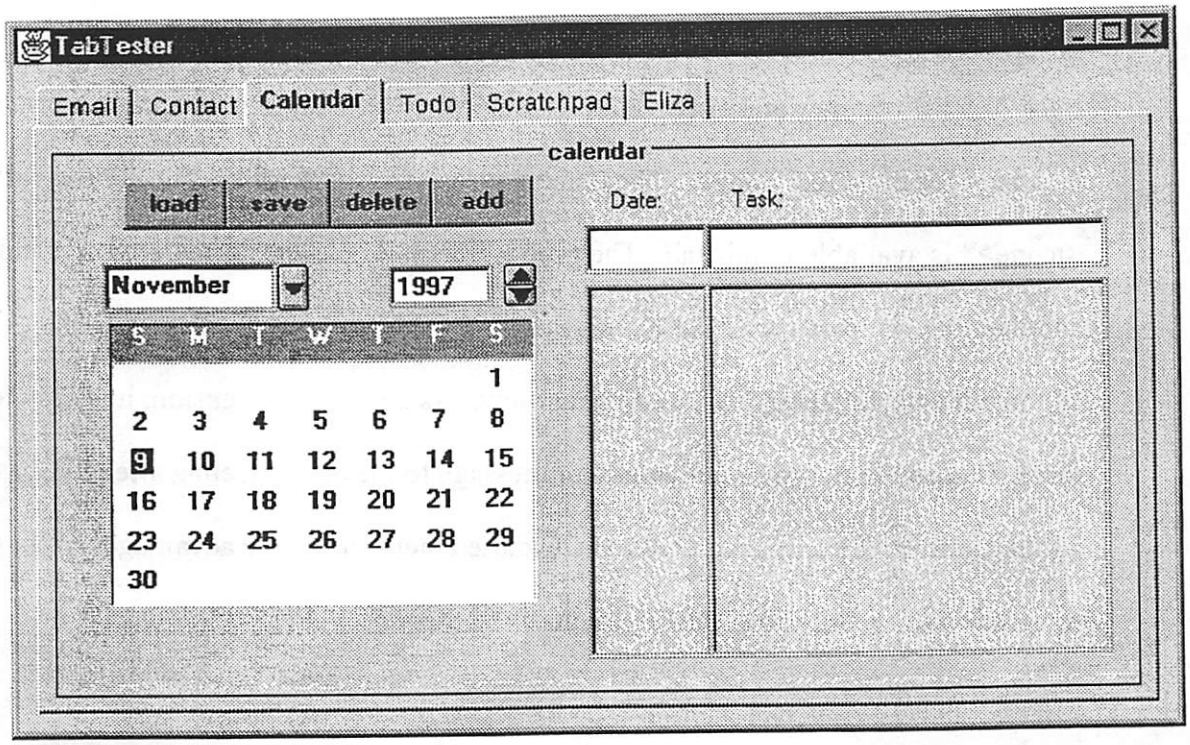

Figure 7-4. PIM Calendar

The calendar tab is interesting in that it has few simple speech commands and is almost always will be used with complex speech input. The most common is switching among dates by saying, "please go to <month> <date> <year>", where the month is the name of the month; date is a number from 1 to 31; and year as implemented can be from 1996 to 1999. That command will automatically update the calendar widget to display the proper month and year as well as move the selection to the proper date.

The Calendar widget itself is designed to be easy to use with a pointer input, and switching among months or years is quite simple. The added value in using speech is in eliminating a set of additional selection or processing, reducing the number of steps for switching a date from essentially three (using pen to select month, then day, then year) to one (the single spoken phrase).

The calendar functionality is simple and unremarkable in that it simply keeps track of a list of times and events for a given date. These essentially take the form of meetings for most individuals. The command "please add a meeting at <time> with <name>" is available in this tab. The time is internally parsed based on what is inputted, and the name is from the list of names that exist in the original contact list. The command adds this entry to the calendar view. As a possible extension, it is easy to see that it is possible to spawn off an e-mail message to the other meeting attendee, based on a look-up from the contact list. The calendar element shows the advantages of the tight integration of the different PIM elements.

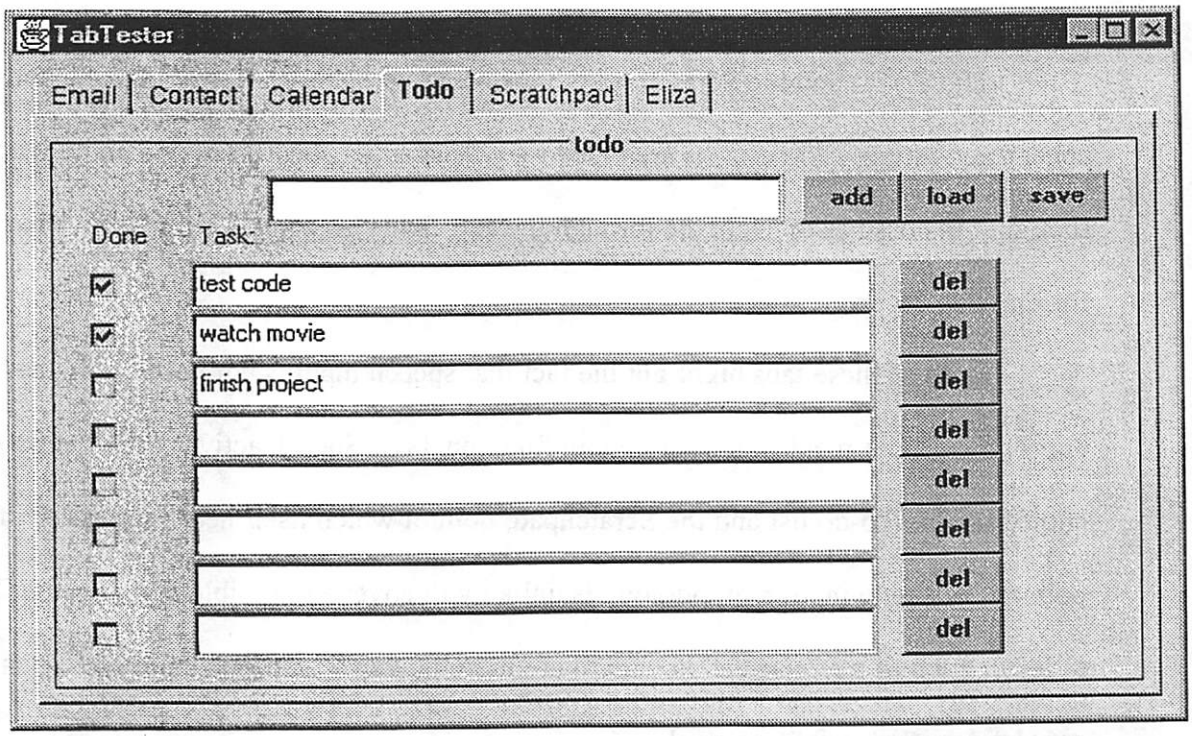

Figure 7-5. PIM To-do

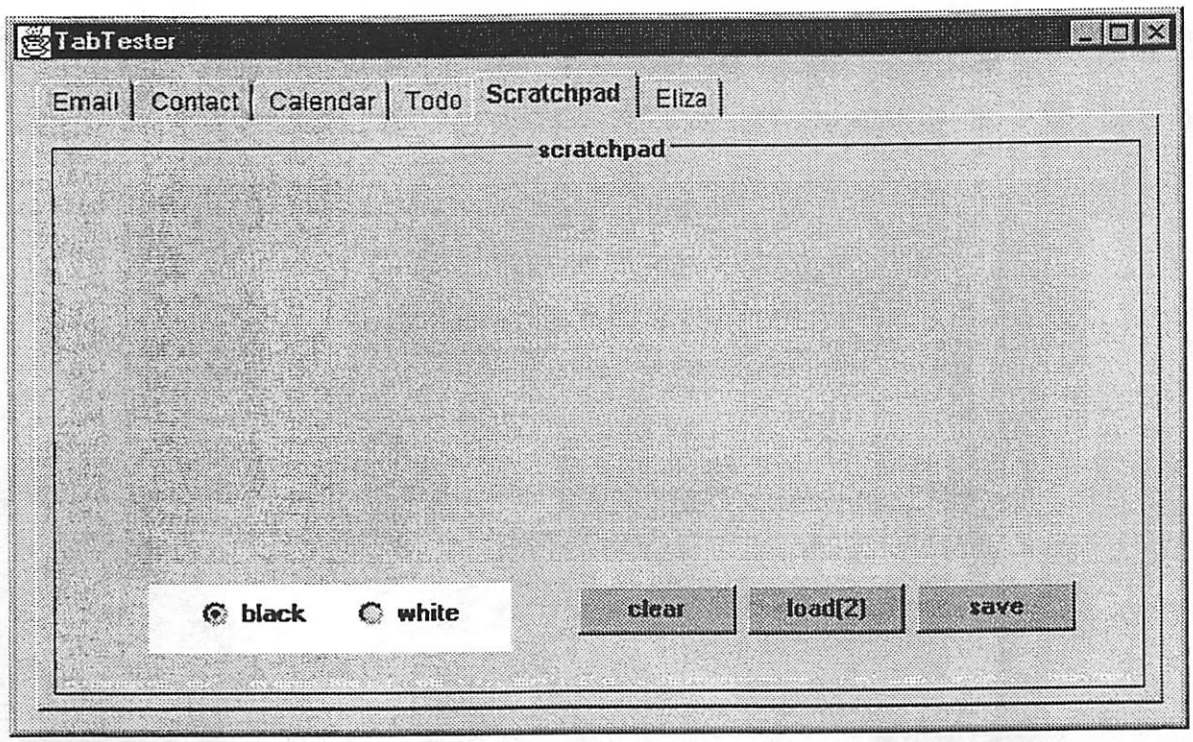

Figure 7-6. PIM Scratchpad

The To-do and the Scratchpad tabs are unremarkable and do not have much speech activation. To-do tasks are typed in by Virtual Keyboard rather than by voice, though it is possible to negotiate through the text fields by voice. Scratchpad is basically for drawing by pen.

Both of these tabs highlight the fact that speech input is not necessarily the appropriate input mechanism for certain functionality. Speech activation is the wrong choice for the To-do list and the Scratchpad, both of which itself necessitates use of the pointer. A single input methodology is unlikely to cover all possible input needs. Two together such as a mouse and keyboard are more powerful. Speech input and a pen input are also together quite powerful.

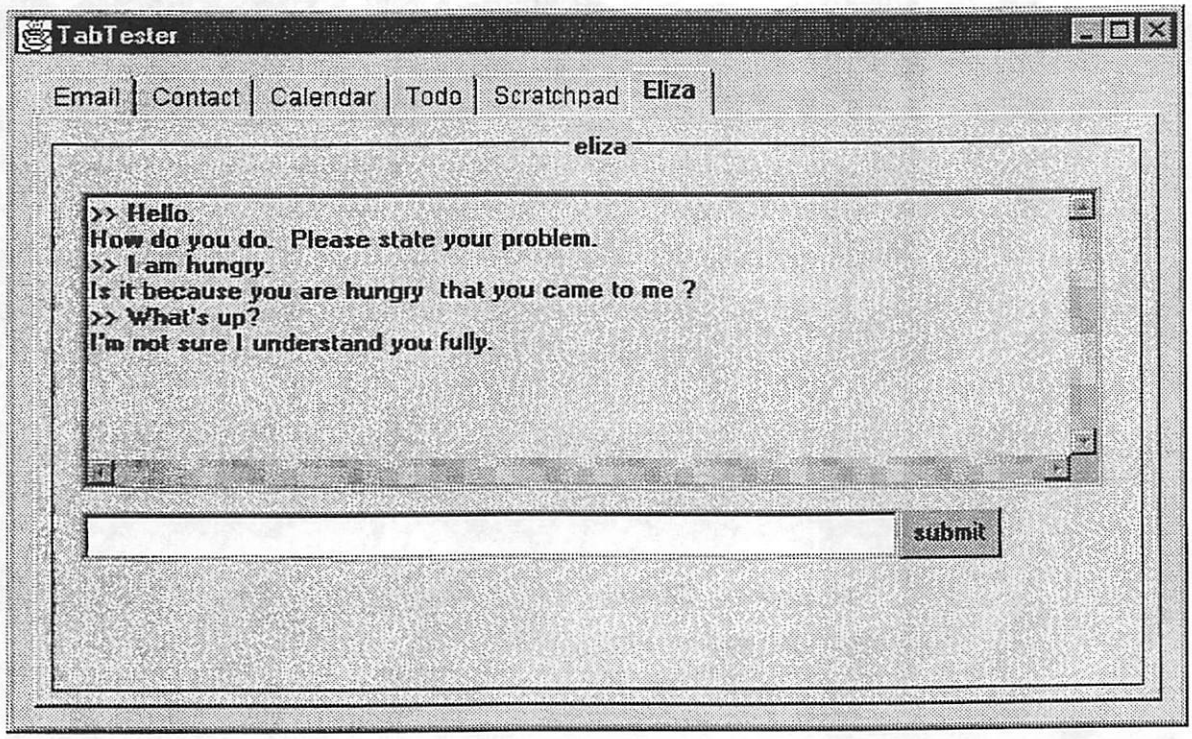

Figure 7-7. PIM Eliza

The last and perhaps most interesting, from a speech-activated perspective, element is the ELIZA tab. ELIZA was designed as a Rogerian psychologist to continue to probe and ask questions based on a given set of rules in response to input from a given user. It is a very simple program in that it does not comprehend information that it receives as input. It actually moves through a state machine, parsing each individual sentence and giving the appropriate response based on how it parses the sentence or what keywords it recognizes.

It is an interesting task for a speech recognition system since most of the commands that are specified in ELIZA involve simple psychological issues such as problems and other queries. The actual ELIZA specification is fairly large, and in some ways unlimited, since ELIZA can parse almost any sentence in someway and give the response, "I'm not sure I understand you fully." In the case of this speech recognition system, it was critical to limit the possible inputs to Eliza to a very limited subset. Limited even to the point of not likely covering the fully range of possibilities in a given conversation. However, enough of the subset was included in the vocabulary to be interesting.

Use of ELIZA certain shows promise for the use of speech in artificial intelligence applications. Once speech recognition capabilities improve enough, ELIZA shows that it will make sense to include free form speech input in a speech recognition system. The free form speech input increases the friendliness and the user's connection with the computer system. The set of spoken phrases that ELIZA as implemented can recognize can be deduced from the grammar specification.

The most important contribution of this PIM was not in the features it included in the PIM, but rather in how it shows that a speech inputted PIM has great potential, especially if natural language phrases are included in the input vocabulary. With speech as input, it is possible to have a great wealth of information transmitted in a single spoken sentence, which might affect all of the different PIM components in a coordinated fashion. This natural input interaction should make PIM's a viable replacement for cumbersome keyboard and mouse commands that are a part of computer PIM's today.

It is clear from this implementation, that training will almost certainly be required to enable users to know about the full range of functionality available in a speech activated system. With some training and accurate recognition, it is possible to maneuver quite readily among the different elements of the PIM, for instance, but some speech activated commands will remain unknown and unused unless enabled by training. Part of the issue in this regard is the design of the grammar and the functionality that the grammar covers as well as the words that the implementation recognizes.

VIII. Java Graph Editor

Another demonstration application developed using the Java Speech architecture was the speech-enabled Java Graph Editor. This application was developed to experiment with voice cues leading to visual results, showing us the potential future of speech activated applications even in a visual environments.

In the case of the Java Graph Editor, the pen as input is an option. However, commands are designed so that the entire functionality of the Java Graph Editor can be performed with speech.

The core Graph Editor code was largely due to work by Shilman [Shil97b]. His toolkit for graph element and editor classes for nodes and edges enabled quick creation of the speech enabled Graph Editor program.

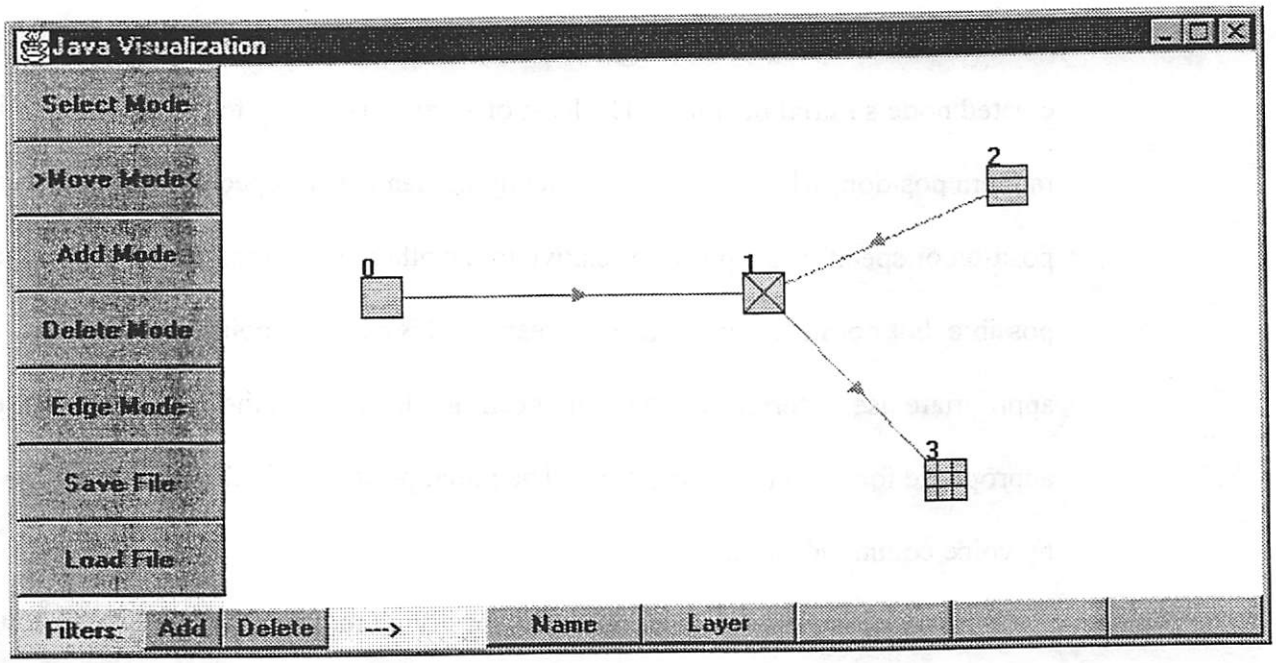

Figure 8-1. Java Graph Editor

The Java Graph Editor program was built using the Java Virtual Keyboard infrastructure. Though a virtual keyboard is not necessary for the program, using the Java Virtual Keyboard provides a consistent user view of the available speech commands in the sayable listbox. The Virtual Keyboard also takes care of the hooks to the speech recognizer.

The basic operations in the Graph Editor are quite simple: add node, delete node, move node, select a node, and create an edge between two nodes. These operations are enumerated in the mode buttons on the left-hand side of the application window. Switching modes by speech is the same as speech enabling the different tabs in the PIM application. "Switch to add mode", or "switch to edge mode" are examples of the required commands.

For this particular application, one point of consideration is how to determine a created node's initial position. The least of all evils is simply to create the node in a random position, which is what is presently implemented. Specifying a Cartesian position or specifying a position relative to the other nodes already in the graph is possible, but not necessarily ideal or clear. In this case, the pointer is the more appropriate user interface method for specifying location on the canvas; speech is not appropriate for this visual operation. The initial position is in fact not readily specified by voice commands alone.

Speech is also not appropriate for specifying movement of objects. Though voice can allow us to specify node movement up, down, left, or right, it is better to rely on the pen for moving nodes.

Speech definitely adds value in the selection of nodes. As implemented, each node has an associated number, to enable identification by speech. Saying "find one" or "find seven" will move the selection to the node number in question. If deletion is the desired operation, "delete one" or "delete seven" will delete the node with number specified if the node exists in the graph. These operations are quick and likely quicker than a visual search, selection, and deletion. They remove the need for a slow visual scan.

Speech is also similarly useful for this application in the connecting of nodes. It is much easier to specify "connect one to two and three" than it is to switch to edge mode and draw edge elements between the nodes one and two and one and three. The connection commands is also remarkable in its ability to concisely contain multiple different visual commands, much like the natural language commands in the PIM specified multiple operations.

Though this graph creation behavior is quite basic, it does show that in certain areas speech commands are not appropriate while in other areas speech commands improve the capabilities of the system. This is likely a general rule in the creation of speech activated system. Translated as a design rule: speech should be chosen for certain situations and applications and not used in others.

One final situation for this application in which speech activated control has good potential is in the use and specification of filters. The use of filters to generate interpretations from a graph is an idea introduced to the author by Shilman and implemented together for a 3-d graph visualization project [Shil97c]. The same ideas were used for this two dimensional graph layout application, in an initial form to show

the potential use of speech for filter specification. In the case of this application, it is possible to speak the command "show me the layer" or "show me the names", two different commands which apply the layer-filter and the name-filter to the graph in question respectively. These filters simply modify the nodes by changing the shape of the node, or the hatchmarks that are used to specify it, based on the internally stored layer value or the name.

It is quite clear that the concept of filters is quite powerful for analyzing a large graph where nodes have internally stored properties and external properties, which can be modified. Specifying filters using speech also appears quite superior to specifying filters by name. A complex filter could be specified with a very natural language command, such as "show me the people older than 30 who are female". The "show me" translates into a filter that the application determines, and the user can be hidden from the details of the specific filter or how that filter is created.

The operation of "show me" is one of most natural commands that can bring speech and visualization into complementary positions. In the particular case of graph layout where often one of the applications desired is data visualization and manipulation, speech activated commands are likely to be useful for specifying complex filters and viewpoints.

IX. Conclusions

This project was successful in introducing a new speech enabled application infrastructure for the Infopad and in showing the viability of speech as an input method for the Infopad. The project showed that speech has strong potential in the area of personal information management, and it is likely that speech-enabled application developed that manage personal information will gain acceptance. The project also showed that the usefulness of speech does not stop with PIMs, however, and can extend to a wide variety of command applications, even those with visual feedback such as a graph editor.

Speech is apparently most useful in conjunction with other input methodologies, and the combination of speech input and a pen is quite powerful for tablet based input. However, the work in this project suggests that a virtual keyboard is the best option, at the moment, for free form text input method. Though speech and pen input together are powerful, the general input provided by the keyboard suggest that it should be included as an alternative in systems requiring generic input.

Ultimately, application developers can easily use this Java-based architecture for their own programs. The speech server wrapper functionality is generic, and users who do not want to use the provided Java Virtual Keyboard or who want to use speech server technology for some other task, can modify the use of the speech recognition server wrapper to their own needs. Though it would require some coding, it is not difficult to use the speech server wrapper around any other sort of recognizer, perhaps even one day a speedy hardware enhanced speech recognition system.

X. Bibliography

[AF] http://gatekeeper.dec.com/pub/DEC/AF/

[Burs96] Burstein, Andrew. "Speech Recognition for Portable Multimedia Terminals." Ph. D. Thesis. Graduate Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, May 1996.

[Eliza] Hayden, Chris. http://24.3.201.16/chayden/eliza/Eliza.html

[Flan96] Flanagan, David. Java in a Nutshell. O'Reilly and Associates, 1996.

[IPA] http://www.dutch.nl/wilbwk/digit.htm

[Long95] Long, Allan Christian, Jr., Shankar Narayanaswamy, Andrew Burstein, Richard Han, Ken Lutz, Brian Richards, Samuel Sheng, Robert Brodersen, Jan Rabaey. "A Prototype User Interface for a Mobile Multimedia Terminal." Proceedings of the 1995 Computer Human Interface Conference, May 1995.

[Nara96a] Narayanaswamy, Shankar. "Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals." Ph. D. Thesis. Graduate Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, May 1996.

[Nara96b] Narayanaswamy, Shankar, Srinivasan Seshan, Eric Brewer, Robert Brodersen, Frederick Brughardt, Andrew Burstein, Yuan-Chi Chang, Armando Fox, Jeffrey Gilbert, Richard Han, Randy Katz, Allan C. Long, David Messerschmitt, Jan Rabaey. "Application and Network Support for Infopad." IEEE Personal Communications Magazine, March 1996.

[Nuan] Nuance Recognizer Documentation.

[Shil97a] Shilman, Michael. http://www-cad.eecs.berkeley.edu/~michaels/sax

[Shil97b] Shilman, Michael. http://www-cad.eecs.berkeley.edu/~michaels/.graph

[Shil97c] Shilman, Michael. http://www-cad.eecs.berkeley.edu/~michaels/courses/vr

[Tisr] TI Dagger System Documentation.

[Young] Young, Douglas. The X Window System: Porgramming and Applications with XT OSF/Motif Edition. Prentice Hall, 1990.

Appendix A. Grammar

```
start(<START>).
export(<START>).
<START> ---> <START1> | <IPA> | <KEYBOARD> | <PIM> | <GRAPH> | <NL>.
<START1> ---> go to sleep | wake-up | stop listening | start feedback | stop
feedback.
<NUMBER> ---> number <NUMBER2> | <NUMBER2>.
<NUMBER2> ---> <DIGITS> | <TEENS> | <TWENTIES> | <THIRTIES> | <NUMBER3>.
<NUMBER3> ---> <FORTIES> | <FIFTIES> | <SIXTIES> | <SEVENTIES> | <NUMBER4>.
<NUMBER4> ---> <EIGHTIES> | <NINETIES>.
<DIGITS> ---> zero | one | two | three | four | five | six | <DIGITS2>.
<DIGITS2> ---> seven | eight | nine | oh.
<TEENS> ---> ten | eleven | twelve | thirteen | fourteen | <TEENS2>.
<TEENS2> ---> fifteen | sixteen | seventeen | eighteen | nineteen.
<TWENTIES> ---> twenty | twenty-one | twenty-two | twenty-three | <TWENTY2>.
<TWENTY2> ---> twenty-four | twenty-five | twenty-six | <TWENTY3>.
<TWENTY3> ---> twenty-seven | twenty-eight | twenty-nine.
<THIRTIES> ---> thirty | thirty-one | thirty-two | thirty-three | <THIRTY2>.
<THIRTY2> ---> thirty-four | thirty-five | thirty-six | thirty-seven |
<THIRTY3>.
<THIRTY3> ---> thirty-eight | thirty-nine.
<FORTIES> ---> forty | forty-one | forty-two | forty-three | <FORTY2>.
<FORTY2> ---> forty-four | forty-five | forty-six | forty-seven | <FORTY3>.
<FORTY3> ---> forty-eight | forty-nine.
<FIFTIES> ---> fifty | fifty-one | fifty-two | fifty-three | <FIFTY2>.
<FIFTY2> ---> fifty-four | fifty-five | fifty-six | fifty-seven | <FIFTY3>.
<FIFTY3> ---> fifty-eight | fifty-nine.
<SIXTIES> ---> sixty | sixty-one | sixty-two | sixty-three | <SIXTY2>.
<SIXTY2> ---> sixty-four | sixty-five | sixty-six | sixty-seven | <SIXTY3>.
<SIXTY3> ---> sixty-eight | sixty-nine.
<SEVENTIES> ---> seventy | seventy-one | seventy-two | seventy-three |
<SEVENTY2>.
<SEVENTY2> ---> seventy-four | seventy-five | seventy-seven | seventy-seven |
<SEVENTY3>.
<SEVENTY3> ---> seventy-eight | seventy-nine.
<EIGHTIES> ---> eighty | eighty-one | eighty-two | eighty-three | <EIGHTY2>.
<EIGHTY2> ---> eighty-four | eighty-five | eighty-eight | eighty-seven |
<EIGHTY3>.
<EIGHTY3> ---> eighty-eight | eighty-nine.
<NINETIES> ---> ninety | ninety-one | ninety-two | ninety-three | <NINETY2>.
<NINETY2> ---> ninety-four | ninety-five | ninety-nine | ninety-seven |
<NINETY3>.
<NINETY3> ---> ninety-eight | ninety-nine.
<IPA> ---> toggle letter mode | letter <IPA1> | letter <KEYBOARD>.
<IPA1> ---> alphabet | bravo | charlie | delta | echo | foxtrot | go-golf |
<IPA2>.
<IPA2> ---> hotel | india | juliett | kilo | lima | mike | november | <IPA3>.
<IPA3> ---> oscar | papa | quebec | romeo | sierra-song | tango | uniform |
<IPA4>.
<IPA4> ---> victor | whiskey | x-ray | yankee | zulu.
<KEYBOARD> ---> space-bar | return | tab-key | backspace |<KEYBOARD2>.
<KEYBOARD2> ---> shift | alt-key | control | clear.
<MONTH> ---> january | february | march | april | may | june | <MONTH2>.
<MONTH2> ---> july | august | september | october | november | december.
<PIM> ---> <DEFAULT> | <EMAIL> | <CONTACT> | <CALENDAR> | <SCRATCH> | <PIM2>.
<PIM2> ---> <CHECK> | <TODO> | quit this program.
<DEFAULT> ---> sleep | wake-up | load | save | add | delete | previous | next |
<DEFAULT2>.
<DEFAULT2> ---> switch to <PANEL>.
<PANEL> ---> e-mail | contact | calendar | to-do | scratch-pad | eliza.
```

```
<EMAIL> ---> <EMAIL1>.
<EMAIL1> ---> deliver | compose | reply | get my e-mail | <EMAILF> text field.
<EMAILF> ---> to | subject | message.
<CONTACT> ---> search | <CONTACT1> text field.
<CONTACT1> ---> search | name | e-mail | business | home | fax | <CONTACT2>.
<CONTACT2> ---> other | title | company | address-one | <CONTACT3>.
<CONTACT3> ---> address-two | city | state | zip | country.
<CALENDAR> ---> <CALENDAR1> text field | <CALENDAR2>.
<CALENDAR1> ---> time | task.
<CALENDAR2> ---> year-up | year-down | <MONTH> | <MONTH> <NUMBER>.
<SCRATCH> ---> <SCRATCH1>.
<SCRATCH1> ---> black | white | clear.
<CHECK> ---> <CHECK1> text field.
<CHECK1> ---> number | date | payee.
<TODO> ---> <TODO1> text field.
<TODO1> ---> task | <DIGITS>.
<NL> ---> please <NL2>.
<NL2> ---> <NLEMAIL> | <NLCONTACT> | <NLCALENDAR> | <ELIZA>.
<NLEMAIL> ---> send an e-mail to <NAMES> | <NLE2>.
<NLE2> ---> send an e-mail to person <NUMBER>| say <SAYING> | <NLE3>.
<NLE3> ---> compose an e-mail | compose an e-mail to person <NUMBER> | <NLE4>.
<NLE4> ---> <NLE5>.
<NLE5> ---> e-mail <NAMES> | <NLE6>.
<NLE6> ---> add this person to my contact list.
<NAMES> ---> anoop | bob | yeh | sam | tony | fred | kevin | group.
<SAYING> ---> <GREETING> | <MEET> | <NAMES>.
<GREETING> ---> hello | good-bye | see ya | take care | sincerely.
<MEET> ---> meet at <MEETR> | meet at <MEETR> on <DATE> | meet on <DATE>.
<MEETR> ---> <NUMBER> o'clock | noon | midnight | <MEETR1>.
<MEETR1> ---> <NUMBER> fifteen | quarter past <NUMBER> | <MEETR2>.
<MEETR2> ---> <NUMBER> thirty | half past <NUMBER> | <MEETR3>.
<MEETR3> ---> <NUMBER> forty-five | quarter til <NUMBER>.
<DATE> ---> <MONTH> <NUMBER2> nineteen <NINETIES> | <MONTH> <NUMBER2>.
<NLCONTACT> ---> send an e-mail to this person | find <NAMES>.
<NLCALENDAR> ---> go to <DATE> | add a meeting at <MEETR> | <NLC2>.
<NLC2> ---> add a meeting at <MEETR> with <NAMES> | with <NAMES>.
<ELIZA> ---> write <ELIZA2> | eliza <ELIZA2> | computer <ELIZA2>.
<ELIZA2> ---> <E1> | <E2> | <E3> | <E4> | <E5> | <E6> | <E7>.
<E1> ---> i remember my <ZNOUN> | i remember my <ZPOS> <ZNOUN>.
<E2> ---> i am <ZADJ> | i was <ZADJ>.
<E3> ---> am i <ZADJ> | are you <ZADJ> | were you <ZADJ>.
<E4> ---> why don't you | why can't i | why not.
<E5> ---> yes | no | what | because | why | how | hello | computer.
<E6> ---> sorry | just because | none | perhaps.
<E7> ---> always | alike | if | bye | good bye.
<ZNOUN> ---> mom | dad | brother | sister | mother | father | <ZN1>.
<ZN1> ---> dog | cat | computer | car | fish | house | everyone | <ZN2>.
<ZN2> ---> everybody | nobody | noone | wife | child | family.
<ZADJ> ---> <ZADV> <ZA1> | <ZA1>.
<ZA1> ---> happy | sad | bad | good | funny | serious | curious | <ZA2>.
<ZA2> ---> bored | scared | excited | experienced | fine | <ZA3>.
<ZA3> ---> ecstatic | excellent | terrible | horrible | unhappy | <ZA4>.
<ZA4> ---> depressed | sick | elated | glad | better.
<ZADV> ---> very | not very | terribly | amazingly | especially | <ZAD1>.
<ZAD1> ---> always | never | forever.
<ZPOS> ---> your | my | our | her | his.
<GRAPH> ---> <GMODES> | save file | load file | look up | create | rename |
<GRAPH2>.
<GMODES> ---> switch to <GMODE1> mode | <GFILTER>.
<GMODE1> ---> move | add | delete | edge | select.
<GRAPH2> ---> <GDO> <NUMBER> | delete selected | remove selected | <GRAPH3>.
<GDO> ---> find | unselect | select | delete | remove.
<GRAPH3> ---> connect <NUMBER> <GCN> | disconnect <NUMBER> <GCN>.
```

```
<GCN> ---> <GN> | <GN> <GN> | <GN> <GN> <GN> | <GN> <GN> <GN> <GN> | <GCN2>.
<GCN2> ---> <GN> <GN> <GN> <GN> <GN> | <GN> <GN> <GN> <GN> <GN> <GN> | <GCN3>.
<GCN3> ---> <GN> <GN> <GN> <GN> <GN> <GN> <GN> | <GN> <GN> <GN> <GN> <GN> <GN>
<GN> <GN>.
<GN> ---> <GCON> <NUMBER>.
<GCON> ---> and | with | to | from.
<GFILTER> ---> add filter | add <GFTYPES> filter | show me the layers | <GF2>.
<GF2> ---> show me the names | delete filter | delete filter <NUMBER> | <GF3>.
<GF3> ---> show me the money | select filter <NUMBER>.
<GFTYPES> ---> reset | name | layer.
```

Location of Files (in Infopad cluster):

```
/tools/ui/speechDemos/
/tools/ui/speechDemos/Gramm/            -- grammar translation program
/tools/ui/speechDemos/Jgraph/           -- Java Graph Editor
/tools/ui/speechDemos/Pim/              -- Java PIM
/tools/ui/speechDemos/Nuance/           -- Nuance recognizer files
/tools/ui/speechDemos/TI/               -- TI recognizer files
/tools/ui/speechDemos/noise/            -- noise reduction files


/tools/ui/speechDemos/servenuance       --    start Nuance server recognizer on
                                              badlands.eecs.berkeley.edu only
/tools/ui/speechDemos/startnuance       --    start Nuance server wrapper
/tools/ui/speechDemos/startti           --    start TI recognizer and server
                                              wrapper
/tools/ui/speechDemos/runpim*           --    start the PIM
/tools/ui/speechDemos/rungedit*         --    start the GraphEdit program
```

Acknowledgments

I would like to thank Prof. Brodersen for his support of this project and his guidance. I would also like to thank Jeff Gilbert for his help in getting me started with this project. I would especially like to express my deep gratitude to Michael Shilman for many productive and enriching brainstorming sessions on this and other projects.