From the book: Integration-Ready Architecture and Design, by Cambridge University Press, ISBN 0521525837, Jeff Zhuk
http://javaschool.com/about/publications.html

"Dave: Open the pod bay door, Hal...
Hal: I'm sorry, Dave. I'm afraid I can't do that."

– A conversation between a man and a computer from the movie "2001: A Space Odyssey" by Stanley Kubrick

Voice Technologies on the way to a Natural User Interface

This chapter is about speech technologies and related APIs: VoiceXML, SALT, Java Speech API, MS Speech SDK. It looks into unified scenarios with audio and video interface definitions; considers design and code examples; and introduces important skills for a new world of wired and wireless speech applications.

What is a Natural User Interface?

Is it another set of tags and rules covered by the nice name? Absolutely not!

This time end users, not a standards committee, make the determination of what they prefer for their methods of interaction. A natural user interface (NUI) allows end users to give their preferences at the time of the service request, and change them flexibly.

Are you a "computer" person?

My guess is that you are, because you are reading this book. "Computer literate" folks like you and me enjoy exploring the capacities of computer programs via traditional interfaces. Even so, there are still times, such as on vacation, on the go, and in the car, when even we would prefer "hands free" conversations over using keyboards to access computerized services.

One person prefers handwriting, and someone else is comfortable with typing. One would like to forget keywords and use common sense terminology instead. Can a computer understand that "find" is the same as "search" and "Bob" is actually "Robert"? Can it understand that someone has chosen a foreign (non-English) language to interact with it?

A natural user interface will be able to offer all these possibilities. Some of these complex tasks can be addressed in a unified way with Audio-Video Interface (AVI) scenarios.

First, let us look into Java details of the voice interface, one part of a NUI. A significant part of the population would like a voice interface as the most natural and preferred way of interaction.

Speaking with style

Chapter 4 introduced text-to-speech conversion. We used the FreeTTS Java Speech API implementation to write simple Java code for a voice component, but the sound of the voice may not have been terribly impressive. What can the Java Speech API (JSAPI) [1] offer to enhance a voice component?

Here are two lines of the source (from the chapter 4) that actually speak for themselves:

Voice talker = new CMUDiphoneVoice();
talker.speak(speech);

Remember that the Voice class is the main talker.

We can create an object of the Voice class with four features: voice name, gender, age, and speaking style. The voice name and speaking style are both String objects, and the synthesizer determines the allowed contents of those strings.

The gender of a voice can be GENDER_FEMALE, GENDER_MALE, GENDER_NEUTRAL, or GENDER_DONT_CARE. Gender neutral means some robotic or artificial voices. We can use the "don't care" value if the feature is not important and we are “OK” with any available voice.

The age can be AGE_CHILD (up to 12 years), AGE_TEENAGER (13-19), AGE_YOUNGER_ADULT (20-40), AGE_MIDDLE_ADULT (40-60), AGE_OLDER_ADULT (60+), AGE_NEUTRAL, and AGE_DONT_CARE.

Here is the example of a woman’s voice selection with a “greeting” voice style:

Voice("Julie", GENDER_FEMALE, AGE_YOUNGER_ADULT, "greeting");

Not all the features of Java Speech API are implemented yet, as of the time of writing. Here is the reality check: the *match()* method of the *Voice* class can test whether an engine-implementation has suitable properties for the voice.

Fig.12-1.java illustrates the point.

```
/**
 * The getVoices() method creates a set of voices according to initial
age and gender parameters
 * The method checks if the voice if available.
 * If requested age or gender is not available the default voice
parameters are set.
 * @param gender
 * @param ages
 * @return voices
 */
    public Voice[] getVoices(int[] gender, int[] ages) {
        // make sure that at least default set
```

```
        if(gender == null) {
            gender = new int[1];
            gender[0] = GENDER_DONT_CARE;
        }
        if(ages == null) {
            ages = new int[1];
            ages[0] = AGE_DONT_CARE;
        }
        Voice[] voices = new Voice[ages.length * gender.length]; // all
combinations
        SynthesizerModeDesc desc = new
SynthesizerModeDesc(Locale.ENGLISH);
        Voice[] availableVoices = desc.getVoices();
        // try to set requested voices
        for (int i = 0; i < ages.length; i++) {
            for (int j = 0; j < gender.length; j++) {
                int k=(i+1)*j; // current voice index
                // try to set voice according to requirements
                // and check availability
                boolean available = false;
                // start from gender
                voices[k].setGender(gender[j]);
                for(int n=0; !available && n<availableVoices.length; n+
+) {
                    if (voices[k].match(availableVoices[n])) {
                        available = true;
                    }
                }
                if(!available) {
                    voices[k].setGender(GENDER_DONT_CARE);
                }
                // continue with ages
                voices[k].setAge(ages[i]);
                for(int n=0; !available && n<availableVoices.length; n+
+) {
                    if (voices[k].match(availableVoices[n])) {
                        available = true;
                    }
                }
                if(!available) {
                    voices[k].setAge(AGE_DONT_CARE);
                }
            }
        }
        // at this point all requested voices set to requested
parameters or to default
        return voices;
    }
```

[Fig.12-1]

The getVoices() method creates a set of voices according to initial age and gender parameters. If the requested age or gender is not available, the default voice parameters are set.

We can use this method in the scenario of multiple actors that have different gender and ages.

<actors name=”age” value=” AGE_TEENAGER | AGE_MIDDLE_ADULT” />
<actors name=”gender” value=”GENDER_FEMALE | GENDER_MALE” />

These two scenario lines define arrays of age and gender arguments that produce four actor voices.

Java™ Speech API Markup Language

An even more intimate control on voice characteristics can be achieved by using the *Java™ Speech API Markup Language* [2] (JSML). JSML is a subset of XML that allows applications to annotate text that is to be spoken, with additional information. We can set prosody rate or speed of speech. For example:

Your <emphasis>United Airlines</emphasis> flight is scheduled tonight <prosody rate="-20%">at 8:80pm</prosody> It is almost 4pm now. Good time to get ready.

This friendly reminder emphasizes the airline company name, and slows down the voice speed 20% while pronouncing the departure time.

According to the Java Speech API, the synthesizer’s *speak()* method understands JSML. JSML has more element names or tags, in addition to *emphasis* and *prosody.*

For example, the *div* marks text content structures such as paragraph and sentences.

*Hello How are you *

The *sayas* tag provides important hints to the synthesizer on how to treat the text that follows. For example, “3/5” can be treated as a number or as a date.

<sayas class="date:dm">3/5</sayas>
<!-- spoken as "May third" -->

<sayas class="number">1/2</sayas>
<!-- spoken as "half" -->

<sayas class="phone">12345678901</sayas>
<!-- spoken as "1-234-567-8901" -->

<sayas class="net:email">jeff.zhuk@javaschool.com</sayas>
<!-- spoken as "Jeff dot Zhuk at Javaschool dot com" -->

<sayas class="currency">$9.95</sayas>
<!-- spoken as "nine dollars ninety five cents" -->

<sayas class="literal">IRS</sayas>
<!-- spoken as character-by-character "I.R.S" -->

<sayas class="measure">65kg</sayas>
<!-- spoken as "sixty five kilograms" -->

In addition, a voice tag specifies the speaking voice.

For example:

<voice gender="male" age="20"> Do you want to send fax?</voice>

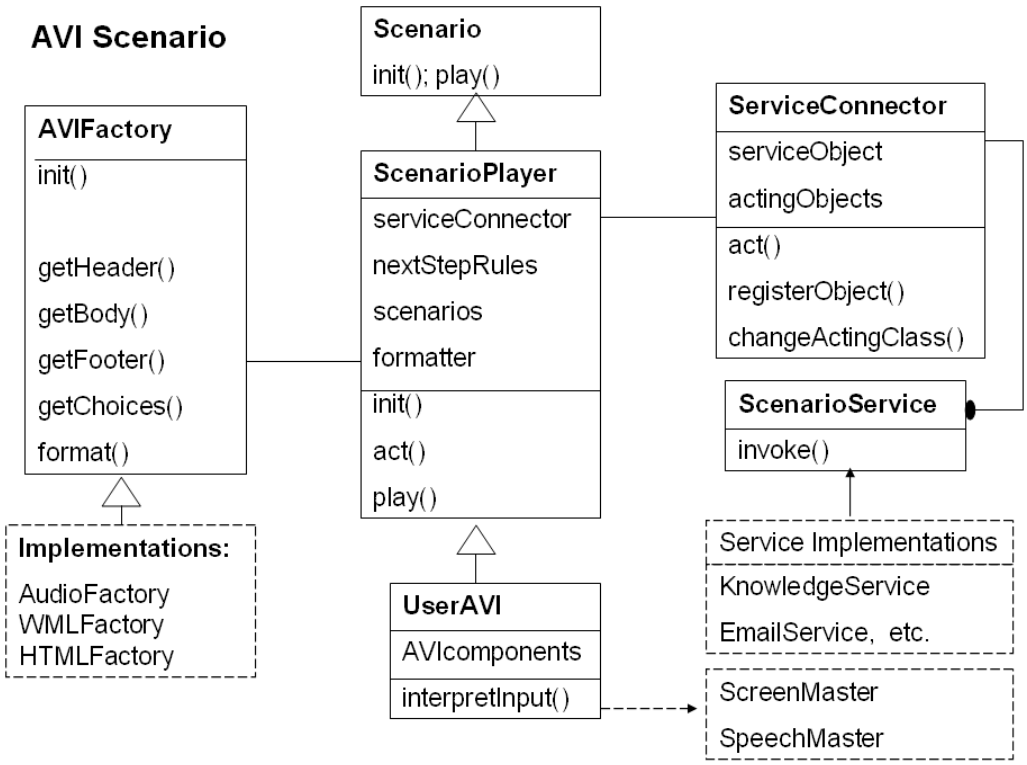
You can see that we can define the speaking voice in our Java code (see Fig.12-1) or in JSML. Which way is preferable?

For the famous “Hello World” application, it is easier to specify voice characteristics directly in the code. JSML would be the better answer for real-life production quality applications; it gives more control of speech synthesis.

JSML Factory as one of the AVIFactory implementations

JSML based text can be generated on the fly by an appropriate lightweight presentation layer component of the application. No adjustment of the core services is required. We can use XML Style Sheet Language for Transformations (XSLT) to automatically convert core service contents into HTML or JSML forms.

Fig.12-2 shows the Object Model Diagram for such an application.



[Fig.12-2]

The AVIFactory interface on the left side of the Fig.12-2 has multiple implementations for audio and video presentation formats. Any AVIFactory implementation transforms data into JSML, HTML, or other presentation formats.

The ScenarioPlayer class plays a selected scenario and uses the ServiceConnector object to invoke any service that retrieves data according to a scenario.

The ScenarioPlayer class creates the proper AVIFactory presenter object to transform data into the proper audio or video presentation format.

In this chapter, we will look into an example of the AudioFactory while in the chapter 14 we will consider implementation examples of the ScenarioPlayer and other classes.

Speech Synthesis Markup Language

Is JSML the best way to represent voice data? We have to ask several more questions before we can answer this one. For example, what are the current standards in the speech synthesis and recognition area?

Speech Synthesis Markup Language (SSML) [3] is an upcoming W3C standard; the SSML Specification Version 1.0 might be already approved by now.

JSML and SSML are not exactly the same (surprise!). Both are XML-based definitions for voice synthesis characteristics. Do not panic! It is relatively easy to map SSML to JSML for at least some voice characteristics.

An example of an SSML document is provided in Fig.12-3.xml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- SSML Document-->

<speak version="1.0"
       xmlns="http://www.w3.org/2001/10/synthesis"
       xml:lang="en-US">
   <sentence>
           Message from <prosody rate="-30%"> Deena Malkin </prosody>
         delivered <say-as type="date"> 02/13/2003 </say-as>
           research paper on <prosody rate="-30%"> SSML </prosody>
   </sentence>
</speak>
```

[Fig.12-3]

The differences between SSML and JSML are very visible but similarities are even more impressive.

Here is a brief review of basic SSML elements.

audio – allows insertion of recorded audio files

break - an empty element to set the prosodic boundaries between words

emphasis - increases the level of stress with which the contained text is spoken

mark - a specific reference point in the text

paragraph - indicates the paragraph structure of the document

phoneme - provides the phonetic pronunciation for the contained text

prosody - controls the pitch, rate, and volume of the speech output

say-as - indicates the type of text contained in the element; for example, date or number

sentence - indicates the sentence structure of the document

speak – includes the document body, the required root element for all SSML documents

sub – substitutes a string of text that should be pronounced in place of the text contained in the element

voice - describes speaking voice characteristics

Here is an example of another SSML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<speak version="1.0"
     xmlns="http://www.w3.org/2001/10/synthesis"
     xml:lang="en-US">

<sentence>

Your friendly reminder
<prosody pitch="high" rate="-20" volume="load">it is only
 <say-as type="number"> 5 </say-as> days left till the Valentine day</prosody>

</sentence>

</speak>
```

From SSML to JSML

A simple example of mapping SSML to JSML is provided with the SSMLtoJSMLAudioFactory class, which implements the AVIFactory interface and represents a version of the AudioFactory, Fig.12-4.java.

```
// SSMLtoJSMLFactory

package com.its.connector;

import javax.speech.*;
import javax.speech.synthesis.*;
import java.util.Locale;

/**
 * The SSMLtoJSMLFactory class is to transform SSML data into JSML
format
 * And present the data
 * @author Jeff.Zhuk@JavaSchool.com
 */
public class SSMLtoJSMLAudioFactory implements AVIFactory, Speakable {
    private String source;
    private String jsml; // formatted JSML string
    private Synthesizer synthesizer;
    // private Voice[] voices; // optional
```

```
    // private int[] ages = {AGE_TEENAGER,AGE_MIDDLE_ADULT};
    // private int[] gender = {GENDER_FEMALE, GENDER_MALE};
    private String[] SSMLmap = {"<sentence>","</sentence>","<say-as
type","</say-as>"};
    private String[] JSMLmap = {"<div type=\"sent\">","</div>","<sayas
class","</sayas>"}
    /**
     * The init() method is to initiate data
     * @param source
     */
    public void init(String source) {
        this.source = source;
    }
    /**
     * The initComponents() method initializes synthesizer
     * The method can optionally use getVoices() to init voice
components
     */
    public void initComponents() {
        try {
            // Create a synthesizer for English language
            synthesizer = Central.createSynthesizer(new
SynthesizerModeDesc(Locale.ENGLISH));
            // Get it ready to speak
            synthesizer.allocate();
            synthesizer.resume();

            // voices = getVoices(gender, ages); // optional
        } catch(Exception e) {
            System.out.println("SSMLtoJSMLAudioFactory.init:ERROR
creating synthesizer");
        }
    /**
     * The getHeader() method is to provide a proper presentation header
     * @return header
     */
    public String getHeader() {
        return "<?xml version=\"1.0\"?>\n<jsml>\n";
    }
    /**
     * The getBody() method is to proved a proper body in the
presentation format
     * @return body
     */
    public String getBody() {
        // extract body from the original source
        String body = Stringer.getStringBetweenTags(source, "speak");

        // replace the SSMLmap cases with the JSML tags
        // Stringer.replaceIgnoreCase is similar to the replaceAll() of
String in 1.4
        for(int i=0; i<SSMLmap.length; i++) {
            body = Stringer.replaceIgnoreCase(body,
SSMLmap[i],JSMLmap[i]);
        }
        return body;
    }
```

```
    /**
     * The getFooter() method is to provide a proper footer
     * @return footer
     */
    public String getFooter() {
        return "</jsml>";
    }
    // more methods
```

[Fig.12-4]

The SSMLtoJSMLAudioFactory class implements the AVIFactory interface. There are at least six methods that must be provided in this class. The Fig.12-4 displays four of them.

The *init()* method initializes an original source. The *initComponents*() method creates the synthesizer object and optionally can use *getVoices*() method considered in Fig.12-1 to initialize voice components. The *getHeader*() method returns the standard JSML header. The *getFooter*() method returns the standard JSML footer.

The *getBody*() method extracts a speakable text from the original SSML source. The “speak” tags frame this text. We use the *Stringer.getStringBetweenTags*() method for body extraction.

The next move is to map SSML tags with appropriate JSML tags. We use two string array-maps: the SSMLmap array and the corresponding JSMLmap array.

A simple loop uses the *Stringer.replaceIgnoreCase*() method to map these tags. (We will consider the Stringer class with its methods in chapter 14.)

Play JSML

Note that the *SSMLtoJSMLAudioFactory* class implements the *Speakable* interface. The *Speakable* interface is the spoken version of the *toString()* method of the Object class. Implementing the *Speakable* interface means to implement the *getJSMLText*() method. The *getJSMLText*() method as well as the *play*() method are shown in Fig.12-5.java

```
    /**
     * The getJSMLText() method returns JSML string
     * @return jsml
     */
    public String getJSMLText() {
        jsml = getFooter() + getBody() + getFooter();
    }
    /**
     * The play() method uses the factory properties to present the
content
     */
```

```
    public void play() {
        if(synthesizer == null) {
            initComponents();
        }
        synthesizer.speak(jsml, null);
    }
    /**
     * The getChoices() method returns an XML string with expected user
input
     * @return xml
     */
    public String getChoices() {
        return null;
    }
    /**
     * The getVoices() method creates a set of voices according to
initial age and gender parameters
     * The method checks if the voice if available.
     * If requested age or gender is not available the default voice
parameters are set.
     * @param gender
     * @param ages
     * @return voices
     */
    public Voice[] getVoices(int[] gender, int[] ages) {
        // make sure that at least default set
        if(gender == null) {
            gender = new int[1];
            gender[0] = GENDER_DONT_CARE;
        }
        if(ages == null) {
            ages = new int[1];
            ages[0] = AGE_DONT_CARE;
        }
        Voice[] voices = new Voice[ages.length * gender.length]; // all
combinations
        SynthesizerModeDesc desc = new
SynthesizerModeDesc(Locale.ENGLISH);
        Voice[] availableVoices = desc.getVoices();
        // try to set requested voices
        for (int i = 0; i < ages.length; i++) {
            for (int j = 0; j < gender.length; j++) {
                int k=(i+1)*j; // current voice index
                // try to set voice according to requirements
                // and check availability
                boolean available = false;
                // start from gender
                voices[k].setGender(gender[j]);
                for(int n=0; !available && n<availableVoices.length; n+
+) {
                    if (voices[k].match(availableVoices[n])) {
                        available = true;
                    }
                }
                if(!available) {
                    voices[k].setGender(GENDER_DONT_CARE);
                }
```

```
                // continue with ages
                voices[k].setAge(ages[i]);
                for(int n=0; !available && n<availableVoices.length; n+
+) {
                    if (voices[k].match(availableVoices[n])) {
                        available = true;
                    }
                }
                if(!available) {
                    voices[k].setAge(AGE_DONT_CARE);
                }
            }
        }
        // at this point all requested voices set to requested
parameters or to default
        return voices;
    }
}   // end of the class
```

[Fig.12-5]

With the source code presented in Fig.12-4, the *getJSMLText*() method can be implemented as a single line that concatenates the header, the transformed body, and the footer of the JSML string.

The *play*() method uses the factory properties to present the JSML content. The *play*() method starts by checking whether the synthesizer is ready to talk. Then it uses the synthesizer object to invoke its *speak*() method, passing the JSML string as one of the arguments.

The other argument is a *SpeakableListener* object. This object can be used to receive and handle events associated with the spoken text. We do not plan to use the *SpeakableListener* object in our example, so we passed the null object as the second argument. It is possible to use the normal mechanisms for attachment and removal of listeners with *addSpeakableListener()* and *removeSpeakableListener()* methods.

Speech Recognition with Java

Speech technologies are not limited by speech synthesis. Speech recognition technologies have matured to the point that the default message “Please repeat your selection” is not so commonplace anymore, and human-computer conversation can go beyond multiple-choice menus.

There are recognizer programs for personal usage, corporate sales, and other purposes. Most personal recognizers support dictation mode. They are speaker-dependent, requiring “program training” that creates a "speaker profile" with a detailed map of the user's speaking patterns and accent. Then the program uses this map to improve recognition accuracy.

The Java Speech API offers a *Recognizer* that may, optionally, provide a *SpeakerManager* object that allows an application to manage the *SpeakerProfiles* of that *Recognizer*. The *getSpeakerManager()* method of the *Recognizer* interface returns the *SpeakerManager* if this option is available for this *Recognizer*. Recognizers that do not maintain speaker profiles - known as speaker-independent recognizers - return null for this method.

A single recognizer may have multiple *SpeakerProfiles* for one user, and may store the profiles of multiple users.

The *SpeakerProfile* class is a reference to data stored with the recognizer. A profile is identified by three values: its unique id, its name, and its variant. The id and the name are self-explanatory. The variant identifies a particular enrollment of a user, and becomes useful when one user has more than one enrollment, or *SpeakerProfile*.

Additional data stored by a recognizer with the profile might include:

- Speaker data such as name, age, gender, etc.
- Speaker preferences
- Data about the words and word patterns of the speaker (language models)
- Data about the pronunciation of words by the speaker (word models)
- Data about the speaker's voice and speaking style (acoustic models)
- Records of previous training and usage history.

Speech recognition systems (SRS) can listen to users and, to some degree, recognize and translate their speech to words and sentences. Current speech technologies have to constrain speech context with *grammars.* Today, the systems can achieve “reasonable” recognition accuracy only within these constraints.

The Java Speech Grammar Format.

The *Java™ Speech Grammar Format* (JSGF) [4] is a platform and vendor independent way of describing a *rule grammar* (also known as a *command and control* grammar or *regular grammar*).

A rule grammar specifies the types of *utterances* a user might say. For example, a service control grammar might include “Service,” and "Action” commands.

A voice application can be based on a set of scenarios. Each scenario knows the context and provides appropriate grammar rules for the context.

Grammar rules can be provided in multi-lingual manner. For example:

<greetings.english.hello>
<greetings.russian.privet>

<greetings.deutsch.gutenTag>

“Hello,” “Privet,” and “GutenTag” are tokens in the grammar rules. Tokens define expected words that may be spoken by a user. The world of tokens forms a *vocabulary* or *lexicon*. Each record in the vocabulary defines the *pronunciation* of the token.

A single file defines a single grammar with its header and body. The header defines the JSGF version and (optionally) encoding. For example:

#JSGF V1.0 ISO8859-5;

The grammar starts with the grammar name and is similar to java package names. For example:

grammar com.its.scenarios.examples.greetings;

We can also *import* grammar rules and packages as is usually done in Java code:

import <com.its.scenarios.examples.cyc.> ; // talk to knowledge base*

The grammar body defines *rules* as a rule name followed by its definition-token. The definition can include several alternatives separated by “|” characters.

For example:

<login> = login ;
<find> = find | search | get | lookup ;
<new> = new | create | add ;
<command> = <find> | <new> | <login> ;

We can use the Kleene star (after Stephen Cole Kleene, originator) or the “+” character to set expectations that user can repeat a word multiple times.

*<agree> = I * agree | yes | OK ; //* “I agree” and “agree” - both covered
<disagree> = no + ; // no can be spoken 1 or more times

The Kleene star and the plus operator are both *unary operators* in the JSGF. There is also the tag unary operator that helps to return application-specific information as the result of recognition.

For example:

<service> = (mail | email) {email} | (search | research | find) {find} ;

The system returns the word “email” if “mail” or “email” was spoken. In the case when one of the words “search,” “research,” or “find” was spoken, the system returns the word “find.”

The mainstream of speech recognition technologies lies outside of the Java Speech API today. (This may be different next year.) One example is the open source Sphinx [5] project written in C++ by a group from Carnegie Mellon University.

In the Sphinx system, recognition takes place in three passes (the last two are optional): lexical-tree Viterbi search, flat-structured Viterbi search, and global best-path search.

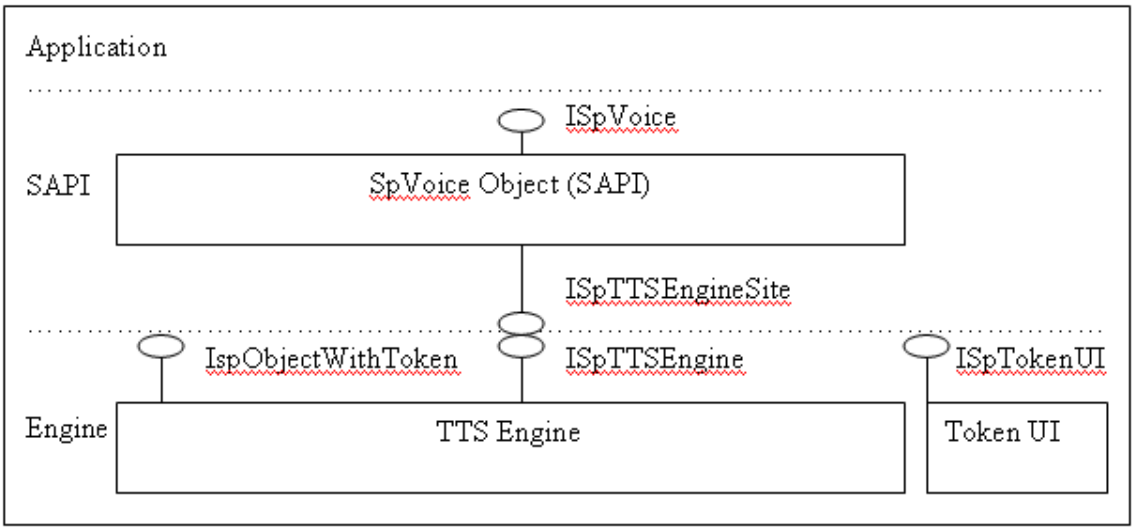
Improving Sphinx recognition rate with training.

Sphinx can be trained to better satisfy a specific client with the SphinxTrain program. Even after training, the rate of accuracy for Sphinx II is about 50%, and for Sphinx 3 delivered at the end of 2002, the rate is about 70%. In comparison, the rate of accuracy in Microsoft’s Speech SDK recognition engine is 95% after voice training and microphone calibration.

Microsoft Speech Software Development Kit

The Microsoft Speech Software Development Kit (SDK) [6] is based on the Microsoft Speech API (SAPI), a layer of software that allows applications and speech engines to communicate in a standardized way. The MS Speech SDK provides both text-to-speech (TTS) and speech recognition (SR) functionality.

Fig.12-6, below, illustrates the TTS synthesis process.



[Fig.12-6]

The main blocks that participate in the text-to-speech conversion are:

ISpVoice - The interface, which is used by the application to access TTS functionality

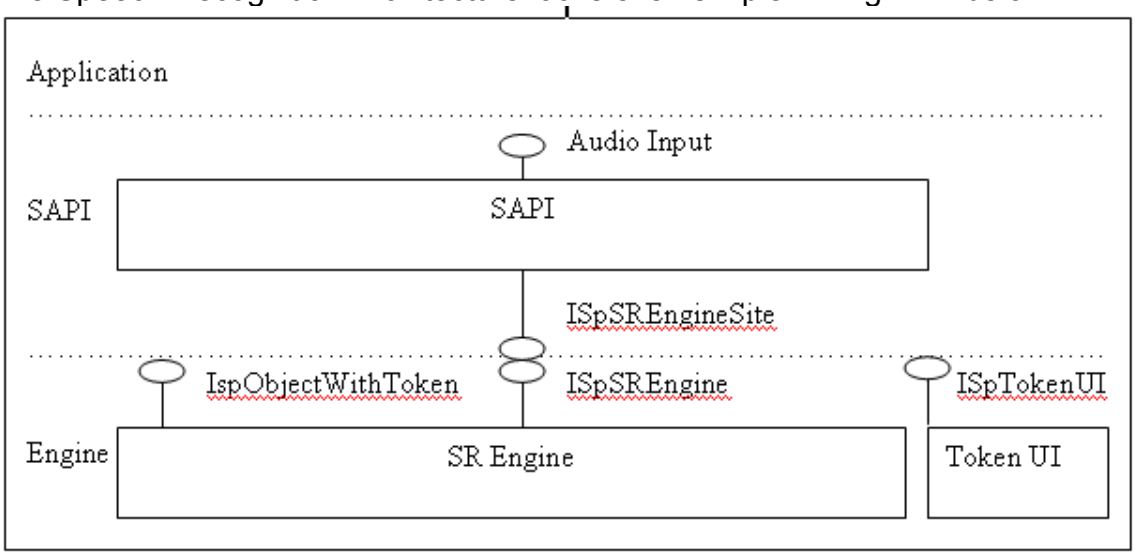
ISpTTSEngineSite - The engine interface to speak data and queue events

IspObjectWithToken - The interface to create and initialize the engine

ISpTTSEngine - The interface to call the engine

IspTokenUI - The way for the SAPI control panel to access the User Interface

The Speech Recognition Architecture looks even simpler in Fig.12-7 below.



[Fig.12-7]

The main speech recognition blocks interact in the following way.

1. The engine uses the ISpSREngineSite interface to call SAPI to read audio, and returns recognition results.
2. SAPI calls the engine using the methods of the *ISpSREngine* interface to pass details of recognition grammars. SAPI also uses these methods to start and stop recognition.
3. The *IspObjectWithToken* interface provides a mechanism for the engine to query and edit information about the object token.
4. *ISpTokenUI* represents User Interface components that are callable from an application.

SAPI 5 synthesis markup is not exactly SSML; it is closer to the format published by the SABLE Consortium. SAPI XML tags provide functionality such as volume control and word emphasis. These tags can be inserted into text

passed into *ISpVoice::Speak* and text streams of format SPDFID_XML, which are then passed into *ISpVoice::SpeakStream* and auto-detected (by default) by the SAPI XML parser. In the case of an invalid XML structure, a speak error may be returned to the application. We can change rate and volume attributes in real time using *ISpVoice::SetRate* and *ISpVoice::SetVolume*.

Volume

The Volume tag controls the volume of a voice and requires just one attribute: Level; an integer between zero and one hundred. The tag can be empty to apply to all following text, or it can frame a content, to which alone it applies.

<volume level="50">This text should be spoken at volume level fifty.

```
  <volume level="100">
    This text should be spoken at volume level one hundred.
  </volume>

</volume>
```

<volume level="80"/>All text which follows should be spoken at volume level eighty.

Rate

The Rate tag defines the rate of a voice with one of two attributes, Speed and AbsSpeed. The Speed attribute defines relative increase or decrease of the speed value, while AbsSpeed defines its absolute rate value; an integer between negative ten and ten. The tag can be empty to apply to all following text, or it can frame content to which alone it applies.

```
<rate absspeed="5">
  This text should be spoken at rate five.
  <rate speed="-5">
    Decrease the rate to level zero.
  </rate>
</rate>
<rate absspeed="10"/> Speak the rest with the rate 10.
```

Pitch

In a very similar manner, the Pitch tag controls the pitch of a voice with one of two attributes, Middle and AbsMiddle; an integer between negative ten and ten can represent an absolute as well as relative value.

```
<pitch absmiddle="5">
```

This text should be spoken at pitch five.
* <pitch middle="-5">*
*   This text should be spoken at pitch zero.*
* </pitch>*
</pitch>
<pitch absmiddle="10"/> All the rest should be spoken at pitch ten.

Zero represents the default level for rate, volume, and pitch values.

Emph

The Emph tag instructs the voice to emphasize a word or section of text. The Emph tag cannot be empty.

Your <emph>American Airline </emph> flight departs at <emph>eight </emph> tonight

Voice

The Voice tag defines a voice based with its *Age, Gender, Language*, *Name, Vendor*, and *VendorPreferred* attributes, that can be *Required* and *Optional*. These correspond exactly to the required and optional attributes parameters to *ISpObjectTokenCategory_EnumerateToken* and *SpFindBestToken* functions.

If no voice is found that matches all of the required attributes, no voice change will occur. Optional attributes are treated differently. In this case, the exact match is not necessarily expected. A voice that is closer to the provided attributes will be selected over one that is less similar.

Example:

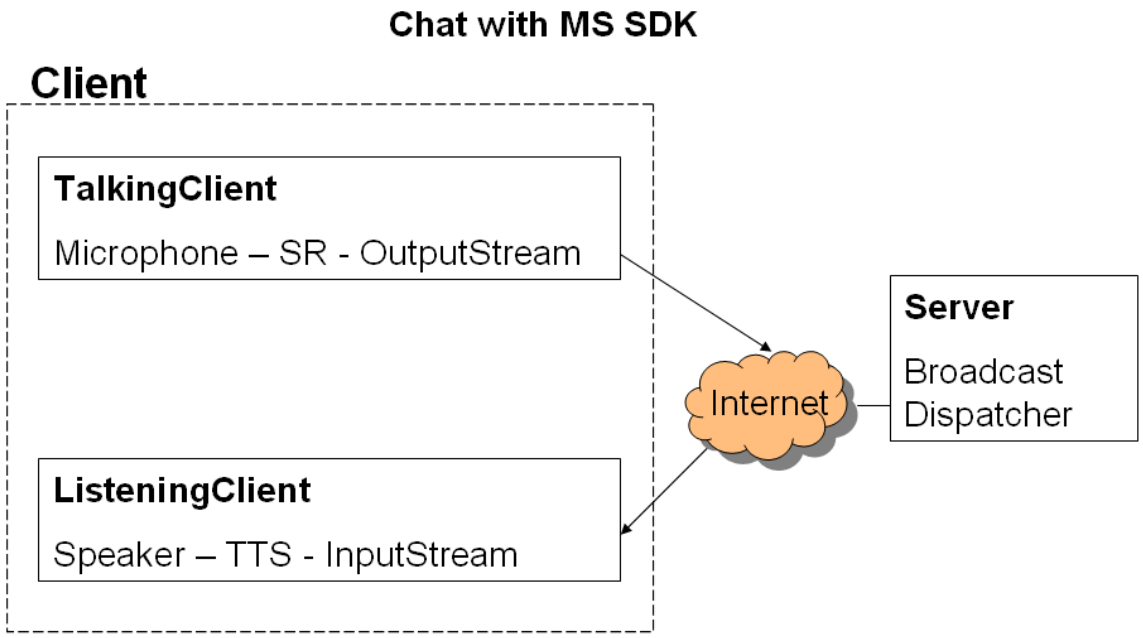
The default voice should speak this sentence.
<voice required="Gender=Female;Age!=Child">
A female non-child should speak this sentence, if one exists.
<voice required="Age=Teen">
* A teen should speak this sentence. If a female, non-child teen voice is present; this voice will be selected over a male teen voice, for example.*
* </voice>*
</voice>

Let us consider a demonstration program that uses text-to-speech and speech recognition facilities of the Microsoft Speech SDK.

Speech technology to decrease network bandwidth

The application is a conference between multiple clients over the Internet. The application utilizes speech technology to significantly decrease network bandwidth. Client programs intercept a user's speech, translate it to text, and send ASCII text over the Internet to the server-dispatcher. The server broadcasts the text it receives to the other clients (a regular chat schema). Client programs receive text from the server and convert it back to speech.

Fig.12-8 illustrates the application with a diagram.



[Fig.12-8]

More details:

- Speech recognition and TTS is done on the client side
- The client recognizes a phrase
- Plain text is transmitted between the client and the server
- The client program appends meta-data like the user's name in SSML format
- The server side can be implemented in C++, C#, or Java, using TCP/IP sockets

An example of the *TalkingClient* program can be found in Fig.12-9.cpp

```
/*****************
***TalkingClient.cpp***
*****************/
#include <iostream>
#include "Socket.h"

#include <windows.h>
#include <sapi.h>
```

```
#include <stdio.h>
#include <string.h>
#include <atlbase.h>
#include "sphelper.h"

using namespace std;

// Provided by Deena Malkina with use and modification of Microsoft Speech SDK examples

inline HRESULT ReadVoiceData(ISpRecoContext * voiceDataContext, ISpRecoResult **
recognitionData)
{
   HRESULT successLevel = S_OK;
   CSpEvent speechEvent;

   while (SUCCEEDED(successLevel) &&
         SUCCEEDED(successLevel = speechEvent.GetFrom(voiceDataContext)) &&
         successLevel == S_FALSE)
   {
      successLevel = voiceDataContext->WaitForNotifyEvent(INFINITE);
   }

   *recognitionData = speechEvent.RecoResult();
   if (*recognitionData)
   {
      (*recognitionData)->AddRef();
   }

   return successLevel;
}

int main(int argc, char* argv[])
{
   HRESULT successLevel = E_FAIL;
   // user name can be obtained from argv[1] or from system.properties
   const WCHAR * userName = L"Deena"; // from user profile
   const WCHAR * userGender = L"Female"; // from user profile
   const WCHAR * userAge = L"Teen";  // from user profile
   const WCHAR * voiceTag = sprintf(
     "<voice optional=\"Gender=%s;Age=%s;Name=%s\">",
     userGender,userAge,userAge);


   if(argc == 2) { // hopefully there is the voice profile on the name
      userName = argv[1];
   }
   try {
      // code to connect to recipient, for example, "ipserve.com, port=7445" via SocketClient
      SocketClient sender("ipserve.com",7445);
      // initialize Speech Engine
      if (SUCCEEDED(successLevel = ::CoInitialize(NULL)))
      {
         CComPtr<ISpRecoContext> context;
         CComPtr<ISpRecoGrammar> grammar;

         successLevel = context.CoCreateInstance(CLSID_SpSharedRecoContext);
```

```
        if (context &&
           SUCCEEDED(successLevel = context->SetNotifyWin32Event()) &&
           SUCCEEDED(successLevel = context->SetInterest(SPFEI(SPEI_RECOGNITION),
SPFEI(SPEI_RECOGNITION))) &&
           SUCCEEDED(successLevel = context->SetAudioOptions(SPAO_RETAIN_AUDIO,
NULL, NULL)) &&
           SUCCEEDED(successLevel = context->CreateGrammar(0, &grammar)) &&
           SUCCEEDED(successLevel = grammar->LoadDictation(NULL, SPLO_STATIC)) &&
           SUCCEEDED(successLevel = grammar->SetDictationState(SPRS_ACTIVE)))
        {
           USES_CONVERSION;

           // define the break signal that will send the sentence to the server
           const WCHAR * const breakSign = L"okey";
           const WCHAR * const exitSign = L"exit the program";
           CComPtr<ISpRecoResult> resultObject;

           printf( "You can start talking.\nSay \"%s\" to send your phrase.\n",
             W2A(breakSign) );

           while (SUCCEEDED(successLevel = ReadVoiceData(context, &resultObject)))
           {
              grammar->SetDictationState( SPRS_INACTIVE );

              CSpDynamicString resultingText;

              if (SUCCEEDED(resultObject->GetText(SP_GETWHOLEPHRASE,
SP_GETWHOLEPHRASE,
                                  TRUE, &resultingText, NULL)))
              {
                 printf("Said by %s:  %s\n", W2A(userName), W2A(resultingText));

                 // send text to server: voiceTag + resultingText + "</voice>";
                 char * textToSend = sprintf("%s%s</voice>", voiceTag,resultingText);
                 sender.SendLine(textToSend);

                 resultObject->SpeakAudio(NULL, 0, NULL, NULL);
                 resultObject.Release();
              }
              if (_wcsicmp(resultingText, breakSign) == 0)
              {
                 break;
              }

              grammar->SetDictationState( SPRS_ACTIVE );
           }
        }
     } catch(const char * e) {
        cerr << e <<endl;
     }
     ::CoUninitialize();
  }
  return successLevel;
}
```

[Fig.12-9]

The *TalkingClient* program takes user's name as an optional argument. If the argument is not provided, the "default" voice will be used.

The main routine starts with the socket connection to a recipient. We then initialize the speech engine and define the break signal that will be used to indicate when to send to the server what the user has said. In the example, the break signal is the word "okay."

The main processing happens in the while loop. The speech engine recognizes the user's sentence and prints it on the screen. Then the program sends the resulting text to the server with the additional SAPI XML Voice tag:

<voice optional="Gender=userGender;Age=userAge;Name=userName">
resultingText</voice>

The other part of the application is presented in Fig.12-10.cpp

```
/**
 * ListeningClient.cpp
 * By Deena.Malkina@javaschool.com
 * with use and modification of Microsoft Speech SDK examples
 */
#include "Socket.h"
#include <string>
#include <iostream>
#include <windows.h>
#include <sapi.h>
#include <stdio.h>

#define _ATL_APARTMENT_THREADED
#include <atlbase.h>
//You may derive a class from CComModule and use it if you want to override something,
//but do not change the name of _Module
extern CComModule _Module;
#include <atlcom.h>

#include <string.h>
//#include <atlbase.h>
#include "sphelper.h"

using namespace std;

int main() {
   const string machine="javaschool.com";
   const int port=7554;

   ISpVoice * pVoice = NULL;

   if (FAILED(::CoInitialize(NULL)))
    return FALSE;
```

```
    HRESULT hr = CoCreateInstance(
      CLSID_SpVoice, NULL, CLSCTX_ALL, IID_ISpVoice, (void **)&pVoice);


    try {
       SocketClient s(machine, port);
       String textToSpeak;

       if( SUCCEEDED( hr ) ) {
          while (1)  {
             // read one char at a time
             String c = s.ReceiveChar();
             if (c.empty()) break;

             cout << c;
             cout.flush();

             if(c=="." || c=="!" || c=="?") {
               // transform textToSpeak to WCHAR

               hr = pVoice->Speak(textToSpeak, 0, NULL);
             } else {
               textToSpeak.append(c);
             }
          }   // end of while loop
          pVoice->Release();
          pVoice = NULL;
       }
       // Reset or Uninitialize
       ::CoUninitialize();
    } catch (const char* s) {
       cerr << s << endl;
    } catch (String s) {
       cerr << s << endl;
    } catch (...) {
       cerr << "unhandled exception\n";
    }
    char q;
    cin>>q;
    return TRUE;
}
```

[Fig.12-10]

The *ListeningClient* requirements*:*

- Receive text from the server
- Transform text to speech using the voice profile if available

The *ListeningClient* program starts in a very similar manner. It uses the *ReceiveLine* method of the *SocketClient* to listen to messages coming from the

server. The program converts every unit of speech into voice and displays the line on the screen.

Examples of Socket classes for Windows can be found online [7].

Appendix 3, Sources, provides examples of text-to-speech and speech recognition programs written in C# using SAPI5.

Standards for scenarios for speech applications

Let us stop this overview of the parts of speech recognition technology for a minute. They are all important. At the same time, some of them are more important for system programmers who do the groundwork. Others pieces of the technology target application developers. Application developers can use this groundwork to describe application flow and write interpretation scenarios.

Our next step is to write scenarios for speech applications. Let us consider current and upcoming standards that can help. Note that the Microsoft .NET Speech SDK uses SSML, which is a part of W3C Speech Interface Framework (unlike the Microsoft Speech SDK that uses SAPI XML). SSML is a markup language to define text-to-speech processing, which is the simplest part of speech technology. There are two markup languages able to describe a complete speech interface: Speech Application Language Tags (SALT) [8], a relatively new upcoming standard, and VoiceXML [9], a well established technology with multiple implementations.

VoiceXML was developed for telephony applications as a high-level dialog markup language that integrates speech interface with data and control flow.

Unlike VoiceXML, SALT offers a lower level interface that strictly focuses on speech tags, but targets multiple devices, including but not limited to telephone systems.

VoiceXML, as well as SALT, uses such standards of the W3C Speech Interface Framework as SSML and Speech Recognition Grammar Standard (SRGS) [10]. SALT also includes recommendations on Natural Language Semantics Markup Language (NLSML) [11] as a recognition result format, and Call Control XML (CCXML) [12] as a call control language.

In a nutshell: NLSML is an XML-based markup for representing the meaning of a natural language utterance, and CCXML provides telephony call control support for VoiceXML or SALT, and other dialog systems.

NLSML uses an XForms data model for the semantic information being returned in the interpretation. (See NLSML and XForms overviews in the Appendix 2, XML Crossroads.)

SALT provides facilities for multi-modal applications that can include not only voice but also screen interfaces. SALT also gives developers the freedom to embed SALT tags into other languages. This allows for more flexibility in writing speak-and-display scenarios.

Speech Application Language Tags

SALT consists of a relatively small set of XML elements. Each XML element has associated attributes and DOM object properties, events and methods. One can write speech interfaces for voice-only and multi-modal applications using SALT with HTML, XHTML, and other standards. SALT controls dialog scenarios through the DOM event model that is popular in web software.

Three top-level elements in SALT are: *<listen …>, <prompt…>,* and *<dtmf..>.* First two XML elements define speech engine parameters.

<listen …> configures the speech recognizer, executes recognitions, and handles speech input events

<prompt …> configures the speech synthesizer and plays out prompts

The third XML element plays a significant role in call controls for telephony applications

<dtmf …> configures and controls dual-tone multi-frequency (DTMF) signaling in telephony applications. Telephony systems use DTMF to signal which key has been pressed by a client. Regular phones usually have 12 keys: ten decimal digit keys, and additional "#", and "*" keys. Each key corresponds to a different pair of frequencies.

The *listen* and the *dtmf* element may contain *<grammar>* and *<bind>* elements. The *listen* element can also include the *<record>* element.

The *<grammar>* element defines grammars. A single *<listen>* element can include multiple grammars. The *<listen>* element can have methods to activate an individual grammar before starting recognition. SALT itself is independent of the grammar formats, but for interoperability it recommends supporting at least the XML form of the W3C Speech Recognition Grammar Specification.

The *<bind>* element can inspect the results of recognition and provide conditional copy-actions. The *<bind>* element can cause the relevant data to be copied to values in the containing page. A single *<listen>* element may contain multiple binds. *<bind>* can have a conditional test attribute as well as a value attribute. *<bind>* uses XPath (see Appendix-XML on Xpath and other XML standards mentioned in the book) syntax in its value attribute to point to a particular node of the result. *<bind>* uses an XML pattern query in its conditional test attribute. If the

condition is true, the content of the node is bound into the page element specified by the *targetElement* attribute. The *onReco* event handler with script programming can provide even more complex processing. The <*onReco*> and the <*bind*> elements are triggered on the return of a recognition result.

The <*record*> element can specify parameters related to speech recording. <*bind*> or scripted code can process the results of recording, if necessary.

A spoken message scenario

Fig.12-11.xml demonstrates a scenario in which dialog flow is provided with a client-side script.

```
<!-- HTML -->
<html xmlns:salt="urn:saltforum.org/schemas/020124">
 <body onload="askForService()">
  <form id="messageForm"
    action="http://javaschool.com/school/public/knowledge/SALT/message"
            method="post">
   <input name="fromTextBox" type="text" />
   <input name="subjectTextBox" type="text" />
   <input name="recipientTextBox" type="text" />
   <input name="messageTextBox" type="text" />
  </form>

<!-- Speech Application Language Tags -->
  <salt:prompt id="askName"> What is your name? </salt:prompt>
  <salt:prompt id="askSubject"> What is the subject? </salt:prompt>
  <salt:prompt id="askRecipient"> Who is the recipient? </salt:prompt>
  <salt:prompt id="askMessage"> What is your message? </salt:prompt>
  <salt:prompt id="repeatDefault" onComplete="askForService()">
Please repeat your answer.
  </salt:prompt>
  <salt:listen id="nameRecognition"
 onReco="setName()" onNoReco="repeatDefault.Start()">
         <salt:grammar src="spokenMessage.xml" />
  </salt:listen>
  <salt:listen id="subjectRecognition"
 onReco="setSubject()" onNoReco="repeatDefault.Start()">
         <salt:grammar src="spokenMessage.xml" />
  </salt:listen>
  <salt:listen id="recipientRecognition"
 onReco="setRecipient()" onNoReco="repeatDefault.Start()">
         <salt:grammar src="spokenMessage.xml" />
  </salt:listen>
  <salt:listen id="messageRecognition"
 onReco="setMessage()" onNoReco="repeatDefault.Start()">
         <salt:grammar src="spokenMessage.xml" />
  </salt:listen>

<!-- script -->
  <script>
    // settings are based on user's answers
    function setName() {
```

```
    messageForm.fromTextBox.value = nameRecognition.text;
    askForService();
  }
  function setSubject() {
    messageForm.subjectTextBox.value = subjectRecognition.text;
    askForService();
  }
  function setRecipient() {
    messageForm.recipientTextBox.value = recipientRecognition.text;
    askForService();
  }
  function setMessage() {
    messageForm.messageTextBox.value = messageRecognition.text;
    messageForm.submit();
  }
          // the main script
  function askForService() {
    if messageForm.fromTextBox.value=="") {
      askName.Start();
      nameRecognition.Start();
    } else if (messageForm.subjectTextBox.value=="") {
      askSubject.Start();
      subjectRecognition.Start();
    } else if (messageForm.recipientTextBox.value=="") {
      askRecipient.Start();
      recipientRecognition.Start();
    } else if (messageForm.messageTextBox.value=="") {
      askMessage.Start();
      messageRecognition.Start();
    }
  }
 </script>
</body>
</html>
```

[Fig.12-11]

The scenario is actually an HTML page with embedded SALT tags and script functions. The *askForService*() script activates the SALT *<listen>* and *<prompt>* tags. For example, *askName.Start*() prompts the user with, “What is your name?”, and the following *nameRecognition.Start*() examines the recognition results. The *askForService*() script executes the relevant prompts and recognitions until all values are obtained. Successful message recognition triggers the *submit()* function, which submits the message to the recipient.

The user’s name can serve not only as the user’s signature, but can also invoke a chosen voice profile, if available, on recipient’s side.

Did you notice the reference to the *spokenMessage.xml* grammar file that supports the scenario in the code? How do we define grammar?

Grammar Definition

First, let us look into the existing Command and Control features of the MS Speech SDK. The Command and Control features of Speech API 5 (SAPI 5) are based on context-free grammars (CFGs). A CFG defines a specific set of words, and the sentences that are valid for recognition by the speech recognition (SR) engine.

The CFG format in SAPI 5 uses XML to define the structure of grammars and grammar rules. SAPI 5-compliant SR engines expect grammar definitions in a binary format produced by any CFG/Grammar compiler; for example, *gc.exe*, the SAPI 5 grammar compiler that is included in the Speech SDK. Compilation is usually done before application run-time, but can be done at run-time.

Here is an example of a file that provides grammar rules to navigate through mail messages (“next”, “previous”) and to retrieve the currently selected email (“getMail”).

```
<GRAMMAR LANGID="409">
   <DEFINE>
      <ID NAME="VID_MailNavigationRules" VAL="1"/>
      <ID NAME="VID_MailReceiverRules" VAL="2"/>
   </DEFINE>
   <RULE ID="VID_MailNavigationRules" >
     <L>
      <P VAL="next">
            <o>Please *+</o>
                  <p>next</p>
                  <o>message\email\mail</o>
            </P>
      <P VAL="previous">
            <o>Please *+</o>
                  <p>previous\last\back</p>
                  <o>message\email\mail</o>
            </P>
     </L>
   </RULE>
   <RULE ID="VID_MailReceiverRules" TOPLEVEL="ACTIVE">
      <O>Please</O>
      <P>
         <L>
            <P val="getMail">Retrieve</P>
            <P val="getMail">Receive</P>
            <P val="getMail">Get</P>
         </L>
      </P>
      <O>the mail</O>
   </RULE>
</GRAMMAR>
```

Appendix3, Sources, provides more examples (along with C# program source code) for a speech application based on SAPI5. The grammar file can be dynamically loaded and compiled at run-time. This would decrease the number of choices for any *current* recognition, and improve recognition quality.

VoiceXML

The last but definitely not the least important technology on the list is VoiceXML. Although SALT and VoiceXML have different targets, in some ways they compete in the speech technology arena. Unlike SALT, which is relatively new, VoiceXML started in 1995, within an AT&T project called Phone Markup Language (PML).

The VoiceXML Forum was formed in 1998-1999 by AT&T, IBM, Lucent, and Motorola. At that time Motorola had developed VoxML, and IBM was developing its own SpeechML. The VoiceXML Forum helped integrate the efforts. Since then VoiceXML had a history of successful implementations by multiple vendors.

Unlike SALT, which is a royalty-free upcoming standard, VoiceXML can be subject to royalty payments. Several companies, including IBM, Motorola, and AT&T, have indicated that they could have patent rights in VoiceXML.

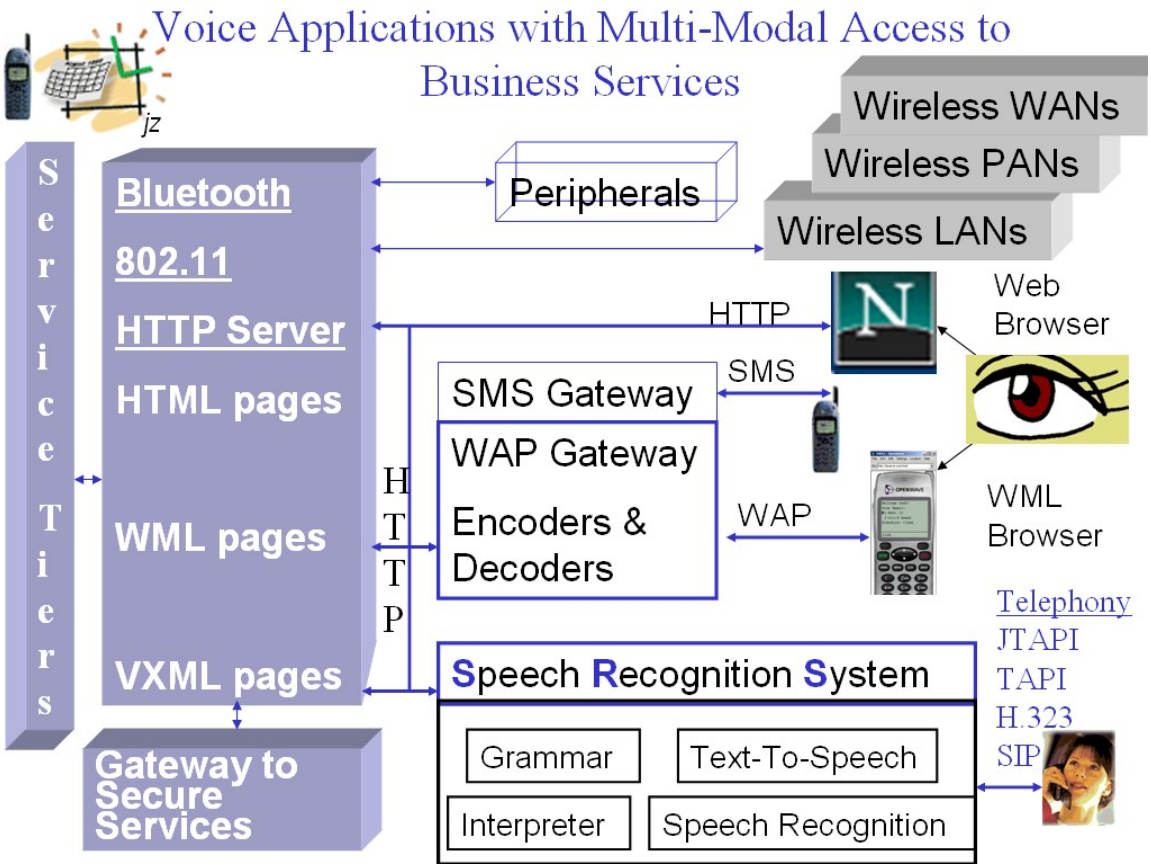
This brief overview of VoiceXML is based on the VoiceXML2.0 Specification submitted to W3C in the beginning of 2003.

What is VoiceXML?

VoiceXML is designed for creating dialog scenarios with digitized audio, speech recognition, and DTMF key input. VoiceXML can record spoken input, telephony, and mixed initiative conversations. The mixed conversation is an extended case of the most common type of computer-human conversations directed by the computer. The main target of VoiceXML is web-based development and content delivery to interactive speech applications.

The VoiceXML interpreter renders VoiceXML documents audibly, just as a web browser renders HTML documents visually. However,
standard web browsers run on the local machine, whereas the VoiceXML interpreter runs at a remote hosting site.

Fig.12-12 displays the enterprise application with multi-modal access to business services.



[Fig.12-12]

Like HTML web pages, VoiceXML documents have web URLs and can be located on any web server.
VoiceXML pages deliver the service content via speech applications using computer telephony protocols like JTAPI, TAPI, H.323 and SIP (Session Initiation Protocol, most widely accepted by the industry)

Main components of speech recognition systems.

Speech Recognition Systems (SRS) in general and VoiceXML systems in particular rely on high-performance server side hardware and software located on or connected to the web container. The web container is the architecture tier responsible for correspondence to clients over HTTP and dispatching client requests to proper business services. In this case, speech recognition services become the client that intercepts voice flow and translates it into HTTP streams. The key hardware factors for delivering reliable, scalable VoiceXML applications are:

- Telephony Connectivity
- Internet Connectivity
- Scalable Architecture

- Caching and Media Streaming
- CODECs - combinations of analog to digital (A/D) with digital to analog (D/A) signal converters

Progress in hardware technologies such as the high-speed, low-power consumption digital signal processor (DSP) has substantially contributed to improving CODEC conversion efficiency.

The SRS platform contains intelligent caching technology that minimizes network traffic by caching VoiceXML, audio files, and compiled grammars.
The VXML Platform makes extensive use of load balancing, resource pooling, and dynamic resource allocation.
SRS servers use multi-threaded C++ implementations, delivering the most performance from available hardware resources.
To prevent unnecessary re-compilation of grammars, the VoiceXML platform uses a high-performance indexing technique to cache and re-use previously compiled grammars.

Voice services offer the following software components to implement an end-to-end solution for phone-accessible Web content:

Telephony platform - software modules for text-to-speech, voice recognition, menuing system, parsing engines and DTMF

Briefly about the main telephony protocols:
JTAPI: The Java Telephony API supports telephony call control from consumer devices to call centers.
TAPI: The Telephony Application Programming Interface was created by Microsoft and Intel to provide computer telephony services
H.323: This standard for call signaling, multimedia transport and control is widely implemented for point-to-point and multi-point voice and videoconferencing over Integrated Services Digital Network (ISDN), Public Switched Telephone Network (PSTN) or Signaling System 7 (SS7), and 3G mobile networks.
SIP: The Session Initiation Protocol is commonly used for voice and video calls over Internet Protocol.

Open Standards support - Open system architecture in compliance with industry standards. VoiceXML, WAP, WML, XHTML, SSML, SRGS, NLSML, etc.

WAP solution - Support for using Wireless Application Protocol to deliver web and audio content to new web phones and enabling seamless integration between web and audio content.

Voice application and activation - User interface and logic (such as personalization) for accessing back-end audio content, and web and email databases for easy phone access

What is the VoiceXML architecture and how does it work?

A document server (a Web server) contains VoiceXML documents or VXML pages with dialog based scenarios. (I try to use the word “scenario” on every other page, but sometimes the word sneaks in-between.)

The document server responds to a client request by sending the VoiceXML document to a Speech Recognition System, or a VoiceXML implementation platform (the VoiceXML interpreter). A voice service scenario is a sequence of interaction dialogs between a user and an implementation platform.

Document servers perform business logic, database and legacy system operations, and produce VoiceXML documents that describe interaction dialogs. User input affects dialog interpretation by the VoiceXML interpreter. The VoiceXML interpreter transforms user input into requests submitted to a document server. The document server replies with other VoiceXML documents describing new sets of dialogs.

What does the VoiceXML document look like?

A VoiceXML document can describe:

- Output of synthesized speech (text-to-speech).
- Output of audio files.
- Recognition of spoken input.
- Recognition of DTMF input.
- Recording of spoken input.
- Control of dialog flow.

The VoiceXML language requires a common grammar format, namely the XML Form of the W3C Speech Recognition Grammar Specification (SRGS), to facilitate interoperability.

A voice application is a collection of one or more VoiceXML documents sharing the same *application root document*. A VoiceXML document is composed of one or more dialogs. The application entry point is the first VoiceXML document that the VoiceXML interpreter loads when it starts the application.
The developer’s task is to provide voice commands to the user in the most comfortable way while offering clearly distinguished possibilities of responses expected from the user through voice or/and telephone keys
.

There are two kinds of dialogs: *forms* and *menus*. Forms define an interaction that collects field-values. Each field may specify a grammar with expected inputs for that field. A menu commonly asks the user to choose one of several options, and then uses the choice to transition to another dialog.

Fig.12-13.vxml presents a very simple example of a VoiceXML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
  <form>
  <field name="service">
     <prompt>Would you like to read your mail, send a message, or check your calendar?</prompt>
     <grammar src="com.its.services.grxml" type="application/srgs+xml"/>
  </field>
  <block>
     <submit next="http://javaschool.com/school/public/speech/vxml/service.jsp"/>
  </block>
 </form>
</vxml>
```

[Fig.12-13]

This VoiceXML document provides a form dialog that offers user a choice of services to fill the service field. Expected answers are provided in the grammar document “com.its.services.grxml”.

Each dialog has one or more speech and/or DTMF *grammars* associated with it. Most of the speech applications today are *machine directed*. A single dialog grammar is active at any current time for machine directed applications, the grammar associated with a current user dialog. In *mixed initiative* applications, the user and the machine alternate in determining what to do next. In this case, more than one dialog grammar can be active, and the user can say something that matches another dialog’s grammar. *Mixed initiative* adds flexibility and power to voice applications.

VoiceXML can handle events not covered by the form mechanism described above. There are default handlers for the predefined events; plus, developers can override these handlers with their own event handlers in any element that can throw an event. The platform throws events, for example, when the user does not respond, does not respond intelligibly, requests help, etc.

The <*catch*>, <*error*>, <*help*>, <*noinput*>, and <*nomatch*> elements are examples of event handlers.

For example, the catch element can detect a disconnect event and provide some action upon the event:

```
 <catch event="connection.disconnect.hangup">
    <submit namelist="disconnect"
next="http://javaschool.com/school/public/speech/vxml/exit.jsp"/>
</catch>
```

Applications can support help by putting the help keyword in a grammar in the application root document.

```
<help>

  <prompt>Say "Retry" to retry authorization, or "Register" to hear the registration
instructions. Say "Exit" or "Goodbye" to exit.
  </prompt>

  <listen/>
</help>
```

A list of VoiceXML elements

<*assign*> - Assign a variable a value

<*audio*> - Play an audio clip within a prompt

<*block*> - A container of (non-interactive) executable code

<*catch*> - Catch an event

<*choice*> - Define a menu item

<*clear*> - Clear one or more form item variables

<*disconnect*> - Disconnect a session

<*else*> - Used in <*if*> elements

<*elseif*> - Used in <*if*> elements

<*enumerate*> - Shorthand for enumerating the choices in a menu

<*error*> - Catch an error event

<*exit*> - Exit a session

<*field*> - Declares an input field in a form

<*filled*> - An action executed when fields are filled

<*form*> - A dialog for presenting information and collecting data

<*goto*> - Go to another dialog in the same or different document

<*grammar*> - Specify a speech recognition or DTMF grammar

<*help*> - Catch a help event

<*if*> - Simple conditional logic

<*initial*> - Declares initial logic upon entry into a mixed initiative form

<*link*> - Specify a transition common to all dialogs in the link's scope

<*log*> - Generate a debug message

<*menu*> - A dialog for choosing amongst alternative destinations

<*meta*> - Define a metadata item as a name/value pair

<*metadata*> - Define metadata information using a metadata schema

<*noinput*> - Catch a *noinput* event

<*nomatch*> - Catch a *nomatch* event

<*object*> - Interact with a custom extension

<*option*> - Specify an option in a <*field*>

<*param*> - Parameter in <*object*> or <*subdialog*>

<*prompt*> - Queue speech synthesis and audio output to the user

<*property*> - Control implementation platform settings.

<*record*> - Record an audio sample

<*reprompt*> - Play a field prompt when a field is re-visited after an event

<*return*> - Return from a subdialog.

<*script*> - Specify a block of ECMAScript client-side scripting logic

<*subdialog*> - Invoke another dialog as a subdialog of the current one

<*submit*> - Submit values to a document server

<*throw*> - Throw an event.

<*transfer*> - Transfer the caller to another destination

<*value*> - Insert the value of an expression in a prompt

<*var*> - Declare a variable

<vxml> - Top-level element in each VoiceXML document

Fig.12-14.vxml introduces a typical VoiceXML document that initiates a brief phone conversation.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
  http://www.w3.org/TR/voicexml20/vxml.xsd"
  version="2.0">

 <form id="training">
  <field name="course">
  <grammar type="application/srgs+xml" src="/grammars/training.grxml"/>
  <prompt>Which course do you want to take?
    Here is the list of courses:
    <!-- list of courses offered -->
    </prompt>

  <if cond="course == 'operator' ">
     <goto next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
  </if>

  <noinput>
     I could not hear you.
     <reprompt/>
  </noinput>

  <nomatch count="1">
    Please select any Java, Wireless, or Ontology course from the list.
    <reprompt/>
  </nomatch>

  <nomatch count="2">
    <prompt>
     I am sorry, we have almost so many types of training courses but not this one.
```

```
      I would recommend you to start with the Ontology Introduction course at this time.
    </prompt>
   </nomatch>

   <nomatch count="3">
    I switch you to the operator.
    Hopefully you will find the course you want.
    Good luck.
    <goto next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
   </nomatch>

   </field>

   <block>
    <submit next="http://javaschool.com/school/public/speech/vxml/training.jsp"/>
   </block>
  </form>
</vxml>
```

[Fig.12-14]

The source code starts with the standard XML, and then has VXML reference lines. The next thing we see is a *form* that looks almost exactly like an HTML form. In fact, the form has exactly the same purpose – to collect information from a user into the form fields. This form has a single field named “course.”

The program prompts the user to choose one of the training courses.

```
<prompt>Which course do you want to take?</prompt>
```

The grammar line above the prompt defines a grammar rules file that will try to resolve the answer.

```
<grammar type="application/srgs+xml" src="/grammars/training.grxml"/>
```

The user might want to talk to a human being. In this case, the grammar rules might resolve user’s desire and return the “operator” word as the user’s selection.

The program uses an <if> element to check on this condition.

```
<if cond="course == 'operator' ">
```

If this condition is true, the program will use the <*goto*> element to jump to another document that transfers the caller to the operator.

Note that all tags are properly closed, as should be done in any XML file.

Looking down the code below the “if” element, we find <*noinput*> and <*nomatch*> event handlers. If the user produces no input during the default time, the program plays the prompt again using the <*reprompt*/> element.

```
<noinput>
   I could not hear you.
   <reprompt/>
</noinput>
```

The most interesting script starts when a user selection is not expected. In this case, the <*nomatch*> event handler is fired. This element can optionally have a counter, which we use here to try to provide a more appropriate response, and possibly decrease the user’s discomfort.

The very first “nomatch” element will provide an additional hint to the user and reprompt the original message.

```
<nomatch count="1">
  Please select a Java, Wireless, or Ontology course from the list.
  <reprompt/>
</nomatch>
```

The next time the user makes a strange selection, the program offers its candid advice.

```
<nomatch count="2">
  <prompt>
    I am sorry, we have so many types of training courses, but not this one.
    I would recommend for you to start with the Ontology Introduction course at
this time.
     Will that work for you?
  </prompt>
</nomatch>
```

The third “nomatch” event will switch user to the operator.

```
<nomatch count="3">
    I will switch you to the operator.
    Hopefully, you will find the course you want.
    Good luck.
  <goto next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
 </nomatch>
```

But what if the user was successful in the course selection?

In this case, the selected course value will fill the “course” field and the value will be submitted to the training page.

```
  <block>
    <submit next="http://javaschool.com/school/public/speech/vxml/training.jsp"/>
   </block>
```

The last two lines close the form and the VoiceXML document.

```
  </form>
</vxml>
```

Wow!

How does VoiceXML do the transfer operation? Here is the code.

```
      <!-- Transfer to the operator -->
       <!-- Say it first -->
       Transferring to the operator according your request.
       <!-- Play music while transfer -->
       <!-- Wait up to 60 seconds for the transfer  -->
       <transfer dest="tel:+1-234-567-8901"
         transferaudio="music.wav" connecttimeout="60s">
       </transfer>
```

The code extract first says, “Transferring to the operator according to your request,” and then actually tries to transfer the user to the operator. The *transfer* element in our example turns on some music and sets the timeout to 60 seconds for the transfer. There is also essential part of the transfer element – the telephone number of the operator.

Here is another transfer example when program catches the “busy” event.

```
  <transfer maxlength="60" dest="8005558355">
   <catch event="event.busy">
    <audio> busy </audio>
    <goto next="_home"/>
   </catch>
</transfer>
```

The <*link*> element below navigates to mail.vxml whenever the user says "mail".

```
<link next="mail.vxml">
   <grammar type="application/srgs+xml" root="root" version="1.0">
     <rule id="root" scope="public">mail</rule>
  </grammar>
```

```
 </link>
```

This example provides in-line grammar rules, unlike most of following examples where we reference gramar rules files.

The <*subdialog*> element helps to create reusable dialog components and decompose an application into multiple documents.

```
<subdialog name="compose" src="newmail.vxml">
     <filled>
       <!-- The "compose" subdialog returns 3 variables below.
         These variables must be specified in the  "return" element
         of the “compose” -->

       <assign name="to_address" expr=" compose.to_address"/>
       <assign name="subject" expr=" compose.subject"/>
       <assign name="message" expr=" compose.body"/>

     </filled>
 </subdialog>
```

Fig.12-15.vxml provides an example of the “new_mail” service request.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
  http://www.w3.org/TR/voicexml20/vxml.xsd"
  version="2.0">
 <form id="new_mail">
   <!-- two variables collected by the "compose" subdialog -->
   <var name="to_name"/>
   <var name="message"/>
   <subdialog name="compose" src="compose.vxml">
     <filled>
       <!-- The "compose" subdialog returns its status and two variables below.
         The status and other variables must be specified in the  "return" element
         of the "compose" -->

       <if cond="compose.status == 'OK'">
          <assign name="to_name" expr="compose.to_name"/>
          <assign name="message" expr="compose.message"/>
       <else/>
          Sorry, the system cannot deliver the message.
          <exit/>
       </if>

     </filled>
   </subdialog>

  <field name="subject">
```

```
    <grammar type="application/srgs+xml" src="/grammars/mail_subject.grxml"/>
     <prompt>
      What is the subject of your message?
     </prompt>
     <filled>
      <submit  next="http://javaschool.com/school/public/speech/send_mail.jsp"/>
     </filled>
    </field>
  </form>
</vxml>
```

[Fig.12-15]

The example uses the “compose” subdialog to fill two fields for the new mail form. The “compose” subdialog returns its status and two requested fields. If the returned status is not “OK,” the service says that the message cannot be delivered and exits.

Otherwise, the service assigns returned values to the “to_name” and “message” fields, and prompts the user for the message subject. It often happens that mail goes out without any subject.

The subject can serve as communication meta-data, which makes even more sense today when computer systems are increasingly involved in the communication process. With this last field, the service is ready to rock-n-roll and submits all the data to the long URL provided in the “submit” element.

The new_mail service listing also illustrates the usage of “if-else” elements with the conditional actions described above.

Fig.12-16 displays the “compose” subdialog.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
  <form id="compose">

    <var name="status" expr="'not_known_name'"/>
    <field name="to_name">
      <grammar type="application/srgs+xml" src="/grammars/names.grxml"/>
      <prompt> What is your <prosody rate="-40%">recipient</prosody> name? </prompt>
        <help>
          Please say first and last name of your message recipient.
          First name first. For example: John Smith.
         <reprompt/>
       </help>
       <nomatch>
        <return namelist="status"/>
```

```
      </nomatch>
    </field>

    <field name="message">
      <grammar type="application/srgs+xml" src="/grammars/phone_numbers.grxml"/>
      <prompt> Provide your <emphasis>message now</emphasis> </prompt>
    </field>

    <block>
      <assign name="status" expr="'OK'"/>
      <return namelist="status to_name message"/>
    </block>

  </form>
</vxml>
```

[Fig.12-16]

In the “compose” dialog, the prompt asks for a recipient’s name. Apparently, the grammar rules behind the scene are working hard to recover the email address from the list of available names. The user can ask for help to hear more detailed prompt messages. If the name recognition fails, the “*nomatch*” element returns the status value “*not_known_name*”, back to the “*new_mail*” service.

In the best-case scenario, when the name recognition succeeds, the “compose” dialog sets the status value to “OK” and prompts the user to fill (answer) the “message” field. The “compose” dialog then returns the “OK” status and two variables (the “*to_name*” and the “*message*”), back to the “*new_mail*” service.

The “*forward_mail*” service will reuse the same “compose” subdialog to collect the “*to_address*” and “message” fields. Fig.12-17 shows the “*forward_mail*” VoiceXML page.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
  <form id="forward_mail">
    <!-- two parameters related to the original mail passed from the mail_service -->
    <var name="subject"/>
    <var name="old_message"/>
    <!-- two variables collected by the "compose" subdialog -->
    <var name="to_name"/>
    <var name="message"/>
    <subdialog name="compose" src="compose.vxml">
      <filled>
        <!-- The "compose" subdialog returns its status and two variables below.
          The status and other variables must be specified in the  "return" element
          of the "compose" -->
```

```
      <if cond="compose.status == 'OK'">
         <assign name="to_name" expr="compose.to_name"/>
         <assign name="message" expr="compose.message"/>

         <!-- use ECMAScript to prepare subject and body fields -->
         <!-- subject will start with "FW: " -->
         <!-- body will include not only current but also original "old_message" -->
         <return namelist="to_name subject message" />
      <else/>
         Sorry, the system cannot deliver the message.
         <exit/>
      </if>

    </filled>
   </subdialog>

  </form>
</vxml>
```

[Fig.12-17]

The “forward_mail” listing includes two additional variables: “subject” and “old_message.” These variables passed as parameters extracted by the “mail_service” dialog from the original mail. The “forward_mail” service behaves similarly to the “new_mail” service.

If the “compose” subdialog returns an “OK” status with the two requested fields (the “to_address” and the “message”), the “forward_mail” service will submit all data, including the two additional fields (“subject” and “old_message”) that came as parameters from the original email, to the final URL. If the status returned by the “compose” subdialog is not as cheerful, the “forward_mail” service will not forward the message but will exit instead.

Parameters can be passed with the <param> elements of a <subdialog>. These parameters must be declared in the subdialog using <var> elements, as displayed in Fig.12-17.

The “mail_service” dialog passes the parameters to the “forward_mail” service with the following lines:

```
<form>
<subdialog name="forward_mail" src="forward_mail.vxml">
   <param name="subject" expr=" ‘ Hello’ "/>
   <param name=”old_message” expr=” ‘How are you?’ “/>
   <filled>
     <submit next="http://javaschool.com/school/public/speech/mail.jsp"/>
   </filled>
  </subdialog>
</form>
```

Looking into the PROMPT examples in Fig.12-16, we can see the tags we learned before as SSML elements. No wonder. The VoiceXML 2.0 Specification models the content of the <prompt> element based on the W3C Speech Synthesis Markup Language 1.0 (SSML), and makes available the following SSML elements:

<*audio*> - Specifies audio files to be played and text to be spoken.

<*break*> - Specifies a pause in the speech output.

<*desc*> - Provides a description of a non-speech audio source in <*audio*>.

<*emphasis*> - Specifies that the enclosed text should be spoken with emphasis.

<*lexicon*> - Specifies a pronunciation lexicon for the prompt.

<*mark*> - Ignored by VoiceXML platforms.

<*metadata*> - Specifies XML metadata content for the prompt.

<*paragraph*>*(*alias <p>) - Identifies the enclosed text as a paragraph, containing zero or more sentences

<*phoneme*> - Specifies a phonetic pronunciation for the contained text.

<*prosody*> - Specifies prosodic information for the enclosed text.

<*say-as*> - Specifies the type of text construct contained within the element.

<*sentence*> (alias <s>) - Identifies the enclosed text as a sentence.

<*sub*> - Specifies replacement spoken text for the contained text.

<*voice*> - Specifies voice characteristics for the spoken text.

The following example in Fig.12-18.vxml uses the <*record*> element to collect an audio recording from the user.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd">
   <form>
     <property name="bargein" value="true"/>

     <record  name="msg" beep="true" maxtime="10s"
```

```
      finalsilence="3000ms" dtmfterm="true" type="audio/x-wav">
      <prompt timeout="5s">
       Record your audio message after the beep.
      </prompt>
      <noinput>
       I didn't hear anything, please try again.
      </noinput>
    </record>

    <submit next="http://javaschool.com/school/public/speech/recording.jsp"
     enctype="multipart/form-data" method="post" namelist="msg"/>

  </form>
</vxml>
```

[Fig.12-18]

This example also uses the *bargein* property that controls whether a user can interrupt a prompt. Setting the *bargein* property to “true” allows the user to interrupt the program, introducing a *mixed initiative*.

The program prompts the user to record her or his message. A reference to the recorded audio is stored in the “msg” variable. There are several important settings in the *record* element, including timeouts and DTMFTERM.

The recording stops under one of the following conditions: a final silence for more than 3 sec occurs, a DTMF key is pressed, the maximum recording time, 10 sec, is exceeded, or the caller hangs up. The audio message will be sent to the web server via the HTTP POST method with the enctype="multipart/form-data”.

Another example, in Fig.12-19.vxml, demonstrates a VoiceXML feature that allows the user to enter text messages using a telephone keypad.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd">
<form id="key_message">
  <object name="message"
      classid="builtin://keypad_text_input">
   <prompt>
    Enter your message with the telephone keys.
    Press star for a space, and the pound sign to end the message.
   </prompt>
  </object>

  <block>
   <assign name="document.key_message" expr="message.text"/>
   <goto next="#send_message"/>
  </block>
</form>
```

```
</vxml>
```

[Fig.12-19]

VoiceXML supports platforms with telephone keys. In the example above the user is prompted to type the message. The <*block*> element copies the message to the variable *document.key_message.*

This example shows the usage of the *object* element, a part of ECMAScript [13].

ECMAScript

Developed under the European Computer Manufacturers Association (ECMA), ECMAScript was modeled after JavaScript but designed as application-independent. The language was divided into two parts: a domain independent core, and a domain specific object model. ECMAScript defines a language core, leaving the design of domain object model to specific vendors.

An ECMAScript object, presented in the example, can have following attributes:

name - When the object is evaluated, it sets this variable to an ECMAScript value whose type is defined by the object.

expr - The initial value of the form item variable; default is the ECMAScript value "*undefined*". If initialized to a value, then the form item will not be visited unless the form item variable is cleared.

cond - An expression that must evaluate to true after conversion to boolean in order for the form item to be visited.

classid - The URI specifying the location of the object's implementation. The URI conventions are platform-dependent.

codebase - The base path used to resolve relative URIs specified by classid, data, and archive. It defaults to the base URI of the current document.

codetype - The content type of data expected when downloading the object specified by *classid*. The default is the value of the type attribute.

data - The URI specifying the location of the object's data. If it is a relative URI, it is interpreted relative to the *codebase* attribute.

type - The content type of the data specified by the data attribute.

archive - A space-separated list of URIs for archives containing resources relevant to the object, which may include the resources specified by the *classid* and data attributes.

ECMAScript provides scripting capabilities for Web-based client-server architecture and makes it possible to distribute computation between the client and server. Each Web browser and Web server that supports ECMAScript supports (in its own way) the ECMAScript execution environment.

Some of the facilities of ECMAScript are similar to Java and Self [14] languages.

An ECMAScript program is a cluster of communicating objects that consist of an unordered collection of *properties* with their *attributes.* Attributes, like "ReadOnly", "DontEnum", "DontDelete", or "Internal", determine how each property can be used.

For example, the property with the "ReadOnly" attribute is not changeable and not executable by ECMAScript programs, the "DontEnum" properties cannot be enumerated in the programming loops, your attempts to delete the "DontDelete" properties will be ignored, and the "Internal" properties are not directly accessible via the property access operators.

ECMAScript properties are containers for objects, *primitive value*s, o*r method*s. A primitive value is a member of one of the following built-in types: *Undefined, Null, Boolean, Number,* and *String*.

ECMAScript defines a collection of *built-in objects* that include the following object names: *Global, Object, Function, Array, String* (yes, there objects with the same names as built-in primitive types), *Boolean, Number, Math, Date, RegExp*, and several *Error* object types.

ECMAScript in VoiceXML documents

Fig.12-20.vxml presents ECMAScript embedded into the "forward_mail" subdialog.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
  http://www.w3.org/TR/voicexml20/vxml.xsd"
  version="2.0">
 <form id="forward_mail">
  <!-- two parameters related to the original mail passed from the mail_service -->
  <var name="subject"/>
  <var name="old_message"/>
  <!-- two variables collected by the "compose" subdialog -->
  <var name="to_name"/>
  <var name="message"/>
  <subdialog name="compose" src="compose.vxml">
   <filled>
    <!-- The "compose" subdialog returns its status and two variables below.
```

```
        The status and other variables must be specified in the  "return" element
        of the "compose" -->

      <if cond="compose.status == 'OK'">
        <assign name="to_name" expr="compose.to_name"/>
        <assign name="message" expr="compose.message"/>
        <!-- use ECMAScript to prepare subject and body fields -->
        <script>
          <![CDATA[
          subject = 'FW: ' + subject;
          message = message + '\n----- Original message ----\n' + old_message;
          ]]>
        </script>
        <!-- return all data -->
        <return namelist="to_name subject message" />
      <else/>
        Sorry, the system cannot deliver the message.
        <exit/>
      </if>

    </filled>
   </subdialog>

  </form>
</vxml>
```

[Fig.12-20]

Several lines of the ECMAScript give the final touch to the “forward_mail” dialog. The subject of the forwarded message will start with “FW:“ and the body of the message will include not only the current message provided by the user, but also the original message that the user wants to forward to another recipient.

Grammar rules

According to the VoiceXML 2.0 Specification, platforms should support the Augmented BNF (ABNF) Form of the W3C Speech Recognition Grammar Specification, although VoiceXML platforms may choose to support grammar formats other than SRGS.

The <grammar> element may specify an *inline* grammar or an *external* grammar. Fig.12-21 demostrates an example of inline grammar.

```
<grammar mode="voice" xml:lang="en-US" version="1.0" root="training">
  <!-- Selection of one of the training courses -->
  <rule id="course" scope="public">
    <one-of>
      <item> Java Introduction </item>
      <item> Advanced Java </item>
      <item> Wireless Introduction </item>
      <item> Java Microedition </item>
      <item> Speech Technologies </item>
```

```
    <item> Ontology Introduction </item>
    <item> Integration Technologies </item>
    <item> Knowledge and Service Integration </item>
    <item> Natural User Interface </item>
   </one-of>
  </rule>
</grammar>
```

[Fig.12-21]

This simple example provides inline grammar rules for the selection of one of many items.

In a similar manner, VoiceXML allows developer to provide DTMF grammar rules.

```
<grammar mode="dtmf" weight="0.3"
src="http://javaschool.com/school/public/speech/vxml/dtmf.number"/>
```

The grammar above includes references to the dtmf grammar file. The extract below shows inline dtmf grammar rules.

```
<grammar mode="dtmf" version="1.0" root="code">
  <rule id="root" scope="public">
    <one-of>
      <item> 1 2 3 </item>
      <item> # </item>
    </one-of>
  </rule>
</grammar>
```

The VoiceXML interpreter evaluates its own performance.

The *application.lastresult$* variable holds information about the last recognition. The *application.lastresult$[i].confidence* can vary from 0.0 to1.0. A value of 0.0 indicates minimum confidence

The application.lastresult$[i].utterance keeps the raw string of words (or digits for DTMF)) that were recognized for this interpretation.

The *application.lastresult$[i].inputmode* stores the last mode value (dtmf or voice).

The *application.lastresult$[i].interpretation* variable contains the last interpretation result.

This self-evaluation feature can be used to provide additional confirmational prompt when necessary.

<if cond="application.lastresult$.confidence < 0.7">
<goto nextitem="confirmationdialog"/>
<else/>

Resources and Caching

A VoiceXML interpreter fetches VoiceXML documents and other resources, such as audio files, grammars, scripts, and objects, using powerful caching mechanisms. Unlike a visual browser, a VoiceXML interpreter lacks end user controls for cache refresh, which is controlled only through appropriate use of the *maxage* and *maxstale* attributes in vxml documents.

The *maxage* attribute indicates that the document is willing to use content whose age is no greater than the specified time in seconds. If the *maxstale* attribute is assigned a value, then the document is willing to accept content that has exceeded its expiration time by no more than the specified number of seconds.

Metadata

VoiceXML does not require metadata information. However, it provides two elements in which metadata information can be expressed: <*meta*> and <*metadata*>, with the recommendation that metadata is expressed using the <*metadata*> element, with information in Resource Description Framework (RDF).

Similarly to HTML, the <*meta*> element can contain a metadata property of the document expressed by the pair of attributes, *name* and *content*.

<meta name="generator" content="http://JavaSchool.com"/>

The <*meta*> element can also specify HTTP response headers with *http-equiv* and *content* attributes.

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />

A VoiceXML document can include the <metadata> element using the Dublin Core version 1.0 RDF schema [15].

Fig.12-22.vxml provides an example of a VoiceXML document with the <*metadata*> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd">
```

```
<metadata>
  <rdf:RDF
      xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs = "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
      xmlns:dc = "http://purl.org/metadata/dublin_core#">

<!-- Metadata about the VoiceXML document -->
  <rdf:Description about="http://javaschool.com/school/public/speech/vxml/training.vxml"
      dc:Title="Training Courses"
      dc:Description="Training Courses List"
      dc:Publisher="ITS"
      dc:Language="en"
      dc:Date="2003-05-05"
      dc:Rights="Copyright 2003 Jeff Zhuk"
      dc:Format="application/voicexml+xml" >
  </rdf:Description>
 </rdf:RDF>
</metadata>

 <form id="training">
  <field name="course">
  <grammar type="application/srgs+xml" src="/grammars/training.grxml"/>
  <prompt>Which course do you want to take?
    Here is the list of courses:
    <!-- list of courses offered -->
    </prompt>

  <if cond="course == 'operator' ">
     <goto next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
  </if>

  <noinput>
      I could not hear you.
      <reprompt/>
  </noinput>

  <nomatch count="1">
    Please select any Java, Wireless, or Ontology course from the list.
    <reprompt/>
  </nomatch>

  <nomatch count="2">
    <prompt>
      I am sorry, we have almost so many types of training courses but not this one.
      I would recommend you to start with the Ontology Introduction course at this time.
    </prompt>
  </nomatch>

  <nomatch count="3">
    I switch you to the operator.
    Hopefully you will find the course you want.
    Good luck.
    <goto next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
  </nomatch>

  </field>
```

```
   <block>
    <submit next="http://javaschool.com/school/public/speech/vxml/training.jsp"/>
   </block>
  </form>
</vxml>
 </form>
</vxml>
```

[Fig.12-22]

The <*metadata*> element provides hidden (and silent) information about the document, which nonetheless serves (or will serve) an extremely important role in the interconnected world. This information feeds search engines and helps end users find the document.

The metadata element ends our voyage into VoiceXML technology, and also ends this chapter.

Summary

This chapter reviewed voice technologies, speech synthesis and recognition, related standards, and some implementations.

VoiceXML-based technology is the most mature, and is prime-time ready for what it was designed for: telephony applications that offer menu driven voice dialogs that eventually lead to services.

Data communications is growing, and wireless devices will begin to exchange more data packets outside than inside of the telephony world. At that point, the lightness and multi-modality of SALT will make it a stronger competitor.

Neither of these technologies was designed for a natural language user interface. One common limitation is the grammar rules standard defined by the SRGS. They fit perfectly into the multiple-choice world, but have no room for the thoughtful process of understanding.

Integrating Questions

What are the common features of speech application architectures?

What role plays XML in speech applications?

Case Study

1. *Create a SALT file, similar to Fig.12-11.xml, that is related to a book order.*
2. *Create a grammar file to support ordering a book.*
3. *Describe an application at your workplace that can benefit from speech technology*

References

1. The Java Speech API - http://java.sun.com/products/java-media/speech
2. The Java™ Speech API Markup Language *(JSML)* - http://java.sun.com/products/java-media/speech/forDevelopers/JSML
3. Speech Synthesis Markup Language - http://www.w3.org/TR/speech-synthesis/
4. The Java™ Speech Grammar Format *(JSGF) Specification* – http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/
5. Sphinx, open source speech recognition project - http://sourceforge.net/projects/cmusphinx/
6. Microsoft Speech SDK - http://download.microsoft.com/download/speechSDK/
7. Rene Nyffenegger, A C++ Socket Class for Windows, online, Internet, Dec. 20, 2002. http://www.adp-gmbh.ch/win/misc/sockets.html
8. Speech Application Language Tags (SALT) Technical White Paper, online, SALTforum, Internet, 01/20/2002. Available: http://www.saltforum.org/spec.asp
9. VoiceXML - www.voicexml.org/spec.html
10. Speech Recognition Grammar Standard (SRGS), http://www.w3.org/TR/speech-grammar
11. Natural Language Semantics Markup Language - http://www.w3.org/TR/nl-spec/
12. Call Control XML - http://www.w3.org/TR/ccxml/
13. Standard ECMA-262 ECMAScript Language Specification http://www.ecma-international.org/
14. Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227–241, Orlando, FL, October 1987
15. "Dublin Core Metadata Initiative ", a Simple Content Description Model for Electronic Resources. - http://purl.org/DC/