Emacspeak — The Complete Audio Desktop

T. V. Raman

Google Inc.

$Id: emacspeak-bc.xml 259 2006-12-21 18:54:05Z raman-tv $

1. Introduction — Defining The Problem

A *desktop* is a workspace that one uses to organize the tools of ones trade. Graphical desktops provide rich visual interaction for performing day-to-day computing tasks; the goal of the *audio desktop* is to enable similar efficiencies in an eyes-free environment. Thus, the primary goal of the *Audio Desktop* is to use the expressiveness of auditory output (both verbal and non-verbal) to enable the end-user perform a full range of computing tasks including:

- Communicate using the full range of electronic messaging services
- Ready access to local documents on the client and global documents on the Web
- Ability to develop software effectively in an eyes-free environment

The emacspeak audio desktop was motivated by the following insight; to provide effective auditory renderings of information, one needs to start from the actual information being presented, rather than a visual presentation of that information. This had earlier led to the development of AsTeR (http://emacspeak.sf.net/raman/aster/aster-toplevel.html) — Audio System For Technical Readings. The primary motivation then was to apply the lessons learnt in the context of aural documents to user interfaces — after all, the document *is* the interface. The primary goal was not to merely reflect the visual interface over to the auditory modality; the overarching design goal is to create an eyes-free user interface that is both pleasant and productive to use. Contrast this with the traditional screen-reader approach where GUI widgets such as sliders and tree controls are directly translated to spoken output. Though such direct translation can give the appearance of providing full eyes-free access, the resulting auditory user interface can become fairly inefficient to use.

These prerequisites meant that the environment selected for the audio desktop needed:

- A core set of speech and non-speech audio output services.
- A rich suite of pre-existing applications to speech-enable.
- Access to application context to produce contextual feedback.

2. Producing Spoken Output

I started implementing Emacspeak in October 1994; the target environments were a Linux laptop, and my office workstation. I used a DECTalk Express (a hardware speech synthesizer) on the laptop, and a software version of the DECTalk on the office workstation to produce speech output.

The most natural way to design the system to leverage both speech options was to first implement a speech server that abstracted away the distinction between the two output solutions. The speech server abstraction has withstood the test of time well; I was able to later add support for the IBM ViaVoice engine in 1999. Moreover, the simplicity of the client/server API has enabled Open Source programmers implement speech servers for other speech engines.

Emacspeak speech servers are implemented in TCL. The speech server for the DECTalk Express communicated with the hardware synthesizer over a serial line. As an example, the command to speak a string of text was a TCL `proc` that took a string argument and wrote it to the serial device. A simplified version of this looks like:

```
proc tts_say  {text} {puts -nonewline  $tts(write) "$text"}
```

The speech server for the software DECTalk implemented an equivalent `tts_say` simplified version of which looks like:

```
proc say  {text} {_say "$text"}
```

where `_say` calls the underlying C implementation provided by the software DECTalk.

The net result of this design was to create separate speech servers for each available engine, where each speech server was a simple TCL script that invoked TCL's default read-eval-print loop after loading in the relevant definitions. The client/server API therefore came down to the client (Emacspeak) launching the appropriate speech server, caching this connection, and invoking server commands by issuing appropriate procedure calls over this connection. Notice that so far nothing explicit has been stated about how this client/server connection was opened; this late-binding proved beneficial later when it came to making Emacspeak network aware. Thus, the initial implementation worked by the Emacspeak client communicating to the speech server using `stdio` . Later, making this client/server communication go over the network required the addition of a few lines of code that opened a server socket and connected `stdin/stdout` to the resulting connection. Thus, designing a clean client/server abstraction, and relying on the power of UNIX I/O made it trivial to later run Emacspeak on a remote machine and have it connect back to a speech server running on a local client. This enables me to run Emacspeak inside `screen` on my work machine, and access this running session from anywhere in the world; Upon connecting, I have the remote Emacspeak session connect to a speech server on my laptop — this is the audio equivalent of setting up X to use a remote display.

3. Speech-enabling Emacs

The simplicity of the speech server abstraction described above meant that version 0 of the speech server was running within an hour after I started implementing the system. This meant that I could then move on to the more interesting part of the project, namely producing good quality spoken output. Version 0 of the speech server was by no means perfect; it got improved upon as I built the Emacspeak speech client.

3.1. A Simple First-Cut Implementation

A friend of mine had pointed me at the marvels of Emacs Lisp advice a few weeks earlier. So when I sat down to speech-enable Emacs, advice was the natural choice. The first task was to have Emacs automatically speak the line under the cursor whenever one pressed the `up/down` arrow keys. In emacs, all user actions invoke appropriate Emacs Lisp functions. In standard editing modes, `down-arrow` invokes function `next-line` while pressing `up-arrow` invokes `previous-line` . To speech enable these commands, version 0 of Emacspeak implemented the following rather simple advice fragment:

```
(defadvice next-line (after emacspeak)
  "Speak line after moving."
  (when (interactive-p) (emacspeak-speak-line)))
```

In the above, function `emacspeak-speak-line` implemented the necessary logic to grab the text of the line under the cursor and send it to the speech server. With the above in place, Emacspeak 0.0 was up and running; it provided the scaffolding for building the actual system.

3.2. Iterating On The First Cut Implementation

The next iteration returned to the speech server to enhance it with a well-defined eventing loop. Rather than simply executing each speech command as it was received, the speech server was updated to queue client requests, and a `launch` command that caused the server to execute queued requests. The server used system call `select` to check for newly arrived commands after sending each clause to the speech engine. This enabled immediate silencing of speech; with the somewhat naive implementation described in version 0 of the speech server, the command to stop speech would not take immediate effect since the speech server would first process previously issued `speak` commands to completion. With the speech queue in place, the client application could now queue up arbitrary amounts of text, and still get a high degree of responsiveness when issuing higher priority commands such as requests to stop speech.

Implementing an event queue inside the speech server also gave finer control to the client application over how text was split into chunks before synthesis. This turns out to be crucial for producing good intonation structure. The rules by which text should be split up into clauses varies depending on the nature of the text being spoken. As an example, newline characters in programming languages like Python are statement delimiters and determine clause boundaries; but newlines do not constitute clause delimiters in English text. As an example, a clause boundary is inserted after each line when speaking the `python` code shown below.

```
i=1
j=2
```

See the section on generating rich auditory output (#rich-aui) for details on how the semantics of the content being `python` code are transferred to the speech layer.

With the speech server now capable of smart text handling, the Emacspeak client could become more sophisticated with respect to its handling of text. Function `emacspeak-speak-line` turned into a library of speech generation functions that implemented the following steps:

- Parse text to split it into a sequence of clauses.
- Preprocess text, e.g., handle repeated strings of punctuation marks.
- Plus a lot more that got added over time
- Queue each clause to the speech server, and issue the `launch` command.

From here on the rest of Emacspeak was implemented using Emacspeak as the development environment. This has been significant in how the codebase has evolved. New features can be tested immediately; badly implemented features can render the entire system unusable. Lisp's incremental code development fits naturally with the former; to cover the latter, the Emacspeak codebase has evolved to be *bushy* i.e., most parts of the higher-level system are mutually independent, and depend on a small core that is *carefully* maintained.

3.3. A Brief Advice Tutorial

Lisp *advice* is key to the Emacspeak implementation, and this article would not be complete without a brief overview. The *advice* facility allows one to modify existing functions *without* changing the original implementation. What is more, once a function `f` has been modified by advice `m`, all calls to function `f` are affected by the advice. Advice comes in three flavors:

before

Advice body is run *before* original function is invoked.

after

Advice body is run *after* the original function has completed.

around

Advice body is run *instead of* the original function. The body of the *around* advice can call the original function if desired.

All advice forms get access to the arguments of the adviced function; in addition, *around* and *after* advice forms get access to the return value computed by the original function. The Lisp implementation achieves this magic by:

- Caching the original implementation of the function
- Evaluating the advice form to generate a new function definition
- And storing this definition as the adviced function

Thus, evaluating the advice fragment shown earlier results in Emacs' original `next-line` function being replaced by a modified version that speaks the current line *after* the original `next-line` function has completed its work.

4. Generating Rich Auditory Output

At this point in its evolution, here is what the overall design looks like:

- Emacs' interactive commands are *adviced* to produce auditory output
- *Advice* definitions are collected into modules, one each for every Emacs application being speech-enabled
- Advice forms forward text to core speech functions
- These extract the text to be spoken and forward it to function `tts-speak`
- Function `tts-speak` produces auditory output by preprocessing it's `text` argument and sending it to the speech server
- Speech-server handles queued requests to produce perceptible output

Preprocessing of text is implemented by placing the text in a special scratch buffer. Buffers acquire specialized behavior via:

- Buffer-specific `syntax tables` that define the *grammar* of buffer contents
- Buffer-local variables that affect behavior

When text is handed off to the Emacspeak core, all of these buffer-specific settings are propagated to the special scratch buffer where the text is preprocessed. This automatically ensures that text is meaningfully parsed into clauses based on its underlying grammar.

4.1. Audio Formatting Using Voice-lock

Emacs uses `font-lock` to syntactically color text. For creating the visual presentation, Emacs adds a text property called `face` to text strings; the value of the `face` property specifies the font, color and style to be used to display that text. Text strings with `face` properties can be thought of as a conceptual *visual display list* .

Emacspeak augments these *visual display lists* with text property `personality` whose value specifies the auditory properties to use when rendering a given piece of text. The value of property *personality* is an Aural CSS (ACSS) setting that encodes various voice properties e.g., `pitch` of the speaking voice. Notice that such ACSS settings are not specific to any given TTS engine. Emacspeak implements ACSS to TTS mappings in engine-specific modules that take care of mapping high-level aural properties e.g., `pitch` or `pitch-range` to engine-specific control codes.

The next few sections describe how Emacspeak augments Emacs to:

- Create aural display lists
- Process these aural display lists to produce engine-specific output

4.1.1. Augmenting Emacs To Create Aural Display Lists

Emacs modules that implement `font-lock` call Emacs built-in function `put-text-property` to attach the relevant `face` property. Emacspeak defines an *advice* fragment that advices function `put-text-property` to add in the corresponding `personality` property when it is asked to add a `face` property. Note that the value of both display properties (`face` and `personality`) can be lists; values of these properties are designed to *cascade* to create the final (visual or auditory) presentation. This also means that different parts of an application can progressively add in display property values. Function `put-text-property` has the following signature. `(put-text-property START END PROPERTY VALUE &optional OBJECT)`

```
(defadvice put-text-property (after emacspeak-personality  pre act)
  "Used by emacspeak to augment font lock."
  (let ((start (ad-get-arg 0)) ;; Bind  arguments
        (end (ad-get-arg 1 ))
        (prop (ad-get-arg 2)) ;; name of property being added
        (value (ad-get-arg 3 ))
        (object (ad-get-arg 4))
        (voice nil))                    ;; voice it maps to
    (when (and (eq prop 'face)       ;; avoid infinite recursion
               (not (= start end))   ;; non-nil text range
               emacspeak-personality-voiceify-faces)
      (condition-case nil ;; safely look up face mapping
          (progn
            (cond
             ((symbolp value)
           (setq voice (voice-setup-get-voice-for-face value)))
          ((ems-plain-cons-p value)) ;;pass on plain cons
          ( (listp value)
            (setq voice
                  (delq nil
                        (mapcar   #'voice-setup-get-voice-for-face value))))
          (t (message "Got %s" value)))
         (when voice ;; voice  holds list of  personalities
           (funcall emacspeak-personality-voiceify-faces start end voice object)))
     (error nil)))))
```

Here is a brief explanation of this advice definition.

Bind Arguments

First, we use advice built-in `ad-get-arg` to locally bind a set of lexical variables to the arguments being passed to the adviced function.

Personality Setter

Mapping of faces to personalities is controlled by user customizable variable `emacspeak-personality-voiceify-faces` . If non-nil, this variable specifies a function with the following signature: `(emacspeak-personality-put START END PERSONALITY OBJECT)` Emacspeak provides different implementations of this function that either append or prepend the new personality value to any existing personality properties.

Guard

Along with checking for a non-nill `emacspeak-personality-voiceify-faces` we perform additional checks to determine whether this advice definition should do anything.

- That the text-range is non-nil
- That the property being added is a `face`

The first of these checks is required to avoid edge cases where function `put-text-property` is called with a 0-length text range. The second ensures that we only attempt to add property `personality` when the property being added is `face` . Notice that failure to include this second test will cause infinite recursion because the eventual `put-text-property` call to add property `personality` also triggers the advice definition.

Get Mapping

Next, we *safely* look up the voice mapping of the face (or faces) being applied. If applying a single `face` , we look up the corresponding personality mapping; if applying a list of faces, we create a corresponding list of personalities.

Apply Personality

Finally, we check that we found a valid voice mapping, and if so, call `emacspeak-personality-voiceify-faces` with the set of personalities saved in variable `voice` .

4.1.2. Audio Formatted Output From Aural Display Lists

With the advice definitions from the previous section in place, text fragments that are visually styled acquire a corresponding `personality` property that holds an ACSS setting for *audio formatting* the content. The result is to turn text in Emacs into rich aural display lists. This section describes how the output layer of Emacspeak is enhanced to convert these aural display lists into perceptible spoken output.

Emacspeak module `tts-speak` handles preprocessing of text before finally sending it to the speech server. As described earlier, this preprocessing comprises of a number of steps including:

- Applying pronunciation rules
- Processing repeated strings of punctuation characters
- Splitting text into appropriate clauses based on context
- Converting property `personality` into audio formatting codes

This section describes function `tts-format-text-and-speak` which handles the conversion of aural display lists into audio formatted output. We begin by showing the code for function `tts-format-text-and-speak` :

```
(defsubst tts-format-text-and-speak (start end )
  "Format and speak text between start and end."
  (when (and emacspeak-use-auditory-icons
             (get-text-property start 'auditory-icon)) ;;queue  icon
    (emacspeak-queue-auditory-icon (get-text-property start 'auditory-icon)))
  (tts-interp-queue (format "%s\n" tts-voice-reset-code))
  (cond
   (voice-lock-mode ;;  audio format only if voice-lock-mode is on
    (let ((last  nil) ;; initialize
          (personality (get-text-property start 'personality )))
      (while (and  (
< start end ) ;; chunk at personality changes
                   (setq last
                         (next-single-property-change  start 'personality
                                                       (current-buffer) end)))
        (if personality ;; audio format chunk
            (tts-speak-using-voice personality (buffer-substring start last ))
          (tts-interp-queue (buffer-substring  start last)))
        (setq start  last ;; prepare for next chunk
              personality (get-text-property last  'personality)))))
   ;; no voice-lock just send the text
   (t (tts-interp-queue (buffer-substring start end  )))))
```

Function `tts-format-text-and-speak` is called a clause at a time, with arguments `start` and `end` set to the start and end of the clause. If `voice-lock-mode` is turned on, this function further splits the clause into chunks at each point in the text where there is a change in value of property `personality` . Once such a transition point has been determined, it calls function `tts-speak-using-voice` with the personality to use and the text to be spoken. This function (described next, looks up the appropriate device specific codes before dispatching the audio formatted output to the speech server.

```
(defsubst tts-speak-using-voice (voice text)
  "Use voice VOICE to speak text TEXT."
  (unless (or (eq 'inaudible voice ) ;; not spoken if voice inaudible
              (and (listp voice) (member 'inaudible voice)))
    (tts-interp-queue
     (format
      "%s%s %s \n"
      (cond
       ((symbolp voice)
        (tts-get-voice-command
         (if (boundp  voice ) (symbol-value voice ) voice)))
       ((listp voice)
        (mapconcat  #'(lambda (v)
                        (tts-get-voice-command
                         (if (boundp  v ) (symbol-value v ) v)))
                    voice
                    " "))
       (t       ""))
      text tts-voice-reset-code))))
```

Function `tts-speak-using-voice` returns immediately if the specified voice is `inaudible` — `inaudible` is a special personality that Emacspeak uses to prevent pieces of text from being spoken. Personality `inaudible` can be used to advantage when selectively hiding portions of text to produce more succinct output. If the specified voice (or list of voices) is not `inaudible` , the function looks up the speech codes for the voice and queues the result of wrapping the text to be spoken between `voice-code` and `tts-reset-code` to the speech server.

4.2. Using Aural CSS (ACSS) For Styling Speech Output

Audio formatting was first formalized within AsTeR where rendering rules were written in a specialized language called Audio Formatting Language (AFL). AFL structured the available parameters in auditory space e.g., pitch of the speaking voice, into a multi-dimensional space, and encapsulated the state of the rendering engine as a point in this multi-dimensional space. AFL provided a block structured language where current rendering state was encapsulated by a lexically scoped variable, and operators to move within this structured space. When these notions were later mapped to the declarative world of HTML and CSS, dimensions making up the AFL rendering state became Aural CSS (http://www.w3.org/Press/1998/CSS2-REC) parameters. Though designed for styling HTML (and in general XML) markup trees, Aural CSS turned out to be a good abstraction for building Emacspeak's audio formatting layer while keeping the implementation independent of any given TTS engine.

Here is the definition of structure `acss` that encapsulates ACSS settings.

```
(defstruct  acss
  family gain left-volume right-volume
  average-pitch pitch-range stress richness punctuations)
```

Emacspeak provides a collection of pre-defined *voice overlays* for use within speech extensions. Voice overlays are designed to *cascade* in the spirit of Aural CSS. As an example, here is the ACSS setting that corresponds to `voice-monotone` .

```
[cl-struct-acss nil nil nil nil nil 0 0 nil all]
```

Notice that most fields of the `acss` structure shown above are `nil` i.e., undefined. The above setting creates a voice overlay that:

- Sets `pitch` to 0 to create a flat voice
- Sets `pitch-range` to 0 to create a monotone voice with no inflection
- Sets `punctuations` to *all* so that all punctuation marks are spoken

This setting is used as the value of property `personality` for audio formatting comments in all programming language modes. Because its value is an overlay, it can interact effectively with other aural display properties. As an example, if portions of a comment are displayed in a bold font, those portions can have personality `voice-bolden` (another predefined overlay) added; this results in the value of property `personality` becoming a list of two values `(voice-bolden voice-monotone)` . The final effect is for the text to get spoken with a distinctive voice that conveys both aspects of the text namely, a sequence of words that are emphasized within a comment.

4.3. Adding Auditory Icons

Rich visual user interfaces are made up of both text and icons; once Emacspeak had the ability to speak intelligently, the next step was to increase the band-width of aural communication by augmenting the output with auditory icons. Auditory icons in Emacspeak are short sound snippets (no more than 2 seconds in duration) and are used to indicate frequently occurring events in the user interface. As an example, every time the user saves a file, the system plays a confirmatory sound. Similarly, opening or closing an object (where object may be anything from a file to a Web site) produces a corresponding auditory icon. The set of auditory icons were arrived at iteratively, and cover common events such as objects being opened, closed or deleted. Next, we describe how these auditory icons are injected into Emacspeak's output stream.

Auditory icons are produced in the following user interaction scenarios:

- To cue explicit user actions
- To add additional cues to spoken output

Auditory icons that confirm user actions e.g., a file being saved successfully, are produced by adding an `after` advice to the various Emacs builtins. To provide a consistent sound and feel across the Emacspeak desktop, such extensions are attached to code that is called from many places in emacs. Here are two examples of such extensions, one implemented via an advice fragment, and the other implemented via an Emacs-provided hook.

```
(defadvice save-buffer (after emacspeak pre act)
  "Produce an auditory icon if possible."
  (when (interactive-p) (emacspeak-auditory-icon 'save-object)
    (or emacspeak-last-message (message "Wrote %s" (buffer-file-name)))))
```

As explained in the brief advice tutorial given earlier, advice is about extending/modifying the behavior of existing software without having to modify the underlying source-code. Emacs is itself an extensible system, and well-written Lisp code has a tradition of providing appropriate extension hooks for common use cases. As an example, Emacspeak attaches auditory feedback to Emacs' default prompting mechanism (the Emacs minibuffer) by adding function `emacspeak-minibuffer-setup-hook` to Emacs' `minibuffer-setup-hook` .

```
(defun emacspeak-minibuffer-setup-hook ()
  "Actions to take when entering the minibuffer."
  (let ((inhibit-field-text-motion t))
    (when emacspeak-minibuffer-enter-auditory-icon
      (emacspeak-auditory-icon 'open-object))
    (tts-with-punctuations 'all (emacspeak-speak-buffer))))
(add-hook  'minibuffer-setup-hook 'emacspeak-minibuffer-setup-hook)
```

This is a good example of using built-in extensibility where available. Emacspeak uses advice in a lot of cases because the Emacspeak requirement of adding auditory feedback to *all* of Emacs was not originally envisioned when Emacs was implemented. Thus, the Emacspeak implementation demonstrates a powerful technique for *discovering* extension points. Lack of an advice-like feature in a programming language often makes experimentation difficult, especially when it comes to discovering useful extension points. This is because software engineers are faced with the following trade-off:

- Make the system arbitrarily extensible, (and arbitrarily complex)
- Guess at some reasonable extension points and hard-code these

Once implemented, experimenting with new extension points requires rewriting existing code, and the resultant inertia often means that over time, such extension points remain mostly undiscovered. Lisp advice, and its Java counterpart Aspects offer software engineers the opportunity to experiment without worrying about adversely affecting an existing body of source-code.

4.3.1. Producing Auditory Icons While Speaking Content

In addition to using auditory icons to cue the results of user interaction, Emacspeak uses auditory icons to augment what is being spoken with additional information. Examples of such auditory icons include:

- A short icon at the beginning of paragraphs
- Auditory icon `mark-object` when moving across source lines that have a breakpoint set on them

This is achieved by attaching text property `emacspeak-auditory-icon` with value equal to the name of the auditory icon to be played on the relevant text. As an example, commands to set breakpoints in Emacs package `GUD` (Grand Unified Debugger) are adviced to add property `emacspeak-auditory-icon` to the line containing the breakpoint. When the user moves across such a line, function `tts-format-text-and-speak` queues the auditory icon at the right point in the output stream.

4.4. Enhancing Spoken Output With Context-sensitive Semantics

To summarize the story so far, Emacspeak has the ability to:

- Produce auditory output from within the context of an application
- Audio format output to increase the band-width of spoken communication
- Augment spoken output with auditory icons

4.4.1. Getting Context-Sensitive Feedback In The Calendar

I started implementing Emacspeak in October 1994 as a quick means of developing a speech solution for Linux. It was when I speech-enabled the Emacs Calendar in the first week of November 1994 that I realized that in fact I had created something far better than any other speech-access solution I had used before.

A calendar is a good example of using a specific type of visual layout that is optimized both for the visual medium as well as for the information that is being conveyed. We intuitively think of in terms of weeks and months when reasoning about dates; using a tabular layout that organizes dates in a grid with each week appearing on a row by itself matches this perfectly. With this form of layout, the human eye can rapidly move by days, weeks or months through the calendar, and easily answer such questions as:

- What day is it tomorrow
- Am I free third Wednesday of next month

Notice however that simply speaking this two-dimensional layout does not transfer the efficiencies achieved in the visual context to auditory interaction. This is a good example of where the right auditory feedback has to be generated directly from the underlying information being conveyed, rather than from its visual representation. When producing auditory output from visually formatted information, one has to *rediscover* the underlying semantics of the information before speaking it. In contrast, when producing spoken feedback via advice definitions that extend the underlying application, one has full access to the application's runtime context. Thus, rather than *guessing* based on visual layout, one can essentially instruct the underlying application to *speak the right thing* !

Module *emacspeak-calendar* speech-enables the Emacs Calendar by:

- Defining utility functions that speak calendar information
- And advising all calendar navigation commands to call these functions

Thus, Emacs Calendar produces specialized behavior by binding the arrow keys to calendar navigation commands rather than the default cursor navigation found in regular editing modes. Emacspeak specializes this behavior by advising the calendar specific commands to speak the relevant information in the context of the calendar. The net effect is that from an end-user's perspective, *things just work* — in regular editing modes, pressing up/down arrows speaks the current line; pressing up/down arrows in the calendar navigates by weeks and speaks the current date.

Function `emacspeak-calendar-speak-date` defined in module `emacspeak-calendar` is shown below. Notice that it uses all of the facilities described so far to access and audio format the relevant contextual information from the calendar.

```
(defsubst emacspeak-calendar-entry-marked-p()
  (member 'diary (mapcar #'overlay-face (overlays-at (point)))))
(defun emacspeak-calendar-speak-date()
  "Speak the date under point when called in Calendar Mode. "
  (let ((date (calendar-date-string (calendar-cursor-to-date t))))
    (cond
     ((emacspeak-calendar-entry-marked-p) (tts-speak-using-voice mark-personality date))
     (t (tts-speak date)))))
```

Emacs marks dates that have a diary entry with a special overlay. In the above, helper function `emacspeak-calendar-entry-marked-p` checks this overlay to implement a predicate that can be used to test if a date has a diary entry. Function `emacspeak-calendar-speak-date` uses this predicate to decide whether the date needs to be rendered in a different voice; dates that have calendar entries are spoken using voice `mark-personality` . Notice that function `emacspeak-calendar-speak-date` accesses the calendar's runtime context in the call `(calendar-date-string (calendar-cursor-to-date t))`.

Function `emacspeak-calendar-speak-date` is called from advice definitions attached to all calendar navigation functions. Here is the advice definition for function `calendar-forward-week`:

```
(defadvice calendar-forward-week (after emacspeak pre act)
  "Speak the date. "
  (when (interactive-p) (emacspeak-speak-calendar-date )
    (emacspeak-auditory-icon 'large-movement)))
```

This is an *after* advice since we want the spoken feedback to be produced *after* the original navigation command has done its work. The body of the advice definition first calls function `emacspeak-calendar-speak-date` to speak the date under the cursor; next, it calls function `emacspeak-auditory-icon` to produce a short sound indicating that we have successfully moved.

5. Painless Access To Online Information

With all the necessary affordances to generate rich auditory output in place, speech-enabling Emacs applications using Emacs Lisp's advice facility requires surprisingly small amounts of specialized code. With the TTS layer and the Emacspeak core handling the complex details of producing good quality output, the speech-enabling extensions focus purely on the specialized semantics of individual applications; this leads to simple and consequently *beautiful* code. This section illustrates this with a few choice examples taken from Emacspeak's rich suite of information access tools.

Right around the time Emacspeak was being started, there was a far more profound revolution taking place in the world of computing; The World Wide Web went from being a tool for academic research to becoming a mainstream tool for every-day tasks. This was 1994, when writing a browser was still a comparatively easy task. The complexity that has been progressively added to the WWW in the subsequent 12 years often tends to obscure the fact that the Web is still a fundamentally simple design where:

- Content creators publish Web resources addressable via URIs
- URI addressable content is retrievable via open protocols
- Retrieved content is in a well-understood markup language like HTML

Notice that the basic architecture of the WWW sketched above says little to nothing about how the content is made available to the end-user. The mid-90's saw the Web move toward increasingly complex visual interaction. The commercial Web with its penchant for flashy visual interaction increasingly moved away from the simple data-oriented interaction that had characterized early Web sites. By 1998, I found that the Web had a lot of useful interactive sites; to my dismay, I also found that I was using progressively fewer of these Web sites because of the time for task completion when using spoken output.

This led to the creation of a suite of Web-oriented tools within Emacspeak that went back to the basics of Web interaction. Emacs was already capable of rendering simple HTML into interactive hypertext documents. As the Web became complex, Emacspeak acquired a collection of interaction wizards built on top of Emacs' HTML rendering capability that progressively factored out the complexity of Web interaction to create an auditory interface that allows the user to quickly and painlessly listen to desired information.

5.1. Basic HTML With Emacs W3 And Aural CSS

Emacs W3 is a bare-boned Web browser implemented in the mid-90's. Emacs W3 implemented CSS (Cascading Style Sheets) early on, and this was the basis of the first Aural CSS implementation at the time I wrote the Aural CSS draft in February 96. Emacspeak speech-enables Emacs W3 via module `emacspeak-w3` which implements the following extensions:

- An `aural media` section in the default stylesheet for Aural CSS
- Advice all interactive commands to produce auditory feedback
- Special patterns to recognize and silence decorative images on Web pages
- Aural rendering of HTML form fields along with the associated `label` — this formed the basis of the design of the `label` element in HTML 4.
- Context-sensitive rendering rules for HTML form controls. As an example, given a group of radio buttons for answering the question "Do you accept? " Emacspeak extends Emacs W3 to produce a spoken message of the form *Radio group `Do you accept?` has `Yes` pressed.* and *Press this to change radio group `Do you accept?` from `Yes` to `No`* .
- Defines a `before` advice for Emacs W3 function `w3-parse-buffer` that applies user requested XSLT transforms to HTML pages

5.2. Module `emacspeak-websearch` For Task Oriented Search

By 1997, interactive sites on the Web ranging from `Altavista` for searching to `Yahoo Maps` for online directions required the user to:

- Fill in a set of form fields
- Submit the resulting form
- And be able to spot the results in the resulting complex HTML page

The first and third of these steps were the ones that took time when using spoken output; I needed to first locate the various form fields on a visually busy page, and wade through a lot of complex boilerplate material on result pages before I found the answer.

Notice that from the software design point of view, these steps neatly map into `pre-action` and `post-action` . Since Web interaction follows a very simple architecture based on URIs, the `pre-action` step of prompting the user for the right pieces of input can be factored out of a Web site and placed in a small piece of code that is run locally; this obviates the need for the user to open the initial launch page and seek out the various input fields. Similarly, the `post-action` step of spotting the actual results amidst the rest of the noise on the resulting page can also be delegated to software. Finally, notice that even though these `pre-action` and `post-action` steps are each specific to particular Web sites, the overall design pattern is one that can be generalized. This insight led to module `emacspeak-websearch` a collection of task-oriented Web tools that each:

- Prompted the user
- Constructed an appropriate URI and pulled the content at that URI
- Filtered the result before rendering the relevant content via Emacs W3

Here is the `emacspeak-websearch` tool for accessing directions from Yahoo Maps:

```
(defsubst emacspeak-websearch-yahoo-map-directions-get-locations ()
  "Convenience function for prompting and constructing the route component."
  (concat
   (format "&newaddr=%s"
           (emacspeak-url-encode (read-from-minibuffer "Start Address: ")))
   (format "&newcsz=%s"
           (emacspeak-url-encode (read-from-minibuffer "City/State or Zip:")))
   (format "&newtaddr=%s"
           (emacspeak-url-encode (read-from-minibuffer "Destination Address: ")))
   (format "&newtcsz=%s"
           (emacspeak-url-encode (read-from-minibuffer "City/State or Zip:")))))
(defun emacspeak-websearch-yahoo-map-directions-search (query )
  "Get driving directions from Yahoo."
  (interactive
   (list (emacspeak-websearch-yahoo-map-directions-get-locations))
   (emacspeak-w3-extract-table-by-match
    "Start"
    (concat emacspeak-websearch-yahoo-maps-uri query))))
```

Here is a brief explanation of the code shown above:

Pre-Action

Function `emacspeak-websearch-yahoo-map-directions-get-locations` prompts the user for the start and end locations. Notice that this function hard-wires the names of the query parameters used by Yahoo Maps; on the surface, this looks like a kluge that is guaranteed to break. In fact, this kluge has not broken since it was first defined in 1997. The reason is obvious; once a Web application has published a set of query parameters, those parameters get hard-coded in a number of places, including within a large number of HTML pages on the originating Web site. Depending on the parameter names may feel brittle to the software architect used to structured, top-down APIs; but use of such URL parameters to define bottom-up web services leads to the notion of RESTful Web APIs.

Retrieve content

The URL for retrieving directions is constructed by concatenating the user input to the *base URI* for Yahoo Maps.

Post-Action

The resulting URI is passed to function `emacspeak-w3-extract-table-by-match` along with a search pattern `Start` to:

- Retrieve the content using Emacs W3
- Apply an XSLT transform to extract the table containing `Start`
- Render this table using Emacs W3's HTML formatter

Unlike the query parameters, the layout of the results page *does* change about once a year on average. But keeping this tool current with Yahoo Maps comes down to maintaining the `post-action` portion of this utility. In over 8 years of use, I have had to modify it about half a dozen times, and given that the underlying platform provides many of the tools for filtering the result page, the actual lines of code that need to be written for each layout change is minimal. Function `emacspeak-w3-extract-table-by-match` uses an XSLT transformation that filters a document to return tables that contain a specified search pattern. For this example, the function constructs the following XPath expression:

```
(/descendant::table[contains(., Start)])[last()]
```

This effectively picks out the list of tables that contain the string `Start` and returns the last element of that list.

7 years after this utility was written, Google launched Google Maps to great excitement in February 2005. Many blogs on the Web put Google Maps under the microscope and quickly discovered the query parameters used by that application. I used that to build a corresponding Google Maps tool in Emacspeak that provides similar functionality. The user experience is smoother with the Google Maps tool because the start and end locations can be specified within the same parameter. Here is the code for the Google Maps wizard:

```
(defun emacspeak-websearch-emaps-search (query &optional use-near)
  "Perform EmapSpeak search. Query is  in plain English."
  (interactive
   (list
    (emacspeak-websearch-read-query
     (if current-prefix-arg
         (format "Find what near  %s: "
                 emacspeak-websearch-emapspeak-my-location)
       "EMap Query: "))
    current-prefix-arg))
  (let ((near-p ;; determine query type
         (unless use-near
           (save-match-data (and (string-match "near" query) (match-end 0)))))
        (near nil)
        (uri nil))
    (when near-p ;; determine location from query
      (setq near (substring query near-p))
      (setq emacspeak-websearch-emapspeak-my-location near))
    (setq uri
          (cond
           (use-near
            (format emacspeak-websearch-google-maps-uri
                    (emacspeak-url-encode
                     (format "%s near %s" query near))))
           (t (format emacspeak-websearch-google-maps-uri
                      (emacspeak-url-encode query)))))
    (add-hook 'emacspeak-w3-post-process-hook 'emacspeak-speak-buffer)
    (add-hook  'emacspeak-w3-post-process-hook
               #'(lambda nil
                   (emacspeak-pronounce-add-buffer-local-dictionary-entry
                    "&#240;mi" " miles ")))
    (browse-url-of-buffer
     (emacspeak-xslt-xml-url
      (expand-file-name "kml2html.xsl" emacspeak-xslt-directory)
      uri))))
```

Here is a brief explanation of the code:

- Parse input to decide whether it's a direction or a search query
- In case of search queries, cache user's location for future use
- Construct URI for retrieving results
- Browse the results of filtering the contents of the URI through XSLT filter `kml2html` which converts the retrieved content into a simple hypertext document
- Set up custom pronunciations in the results to pronounce `mi` as `miles`

Notice that as before, most of the code focuses on application specific tasks. Rich spoken output is produced by creating the results as a well-structured HTML document with the appropriate Aural CSS rules producing an audio formatted presentation.

5.3. The Web Command Line And URL Templates

With more and more services becoming available on the Web, another useful pattern emerged by early 2000; Web sites started creating smart client-side interaction via Javascript. One typical use of such scripts was to construct URLs on the client-side for accessing specific pieces of content based on user input. As an example, Major League Baseball constructs the URL for retrieving scores for a given game by piecing together the date and the names of the home and visiting teams; NPR creates URLs by piecing together the date with the program code of a given NPR show. To enable fast access to such services, I added an `emacspeak-url-template` module in late 2000. This module has become a powerful companion to module `emacspeak-websearch` described in the previous section, and together, these modules turn the Emacs `minibuffer` into a powerful Web Command Line that provides rapid access to Web content.

5.3.1. Obtaining Temporally Information Using Calendar Context

Many web services require the user to specify a date. One can usefully default the date by using the user's calendar to provide the context. Thus, Emacspeak tools for playing an NPR program or retrieve MLB scores default to using the date under the cursor from invoked within the Emacs calendar buffer.

URL templates in Emacspeak are implemented using the following data structure:

```
(defstruct (emacspeak-url-template (:constructor emacspeak-ut-constructor))
  name                                  ;; Human-readable name
  template                              ;; template URL string
  generators;; list of param generator
  post-action                   ;; action to perform after opening
  documentation                         ;; resource  documentation
  fetcher;; custom fetcher
  dont-url-encode)
```

Users invoke `url templates` via Emacspeak command `emacspeak-url-template-fetch` which prompts for a url template and:

- Looks up the named template
- Prompts the user by calling the specified `generator`
- Applies Lisp function `format` to the template string and the collected arguments to create the final URI
- Sets up any `post actions` performed after the content has been rendered
- Applies specified fetcher to render the content

The use of this structure is best explained with an example; here is the `url template` for playing NPR programs.

```
(emacspeak-url-template-define
 "NPR On Demand"
 "http://www.npr.org/dmg/dmg.php?prgCode=%s&showDate=%s&segNum=%s&mediaPref=RM"
 (list
  #'(lambda () (upcase (read-from-minibuffer "Program code:")))
  #'(lambda ()
      (emacspeak-url-template-collect-date "Date:" "%d-%b-%Y"))
  "Segment:")
 nil; no post actions
 "Play NPR shows on demand.
Program is specified as a program code:
ME              Morning Edition
ATC             All Things Considered
day             Day To Day
newsnotes       News And Notes
totn            Talk Of The Nation
fa              Fresh Air
wesat           Weekend Edition Saturday
wesun           Weekend Edition Sunday
fool            The Motley Fool
Segment is specified as a two digit number --specifying a blank value
plays entire program."
 #'(lambda (url)
     (funcall emacspeak-media-player url 'play-list)
     (emacspeak-w3-browse-xml-url-with-style
      (expand-file-name "smil-anchors.xsl" emacspeak-xslt-directory)
      url)))
```

In this example, the custom `fetcher` performs two actions:

- Launches a media player to start playing the audio stream
- Filters the associated `SMIL` document via XSLT `smil-anchors` .

5.4. The Advent Of Feed Readers

At the time modules `emacspeak-websearch` and `emacspeak-url-template` were implemented, Emacspeak needed to *screen-scrape* HTML pages to speak the relevant information. But as the Web grew in complexity, the need to readily get at *real content* became more than something specific to eyes-free access; Users capable of working with complex visual interfaces also found themselves under a serious information overload. This led to the advent of RSS and Atom feeds, and the concomitant arrival of feed reading software. This has had a very positive effect on the emacspeak codebase in that in the last few years, the code has become *more beautiful* as I have progressively deleted screen-scraping logic and replaced it with direct content access. As an example, here is the Emacspeak `url template` for retrieving the weather for a given city/state:

```
(emacspeak-url-template-define
 "rss weather from wunderground"
 "http://www.wunderground.com/auto/rss_full/%s.xml?units=both"
 (list "State/City e.g.: MA/Boston") nil
 "Pull RSS weather feed for specified state/city."
 'emacspeak-rss-display)
```

And here is the `url template` for Google News searches via Atom feeds:

```
(emacspeak-url-template-define
 "Google News Search"
 "http://news.google.com/news?hl=en&ned=tus&q=%s&btnG=Google+Search&output=atom"
 (list "Search news for: ") nil "Search Google news."
 'emacspeak-atom-display )
```

Both of these tools use all of the facilities provided by module `emacspeak-url-template` and consequently need to do very little on their own. Finally, notice that by relying on standardized feed formats like RSS and Atom, these templates now have very little site-specific kluges in contrast to older tools like the Yahoo Maps wizard which had specific patterns from the results page hard-wired into the implementation.

6. Summary

Emacspeak was conceived as a full-fledged eyes-free user interface to every-day computing tasks. To be a *full-fledged* interface, the system needed to provide direct access to every aspect of computing on desktop workstations. To enable fluent *eyes-free* interaction, the system needed to treat spoken output and the auditory medium as a first-class citizen, i.e., merely reading out information displayed on the screen was not sufficient. To provide a *Complete Audio Desktop* , the target environment needed to be an interaction framework that was both widely deployed and fully extensible. To be able to do more than *just speak the screen* , the system needed to *build in* speech interaction capability into the various applications. Finally, this had to be done *without* modifying the source code of any of the underlying applications; The project could not afford to *fork* a suite of applications in the name of adding eyes-free interaction, since I wanted to limit myself to the task of maintaining the speech extensions.

To meet the design requirements spelled out above, I picked Emacs as the user interaction environment. As an interaction framework, Emacs had the advantage of having a very large developer community; unlike other popular interaction frameworks of the time, it had the significant advantage of being an Open Source environment — This was in 1994; 12 years later, Firefox affords similar opportunities. The enormous flexibility afforded by Emacs Lisp as as extension language was an essential prerequisite in speech-enabling the various applications. The Open Source nature of the platform was just as crucial; even though I had made an explicit decision that I would modify *no existing code* , being able to study how various applications were implemented made speech-enabling them tractable. Finally, the availability of a high-quality *advice* implementation for Emacs Lisp (note that Lisp's *advice facility* was the prime motivator behind Aspect Oriented Programming) made it possible to *speech-enable* applications authored in Emacs Lisp without modifying the original source code. Emacspeak is a direct consequence of the matching up of the needs outlined above and the affordances provided by Emacs as a user interaction environment.

6.1. Managing Code Complexity Over Time

The Emacspeak codebase has evolved over a period of 12 years. Except for the first 6 weeks of development, the codebase has been developed and maintained using Emacspeak. This section summarizes some of the lessons learned with respect to managing code complexity over time.

Throughout its 12 years of existence, Emacspeak has *always* remained a spare-time project. Looking at the codebase across time, I believe this has had a significant impact on how the code has evolved. When working on large, complex software systems as a full-time project, one has the luxury of focusing one's entire concentration on the codebase for reasonable stretches of time e.g. 6 to 12 weeks. This results in tightly implemented code that creates *deep* codebases. Despite one's best intentions, this can also result in code that becomes hard to understand with the passage of time. Large software systems where a single engineer focuses exclusively on the project for a number of years are almost non-existent; that form of single-minded focus usually leads to rapid burn out!

In contrast, Emacspeak is an example of a large software system that has had a single engineer focused on it over a period of 12 years, where this focus has been a spare-time activity. A consequence of developing the system single-handedly over a number of years is that the codebase has tended to be naturally *bushy* . Notice the distribution of files and lines of code in the summary distribution shown below:

Table 1. Summary Of Emacspeak Codebase

Layer	Files	Lines	Percentage
TTS Core	6	3866	6.0
Emacspeak Core	16	12174	18.9
Emacspeak Extensions	160	48339	75.0
Total	182	64379	99.9

In the above, notice the following:

- The TTS core responsible for high-quality speech output is isolated in 6 out of 182 files and makes up 6% of the codebase
- The Emacspeak core which provides high-level speech services to Emacspeak extensions, in addition to speech-enabling all basic Emacs functionality, is isolated to 16 files and makes up about 19% of the codebase
- The rest of the system is split across 160 files, which can be independently improved (or broken) without affecting the rest of the system
- Finally, many modules, e.g., module `emacspeak-url-template` are *bushy* within themselves, i.e.,

an individual url template can be modified without affecting any of the other url templates

- Advice reduces code size. The Emacspeak codebase which has approximately 60,000 lines of Lisp code is a fraction of the size of the underlying system being speech-enabled. A rough count at the end of December 2006 shows that Emacs 22 has over a million lines of Lisp code; in addition, Emacspeak speech-enables a large number of applications not bundled by default with emacs

6.2. Conclusion

To conclude this chapter, here is a brief summary of the insights gained from implementing and using Emacspeak:

- Lisp advice, and its object-oriented equivalent Aspect Oriented Programming, are a very effective means for implementing cross-cutting concerns, e.g., speech-enabling a visual interface.
- Advice is a powerful means for discovering potential points of extension in a complex software system.
- Focusing on basic Web architecture and relying on a data-oriented Web backed by standardized protocols and formats leads to powerful spoken Web access.
- Focusing on the final user experience, as opposed to individual interaction widgets such as sliders and tree-controls lead to a highly efficient eyes-free environment.
- Visual interaction relies heavily on the human eye's ability to rapidly scan the visual display. Effective eyes-free interaction requires transferring some of this responsibility to the computer, since listening to large amounts of information is time-consuming. Search in every form on the continuum between
  - Emacs incremental search to find the right item in a local document
  - Google search to quickly find the right document on the global Web

  is critical for delivering effective eyes-free interaction.
- Visual complexity that is an irritant for users capable of using complex visual interfaces are a show-stopper for eyes-free interaction. Conversely, tools that emerge early in an eyes-free environment eventually show up in the mainstream when the nuisance value of complex visual interfaces crosses a certain threshold. Two examples of this from the Emacspeak experience are:
  - Advent of RSS and Atom feeds replace the need for screen-scraping
  - Emacspeak's use of XSLT to filter content in 2000 is paralleled by the advent of Greasemonkey for applying custom client-side Javascript to Web pages in 2005

6.3. Acknowledgements

Emacspeak would not exist without Emacs and the ever-vibrant Emacs developer community that has made it possible to do everything from within Emacs. The Emacspeak implementation would not have been possible without Hans Chalupsky's excellent advice implementation for Emacs Lisp. Project `libxslt` from the Gnome project has helped breathe fresh life into William Perry's Emacs W3

browser — Emacs W3 was one of the early HTML rendering engines but the code has not been updated in over 8 years. That the W3 codebase is still usable and extensible bears testimony to the flexibility and power afforded by Lisp as the implementation language.